IMPLEMENTATION, BENCHMARKING, AND PROTECTION OF LIGHTWEIGHT CRYPTOGRAPHY CANDIDATES

by

Richard Haeussler

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

_____	Dr. Kris Gaj, Thesis Director
_____	Dr. Jens-Peter Kaps, Committee Member
_____	Dr. Avesta Sasan, Committee Member
_____	Dr. Monson Hayes, Chairman, Department of Electrical and Computer Engineering
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date: _____	Spring Semester 2021 George Mason University Fairfax, VA

Implementation, Benchmarking, and Protection of Lightweight Cryptography Candidates

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at George Mason University

By

Richard Haeussler
Bachelor of Science
George Mason University, 2015

Director: Dr. Kris Gaj, Professor
Department of Electrical and Computer Engineering

Spring Semester 2021
George Mason University
Fairfax, VA

Acknowledgments

Worked covered in this thesis is primarily that of the author in collaboration as a member of CERG's LWC team. I want to thank the following members Kamyar Mohajerani, Rishub Nagpal, Abubakr Abdulgadir, Dr. Farnoud Farahmand, Dr. Jens-Peter Kaps, Dr. Kris Gaj, as well as the rest of CERG's LWC team for their help and guidance.

Table of Contents

List of Tables

List of Figures

Abstract

IMPLEMENTATION, BENCHMARKING, AND PROTECTION OF LIGHTWEIGHT CRYPTOGRAPHY CANDIDATES

Richard Haeussler

George Mason University, 2021

Thesis Director: Dr. Kris Gaj

In August 2019, the US National Institute of Standards and Technology (NIST) announced 32 candidates for Round 2 of their Lightweight Cryptography (LWC) standardization process. NIST needed to understand how each of the candidates performed in software and hardware before making their finalist selections. George Mason University's Cryptographic Engineering Research Group (CERG) assisted NIST by organizing the Field-Programmable Gate Array (FPGA) benchmarking of the Round 2 candidates. CERG developed LWC Hardware API compliant implementations for 14 of the Round 2 candidates. This work contains a detailed breakdown of the unprotected hardware implementations of Elephant and Xoodyak, along with figures and tables to illustrate the design choices that were made. It also highlights several new features that CERG added to the LWC Hardware API development package to assist in the FPGA benchmarking. An overview of CERG's benchmarking efforts, along with the results for Elephant and Xoodyak, are contained. From the results, analysis was conducted to determine possible design improvements. On March 29, 2021, NIST announced both Elephant and Xoodyak as LWC finalists. Before NIST announced finalists, Domain Oriented Masking was used to develop side-channel resistant implementations of both Elephant and Xoodyak. The efforts from this work certainly

provide NIST with valuable information for their LWC standardization process.

Chapter 1: Introduction

Every day there are increasingly more and more miniature devices communicating with one another in areas such as healthcare, automotive systems, and the Internet of Things (IoT). Sensitive information transmitted by these devices requires the use of authenticated encryption. Authenticated encryption provides a communication channel with message integrity, confidentiality, and authenticity of the sender. Small devices operate in resource-constrained environments requiring them to be as efficient as possible. The National Institute of Standards and Technology's (NIST) Report on Lightweight Cryptography (LWC) [1] states that most modern cryptographic algorithms were developed for use on desktops and servers. Existing cryptographic standards did not account for the considerations and limitations of running on resource-constrained devices. Therefore, NIST set out to establish new lightweight cryptographic algorithms with the LWC call in 2017.

NIST's LWC call aims to augment the current Authenticated Encryption with Associated Data (AEAD) standard, AES-GCM, with a lightweight algorithm. AEAD algorithms take in a key, nonce (NPUB), associated data (AD), and plaintext (PT). With these inputs, the AEAD produces the ciphertext (CT) and tag as outputs. The key is a shared secret between the sender and receiver. A nonce is a random number that should only be used once. Reuse of the same nonce value reduces the security provided by the algorithm. AD is sent in the clear and verified unmodified because of its contribution to the tag. The tag is used as a message authentication code to prove message integrity and the sender's authenticity. On the other hand, authentic decryption takes in the key, nonce, associated data, ciphertext, and the expected tag. If the calculated tag equals the input tag, then the message has successfully been decrypted, and the contents of the message are unmodified.

1.1 Motivation

Cryptographic standardization is a lengthy process. Typically the standardization is performed as a contest involving multiple algorithms, also known as candidates. These candidate algorithms must prove their security and provide a reference software implementation. From the algorithms and the reference software implementation, hardware designs are developed. Candidates are then be compared against one another to determine the trade-offs that each candidate provides. For example, one candidate may process data faster but requires more energy or is possibly less secure. The resource-constrained environment in which the LWC candidates will operate makes the decision between trade-offs even more critical.

Developing cryptographic candidates versus implementing them in hardware requires a completely different skill set. Since many design teams are small, they do not include members with hardware experience. These teams rely on the cryptographic community to create the hardware designs to allow for comparison. George Mason University's (GMU) Cryptographic Engineering Research Group (CERG) supports the cryptographic community by focusing on our area of expertise, hardware implementations. CERG has a long history of supporting the cryptographic standardization process from NIST's hashing competition SHA-3 [2], to the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [3, 4] and now LWC [5, 6]. The need for lightweight cryptography continues to grow, and the effects from this thesis have certainly aided NIST in its selection process.

Chapter 2: Previous Work

2.1 CAESAR

Before NIST's LWC call in 2012, the cryptographic community established the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR). CAESAR focused on developing cryptographic algorithms to be used in resource-constrained environments. The CAESAR competition consisted of three rounds. As the rounds progressed, the selection committee advanced the best candidates and eliminated inferior candidates. Due to the reduction in the number of candidates and the later rounds being longer, it was feasible to compare hardware implementation designs. CERG contributed to the CAESAR competition by implementing multiple designs and establishing CAESAR Hardware API [7]. In total, 28 of the 29 Round 2 candidates received full or partial submissions. The CAESAR Committee made the hardware implementations obligatory for round 3, and as a result, 27 designs were received for the 15 Round 3 candidates [6]. The CAESAR competition ended in 2019 with Ascon declared the winner for the use case of lightweight applications [8].

2.2 Hardware Application Programming Interface (API)

Hardware implementations must adhere to the same API, ensuring the same set of features to allow for comparison. CERG developed the CAESAR Hardware API [7], which was approved by the CAESAR selection committee. To go along with the API, CERG created the Implementer's Guide to Hardware Implementations Compliant with CAESAR Hardware API [9] and a VHDL-based development package for the CAESAR Hardware API [10] to aid developers in making compliant designs. The development package provides VHDL code for a generic PreProcessor, PostProcessor, and Header FIFO to assist implementors. The

development package also includes a universal testbench that can ensure that designs are compliant and functional. Test vectors used by the testbench are generated by compiling the reference C implementations and using a Python3 script to generate test vectors.

2.3 NIST LWC

NIST's LWC competition began in 2017 to supplement the current encryption standard, Advanced Encryption Standard (AES), with an alternative that performs better in a resource-constrained environment. Some of the high-level requirements NIST placed on these algorithms include that all AEAD must use a key at least 128-bits in length. Additionally, both hash and AEAD algorithms must have computational complexity of at least 2^{112} when attacked by classical computers [11]. NIST also desires that algorithms be designed with side-channel resistance in mind [11].

In April 2019, NIST selected 56 of the 57 candidate submissions for Round 1 of the standardization process [12]. Round 1 submissions included the algorithm specification along with a C software reference implementation of the candidate. During the first round, the cryptographic community investigated algorithms for security concerns, and the software reference implementations were benchmarked [13]. By the start of Round 2, in August 2019, NIST reduced the number of candidates to 32. In this round, candidates continued to be investigated for security concerns, and reference implementations were updated as required. In keeping with the previous competition, CERG began implementing and preparing to benchmark the candidates in Round 2. Chapter 3 focuses on the two Round 2 submissions developed as part of this thesis: Elephant and Xoodyak, [14, 15]. Chapter 4 highlights contributions made to the CERG team's efforts to benchmark Round 2 submissions. Additionally, Chapter 4 includes an analysis of how the developed designs performed in both the FPGA and application-specific integrated circuit (ASIC) benchmarking that occurred during Round 2.

2.4 Side Channel Protection

Side-channel analysis (SCA) poses a severe threat to the security of cryptographic algorithms. LWC applications will be typically deployed in locations that provide little to no physical protection to prevent attackers from accessing the device. Physical access allows attackers to collect information about how the cryptosystem is operating via side channels.

Simple Power Analysis (SPA) and Differential Power Analysis (DPA) are two examples of side-channel techniques that attackers use to gather information about the secret key. An attacker performing SPA measures the power being consumed to reveal information about what is occurring within the circuit. One limitation of the SPA attack is that it requires a high signal-to-noise ratio to be effective. When performing a DPA attack, the goal is to determine a small subset of the key. The attacker analyzes the device's power consumption and predicts possible values of the key. If the attacker incorrectly guesses the key's value, then when the power traces are differentially compared, the signals will mostly cancel out, and the only thing left is the noise of the signal. However, if the attacker correctly guesses that part of the key, a spike will occur when the signals are differentially compared.

One way to protect against SCA is using masking techniques. Masking techniques produce side-channel leakage that is statically independent of the data being processed [16]. A basic example of masking is shown by the following equation: $x_m = x \oplus m_0 \oplus m_1$. In this equation, the mask values m_0, and m_1 are used to modify the original value of x to obtain x_m. By doing this, even if the attacker were to obtain two of the three values of x_m, m_0, and m_1 the attacker still would not know the original value of x. In essence, the attacker must know all of the mask shares to be able to determine the data being processed. Two of the most commonly implemented variants of these masking techniques are Threshold Implementations (TI) [17], and Domain-oriented Masking (DOM) [18, 19].

The protected designs covered in this thesis use DOM for protection. DOM requires $d + 1$ shares to achieve d^{th} order of protection. d^{th} order of protections implies that the attacker can place up to d probes to gather information. Protection of linear functions is trivial for all masking techniques because the result is independent of the other shares.

Non-linear functions require information from domains to cross to perform the calculation. DOM designs protect domain crossing by adding a new random share and using registers to prevent propagating glitches. Glitches occur in Complementary Metal Oxide Semiconductor (CMOS) when the output signal transitions based on one part of the input signal arriving before the other. Using a flip-flop (FF) ensures the signal is steady before propagating further in the circuit.

Gross in [16] uses the example of a simple multiplier shown in Figure 2.1 to show how DOM works. This example shows the two inputs x and y have both been split into random and independent shares of each other. The multiplication produces inner-domains product terms (x_0y_0, x_1y_1) and cross-domain terms (x_0y_1, x_1y_0). Notice in Figure 2.1 that cross-domain terms receive extra randomness and that the output is stored in a FF to prevent the glitches.

