A Hardware Evaluation of a NIST Lightweight Cryptography Candidate

Flora A. Coleman

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

William J. Diehl, Chair

Leyla Nazhandali

Joseph G. Tront

April 28, 2020

Blacksburg, Virginia

Keywords: Lightweight Cryptography, Side Channel Analysis, FPGA, ARX

A Hardware Evaluation of a NIST Lightweight Cryptography Candidate

Flora A. Coleman

(ABSTRACT)

The continued expansion of the Internet of Things (IoT) in recent years has introduced a myriad of concerns about its security. There have been numerous examples of IoT devices being attacked, demonstrating the need for integrated security. The vulnerability of data transfers in the IoT can be addressed using cryptographic protocols. However, IoT devices are resource-constrained which makes it difficult for them to support existing standards. To address the need for new, standardized lightweight cryptographic algorithms, the National Institute of Standards and Technology (NIST) began a Lightweight Cryptography Standardization Process. This work analyzes the Sparkle (Schwaemm and Esch) submission to the process from a hardware based perspective. Two baseline implementations are created, along with one implementation designed to be resistant to side channel analysis and an incremental implementation included for analysis purposes. The implementations use the Hardware API for Lightweight Cryptography to facilitate an impartial evaluation. The results indicate that the side channel resistant implementation resists leaking data while consuming approximately three times the area of the unprotected, incremental implementation and experiencing a 27% decrease in throughput. This work examines how all of these implementations perform, and additionally provides analysis of how they compare to other works of a similar nature.

A Hardware Evaluation of a NIST Lightweight Cryptography Candidate

Flora A. Coleman

(GENERAL AUDIENCE ABSTRACT)

In today's society, interactions with connected, data-sharing devices have become common. For example, devices like "smart" watches, remote access home security systems, and even connected vending machines have been adopted into many people's day to day routines. The Internet of Things (IoT) is the term used to describe networks of these interconnected devices. As the number of these connected devices continues to grow, there is an increased focus on the security of the IoT. Depending on the type of IoT application, a variety of different types of data can be transmitted. One way in which these data transfers can be protected is through the use of cryptographic protocols. The use of cryptography can provide assurances during data transfers. For example, it can prevent an attacker from reading the contents of a sensitive message. There are several well studied cryptographic protocols in use today. However, many of these protocols were intended for use in more traditional computing platforms. IoT devices are typically much smaller in size than traditional computing platforms. This makes it difficult for them to support these well studied protocols. Therefore, there have been efforts to investigate and standardize new lightweight cryptographic protocols which are well suited for smaller IoT devices. This work analyzes several hardware implementations of an algorithm which was proposed as a submission to the National Institute of Standards and Technology (NIST) Lightweight Cryptography Standardization Process. The analysis focuses on metrics which can be used to evaluate its suitability for IoT devices.

Dedication

To my family for their unwavering support.

Acknowledgments

I would like to thank my advisor, Dr. William Diehl for his guidance and patience during this process. This work significantly expanded my knowledge and interest in this field and Dr. Diehl's assistance played a pivotal role. Thank you to Dr. Leyla Nazhandali for her continued support throughout my undergraduate and graduate career, and for serving as a member of my committee. Thank you to Dr. Joseph Tront for serving as a member of my committee and a special thank you to both Dr. Tront and Dr. Ingrid Burbey for their support over the past three years.

I would also like to acknowledge my Signatures Analysis Lab team members Behnaz Rezvani and Sachin for their contributions. Thank you to Behnaz for her work on the extension to the LWC HW DP and her willingness to answer questions. Thank you to Sachin for testing my implementations on hardware to verify their correct operation and running the statistical analysis tests.

This material is based upon work supported by the National Science Foundation under Grant Number 1303297 and the U.S. Department of Commerce, National Institute of Standards and Technology (NIST) Award 70NANB18H219 for Lightweight Cryptography in Hardware and Embedded Systems. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding parties.

Contents

List of Figures

List of Tables

List of Abbreviations

AEAD: Authenticated encryption with associated data

API: Application Programming Interface

ARX: Addition-Rotation-XOR

CPA: Correlation Power Analysis

DPA: Differential Power Analysis

FPGA: Field Programmable Gate Array

HW DP: Hardware Development Package

IoT: Internet of Things

LUT: Lookup Table

LWC: Lightweight cryptography

NIST: National Institute of Standards and Technology

SCA: Side channel analysis

SPA: Simple Power Analysis

SPN: Substitution-Permutation Network

XOR: Exclusive OR

Chapter 1

Introduction

As the Internet of Things (IoT) continues to grow, there is a heightened need for integrated security measures. One aspect of a network of IoT devices which requires attention is the protection of data transfers. Existing cryptographic standards have been thoroughly studied and vetted. However, they are not well suited for the IoT as they usually require a level of resource utilization that is not feasible for smaller devices. Alternatively, lightweight cryptographic algorithms are designed specifically for resource constrained devices. In an effort to evaluate and standardize these algorithms, the National Institute of Standards and Technology (NIST) began the Lightweight Cryptography (LWC) Standardization Process in 2018. In the call for algorithms, NIST specified several evaluation metrics for the candidates including the suitability of an algorithm for software and hardware, its performance, and its ability to resist side channel attacks [19]. Of the candidates still in consideration, only two families use ARX-based primitives. As discussed in Chapter 2, ARX cryptography has some interesting properties and potential for more exploration. Considering these factors, this work analyzes an ARX-based candidate from a hardware perspective. The central research question focuses on how a selected NIST LWC Candidate satisfies the performance and resistance to side channel attack metrics in hardware.

The main contributions of this work include:

- **Hardware Implementations of a NIST LWC Candidate:** Four Register Transfer Level (RTL) implementations of the selected candidate are detailed. These implementations include two baseline implementations, one side channel resistant implementation, and an incremental implementation included for analysis purposes. All implementations are compliant with the Hardware Application Programming Interface for Lightweight Cryptography [43]. To the best of my knowledge, this is the first hardware based side channel resistant implementation of the NIST Candidate SCHWAEMM.
- **Third party analysis in support of the NIST LWC Standardization Process:** The results of the implementations developed and a comparison of these results to other similar works provide additional data and impartial insight in aid of the ongoing process.
- **Protected extension to the Development Package for the Hardware API for Lightweight Cryptography:** The Python script genShared.py enables the creation of n-share text vectors which are compatible with the modified test bench used in the extension to the Development Package.

All necessary background information for an understanding of this work is provided in Chapter 2. Chapter 3 details the methodology used to create the RTL implementations and Chapter 4 will provide an overview of the results. Finally, Chapter 5 will provide a synopsis of how the results compare to related works and Chapter 6 will supply some closing thoughts.

Chapter 2

Background

2.1 The Internet of Things

The Internet of Things (IoT) encompasses vast networks of interconnected devices which share data. It has continued to expand in recent years, with the number of IoT devices connected worldwide expected to increase to 43 billion by 2023, roughly three times the number connected in 2018 [13]. The IoT can be leveraged for a variety of different applications, for example commercial products like "smart" appliances or even solutions in the healthcare industry. A 2015 report from the McKinsey Global Institute analyzes the IoT over nine different application fields [30]. The report indicates that the IoT will have anywhere from a $3.9 trillion to $11.1 trillion impact on the worldwide economy in 2025. With the proliferation of IoT devices being introduced, the security of the IoT is increasingly called into question.

To date, there have been numerous examples of IoT products falling victim to attacks. In a 2016 study on the security of four different IoT connected surveillance systems, authors discovered serious vulnerabilities in the cameras [37]. Of interest was a security risk associated with the encryption scheme used by one of the cameras. The manufacturers of the camera had used a proprietary encryption algorithm that the authors of the study were able to analyze and break. They also explained that it was possible for an attacker to watch users' video streams [37]. Even when considering non-critical IoT applications, it is important to

ensure the privacy of messages being transmitted.

Encrypting data transmissions can provide security for IoT devices. However, the use of established cryptographic standards is not an ideal solution for IoT devices. Existing standards, such as AES or DES, were designed with target devices like desktop computers in mind [31]. These standards can quickly consume platform resources like memory and power. Small IoT devices do not offer these resources on the same scale, making it impractical for them to support these existing standards. Consequently, there is a need for cryptographic algorithms that require fewer resources while still offering communication security.

2.2 Lightweight Cryptography

Lightweight cryptography (LWC) can be distinguished from traditional cryptographic algorithms by its comparatively small resource footprint. LWC algorithms are specifically designed to consume fewer device resources, offering an appealing security solution that can reasonably be integrated into the IoT. Several examples of lightweight cryptographic algorithms which have been examined in research include PRESENT, SIMON, and SPECK [28]. Lightweight cryptography continues to evolve alongside the growth of the IoT. In 2017, NIST released their Report on Lightweight Cryptography [31]. The report identifies a need for LWC standards and cites the rapid evolution of lightweight cryptography as the reason for compiling a portfolio of new lightweight solutions.

2.2.1 NIST Lightweight Cryptography Standardization Process

In August of 2018, the National Institute of Standards and Technology (NIST) began the Lightweight Cryptography Standardization Process to collect and evaluate lightweight cryp-

tographic algorithm submissions [19]. All submissions were required to provide an authenticated encryption with associated data (AEAD) scheme. The inclusion of a hashing scheme was optional. The advantage of AEAD [40] is that it effectively ties associated data to the encrypted message while providing confidentiality, authenticity, and integrity. In other words, an AEAD scheme can hide the contents of a message from unauthorized parties, verify the identity of the transmitting party, and ensure that no part of the message has been modified in transit. The NIST LWC call for algorithms specified several criteria for evaluation including but not limited to performance, cost, resistance to side channel analysis and fault attacks, and "suitability for hardware and software implementations" [19]. NIST tasked a group of their own researchers with evaluating these submissions, but public third party analysis was also "strongly encourage[d]" [19]. The first round of evaluation began in April 2019 with 56 candidates. In October 2019, the Status Report on the First Round of the NIST Lightweight Cryptography Standardization Process was released [45]. Of the original submissions, 32 advanced to the second round of evaluation. Many of the algorithms which did not advance to the second round were shown to be vulnerable to practical attacks. It is important to note that side-channel resistance was not discussed as a major factor in the first round results, suggesting that it will play a larger role in ongoing rounds. The report also made it clear that the performance criterion will be heavily emphasized during the second round of evaluation. As the second round is currently ongoing, the content of this work will examine the cost and trade-offs associated with creating a side-channel resistant implementation of a NIST LWC Round 2 candidate.

2.2.2 The Hardware API for Lightweight Cryptography

The Hardware Application Programming Interface for Lightweight Cryptography (LWC API), first released in October 2019, facilitates the impartial evaluation and comparison

of hardware implementations of candidates in the NIST LWC Standardization Process [24]. The corresponding Hardware Development Package (LWC HW DP), available at [43] and detailed in [44], aids in the creation of LWC API compliant implementations. The LWC HP DP has a few central components. Figure 2.1 provides a visual representation of the structure of these components.

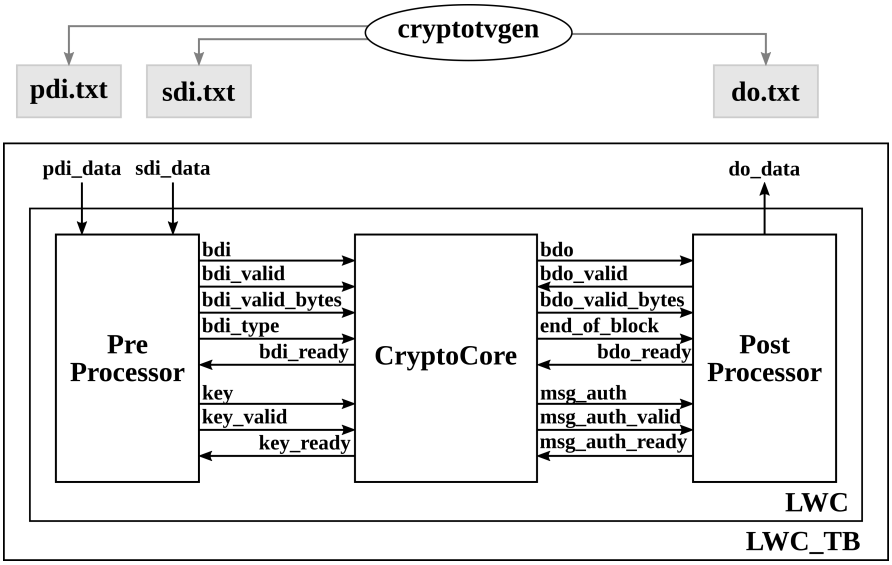

Figure 2.1: Development Package for Hardware Implementations, compliant with the LWC API. Some details are omitted for ease of understanding. [1]

A set of generic VHDL modules establishes a hardware interface for cipher implementations. The LWC top module consists of a PreProcessor, CryptoCore, and PostProcessor. The PreProcessor and PostProcessor handle the processing of test vector inputs and CryptoCore outputs respectively. Cipher implementations are contained within the CryptoCore module. Additionally, the LWC HW DP provides a method for creating test vectors, and a test bench which verifies that CryptoCore outputs match the expected results. The test vector generation process uses the Python module `cryptotvgen` to generate test bench compatible test vectors using the C reference implementations of LWC Candidates. Design using the LWC API will be discussed in more detail later.

[1] Figure 2.1 is an adaptation of a figure from [44]

2.3 Sparkle (Schwaemm and Esch)

One of the candidates which advanced to the second round of the NIST Lightweight Cryptography Standardization Process was Sparkle (Schwaemm and Esch), submitted by Beierle et al. [4]. Both the AEAD and hashing families rely upon the underlying Sparkle family of permutations. A total of two hashing instances and four AEAD instances were proposed, each using one of the three instances of the permutation. The instances analyzed in this work are the primary recommended instances, Schwaemm256-128 and Esch256. These instances both employ the Sparkle384 permutation.

2.3.1 An Overview

The Sparkle permutation is based upon its predecessor Sparx [16], with the main difference being the use of a fixed key and a larger block size in Sparkle. The general structure of Sparkle is a Substitution-Permutation Network (SPN) which uses an Addition-Rotation-XOR (ARX) primitive to achieve the substitution layer. The internal state is processed in 64-bit branches, with one branch consisting of two consecutive words of the state. Each branch experiences a non-linear transformation through the application of the four round ARX-box *Alzette*. Each of the four rounds uses a 32-bit round constant specific to the branch and the rotation amounts used per round vary. After the completion of the four ARX rounds, the state is processed through a linear layer. The linear layer applies a Feistel structure, with a rotation of the rightmost branches of the state left by one branch before swapping the leftmost branches of the state with the rightmost branches. The design of this linear layer leaves half of the branches of the state unmodified [4].

Another feature which the AEAD family and hashing family share is their adoption of a sponge-based mode of operation. The basic sponge mode of operation processes a state of

r rate bits and c capacity bits [6]. The initial state is zero filled and input blocks of length r are processed during the absorption phase. The squeezing phase starts after all input blocks have been processed. The output of the sponge process is released during this phase. A duplexed sponge construction can be used for AEAD schemes. It uses the basics of the sponge mode of operation while allowing output blocks to be released as each input block is finished processing [6].

The SCHWAEMM AEAD family employs a modified version of Beetle [10] as its mode of operation. Beetle, which is based on a duplexed sponge approach, leverages the use of combined feedback to provide a difference between the input to the permutation calls and the ciphertext output. As explored in [10], this provides some additional security. The specific mode of operation used in SCHWAEMM makes several modifications to the scheme used in Beetle. One major difference is the use of rate whitening which XORs the capacity to the rate prior to the start of the permutation. As mentioned above, the linear layer allows half of the branches of the state to remain unmodified after the completion of *Alzette*. Without rate whitening, attackers could potentially use this characteristic to gain partial information about the permutation [4]. Rate whitening prevents this from happening by making it so that the attacker would need knowledge of the capacity. Additional modifications to Beetle include making the key used the same size as the capacity, altering the way the tag is handled, and changing the constants used for domain separation to encode the capacity size [4].

The ESCH hashing family employs a modified version of the sponge mode of operation [6]. Specifically, minimum-size padding is supported and the message blocks are processed using "indirect injection" [4]. The indirect injection is a departure from the normal sponge mode because rather than simply XORing the message blocks into the rate, first the Feistel function used in the linear layer of the SPARKLE permutation is applied. This modification is used to improve upon the security properties of ESCH [4].

2.3.2 ARX Cryptography

A key feature of the Sparkle family of permutations is that it relies upon an ARX primitive for its source of non-linearity. ARX primitives supply fast performance in software, but do not necessarily provide the best trade-off in hardware [33]. Specifically, software implementations of ARX primitives offer better performance than traditional SPN-based ciphers [34]. One example of an ARX-based cipher in use today is the ChaCha family of ciphers [5]. The ChaCha20_Poly1305 AEAD combination [36] is one of the three currently supported options in the Transport Layer Security (TLS) Protocol Version 1.3 [2, 39]. The wide acceptance of TLS demonstrates the relevance of ARX-based cryptography. Several works investigate ARX-based ciphers in software, including an investigation of SPARX and CHAM (another ARX-based primitive which is used by a NIST LWC Round 2 Candidate) in [42]. However, in the literature reviewed there were fewer works available that considered ARX-based implementations on hardware, especially implementations resistant to side-channel attacks. This lack of assessment contributed to the motivation for this work.

2.4 Side Channel Analysis

Two methods which can be used to consider the security of a cryptographic algorithm include cryptanalysis and side channel analysis. Cryptanalysis focuses on identifying weaknesses in the mathematical properties of an algorithm. Even when a cryptographic algorithm is secure against methods of cryptanalysis, unintentional leakage of sensitive material is possible through side channels. Side channel analysis (SCA) takes advantage of the physical characteristics of an algorithm during run time. For example, an algorithm's power consumption, timing characteristics or electromagnetic transmissions can be used to extract pertinent information [32].

2.4.1 Attacks

There are several SCA attacks which use power consumption data as the basis for their technique. Simple Power Analysis (SPA) and Differential Power Analysis (DPA) were first introduced by Kocher et al. in 1999 [25]. The SPA technique collects the power consumption data of an algorithm during runtime. It then uses visual patterns present in the traces to form conclusions about the cryptographic operations taking place. This method benefits from conditional statements yielding noticeable differences in power consumption [25]. DPA identifies correlations between power consumption data subsets. The technique uses a "selection function" to separate power data into different subsets and then finds correlations by identifying significant differences between the subset averages [26]. DPA can be used to recover the key used. Another example is Correlation Power Analysis (CPA) [9]. CPA can be used to predict the value of some intermediate state of an algorithm. To do this, it uses a Hamming weight or distance model to determine the leakage and then calculates correlation using different values for the intermediate state. A correct value for the intermediate state will return higher correlation. This same idea can be applied to electromagnetic transmissions [2]. In [2], authors apply Correlation ElectroMagnetic Analysis (CEMA) to the ChaCha cipher. They perform side channel analysis on software based implementations of ChaCha and focus on the leakage associated with memory access.