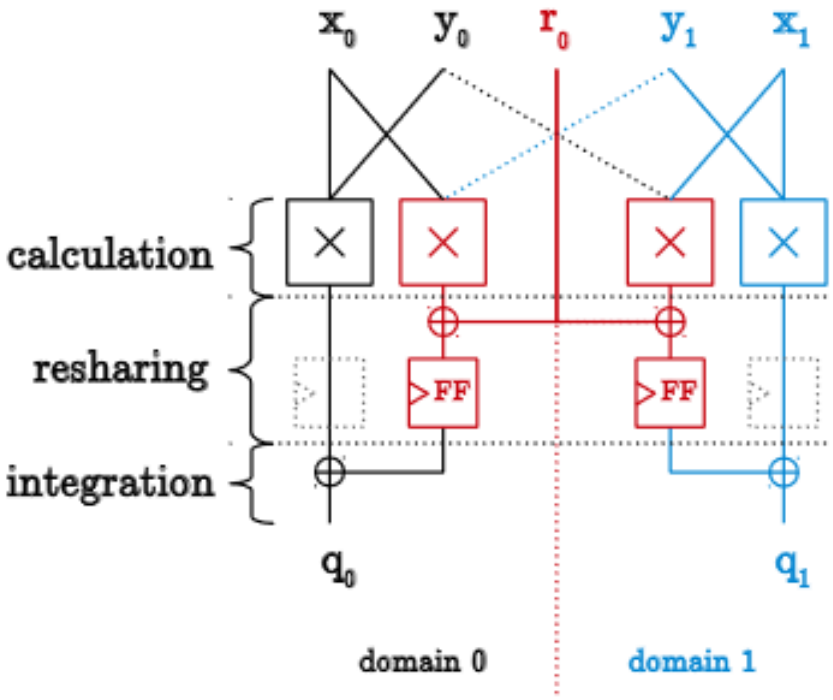


Figure 2.1: First-order DOM GF(2^n) Multiplier

A verification method is required to ensure that protection requirements are achieved. One such approach is Test Vector Leakage Assessment (TVLA) [20,21] using Welch's t-test. The goal behind the t-test is to determine if the data sampled from the same population is distinguishable. If the attacker cannot determine the difference between the sets of data, then the attacker cannot determine if the guess was correct when performing their DPA attacks.

Equation 2.1 shows how the results of the t-test are computed. In this equation, the μ stands for the sample mean, s for the sample variance, and n the number of samples in each set. In [21] the author goes through the mathematical justification to state that $|t| > 4.5$ fails the t-test because the samples are distinguishable with confidence of 99.999%.

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{{s_0}^2}{n_0} + \frac{{s_1}^2}{n_1}}} \tag{2.1}$$

Chapter 3: Implementations of LWC Candidates

When comparing hardware designs, it is not easy to achieve a one-to-one comparison. For instance, one design team might report their ASICs results and while another team reports FPGA results. There is no direct comparison between kilo-gate equivalent (kGE) and lookup tables (LUTs), so comparing the results is challenging. Another variability that may exist between designs is the supported features. In Why Does Hardware API Matter [22] one example of feature mismatch given is that key scheduling and padding are performed external to the circuit in one design and internally in another. A design that pads and schedules the key is likely larger, slower, and uses more energy.

Even if all designers create designs for FPGAs, comparing results may still be difficult based on the chips targeted and the resources used. One implementation may use Digital Signal Processing (DSP) units and Block RAMs (BRAMs), while another uses only LUTs. To allow for comparison, CERG provided FPGA design goals in [23]. Each design presented in this work achieved the highly recommend goals of:

1. 2000 or less LUTs
2. 4000 or less FFs
3. No BRAMs or DSP units

To support comparison between designs and CERG's FPGA benchmarking effort, all the designs covered in this thesis are using the LWC Hardware API[24]. Figure 3.1 provides a high-level view of the interface that the development package offers developers. Developers implement the features of their cipher in the CryptoCore block. The CryptoCore contains all required operations for an algorithm to perform encryption and decryption. Using the LWC development package allows developers to leverage PreProcessor and PostProcessor to achieve message header processing transparently.

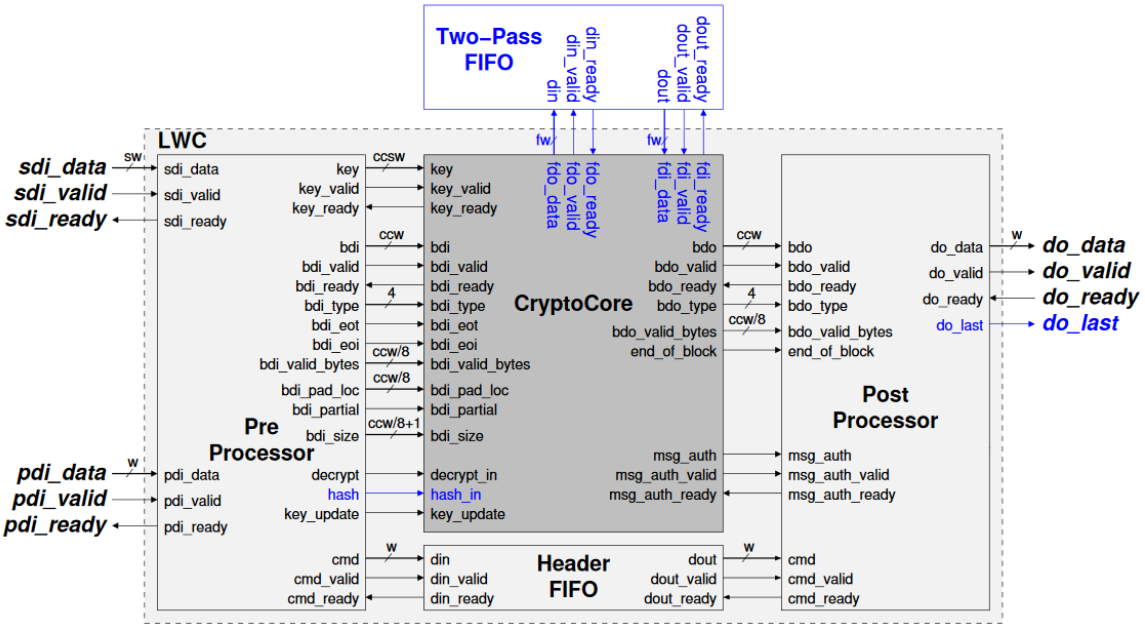


Figure 3.1: Top-Level Block Diagram of LWC Core

The two LWC candidates targeted for implementation as part of this thesis were Elephant and Xoodyak. Multiple variants of each algorithm were developed for both candidates. Each algorithm has one variant that is a basic-iterative architecture design. Xoodyak has an additional folded version attempting to reduce area. No unrolled variants of Xoodyak were implemented as part of this work, but only slight modification would be required to support it. The Elephant basic-iterative design contains several variants that change the unrolling factor of the permutation. A more complex pipelined implementation of Elephant was also created that processes the message and tag in parallel. This pipelined implementation has several unrolled versions of the permutation as well. The following are common features to all of the designs presented in this paper:

1. Support authenticated encryption/decryption
2. Two-Pass FIFO is not required
3. LWC IO width is 32-bits

4. Messages sizes up to $2^{50} - 1$ are supported

3.1 Implementation of Xoodyak

Xoodyak only has one variant submitted to NIST's LWC that supports both AEAD and Hashing. Xoodyak uses the Xoodoo permutation, which is inspired by the Keccak permutation. Xoodyak has a sponge construction that performs its Cyclist operations. Keyak's Motorist mode inspires the Cyclist operations. For full details of Xoodyak, see the specification [15].

At a high level, Xoodyak has a 384-bit state that consists of 3 128-bit planes. The Xoodoo permutation consists of the following 5 steps: θ mixing planes, ρ_{west} shift planes, ι adding round constants, χ non-linear layer and ρ_{east} shift planes. Cyclist uses the Xoodoo permutations in a sponge construction to absorb and squeeze the data. By specification, Cyclist has the following operations Absorb, Encrypt, Decrypt, Squeeze, SqueezeKey, and Ratchet. Based on the source code submitted for Xoodyak, the SqueezeKey and Ratchet operations are not required to match the reference source code implementation for AEAD and hashing functionality. Figure 3.2 contains a detailed breakdown of the different constants required to be added in the various Cyclist operations.

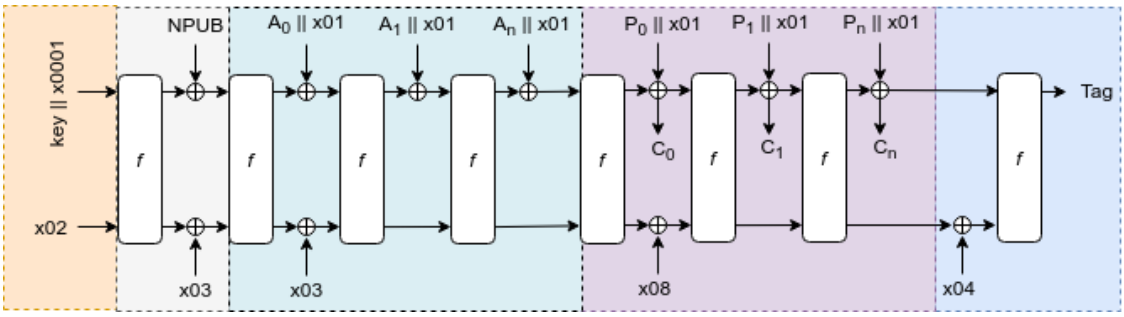


Figure 3.2: Xoodyak Flow for AEAD Encryption

Xoodyak was one of the first designs that both the CERG team and myself attempted to implement as part of our work with LWC Round 2. At first, we attempted to fold designs as

much as possible to reduce the circuit area. As a result, the first attempt at implementing Xoodyak was a 32-bit datapath that took over 300 cycles to complete a single round of the Xoodoo permutation. Due to this design's inefficiency, it is not covered in this work, and CERG learned that most of the LWC candidates should not be folded. As time progressed, we began increasing the width of the datapath. This work contains two Xoodyak designs, a 384-bit basic iterative architecture, and a folded 128-bit datapath. Even though the folded design was developed first, the basic iterative design is easier to understand and, therefore, discussed first.

The three-plane design of the Xoodoo permutation can be seen in Figure 3.3. The *theta* portion of the permutation starts with the different planes being XORed together. The result is shifted and then XORed back into each of the planes. ρ_{west} operates on the second and third planes only, shifting the bits within those planes. ι adds the round constant to the first plane. The round constant is obtained from a lookup table based on the current round. χ adds the non-linear AND gate operations to the permutation. For each of the rows, the other respective rows are combined through an AND gate then XORed with the current row. ρ_{east} is the final operation of the permutation which, similarly to ρ_{west}, only operates on the second and third planes, shifting the bits within these planes.