2.4.2 Countermeasures

Several methods have been proposed to combat side channel leakage in cryptographic algorithms. At the core of some of these methods is the concept of secret sharing. Secret sharing provides additional security by splitting an algorithm's sensitive data into multiple shares. In order to recreate the sensitive data, a threshold number of shares is required.

Any collection of shares less than the number of threshold shares cannot be used to discern the value of the key [8]. One of the first proposed methods for preventing DPA was suggested by Goubin and Patarin [21]. The authors demonstrated a protection technique on the DES algorithm. Rather than performing cryptographic operations on intermediate values directly, the operations were applied to shares of the intermediate values. Shares can be created in a couple of different ways. Boolean masked shares must be XORed together to recreate the original value. Boolean masking can easily be applied to linear operations. Arithmetic masked shares must be added together to recreate the original value [11]. These different forms of masking are important to consider for the non-linear addition operation of an ARX-based cipher. Conversion techniques between Boolean and Arithmetic masks have been explored in other works [11, 12].

The Threshold Implementation (TI) method [35] for guarding against side channel attacks employs the concept of secret sharing. TI is well suited for hardware based implementations because it considers glitches. When a glitch occurs in hardware, the power consumption associated with the glitch is comparatively large. As a glitch propagates through gates, this can be exploited to deduce internal values. In order to provide security, TI relies upon the three properties of non-completeness, correctness, and uniformity [35]. Non-completeness ensures that at least one share is omitted from each function calculation. Correctness is achieved when the summation of the output shares provides an accurate result. Uniformity guarantees that the input probability distribution of a function matches its output probability distribution. Finally, it is important to note that the minimum number of shares needed to uphold these properties for a given function is one more than the degree of the function [35].

Chapter 3

Methods

3.1 Baseline Implementations

The following section will detail the steps taken to create LWC API compliant baseline implementations of Schwaemm and Esch. These implementations were built using the LWC API Development Package for Hardware Implementations [43], which was described in Subsection 2.2.2 and depicted in Figure 2.1. As recommended in the Hardware API for Lightweight Cryptography documentation [24], a standalone implementation was created for the AEAD scheme and a combined implementation was created to provide both AEAD and hashing functionality. The Schwaemm256-128 and Esch256 implementations both used the same Sparkle384 implementation. All code was developed in VHDL using Register Transfer Level (RTL) design. The Xilinx Vivado 2019.1 design suite was used for the development and functional verification of these implementations.

3.1.1 The Sparkle Permutation

To process the 384-bit state, this Sparkle implementation uses six separate instances of one round of the ARX-box *Alzette* which run in parallel. Each ARX round processes one 64-bit branch which encompasses two consecutive words of the state. The leftmost word is referred to as the "x" word input and the rightmost word is referred to as the "y" word input. One

clock cycle is allocated per round of the ARX-box *Alzette*, resulting in four clock cycles being used for each step of the permutation. One round of the ARX box is depicted in Figure 3.1 below. As discussed above, the rotation amounts "x_rot" and "y_rot" change with each ARX round. A different round constant is used for each branch of the state. The constants used per branch are fixed throughout the four rounds of *Alzette*, and update with each step of the permutation [4].

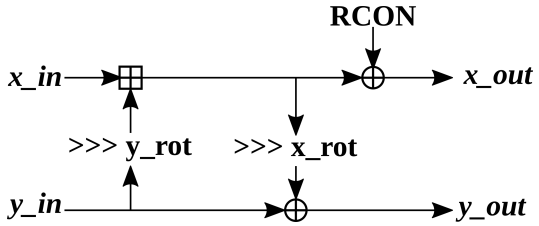

Figure 3.1: One round of the ARX-box *Alzette*. All bus widths are 32 bits. [2]

After all four rounds of *Alzette* complete, the linear layer is applied. As mentioned in Section 2.3, the linear layer employs a Feistel structure using the function "M_3" as shown in Figure 3.2. The rightmost branches of the state are rotated to the left by one branch before being swapped with the leftmost branches.

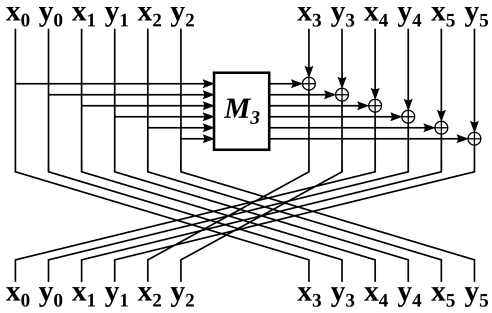

Figure 3.2: The Linear Layer. All bus widths are 32 bits. [2]

The general structure of this Sparkle384 implementation can be seen in Figure 3.3. Each of the ARX box instances shown in the figure represent one round of *Alzette*. Prior to the start of each step, the "y" input words of the 2 leftmost branches are XORed with a round constant

[2]Figures 3.1 and 3.2 are close adaptations of figures from [4].

and the step counter respectively. In this implementation, these updates are applied after the registering of the state to allow for the correct step index to be used. The SPARKLE384 permutation takes either 7 or 11 steps to complete depending upon the stage of the AEAD or hashing process. For example, 11 steps (steps_big) are used during hashing for separation at the end of absorption, while all other calls to the permutation use 7 steps (steps_slim) [4]. This implementation takes 4×11 = 44 clock cycles for an 11 step permutation call, and 4×7 = 28 clock cycles for a 7 step permutation call. An additional clock cycle at the start and the end of the permutation calls is used for registering the state.

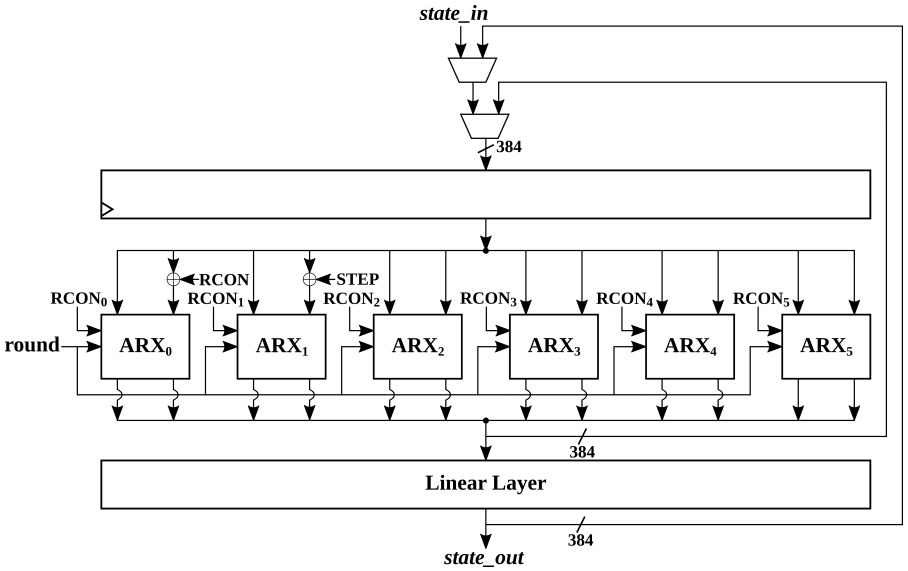

Figure 3.3: SPARKLE384 Implementation. All bus widths are 384 bits except those which are inputs and outputs to the ARX rounds which are 32 bits.

3.1.2 SCHWAEMM

Figure 3.4 depicts an LWC API compliant implementation of SCHWAEMM256-128. Components of SCHWAEMM which can be seen in the datapath include the combined feedback function ρ, the rate whitening shown as W_3, and the constant injection. Other signals to note in Figure 3.4 include `bdi`, `key`, `bdo` and `msg_auth`. These signals all pertain to the LWC

HW DP modules, as documented in [44]. Test vector data including the nonce, plaintext, ciphertext and tag arrive on the `bdo` bus from the PreProcessor. Key data arrives on the `key` bus. All output is transferred to the PostProcessor using the `bdo` data bus. The `msg_auth` signal indicates whether a calculated tag matches the input tag during a decryption process. Other control signals not pictured in Figure 3.4 exist to aid in loading and processing of the data. These signals and the general control architecture of the implementation can be better understood by examining the algorithmic state machine in Appendix Figure A.1.

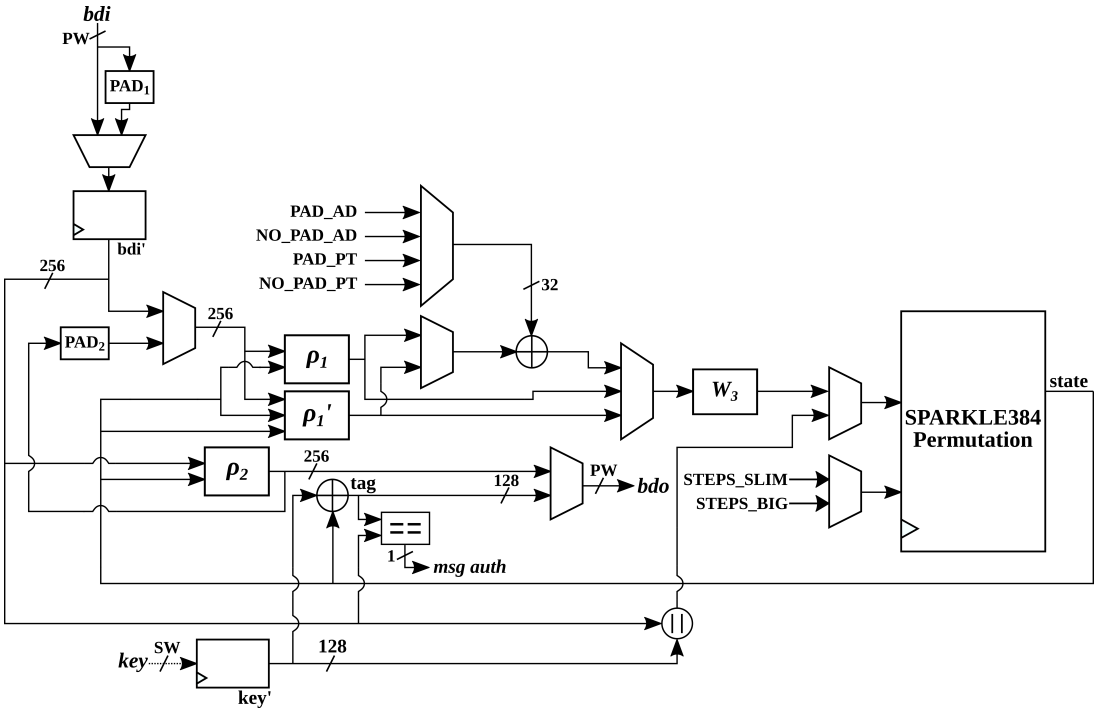

Figure 3.4: Schwaemm256-128 Implementation, compliant with the LWC API. All bus widths are 384 bits unless otherwise indicated.

3.1.3 Schwaemm and Esch

Figure 3.5 depicts an LWC API compliant implementation of Schwaemm256-128 and Esch256. The datapath largely reflects the one used for the Schwaemm256-128 only implementation, with a few additions. When the input data on the `bdo` data bus is a message input for

hashing, the data will be routed through an instance of the same Feistel Function, "M_3", used by the linear layer of SPARKLE384. Different constants are injected to indicate if the last block of the hash is padded or not padded. Additionally, to achieve the squeezing portion of the hash functionality, after the last block of data has been processed, the leftmost 128 bits of the state will be registered and the unmodified output of the state will be fed back into the state input. A seven step permutation is run, and the leftmost 128 bits of the output state will be concatenated to the previously stored 128 bit state output to form the 256 bit message digest.

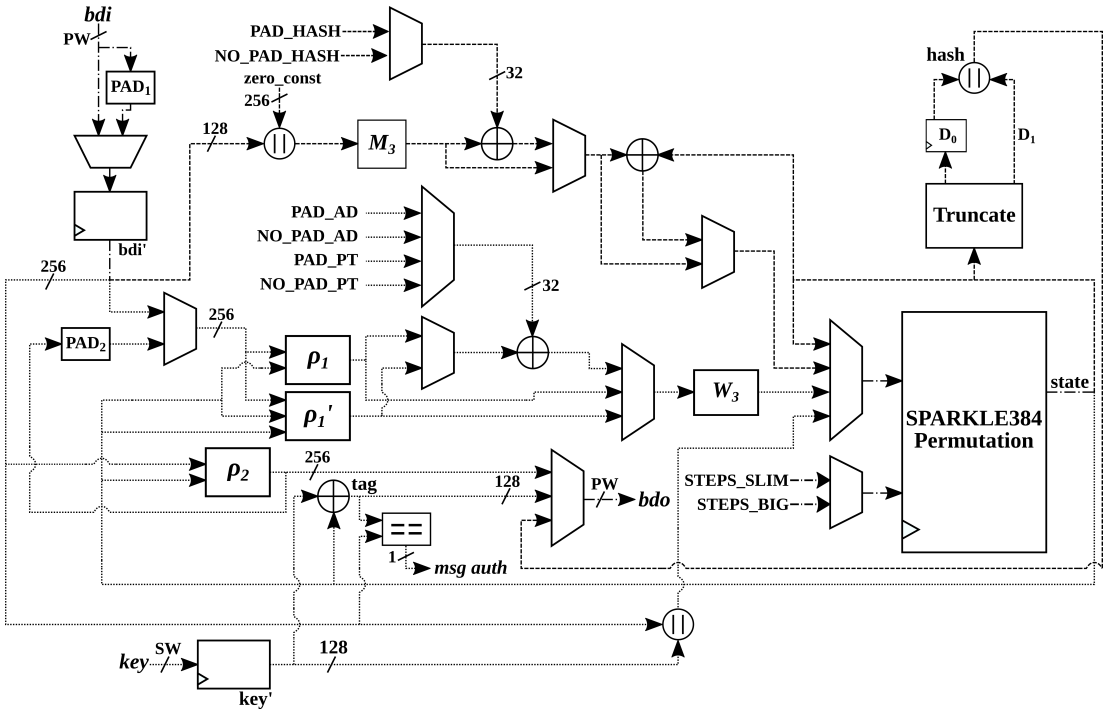

Figure 3.5: SCHWAEMM256-128 and ESCH256 Implementation, compliant with the LWC API. All bus widths are 384 bits unless otherwise indicated. The dashed lines indicate buses used for hash operations, the dotted lines indicate buses used for AEAD operations, and the dash-dot lines indicate the buses used in both.

3.1.4 Functional Verification

To verify the correct functionality of each of the baseline implementations above, multiple sets of test vectors were generated using the Python `cryptotvgen` module and used as input to the LWC TB [43]. As shown in Figure 2.1, each run of `cryptotvgen` produces a PDI, SDI and DO test vector file. Different modes of `cryptotvgen` were used, creating routine AEAD only test vectors, hash only test vectors, combined AEAD and hash test vectors, and finally randomly generated AEAD test vectors. Passing these test vectors in the LWC TB provided assurance that the implementations were behaving correctly.

3.2 Side-Channel Resistant Schwaemm Implementation

In [41], Schneider et al. propose a method for carrying out arithmetic operations over Boolean masked shares. They suggest the use of either a Ripple Carry Adder (RCA) or a Kogge Stone Adder (KSA) [27] and their approach aligns with the principles of TI. The KSA option completes the addition in fewer clock cycles therefore it is the method adopted here. An advantage of the use of this 3-TI KSA method is that it eliminates the need to convert between Boolean and Arithmetic masks, which was discussed in Subsection 2.4.2.

To transition towards side channel resistance, first an implementation with a 32-bit registered KSA was created. This implementation is used as a point of comparison in the results as it uses the same number of clock cycles for the ARX primitive as the final protected version. Figure 3.6 depicts the 32-bit registered KSA and shows its six levels of registers.

Once the ARX primitive was operational using the KSA, the 3-TI KSA scheme explored in [41] was applied. All sensitive data used was partitioned into three Boolean masked shares. The basic steps for the 3-TI KSA scheme include replacing all AND gates within the Kogge

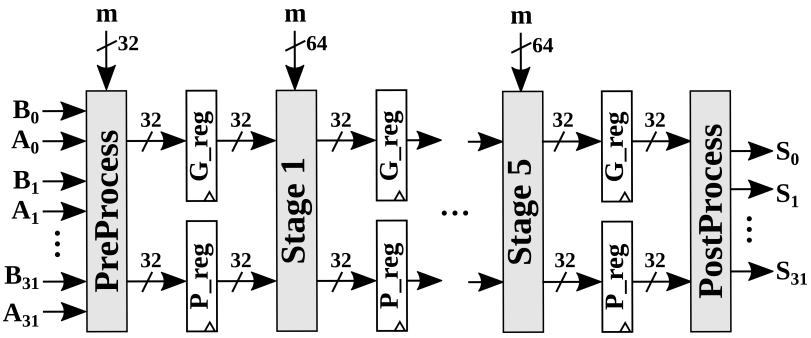

Figure 3.6: 32-bit Registered Kogge Stone Adder. Bus widths are 1 bit unless otherwise indicated. The "m" input alludes to the randomness needed for a protected KSA. [3]

Stone Adder with 3-TI AND gates and registering the output at each stage of the adder. Each TI-AND gate takes in two 32-bit values and a 32-bit random value. The random data input used for these implementations is 64 bits in length so that a unique 32 bit random value can be applied to each TI-AND gate at each stage of the KSA. This method of refreshing the randomness is used to uphold the uniformity property of the TI method. The value "m" in Figure 3.6 indicates which stages of the KSA would use the random data input in the protected 3-TI implementation. At the conclusion of each stage of the adder, the output is registered before starting the next stage. This prevents the propagation of glitches through the design [41]. What is important to note about these changes to the Kogge Stone Adder is that they require 6 additional clock cycles in order to register the output values at each stage. After the conclusion of the addition operation, the remaining rotate and XOR transformations are applied and then registered prior to the start of the next round. This impacts the throughput of the implementations. The number of clock cycles used is now $4\times11\times7 = 308$ clock cycles for an 11 step permutation call and $4\times7\times7 = 196$ clock cycles for a 7 step permutation call.

To handle the protected implementation of SCHWAEMM, an extension to the LWC Hardware Development Package, available at [29], was used. The extension to the LWC HW DP

[3] Figure 3.6 is an adaptation of a figure from [41].

uses an external pseudo random number generator based on the Trivium stream cipher [14]. The LWC top module uses an additional input bus, `rdi`, to take in the random data. The random data refreshes with each clock cycle, which upholds the principles of the TI method. Additionally, the extension to the LWC HW DP ensures that the shares are not XORed together anywhere in the LWC module.