To complete the basic-iterative design of Xoodyak requires the implementation of the Cyclist features. Cyclist controls how specific constants shown in Figure 3.2 are added to the current state of Xoodyak after each block of data is processed. Figure 3.4 contains the full 384-bit datapath of Xoodyak. Muxes choose between the different planes, and then additional muxes are used to determine the correct 32-bit portion of those planes to update. Those 32-bit chunks are then XORed with the input BDI and fed back into the state through a stream of muxes to ensure the correct location is updated. The notation of 0^{384} is indicating 384-bits that are all zero.

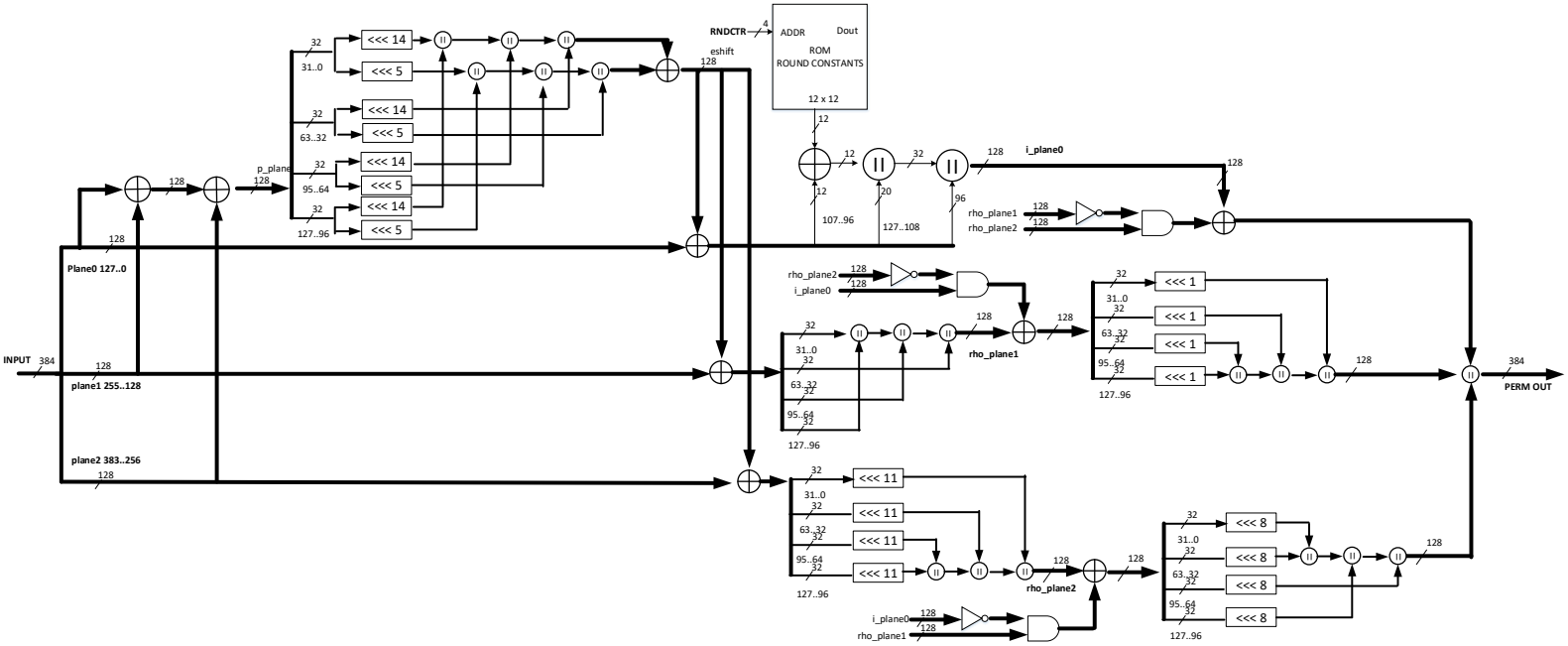


Figure 3.3: Basic Iterative Xoodoo Permutation

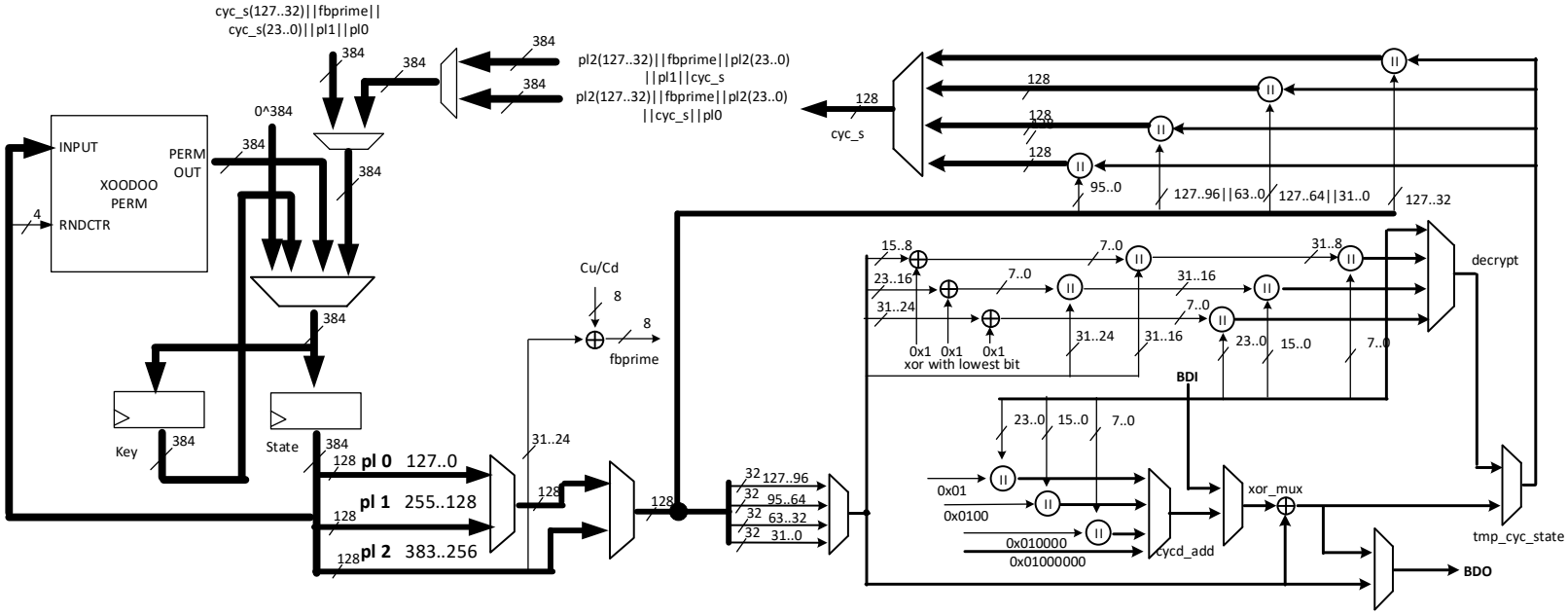


Figure 3.4: Xoodyak 384-bit Datapath

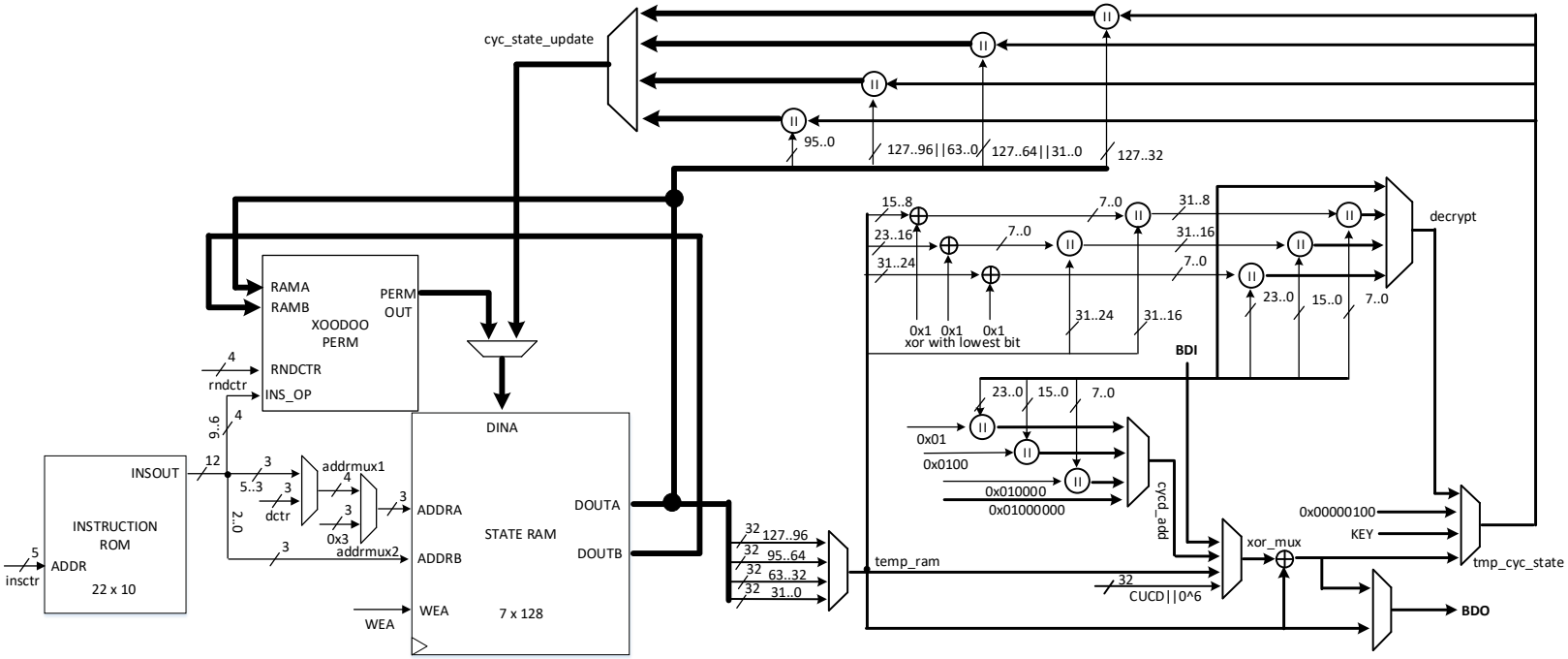


Figure 3.5: Xoodyak 128-bit Folded

Although the 128-bit design is less efficient, the folded design provides some interesting discussion points. Folding Xoodyak requires parts of the Xoodoo permutation to be performed serially using an arithmetic log unit (ALU). Figure 3.5 shows the construction of the full datapath of this design. A 16 x 128 STATE RAM stores the state and intermediate values of the Xoodoo permutation. Without key reuse, the STATE RAM reduces to 8 x 128. Storage locations within the STATE RAM are controlled by a 32x12 instruction ROM. These instructions have the following format: bits 0-3 and 4-7 select the STATE RAM's current operation addresses. The highest 4 bits, 8-11, control the action performed by the 128-bit Xoodoo ALU. Much of the right-hand portion of the 128-bit Xoodyak design in Figure 3.5 is similar to the 384-bit version. The 128-bit version does not require the additional muxes to select between the different planes because all operations occur on a per plane basis out of the state RAM.