3.2.1 Functional Verification

To verify the correct functionality of the protected implementation, modifications of the test vectors were required. To handle this a new Python script, `genShared.py`, was developed to produce a modified set of text vectors that were compatible with the extension to the LWC HW DP. To generate these test vectors, first one must run the original `cryptotvgen` Python module on a reference implementation of a NIST LWC candidate. This yields two input data sets (PDI and SDI) and one output data set (DO) [44]. The `genShared.py` parses the PDI and SDI input test vectors to create n-share Boolean masked versions of the input data streams. This results in the test vector set {sharedPDI.txt, sharedSDI.txt, do.txt}. While the number of shares used in this work was three, this parameterized script supports the creation of anywhere from two to four shares. The format of the files created used the same headers as the original test vectors to minimize the changes needed to the original LWC test bench parser. A screenshot of the help menu for genShared.py can be seen in Figure 3.7.

```
usage: genShared.py [-h] -iow {8,16,32} [-dest DEST] [-path PATH] -num {2,3,4}
                    [-pdi PDI] [-sdi SDI] [-fixed FIXED]

Script parses existing cryptotvgen test vectors and creates shared versions.

optional arguments:
  -h, --help      show this help message and exit
  -iow {8,16,32}  I/O Width. (default: None)
  -dest DEST      Name of destination folder for shared tv files. (default: .)
  -path PATH      Path to existing test vectors. (default: .)
  -num {2,3,4}    Number of shares to create. (default: None)
  -pdi PDI        Name of input PDI file. (default: pdi.txt)
  -sdi SDI        Name of input SDI file. (default: sdi.txt)
  -fixed FIXED    Method of creating shared test vectors: Select True for
                  fixed mask values across all PDI TVs and fixed mask values
                  across all SDI TVs. Defaults to False, with every TV having
                  a new random mask. (default: False)
```

Figure 3.7: genShared.py. This figure depicts a screenshot of the help menu for the Python script.

Chapter 4

Results

A few different techniques were used to evaluate the implementations presented in this work. In Section 4.1, synthesis results will be presented. Metrics including maximum frequency, throughput, area, throughput to area ratio, and power consumption data are provided. The iterative versions of Schwaemm are referred to as unprotected (UnPr), unprotected using the Kogge Stone Adder (UnPr KSA) and protected (Pr). In Section 4.2, results collected on actual hardware will be presented, detailing statistical tests performed on the unprotected and protected Schwaemm implementations.

4.1 Implementation Results

To establish an understanding of how the performance results were calculated, Table 4.1 provides the total clock cycles used per operation for each of the implementations. These formulas capture the clock cycles used from the start of the respective operation to its completion. This includes clock cycles for interacting with the Pre and Post Processor modules as well as the cycles needed for state initialization and other permutation calls. For a further breakdown of the clock cycles used, please see Appendix Section A.1. Table 4.1 also provides the formula used for throughput calculations. To calculate throughput, an assumption is made that a large number of plaintext, ciphertext, or hash input blocks are being processed. With a large number of input blocks, the constant number of clock

cycles needed for processes like state initialization become negligible in these calculations. As a result, throughput is calculated as the maximum achievable frequency multiplied by the number of bits per block divided by the cycles needed for one block of input.

Table 4.1: Clock Cycles. An overview of the clock cycles used in each implementation. Na denotes the number of blocks of associated data. Nm denotes the number of blocks of plaintext or ciphertext. Nh denotes the number of message blocks for hash input.

Implementation	Operation	Clock Cycles	Bits/Block	TP Formula
SCHWAEMM UNPR	Encryption	$38\times Na + 47\times Nm + 97$	256	$f_{clk}\times$ 256/47
	Decryption	$38\times Na + 47\times Nm + 98$		
SCHWAEMM UNPR KSA	Encryption	$206\times Na + 215\times Nm + 553$		$f_{clk}\times$ 256/215
	Decryption	$206\times Na + 215\times Nm + 554$		
SCHWAEMM PR	Encryption	$227\times Na + 257\times Nm + 598$		$f_{clk}\times$ 256/257
	Decryption	$227\times Na + 257\times Nm + 605$		
SCHWAEMM & ESCH	Encryption	$38\times Na + 47\times Nm + 97$		$f_{clk}\times$ 256/47
	Decryption	$38\times Na + 47\times Nm + 98$		
	Hashing	$34\times Nh + 53$	128	$f_{clk}\times$ 128/34

Table 4.2 provides several cost and performance metrics for each implementation. To develop a better understanding of the resources used by each implementation, this work uses the Minerva automated tool for hardware optimization [18]. Minerva applies static timing analysis to post-synthesis results to achieve optimal throughput to area (TPA) ratio. All results listed with the Artix-7 (xc7a100tcsg324-3) FPGA were collected using Minerva. Additionally, synthesis results were collected for the Cyclone V DE1-SoC board (5CSEMA5F31C6) using Quartus Prime Lite 19.1. Area and throughput to area results in Table 4.2 use look up tables (LUTs) and adaptive logic modules (ALMs) for the Artix-7 and the Cyclone-5 respectively. TPA is not calculated for **SCHWAEMM & ESCH** as the combined implementations area should not logically be applied to different hash or AEAD only throughput formulas.

Additionally, power consumption data was collected for the Artix-7 (xc7a100tftg256-3) using the Xilinx Vivado power reporting feature. The top level clocks were constrained with clock periods which yielded frequencies of 10MHz, 25MHz, and 50MHz. The power and energy data for each implementation running at 50MHz is shown in Table 4.3 with the exception

Table 4.2: Implementation Results. For area and throughput to area results, LUTs is the unit for the Artix-7 and ALMs is the unit for the Cyclone-5

Implementation	FPGA	Freq. (MHz)	Reg. (FFs)	Area (LUTs) (ALMs)	Area Ratio	TP (Mbps)	TP Ratio	TPA (Mbps/LUT) (Mbps/ALM)	TPA Ratio
Schwaemm UnPr	Artix-7	141.00	1396	3107	0.62	768.00	3.96	0.2472	6.43
	Cyclone V	104.48	1431	2276	0.73	569.08	3.83	0.2500	5.28
Schwaemm UnPr KSA	Artix-7	163.00	3295	5045	1.00	194.08	1.00	0.0385	1.00
	Cyclone V	124.63	3298	3131	1.00	148.40	1.00	0.0474	1.00
Schwaemm Pr	Artix-7	142.00	10480	14573	2.89	141.45	0.73	0.0097	0.25
	Cyclone V	106.87	10518	10327	3.30	106.45	0.72	0.0103	0.22
Schwaemm & Esch	Artix-7	140.00	1032	3757	1.00	527.06	1.00	N/A	N/A
	Cyclone V	100.24	1566	2855	1.00	377.37	1.00	N/A	N/A

of the gradient, which is computed as the slope of the average power consumption over data collected at 10MHz, 25MHz, and 50MHz. The energy is computed as the average power at 50MHz divided by the throughput at 50MHz.

Table 4.3: Power and Energy Results, on the Artix-7

Implementation	Avg. Power (mW)	Gradient (mW/MHz)	Energy (nJ/Bit)
Schwaemm UnPr	32.00	0.6265	0.118
Schwaemm UnPr KSA	30.00	0.6000	0.504
Schwaemm Pr	119.00	2.373	2.398
Schwaemm & Esch	40.00	0.8000	0.213

4.2 Hardware Testing

Along with synthesis results, select implementations were also verified on an Artix-7 FPGA (xc7a100tftg256-3). To determine whether the protected implementation Schwaemm256-128 would resist side channel attacks, a series of tests were run using the Test Vector Leakage Assessment (TVLA) method [3]. As discussed in Subsection 2.4.1, DPA exploits significant differences in subsets of data. The TVLA method uses statistical analysis (t-tests) to determine whether significant differences occur "with high confidence" [20]. A t-value of less

than -4.5 or greater than 4.5 constitutes a failure of the test and indicates that leakage has occurred. The Flexible Open-source workBench fOr Side-channel analysis (FOBOS) [1] environment has integrated support for TVLA. Using FOBOS, four t-tests were executed on the implementations of SCHWAEMM256-128. Tests were run at 1MHz and 10MHz on both the unprotected, baseline implementation and the protected implementation. The results can be seen in Figure 4.1, with the unprotected results labeled with **A** and the protected results labeled with **B**.

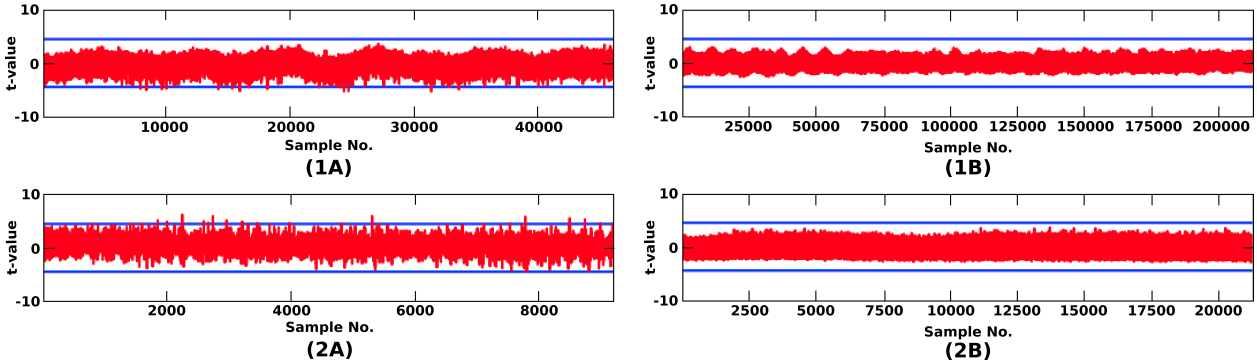

Figure 4.1: Collection of SCHWAEMM t-tests. Charts labeled A denote the results of the unprotected implementation while charts labeled B denote the results of the protected implementation. Tests labeled 1 and 2 were run at 1MHz and 10MHz respectively. [4]

4.3 Discussion

The results collected display some interesting properties of these implementations. One key takeaway from the hardware testing is that the protected implementation of SCHWAEMM256-128 passes the t-tests at both 1MHz and 10MHz frequencies. As shown in Figure 4.1, the red traces of the **B** tests stay within the blue t-value bounds. This suggests the successful application of the side channel resistant methodology discussed in Section 2.4. The unprotected tests show several instances of leakage in tests **1A** and **2A**. This indicates that the

[4]Please note that the protected implementation run in all t-tests labeled B was implemented within the CAESAR Hardware API [23]

changes made did have a significant impact on the physical runtime characteristics of these implementations.

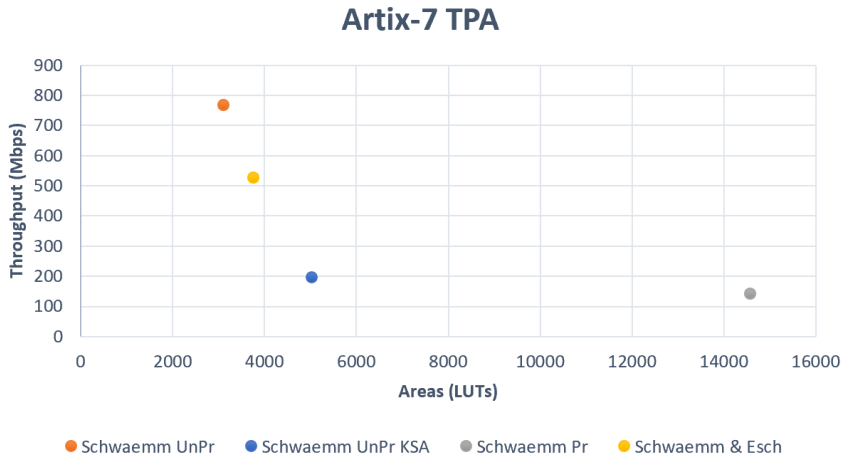

Figure 4.2: Artix-7 TPA Results. Throughput to area results on the Artix-7 FPGA.

While the protected implementation passes the leakage detection test, it is important to consider the cost of achieving this additional security. Table 4.2 provides cost and performance metrics of each implementation and Figure 4.2 provides a visual representation of the Artix-7 TPA results. The protected Schwaemm implementation consumes the largest area of any of the implementations, using 14573 look up tables on the Artix-7. This is 2.89× greater than the area of the unprotected Schwaemm implementation which uses the Kogge Stone Adder. When compared to the baseline implementation, the Schwaemm Pr implementation consumes 4.69× the area. These results closely reflect expected increases in resources. For protection, resources are expected to increase quadratically depending on the order of protection required. For example, protection against first order DPA, $(d = 1)$, requires approximately $(d + 1)^2 = 4$ times the amount of resources required for non-linear operations [15]. For protecting linear operations, the increase in resources is expected to scale linearly, suggesting a 3× increase for three shares. Therefore, resource utilization can be expected to increase anywhere between 3× and 4×. The scaling factors of 2.89× and

4.69× are representative of this range. Additionally, the extension to the LWC HW DP accounts for some additional resource utilization in the protected implementation. This may contribute to the scaling factor falling slightly above the expected range. Another aspect to consider is that the throughput achieved is only 72.88% and 18.42% of the throughput of the **SCHWAEMM UNPR KSA** and **SCHWAEMM UNPR** implementations respectively.

The **SCHWAEMM & ESCH** implementation incurs a 1.21× increase in area when compared to the **SCHWAEMM** only implementation. Also, its throughput is only 68.63% of the previous throughput. The increase in area is due to some additional hardware used to support the hashing mode. The decrease in throughput is affected by **ESCH** only processing 128 bit blocks as opposed to 256 bit blocks.

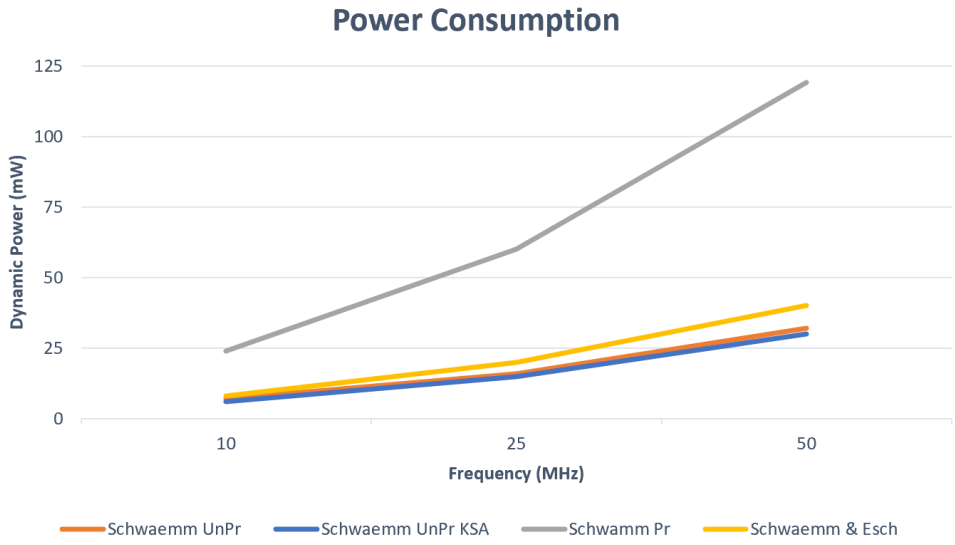

Figure 4.3: Artix-7 Power Results. A graphical representation of the power consumption data on the Artix-7 FPGA.

Another interesting trend which appears in Table 4.3 is the use of less power in the **SCHWAEMM UNPR KSA** implementation when compared to the **SCHWAEMM UNPR** implementation, even though it consumes larger area. This trend can be seen in Figure 4.3. Not only is less power used at 50MHz, the **SCHWAEMM UNPR KSA** gradient of 0.6000 mW/MHz is smaller

than the Schwaemm UnPr gradient, which shows that the power usage for Schwaemm UnPr KSA grows at a smaller rate. This can be attributed to the change in the addition operation of the ARX boxes. The Schwaemm UnPr KSA implementation distributes the addition operation over six clock cycles as opposed to completing it in one, which effectively reduces the power consumption associated with the ARX boxes. However, the energy per bit used by the Schwaemm UnPr KSA implementation is 4.27× larger than the energy used by the Schwaemm UnPr implementation. This is due to the reduction in the throughput achieved by the Schwaemm UnPr KSA implementation.

Another important takeaway from the power consumption data is the significantly larger gradient of 2.373 mW/MHz associated with Schwaemm Pr. This is a 3.96× increase from the gradient of the Schwaemm UnPr KSA implementation. As discussed above, the gradient increase aligns with the quadratic increase in the resource utilization associated with protection.

Chapter 5

Related Work

Table 5.1: Data from Related Works

Cipher	Implementation	FPGA	Freq. (MHz)	Area (LUTs) (GEs)	TP (Mbps)	TPA (Mbps/LUT) (Mbps/GE)	Ref.
Ascon	Unprotected	-	-	7950	5524	0.694	[22]
	Protected		-	30420	3774	0.124	
PRINCE	Unprotected	-	393	3589	-	-	[7]
	Protected		376	11596	-	-	
Ascon	Unprotected	Spartan-6	195.5	2048	255.4	0.125	[15]
	Protected		103.1	6364	134.6	0.021	
Acorn	Unprotected	Spartan-6	226.6	549	906.2	1.651	
	Protected		142.7	2732	570.6	0.209	
JAMBU-AES	Unprotected	Spartan-6	163.1	1073	50.9	0.048	
	Protected		122.4	2869	38.2	0.013	
JAMBU-SIMON	Unprotected	Spartan-6	137.9	1105	509.3	0.461	
	Protected		58.7	3140	216.7	0.069	
SPARX	Unprotected	Spartan-3	173	114	10.81	0.095	[38]
	Protected		144	231	9	0.039	

Previous investigations into hardware implementations of lightweight cryptographic algorithms have employed the TI method for providing side channel resistance. A selection of these works and their results is compiled in Table 5.1. Ciphers which do not have an FPGA listed are ASIC based implementations. In [22], the authors produced a protected version of Ascon [17], another NIST LWC Round 2 Candidate, using TI. Similarly, this study compared the trade offs of protected versus unprotected implementations. A threshold implementation of PRINCE was presented in [7]. This specific implementation focuses on optimizing TI to improve upon the latency of the results. A threshold implementation of SPARX was presented in [38]. This is of interest as the Sparkle permutation is based off of SPARX [4].