The construction of the 128-bit Xoodoo permutation, shown in Figure 3.6, is much simpler to look at than its 384-bit counterpart. Each phase of the permutation is split into possible instructions stored in the instruction ROM. The instruction ROM selects the correct operation of the ALU. Each operation is performed on one plane at a time, which means that to perform both the θ and χ functions, each plane's original values must be saved off before being updated.

3.2 Implementation of Elephant

Elephant is a lightweight permutation-based authenticated encryption scheme that contains three variants.

1. Dumbo: Elephant-Spongent-π[160] 112-bit security
2. Jumbo: Elephant-Spongent-π[176] 127-bit security
3. Delirium: Elephant-Keccak-f[200] 127-bit security

Dumbo is the primary variant of the submission package, making it the logical choice for hardware implementation. For full specification of the algorithm, see [14]. Dumbo's

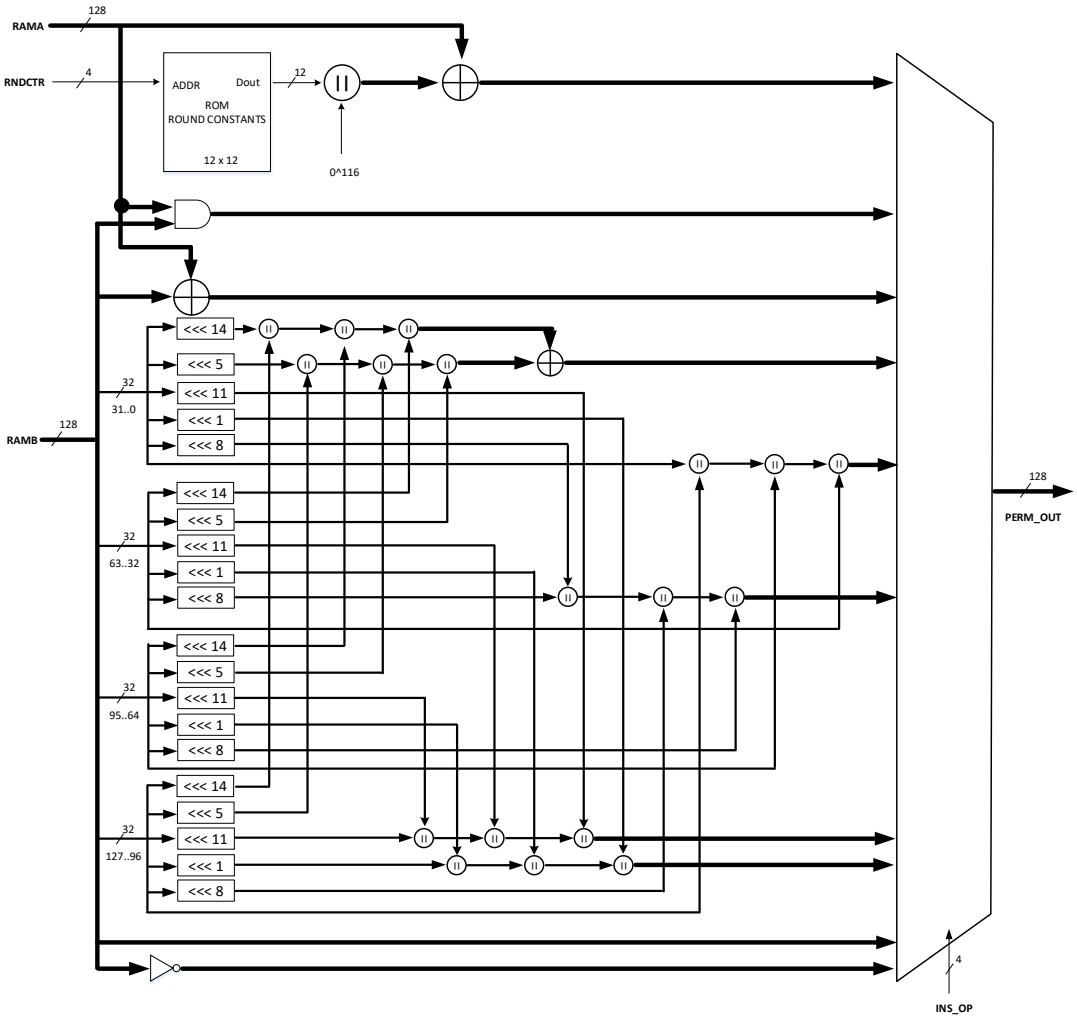


Figure 3.6: Xoodoo 128-bit Folded

permutation consists of 80 rounds of the Spongent lightweight hash function. Each round of Spongent consists of 3 layers.

1. Xor with the ICounter
2. S-box Layer
3. pLayer

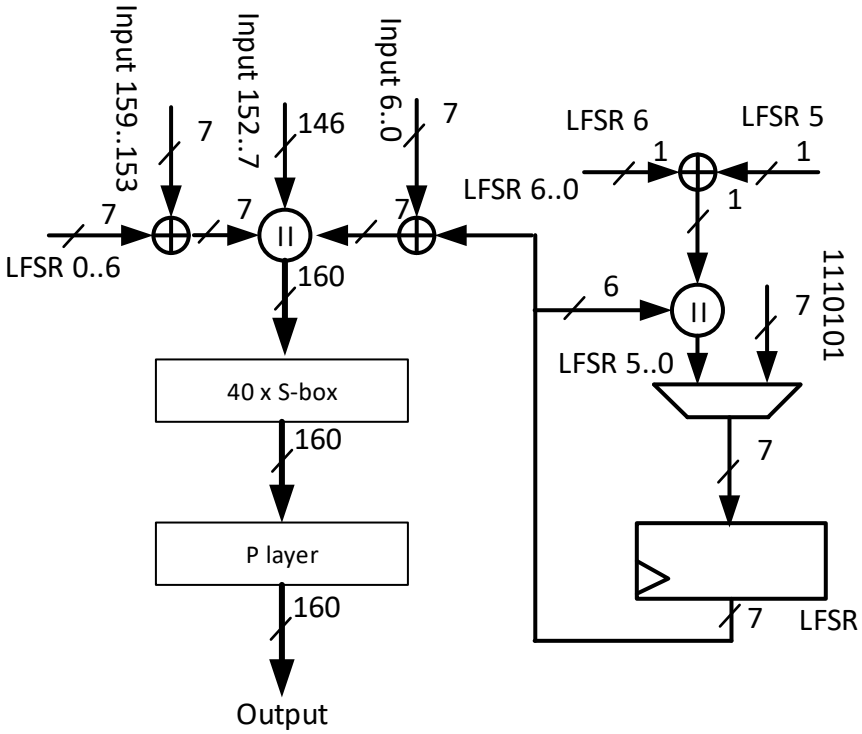


Figure 3.7: Round of Spongent Permutation

Initialization of the permutation sets a 7-bit Linear Feedback Shift Register (LFSR) to 1111010. Figure 3.7 shows the implementation of the Spongent permutation. The right-hand portion of this figure shows how the LFSR is updated each round. The ICounter layer takes the LFSR's output and xors it with the lowest 7-bits of the input. Additionally,

Table 3.1: S-box

x	0 1 2 3 4 5 6 7 8 9 A B C D E F
$S[x]$	E D B 0 2 1 4 F 7 A 8 5 9 C 3 6

the LFSR's output bits are reversed and then XORed with the highest 7-bits of the input. The output of the ICounter layer proceeds into the S-box layer, which consists of 40 4-bit S-boxes. Table 3.1 gives the construction of the S-boxes. The S-boxes are implemented as lookup tables in all of the Elephant designs submitted with this work. The permutation's final layer is the pLayer, which shifts bits locations based on the following equation.

$$P_{160}(x) = \begin{cases} 40 \cdot x \bmod 159 & \text{if } x \in \{0, ..., 158\} \\ 159, & \text{if } x = 159 \end{cases} \tag{3.1}$$

At first glance, Elephant acts similarly to other candidates but has the additional feature of being parallelizable, allowing Elephant to process inputs of AD and PT/CT simultaneously. Figure 3.8 taken from the specification [14] shows how blocks can be processed at the same time. Unfortunately, the LWC Hardware API does not support processing multiple input types simultaneously, artificially limiting the Elephant algorithm.

In Figure 3.8 the top blocks are processing the message, and the output is C_M. C_M is then the input into the lower blocks, which contribute to the tag. In the basic iterative design, these blocks must be sequentially computed, requiring that the bottom block follow the top block. In the pipelined implementation, these blocks are processed in parallel, allowing the top block to process $M + 1$ while the lower block is computing the tag for M. Although this does not take full advantage of processing AD and PT/CT simultaneously, it improves throughput.

The mask values in Figure 3.8 come from an LFSR that is initialized with the output of the $P(K||0^{n-k})$, where n is the number of bits in the state (160), and k is the number of

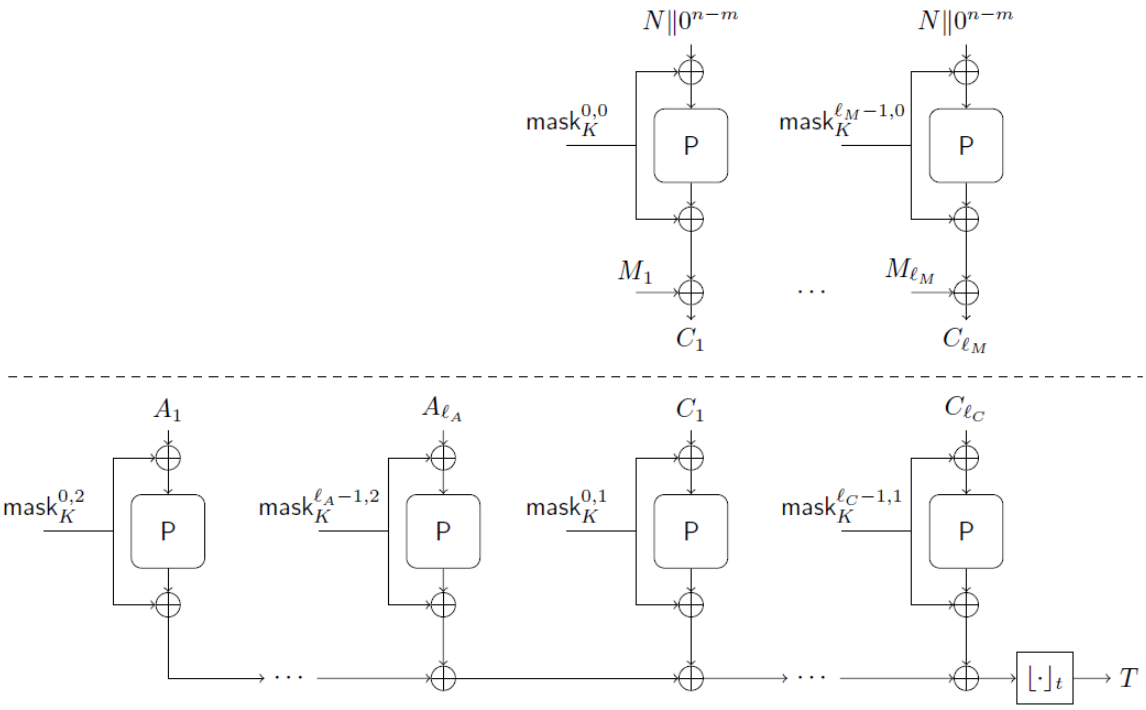


Figure 3.8: Elephant Encryption

bits in the key (128). Once initialized, the following equation updates the LFSR:

$$(x_0, ..., x_{19}) \rightarrow (x_1, ..., x_{19}, x_0 \lll 3 \oplus x_3 \ll 7 \oplus x_{13} \gg 7) \tag{3.2}$$

The $mask_k$ values in Figure 3.8 are written such that $mask_k^0$ is the previous mask, $mask_k^1$ is the current mask, and $mask_k^2$ is the next mask. Instead of storing these masks in multiple registers, these masks are saved by extending the size of the LFSR. The LFSR is updated after each block of data is processed. Figure 3.9 contains the LFSR implementation used to update the mask. The previous mask is stored in bits 0-159, the current mask in 8-167 and the next mask in 16-175.