A collection of AEAD algorithms with both unprotected and TI protected implementations were analyzed in [15]. These implementations used the CAESAR Hardware API [23], which is a similar interface to the LWC API. Due to the similar interface which these implementations use, it is reasonable to directly compare the results of this work with those found by the authors of [15]. A visual representation of the throughput versus area data, and the impact of applying a SCA protection scheme to these works can be seen in Figure 5.1.

Figure 5.1: Impact of Protection Schemes. The throughput versus area affect of applying side channel resistance protection schemes. All data not generated by this work was from [15] in which authors used the Spartan-6 FPGA to collect their results.

When examining how the Schwaemm implementations perform in comparison to all other implementations shown in Figure 5.1, there are several key trends. Regardless of the algorithm, there is the clear reduction in throughput and increase in area that can be expected to achieve SCA resistance. Of the unprotected implementations, the 768 Mbps throughput on the Artix-7 which the Schwaemm implementation offers is very competitive. It exceeds all other unprotected throughput with the exception of ACORN. It also offers a TPA ratio

that is 5.15× greater than unprotected JAMBU-AES, which had the smallest reported TPA ratio for an unprotected implementation in Table 5.1. When considering area, both the unprotected and protected implementations of SCHWAEMM consume the largest number of look up tables. Different architectures could be used to further reduce the area of an unprotected and protected SCHWAEMM implementation. This could potentially change the impact of the application of the protection scheme to be more similar to the trends shown by the other implementations, but an investigation into this was outside the scope of this work.

When comparing this effort to the other works in Table 5.1, a direct comparison is less appropriate. The SPARX implementation discussed in [38] was a block cipher implementation of SPARX-64/128 [16]. The state size of SPARX-64/128 is 64 bits as opposed to 384 bits and the implementation was not created using either of the Hardware APIs discussed in this work. Due to the disparity in these works, a direct comparison is not considered here. Additionally, the ASIC based implementations are valuable points of reference for the impact of TI protection schemes, but are not directly comparable.

Overall, the data discussed in this section is important for framing the performance of the SCHWAEMM implementations presented. This understanding aids in the overall evaluation of the candidate.

Chapter 6

Conclusions

6.1 Summary

Lightweight cryptography is a promising field for the security of the Internet of Things. This work presented hardware implementations of an ARX-based NIST Lightweight Cryptography candidate. A baseline implementation of both Schwaemm, and Schwaemm & Esch was provided. An implementation of Schwaemm which integrates a registered Kogge Stone Adder is detailed, which aids in the analysis presented in Section 4.3. Finally, a side-channel resistant implementation of Schwaemm was evaluated. All implementations were compatible with the LWC API to enable impartial comparisons of the work.

The side-channel resistant implementation was shown to be resistant to leakage when applying the TVLA method. However, this resistance came at an incurred cost. When contrasting the protected implementation of Schwaemm to the unprotected implementation of Schwaemm which integrates a KSA, the area of implementation was 2.89× greater and the throughput to area ratio reflects a 27% decrease. These results show that it is feasible to create side channel resistant implementations of Schwaemm and suggest how the implementation metrics of other members of the AEAD family may scale when applying SCA resistant methods in hardware.

Additionally, a comparison to other lightweight ciphers was included in Chapter 5. The

SCHWAEMM and ESCH implementations displayed competitive throughput and throughput to area ratio results against the subgroup of ciphers included. However, the area of both the unprotected and protected implementations was larger than most other areas reported. The discussion in Chapter 5 enables an understanding of the relative characteristics of this work.

As discussed in Subsection 2.3.2, ARX cryptography has been studied in software. The results presented in this work add to the data available about ARX-based ciphers when implemented in hardware, especially when considering side channel resistant implementations.

6.2 Future Work

There are certain aspects of this work which provide gateways for future research. One main focus of future efforts would be further optimizations of the implementations. When reviewing the baseline implementations, there are features which can still be simplified to consume less area while maintaining the structure of the algorithms. Additionally, there are different types of architectures which can be explored for different use cases. For example, further serialized implementations could reduce the hardware area. Regardless of the architecture used, there will be trade offs between different metrics. The decision in this work to process all 384 bits of the state per clock cycle of the permutation results in a higher area cost but a more favorable throughput. When transitioning to a protected implementation, starting with a lower area implementation may help to scale down the relative cost of protection. There are many different approaches to this process and a large number of opportunities for continuing research.

Another avenue which could be explored is in software. In the specification for SPARKLE (SCHWAEMM and ESCH) [4], the authors focus a larger portion of their design discussion on a software perspective. Other works which consider ARX-based primitives also devote a great

deal of attention to how these algorithms are realized in software. To further expand upon the understanding of SCA resistance for ARX primitives, it would be interesting to evaluate software and hardware implementations together. There are several schemes for side channel resistance in software which would support this pursuit.

Though there are many possibilities for future work, this project remained focused on a small subset of hardware evaluation metrics. This facilitated an understanding of the material and a well defined research scope.

Bibliography

[1] Abubakr Abdulgadir, William Diehl, and Jens-Peter Kaps. *Flexible, Opensource workBench fOr Side-channel analysis (FOBOS), User Guide*. Cryptographic Engineering Research Group, v2.0 edition, Dec 2019.

[2] Alexandre Adomnicai, Jacques J. A. Fournier, and Laurent Masson. Bricklayer Attack: A Side-Channel Analysis on the ChaCha Quarter Round. In Arpita Patra and Nigel P. Smart, editors, *Progress in Cryptology – INDOCRYPT 2017*, pages 65–84, Cham, 2017. Springer International Publishing. ISBN 978-3-319-71667-1.

[3] G Becker, J Cooper, E DeMulder, G Goodwill, J Jaffe, G Kenworthy, T Kouzminov, A Leiserson, P Rohatgi, and S Saab. Test Vector Leakage Assessment (TVLA) methodology in practice. volume 1001, pages 1–13, Gaithersburg, MD, November 2013.

[4] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Leo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing using the Sparkle Permutation Family, September 2019. URL https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates.

[5] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, pages 1–6, 2008. URL https://cr.yp.to/chacha/chacha-20080120.pdf.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, January 2011. URL https://keccak.team/files/CSF-0.1.pdf.

[7] Dušan Božilov, Miroslav Knežević, and Ventzislav Nikov. Optimized Threshold Imple-

mentations: Minimizing the Latency of Secure Cryptographic Accelerators. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications*, pages 20–39, Cham, March 2020. Springer International Publishing. ISBN 978-3-030-42068-0.

[8] Luís T.A.N. Brandão, Nicky Mouha, and Apostol Vassilev. Threshold Schemes for Cryptographic Primitives: Challenges and Opportunities in Standardization and Validation of Threshold Cryptography. Technical Report NIST IR 8214, National Institute of Standards and Technology, Gaithersburg, MD, March 2019. URL https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8214.pdf.

[9] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-28632-5.

[10] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):1–24, 2018.

[11] Jean-Sébastien Coron. High-Order Conversion from Boolean to Arithmetic Masking. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 93–114, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66787-4.

[12] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In Gregor Leander, editor, *Fast Software Encryption*, pages 130–149, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48116-5.

[13] Fredrik Dahlqvist, Mark Patel, Alexander Rajko, and Jonathan Shulman. Growing opportunities in the Internet of Things, July 2019. URL https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things.

[14] Christophe De Cannière. Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security*, pages 171–186, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-38343-7. doi: 10.1007/11836810_13. Series Title: Lecture Notes in Computer Science.

[15] William Diehl, Abubakr Abdulgadir, Farnoud Farahmand, Jens-Peter Kaps, and Kris Gaj. Comparison of Cost of Protection Against Differential Power Analysis of Selected Authenticated Ciphers. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 147–152. IEEE, April 2018. ISBN 978-1-5386-4731-8. doi: 10.1109/HST.2018.8383904.

[16] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design Strategies for ARX with Provable Bounds: Sparx and LAX. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 484–513, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53887-6. doi: 10.1007/978-3-662-53887-6_18.

[17] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon. November 2019. URL https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates.

[18] Farnoud Farahmand, Ahmed Ferozpuri, William Diehl, and Kris Gaj. Minerva, December 2017. URL https://cryptography.gmu.edu/athena/index.php?id=Minerva.

[19] National Institute for Standards and Technology. Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process, August 2018. URL https://csrc.nist.gov/projects/lightweight-cryptography.

[20] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, pages 1–15. URL https://csrc.nist.gov/CSRC/media/Events/Non-Invasive-Attack-Testing-Workshop/documents/08_Goodwill.pdf.

[21] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis The "Duplication" Method. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, pages 158–172, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48059-4.

[22] Hannes Gross, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. Ascon hardware implementations and side-channel evaluation. *Microprocessors and Microsystems*, 52:470–479, July 2017. ISSN 01419331. doi: 10.1016/j.micpro.2016.10.006.

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

[24] Jens-Peter Kaps, William Diehl, Michael Tempelmeier, Ekawat Homsirikamol, and Kris Gaj. Hardware API for Lightweight Cryptography. pages 1–26. URL https://cryptography.gmu.edu/athena/index.php?id=LWC.

[25] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48405-9. doi: 10.1007/3-540-48405-1_25.

[26] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, April 2011. doi: 10.1007/s13389-011-0006-y.

[27] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8): 786–793, August 1973. doi: 10.1109/TC.1973.5009159.