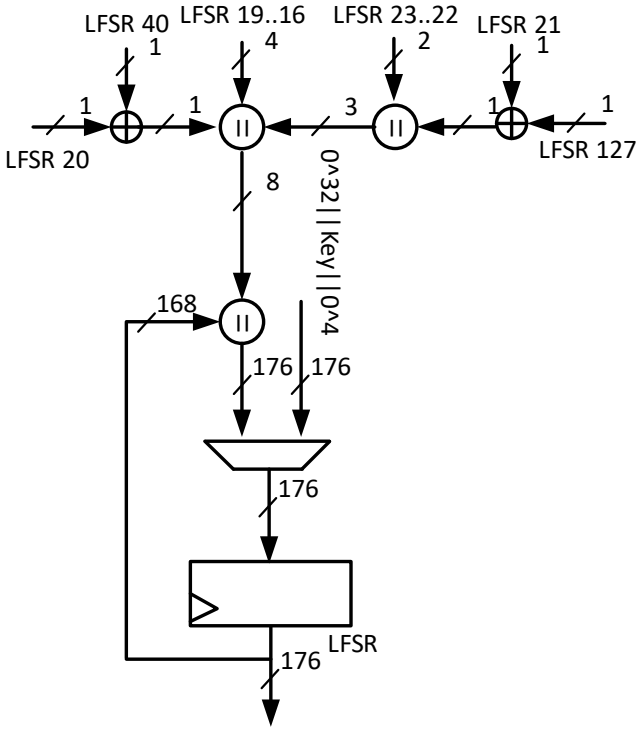


Figure 3.9: Mask LFSR

The basic iterative design of Elephant is shown in Figure 3.10. Inputs come into the circuit through the muxes located in the upper left-hand portion of the diagram. This

data is stored in the load_data resister until it is ready for processing. Storing data in the load_date register is required because when encrypting the PT, the CT is produced, which must be stored because of its contribution to the tag. Since Npub is required for the processing of each message block, it is stored in a register. Once a full block is received, or the end of a data type is reached, data is loaded into the main state register (MS) via the output of the lfsr_mux. The other possible outputs of the lfsr_mux are the different mask values previous, current, and next. BDO output selects the value stored in the tag register or BDI XORed with the appropriate MS register location that produces the CT or PT.

Unrolling the Spongent permutation is accomplished by feeding the output of the permutation into the next round. Figure 3.11 contains a diagram of the Spongent permutation unrolled 4 times. A critical difference between the basic iterative permutation and the unrolled permutation is that the permutation LFSR is implemented as a lookup table. Unrolled variants of the Spongent permutation completed include 2x, 4x, 5x, 8x, and 16x times unrolled. Larger unrolled versions would undoubtedly exceed the 2000 LUT limit and likely have a reduced throughput due to a longer critical path.

The pipelined design shown in Figure 3.12 is significantly different than the original version. There is a serial-in-parallel-out (SIPO) register receiving inputs and a parallel-in-serial-out (PISO) register on the output in this design. These registers allow data to be obtained and released while the circuit is processing its current block. Additionally, having these registers ensures that other parts of the LWC development package are not contributing to the circuit's critical path. To support pipelined processing of PT/CT blocks requires duplication of the Spongent permutation block and state register. The PT/CT data released from the circuit is processed on the left-hand side of the figure. Once this data is processed, it is stored in the PISO and released. This same data goes through another padding circuit before going to the right-hand side, where the tag is calculated.