[28] R Kousalya and G.A. Sathish Kumar. A Survey of Light-weight Cryptographic Algorithm for Information Security and Hardware Efficiency in Resource Constrained Devices. In *International Conference on Vision Towards Emerging Trends In Communication and Networking (ViTECoN 2019)*, pages 1–5, March 2019.

[29] VT Signatures Analysis Laboratory. comet_cham_lwc_protected. URL https://github.com/vtsal/comet_cham_lwc_protected.

[30] James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. The Internet of Things: Mapping the Value Beyond the Hype, June 2015.

[31] Kerry A McKay, Larry Bassham, Meltem Sonmez Turan, and Nicky Mouha. Report on Lightweight Cryptography. Technical Report NIST IR 8114, National Institute of Standards and Technology, Gaithersburg, MD, March 2017. URL https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf.

[32] Seyedeh Sharareh Mirzargar and Mirjana Stojilović. Physical Side-Channel Attacks and Covert Communication on FPGAs: A Survey. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 202–210, Barcelona, Spain, September 2019. IEEE. ISBN 978-1-72814-884-7. doi: 10.1109/FPL.2019.00039.

[33] Nicky Mouha. ARX-based Cryptography, June 2011. URL https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/nicky_mouha_arx-slides.pdf.

[34] Nicky Mouha and Bart Preneel. A Proof that the ARX Cipher Salsa20 is Secure against Differential Cryptanalysis. Cryptology ePrint Archive, Report 2013/328, 2013. https://eprint.iacr.org/2013/328.

[35] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, pages 529–545. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-49497-3. doi: 10.1007/11935308_38.

[36] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols, June 2018. URL https://tools.ietf.org/html/rfc8439.

[37] Johannes Obermaier and Martin Hutle. Analyzing the Security and Privacy of Cloud-based Video Surveillance Systems. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security - IoTPTS '16*, pages 22–28, Xi'an, China, 2016. ACM Press. ISBN 978-1-4503-4283-4. doi: 10.1145/2899007.2899008.

[38] Sumesh Manjunath Ramesh and Hoda AlKhzaimi. Side Channel Analysis of SPARX-64/128: Cryptanalysis and Countermeasures. In Johannes Buchmann, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2019*, pages 352–369. Springer International Publishing, Cham, June 2019. ISBN 978-3-030-23696-0. doi: 10.1007/978-3-030-23696-0_18.

[39] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, August 2018. URL https://tools.ietf.org/html/rfc8446.

[40] Phillip Rogaway. Authenticated-Encryption with Associated-Data. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 98–107, New York, NY, USA, November 2002. Association for Computing Machinery. ISBN 1581136129. doi: 10.1145/586110.586125.

[41] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic Addition over Boolean Masking. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security*, pages 559–578, Cham, 2015. Springer International Publishing. ISBN 978-3-319-28166-7.

[42] Byoungjin Seok and Changhoon Lee. Fast implementations of ARX-based lightweight block ciphers (SPARX, CHAM) on 32-bit processor. *International Journal of Distributed Sensor Networks*, 15(9):10, September 2019. doi: 10.1177/1550147719874180.

[43] Michael Tempelmeier, Farnoud Farahmand, Ekawat Homsirikamol, William Diehl, Jens-Peter Kaps, and Kris Gaj. Development Package for the Hardware API for Lightweight Cryptography, November 2019. URL https://github.com/GMUCERG/LWC.

[44] Michael Tempelmeier, Farnoud Farahmand, Ekawat Homsirikamol, William Diehl, Jens-Peter Kaps, and Kris Gaj. Implementer's Guide to Hardware Implementations Compliant with the Hardware API for Lightweight Cryptography, November 2019. URL https://cryptography.gmu.edu/athena/index.php?id=LWC.

[45] Meltem Sönmez Turan, Kerry A McKay, Çağdaş Çalık, Donghoon Chang, and Larry Bassham. Status Report on the First Round of the NIST Lightweight Cryptography Standardization Process. Technical Report NIST IR 8268, National Institute of Standards and Technology, Gaithersburg, MD, October 2019. URL https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8268.pdf.

Appendices

Appendix A

Supplemental Material

A.1 Formulas

```
Sparkle -- Schwaemm256128 : Clock Cycles in LWC API

Na       = # blocks associated data   ( |block| = 256 bits )
Nm       = # blocks pt/ct data        ( |block| = 256 bits )
Nh       = # blocks hash data         ( |block| = 128 bits )
LbWords  = # words in last data block

Encryption:

    Load key:                   4
    Wait npub:                  3
    Load npub:                  8
    Initialize state:           46
    Load ad:                    8*Na
    Process ad:                 30*(Na - 1) + 46
    Load dat:                   8*Nm
    Process dat:                30*(Nm - 1) + 46
    Output dat:                 9*Nm
    Output tag:                 4

    Total:                      38*Na + 47*Nm + 97
    No AD:                      47*Nm + 81
    No DAT:                     38*Na + 81
    No AD & No DAT:             65
    No new key:                 {All formulas above, subtract 7 clock cycles}

    **Note, for a data input in which the last block is not a full block
    (Length is less than 256 bits) the total number of clock cycles will
```

```
    decrease because fewer clock cycles are required for the output of
    the last block.

    Revised Total Formula:      38*Na + 47*Nm + 89 + LbWords
    No AD:                      47*Nm + 73 + LbWords

Decryption:

    Load key:                   4
    Wait npub:                  3
    Load npub:                  8
    Initialize state:           46
    Load ad:                    8*Na
    Process ad:                 30*(Na - 1) + 46
    Load dat:                   8*Nm
    Process dat:                30*(Nm - 1) + 46
    Output dat:                 9*Nm
    Load tag:                   4
    Output tag valid:           1

    Total:                      38*Na + 47*Nm + 98
    No AD:                      47*Nm + 82
    No DAT:                     38*Na + 82
    No AD & No DAT:             66
    No new key:                 {All formulas above, subtract 7 clock cycles}

    **Note, for a data input in which the last block is not a full block
    (Length is less than 256 bits) the total number of clock cycles will
    decrease because fewer clock cycles are required for the output of
    the last block.

    Revised Total:              38*Na + 47*Nm + 90 + LbWords
    Revised No AD:              47*Nm + 74 + LbWords
```

A.2 Figures

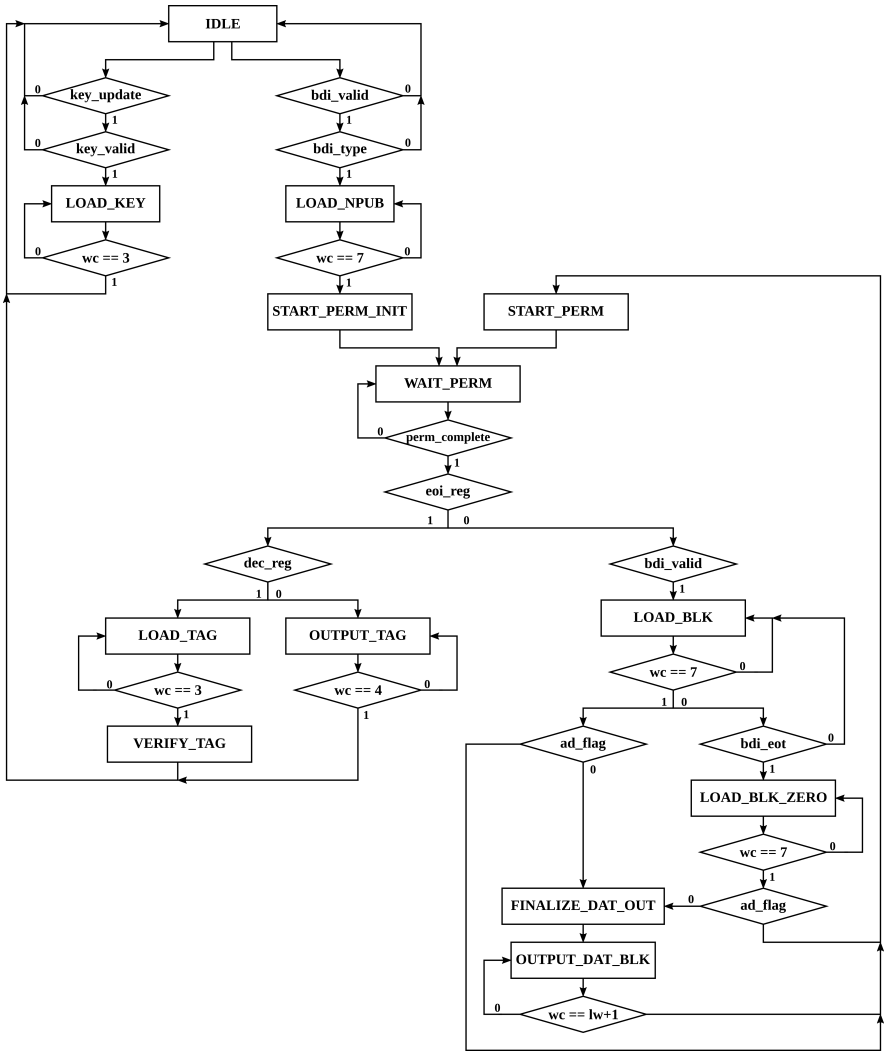

Figure A.1: Control Logic State Machine, SCHWAEMM256-128 Implementation. This is based off of the suggested control flow detailed in [44]. State names are displayed within the boxes. "wc" refers to the word counter used and "lw" refers to the last word. A similar state machine was used for the combined SCHWAEMM256-128 and ESCH256 Implementation, with additional states included to handle hashing.