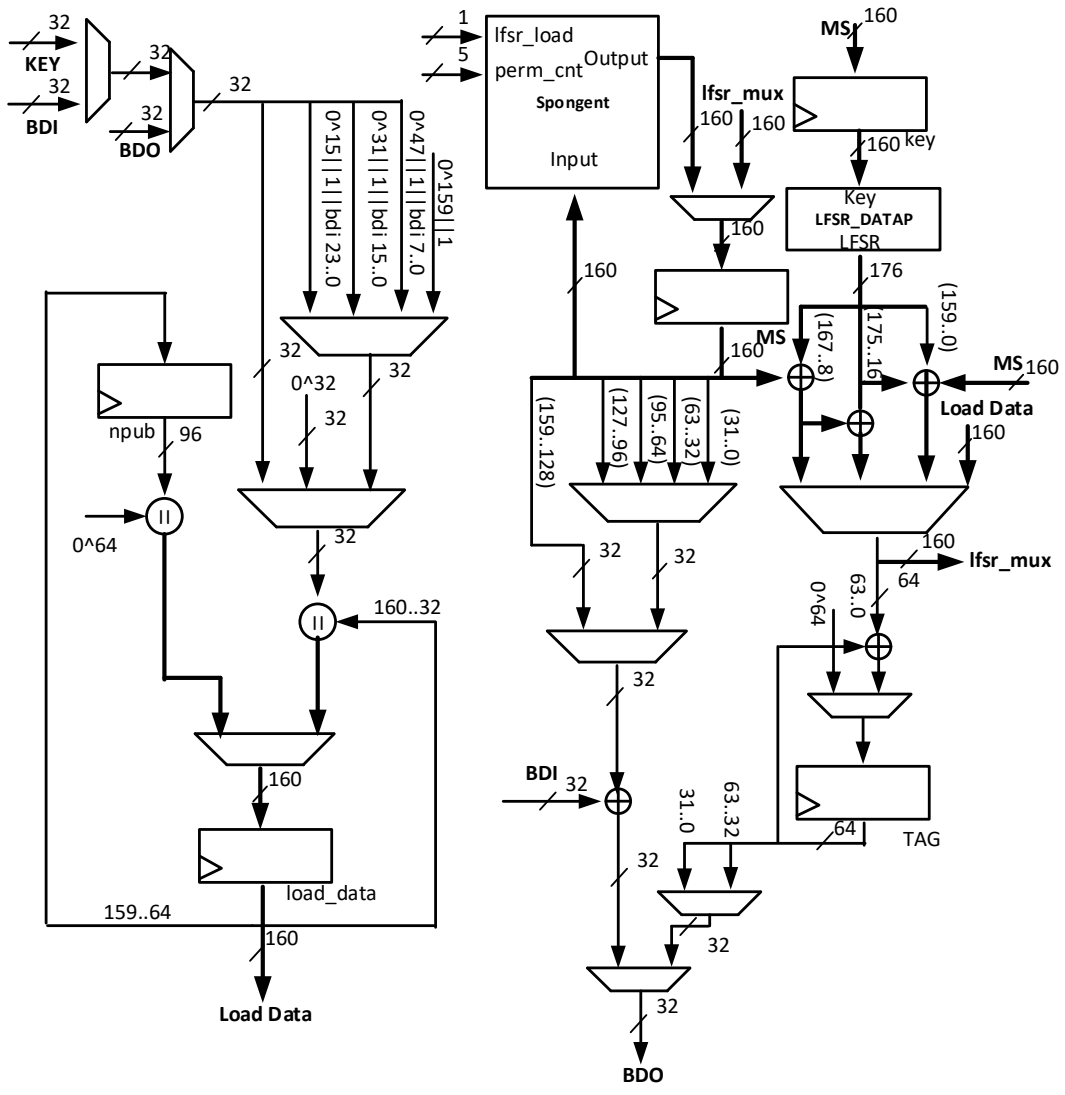


Figure 3.10: Elephant Datapath

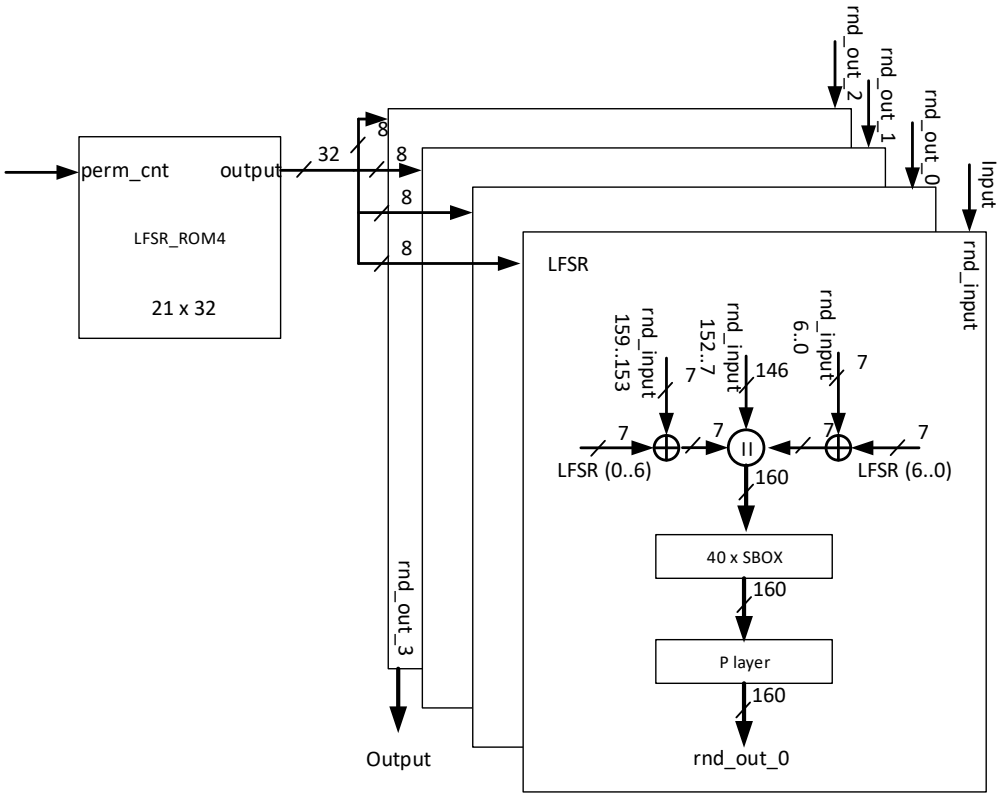


Figure 3.11: Spongent Unrolled 4 Times

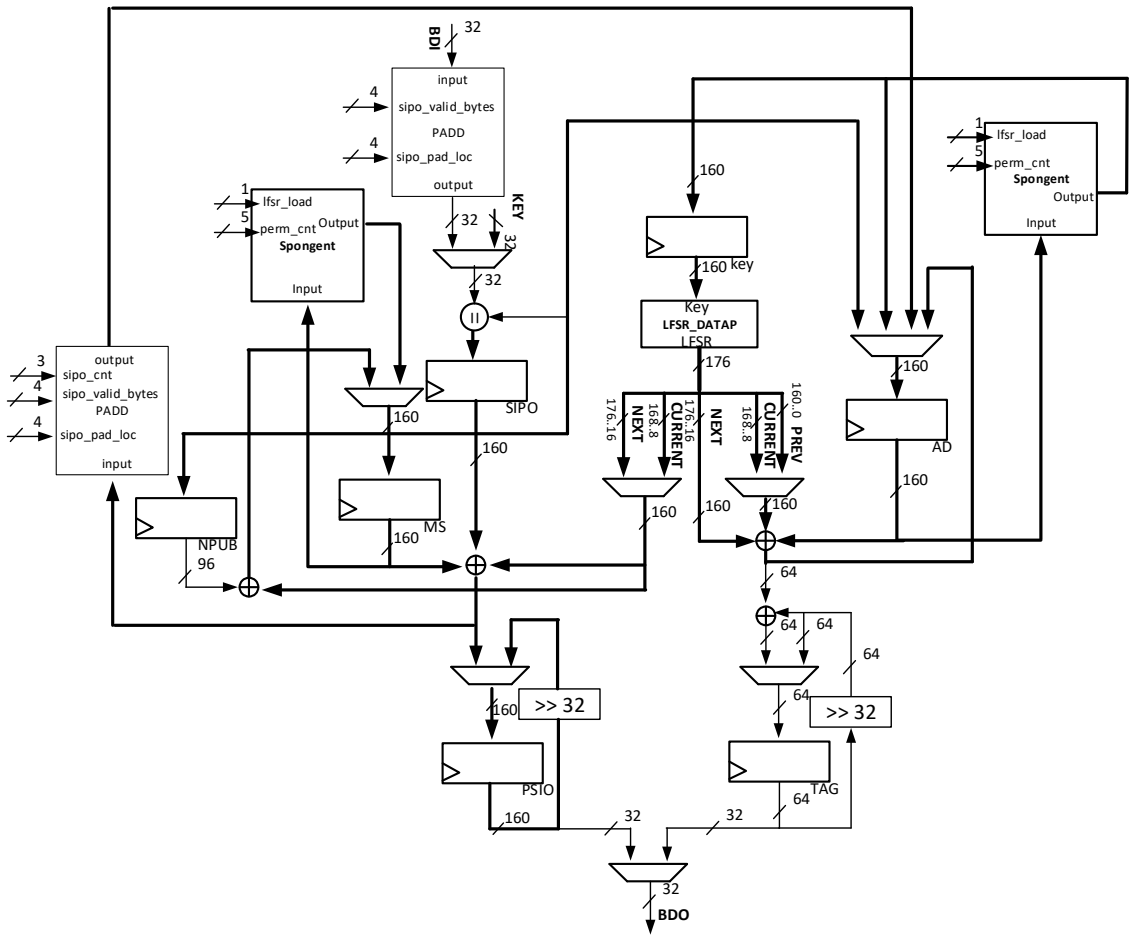


Figure 3.12: Elephant Datapath Parallel Processing

Chapter 4: FPGA Benchmarking of LWC Candidates

CERG has put an extensive amount of effort into supporting the benchmarking of lightweight cryptographic algorithms. In 2019, CERG proposed a framework to benchmarking hardware implementations of LWC [5], which assumed using the LWC Hardware API [24], and the corresponding LWC Development Package. The adoption of this framework required a significant revision of the CAESAR Development Package [25, 26]. In 2020, additional improvements were made to support:

1. Different synthesis tools
2. Improved functional verification
3. Execution timing measurements

The LWC Hardware API development package is maintained on GitHub [27]. Release v1.1.0 of the development package is the first release to contain the features listed. The Implementer's Guide [28] has been used to document these additional features.

Starting in the fall of 2020, as the end of LWC Round 2 approached, CERG began benchmarking candidates submitted by the LWC community. In total, CERG benchmarked 40 different submission packages. These 40 submissions covered 27 out of the total 32 Round 2 candidates. CERG contributed 14 designs covering the following candidates: ACE, Ascon, Elephant, GIFT-COFB, Gimli, mixFeed, PHOTON-Beetle, Pyjamask, Saturnin, SKINNY-AEAD, SPIX, Subterranean 2.0, TinyJAMBU, and Xoodyak.

CERG selected one FPGA family from the following three vendors: Xilinx, Intel, and Lattice Semiconductor. CERG used the following criteria to select the FPGA family: [6] must be widely used, low-cost, and low-power, capable of holding SCA-protected designs,

and supported by free versions of state-of-the-art industry tools. The following families and specific devices were selected:

1. Xilinx: Artix-7 xc7a12tcsg325-3
2. Intel: Cyclone 10 LP 10CL016-YF484C6
3. Lattice Semiconductor: ECP5 LFE5U-25F-6BG381C

4.1 Support for Different Synthesis Tools

Each synthesis tool has different support levels for language features and varies in enforcing restrictions on intermixing values with different data types (strong typing). Some examples are that only some VHDL-2008 features are supported, or assignment between objects of different sizes is not allowed. When testing which tools CERG would use for benchmarking, several of the LWC development package files needed minor code cleanup to eliminate warnings and errors. This code cleanup is why almost every file in the LWC Hardware API development package experienced some change in the latest release.

4.2 Improved Functional Verification

Extensive testing is required to ensure that designs are fully functional. In particular, the following cases need testing because they are some of the most difficult to implement.

1. Empty AD or empty PT/CT or both
2. Incomplete blocks
3. Support for key reuse
4. Crossing block boundaries
5. Hashing (if supported)

CERG automated the generation of test vectors by adding gen_test_routine to cryptotv-gen. Test vector generation requires several design parameters to be known, such as key

size, Npub size, and hash size. The api.h file, in the designs software reference implementation, contains many of these required parameters. This file is parsed when running the gen_test_routine function reducing the number of required input parameters. CERG did not provide this parsing to all the different cryptotvgen functions but may in the future.

The gen_test_routine option produces several sets of test vectors. One of these sets is used to determine the number of cycles per block. The number of cycles per block is calculated by taking the differences in cycles between messages that are four and five times the block size. Block sizes of the cipher are not contained in the api.h and therefore require the user to provide it as an argument to gen_test_rountine. Ideally, block sizes would be reported as constants in api.h to improve automation.

Another added feature to support functional verification was the ability to control the LWC_TB's behavior when a set of test vectors fails. Initially, the testbench would halt whenever there was unexpected output. Immediately halting is convenient when developing, but not when providing feedback to developers about possible issues. It is critical to report as many of the potential problems at one time to the developer as possible. To support this G_MAX_FAILURES was added to the testbench, which controls the number of failures allowed before the testbench halts. Besides allowing the simulation to continue, failed test vectors are logged to the file specified by G_FNAME_FAILED_TVS. These features aid in providing feedback to developers about possible bugs in their implementations.

4.3 Timing Measurements

When comparing designs, one of the more interesting metrics to analyze is maximum throughput. To determine maximum throughput requires knowledge of the maximum clock frequency, the number of clock cycles to complete the message, and the message's size. CERG used the automation tools ATHENA[29], Minerva[30], and Xeda[31] to determine the maximum frequency for the different FPGA families. CERG added Measurement Mode to LWC_TB.vhd to provide metrics about the number of cycles required for different messages.

Measurement Mode adds a counter that tracks how long a given test vector takes to travel from the Pre-Processor input to the output of the Post-Processor. One of the more difficult parts of implementing this feature was preventing the testbench from moving on to the next test vector before completing the first. To accomplish this, once a given test vector is read, the testbench blocks new input data until the PostProcessor sets the do_last flag. The number of cycles required for a message is calculated by taking the counter's value when do_last is set and subtracting the value present at the beginning of the message. Measurement Mode logs the following information into both a text file and CSV file:

- Total size of the different data types
- Total number of complete blocks of each data type
- Indication of the presence of an incomplete block
- Size of the incomplete block
- Execution time in cycles

4.4 Troubleshooting Designs

The most time-consuming part of benchmarking the Round 2 candidates is the effort put into troubleshooting designs that experienced issues. CERG spent many hours looking at designs attempting to determine why each was failing. If the problem could be determined, CERG provided authors with a detailed response about how to fix the issue. Otherwise, CERG provided the authors with the failing test vectors. Although these troubleshooting efforts did not result in contributions to this thesis, they are still worth noting.

4.5 FPGA Benchmarking Results

CERG's work on FPGA benchmarking Round 2 candidates resulted in the following publications [32] [33]. In these publications, candidates are compared against one another in the following categories: throughput, area, energy, and power. Each candidate is ranked first

to last for throughput of the following types of message AD, PT, AD + PT, and Hashing. CERG calculated each candidates' throughput for each message type for each of the following input sizes: 16 bytes, 64 bytes, 1536 bytes, and long message. The following formula calculates throughput for long message: $\frac{\frac{bits}{block} \times Frequency \frac{cycles}{sec}}{\frac{cycles}{block}}$. CERG reported these results for each of the FPGA families.

For power and energy estimation, CERG used Xilinx's Vivado v2020.1 power analysis feature to run vector-based estimations for an Artix-7 xc7a12tcsg325-3 at a target frequency of 75 MHz [32]. CERG reported that nearly all candidates had the same static power of 60mW and focused their analysis on the candidates' dynamic power. CERG accounted for the variance in dynamic power by measuring multiple messages. CERG used 20 messages of 16 bytes when measuring short messages and five messages of 1536 bytes for long messages. The reported dynamic power is the average of these runs for each message size.

Due to the effort required to benchmark, design variants developed by this thesis were limited to the following designs:

- Elephant-v1 Basic iterative architecture 1x unrolled
- Elephant-v2 Basic iterative architecture 5x unrolled
- Elephant-v3 Basic iterative architecture slight improvements 4x unrolled
- Elephant-v4 Pipelined 2x unrolled
- Elephant-v5 Pipelined 4x unrolled
- Xoodyak_GMU-v1 Basic iterative architecture 1x unrolled
- Xoodyak_GMU-v2 Folded 128-bit datapath

Besides the Xoodyak designs created as part of this thesis, the Xoodyak Team submitted additional variants labeled Xoodayk_XT and another submission from GMU by Kamyar Mohajerani labeled Xoodyak_GMU2. The following designs of Xoodyak are interesting for comparison because they are all basic iterative architectures that are unrolled only one time. No other submission attempted to fold Xoodyak.

- Xoodyak_GMU-v1
- Xoodyak_XT-v1
- Xoodyak_XT-v7
- Xoodyak_GMU2-v1

CERG's FPGA benchmarking effort produced a substantial amount of data for the comparison of candidates. The block sizes and cycles per block for each Elephant and Xoodyak variant benchmarked are contained in Table 4.1. Elephant-v4 and Elephant-v5 provided slight improvement when processing AD compared to the non-pipelined implementation. The slight improvement was due to the FIFO on the input. The benefit of pipelining Elephant was evident when processing PT-CT because it reduces the number of cycles required to process a block by one-half.

Xoodyak_GMU-v1, Xoodyak_XT-v1, and Xoodyak_XT-v7 all require the same number of cycles to process a block of AD, PT-CT, and hash. Xoodyak_GMU2-v1 uses a FIFO on the input to reduce the number of cycles by almost one-half. The FIFO on the input allows the Xoodyak_GMU2 designs to process a block in $1 + \frac{12}{unrolling\ factor}$ cycles while the other basic iterative designs use $1 + \frac{block\ size\ bits}{32} + \frac{12}{unrolling\ factor}$ cyles. The number of cycles per block for Xoodyak_GMU-v2 is not on the same order of magnitude as the other designs because it is folded.

Table 4.1: Basic Properties of Elephant and Xoodyak Variants

Variant Name	**AD Block Size [bits]**	**Cycles per AD Block**	**PT-CT Block Size [bits]**	**Cycles per PT-CT Block**	**Hash Msg. Block Size [bits]**	**Cycles per Hash Msg Block**
Elephant-v1	160	88	160	171		
Elephant-v2	160	24	160	43		
Elephant-v3	160	28	160	51		
Elephant-v4	160	43	160	42		

Table 4.1: Continued From Previous Page

Variant Name	AD Block Size [bits]	Cycles per AD Block	PT-CT Block Size [bits]	Cycles per PT-CT Block	Hash Msg. Block Size [bits]	Cycles per Hash Msg Block
Elephant-v5	160	23	160	22		
Xoodyak_GMU-v1	352	24	192	19	128	17
Xoodyak_GMU-v2	352	266	192	261	128	259
Xoodyak_XT-v1	352	24	192	19		
Xoodyak_XT-v2	352	18	192	13		
Xoodyak_XT-v3	352	16	192	11		
Xoodyak_XT-v4	352	15	192	10		
Xoodyak_XT-v5	352	14	192	9		
Xoodyak_XT-v6	352	13	192	8		
Xoodyak_XT-v7	352	24	192	19	128	17
Xoodyak_XT-v8	352	18	192	13	128	11
Xoodyak_XT-v9	352	16	192	11	128	9
Xoodyak_XT-v10	352	15	192	10	128	8
Xoodyak_XT-v11	352	14	192	9	128	7
Xoodyak_XT-v12	352	13	192	8	128	6
Xoodyak_GMU2-v1	352	13	192	13	128	13
Xoodyak_GMU2-v2	352	12	192	7	128	7

A condensed version of CERG's results for resource usage and maximum frequency are reported in Table 4.2 [32]. The Elephant results show that each layer of the Spongent permutation used approximately 170 LUTs for Artix-7. Results for Elephant-v2 and Elephant-v5, which are both 4x unrolled, show that pipelining in Elephant-v5 led to an increase of approximately $4x \times 170$ LUTs. Elephant-v5's second permutation block accounts for most of this increase in size. The effects of the FIFO and PISO in the pipelined versions appear in two locations in these results. First, the number of FFs, and second, the maximum frequency achieved. The FIFO and PISO each add 160 more FFs. An additional 160 FFs come from the second state register in the pipelined version. Adding the FIFO and PISO ensures the critical path is either within the CryptoCore or external to the CryptoCore and not a combination of the two parts. Since Elephant-v3 and Elephant-v5 are both four

times unrolled, one would expect that the critical path would be within the permutation. However, since Elephant-v5 has a higher maximum frequency, it shows that input or output adds to the critical path of Elephant-v3.

When comparing the Xoodyak variants, Xoodyak_GMU-v1 did not perform as well as the other similar designs. One notable difference is the number of FFs used in Xoodyak_GMU-v1 is significantly more than the similar Xoodyak_XT design due to storing the key for key reuse. Xoodyak_GMU-v1 and Xoodyak_GMU-v2 are the only Xoodyak designs that supported key reuse. The increase in LUTs or LEs and the decrease in maximum frequency for Xoodyak_GMU-v1 compared to the other designs is due to an inefficient implementation of the Cyclist portion of Xoodyak. A detailed discussion of different Xoodyak designs is contained in section 4.7. Xoodyak_GMU-v2 is unique compared to the other Xoodyak designs because of its use of a RAM. Using a RAM led to a significant reduction in FFs and slices present in the circuit for both the Artix-7 and ECP5 results. Unfortunately, the RAM significantly increased resources for Cyclone 10 LP.

Table 4.2: Resource Usage and Maximum Frequency All Platforms

Variant	**LUTs\LEs**	**FFs**	**Slices**	**Frequency MHz**
Xilinx Artix-7				
Elephant-v1	1,291	910	379	229
Elephant-v2	1,884	900	541	181
Elephant-v3	1,717	982	501	200
Elephant-v4	1,901	1,501	567	263
Elephant-v5	2,645	1,502	759	217
Xoodyak_GMU-v1	1,808	851	495	170
Xoodyak_GMU-v2	1,234	98	323	168
Xoodyak_GMU2-v1	1,608	1,249	513	314
Xoodyak_GMU2-v2	2,322	1,228	692	199
Xoodyak_XT-v1	1,355	555	407	234
Xoodyak_XT-v2	2,025	557	579	188
Xoodyak_XT-v7	1,392	559	402	226
Xoodyak_XT-v8	2,143	559	618	181
Lattice ECP5				
Elephant-v1	2,368	923	1,464	97.5
Elephant-v2	3,073	916	1,823	85.5

Table 4.2: Continued From Previous Page

Variant	LUTs\LEs	FFs	Slices	Frequency MHz
Elephant-v3	2,901	915	1,874	88.3
Elephant-v4	3,157	1,421	1,855	97.6
Elephant-v5	4,145	1,422	2,389	90.1
Xoodyak_GMU-v1	3,172	878	1,990	74.0
Xoodyak_GMU-v2	2,316	114	1,286	74.8
Xoodyak_GMU2-v1	3,248	1,261	1,834	150.5
Xoodyak_GMU2-v2	4,058	1,233	2,351	69.7
Xoodyak_XT-v1	2,402	489	1,521	95.7
Xoodyak_XT-v2	4,077	489	2,095	70.3
Xoodyak_XT-v7	2,489	499	1,536	88.4
Xoodyak_XT-v8	4,121	499	2,125	71.3
Intel Cyclone 10 LP				
Elephant-v1	2,056	1,005		163.1
Elephant-v2	2,729	998		113.2
Elephant-v3	2,504	996		123.2
Elephant-v4	3,050	1,485		157.6
Elephant-v5	3,926	1,507		126.9
Xoodyak_GMU-v1	3,135	947		106.8
Xoodyak_GMU-v2	5,871	2,237		77.0
Xoodyak_GMU2-v1	2,575	1,256		170.3
Xoodyak_GMU2-v2	5,058	1,237		97.2
Xoodyak_XT-v1	2,231	573		136.3
Xoodyak_XT-v2	3,541	573		88.8
Xoodyak_XT-v7	2,272	583		128.5
Xoodyak_XT-v8	3,630	583		90.0

CERG's report [32] ranks candidates against each other for different message sizes across the different message types. Figures 4.1, 4.2 and 4.3 show the throughput of long AD + PT messages compared to the area for the top variants of each algorithm on the Artix-7, ECP5, and Cyclone 10 LP platforms respectively. Either Elephant-v4 or Elephant-v5 appears in these graphs depending on the size limitations applied to each FPGA family. Elephant placed in or around the top 10 in terms of throughput for all message types across all platforms. These figures only contain the best variant of a candidate; therefore, none of the Xoodyak designs produced by this work are contained within these graphs.

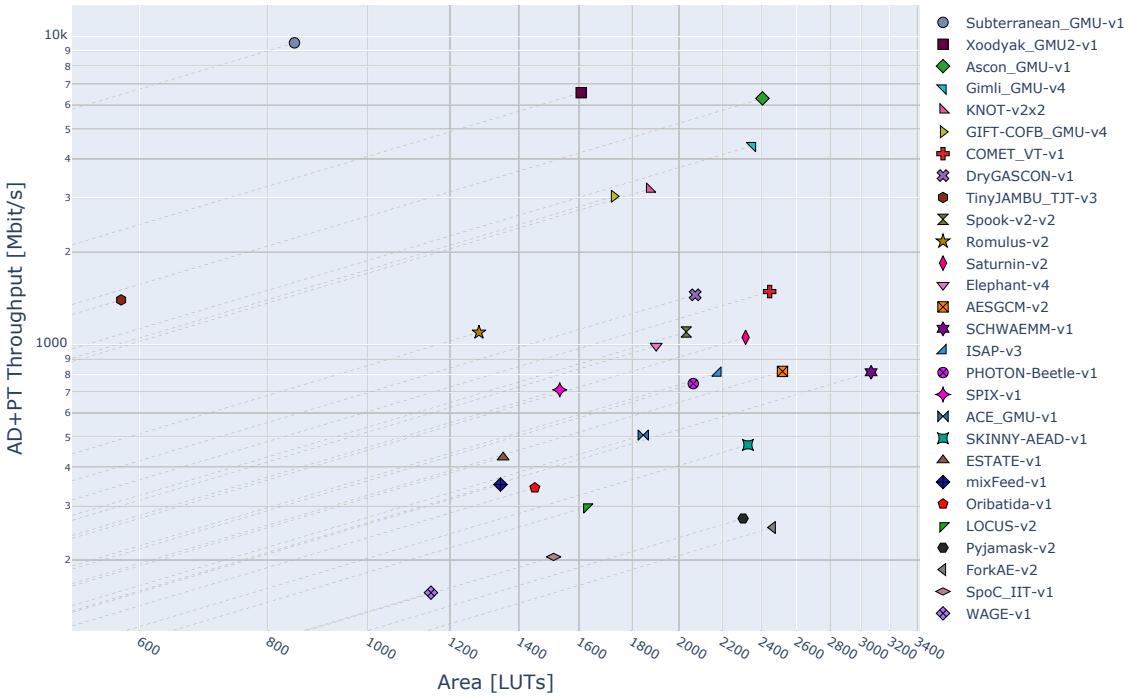


Figure 4.1: Artix-7 Encryption AD+PT Throughput for Long Messages vs LUTs

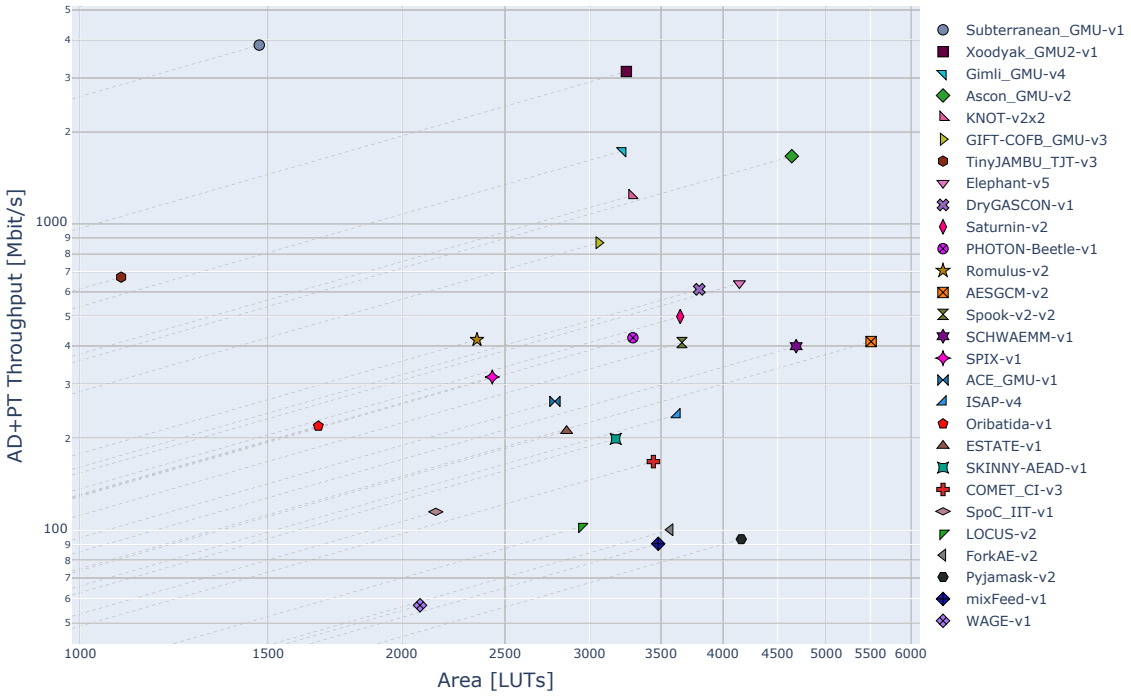


Figure 4.2: ECP5 Encryption AD+PT Throughput for Long Messages vs LUTs

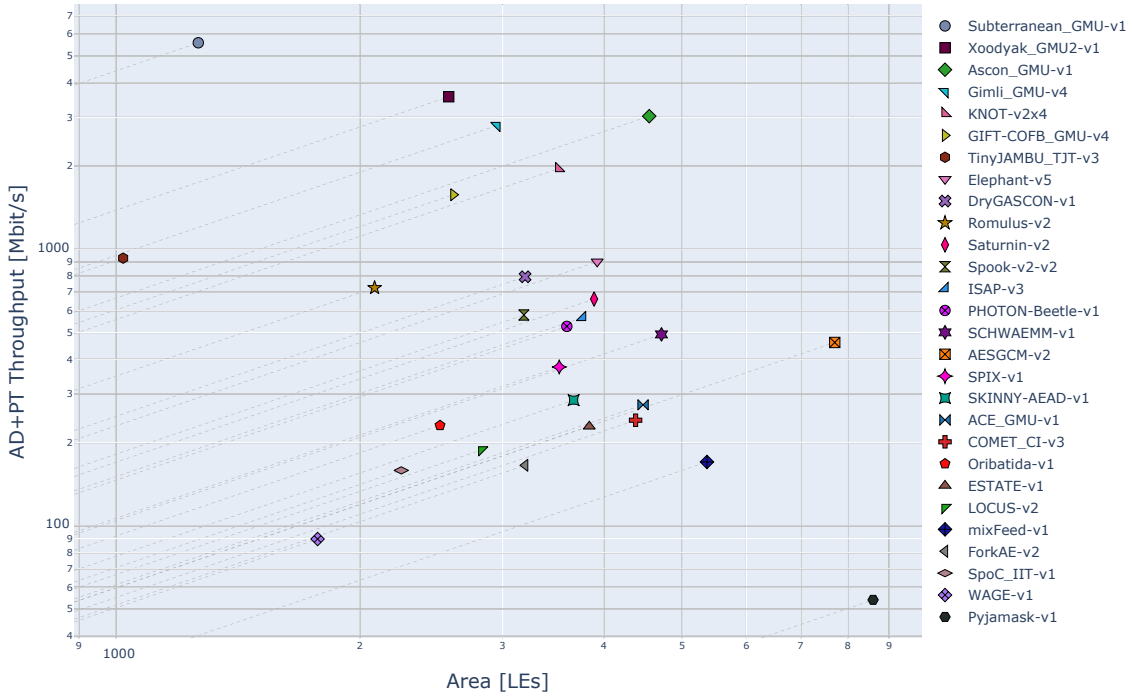


Figure 4.3: Cyclone 10 LP Encryption AD+PT Throughput for Long Messages vs LEs

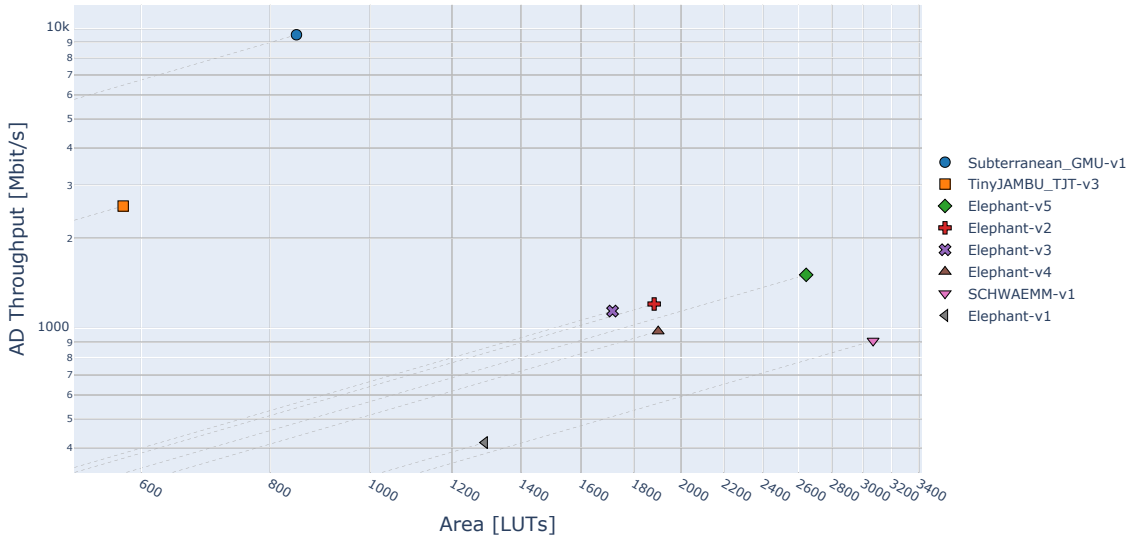


Figure 4.4: Artix-7 Elephant Variants Throughput AD vs LUTs

To understand the trade-offs that each variant provides, they need to be viewed in the same figure. Besides all of the designs of Elephant or Xoodyak, the following figures also contain the smallest candidate TinyJAMBU the fastest Subterranean, and the largest SCHWAEMM for comparison. All versions of Elephant processing AD are contained in Figure 4.4 and processing PT only in Figure 4.5. One conclusion that can be made by looking at these figures is that there is only a slight improvement in throughput when going from a factor of four times unrolled to five. Going beyond five times unrolled increases the critical path of the circuit, which decreases the maximum frequency to the point that the maximum throughput decreases. Additionally, pipelining benefits are evident in Figure 4.5 because Elephant-v4 and Elephant-v5 are significantly faster.

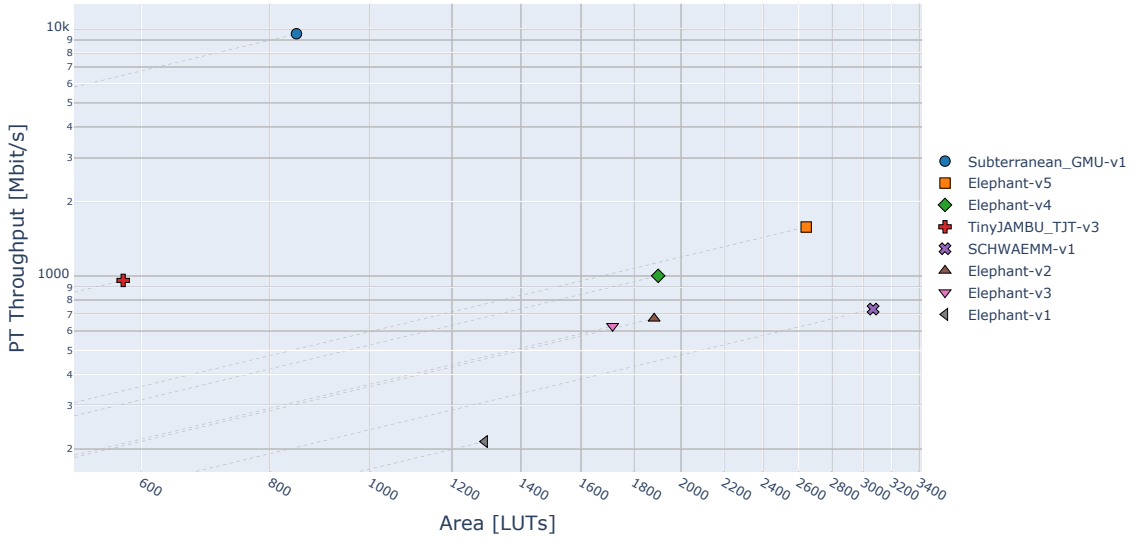


Figure 4.5: Artix-7 Elephant Variants Throughput PT vs LUTs

Figures 4.7, 4.6 and 4.8 provide a similar comparison of the Xoodyak variants for AD, PT and hashing. The results of Xoodyak_GMU-v2 show if a reduction in area is desired, there will be a significant reduction in throughput. These figures also point out that Xoodyak is very competitive when it comes to being one of the fastest candidates. One final observation is that little additional area is required to support hashing.

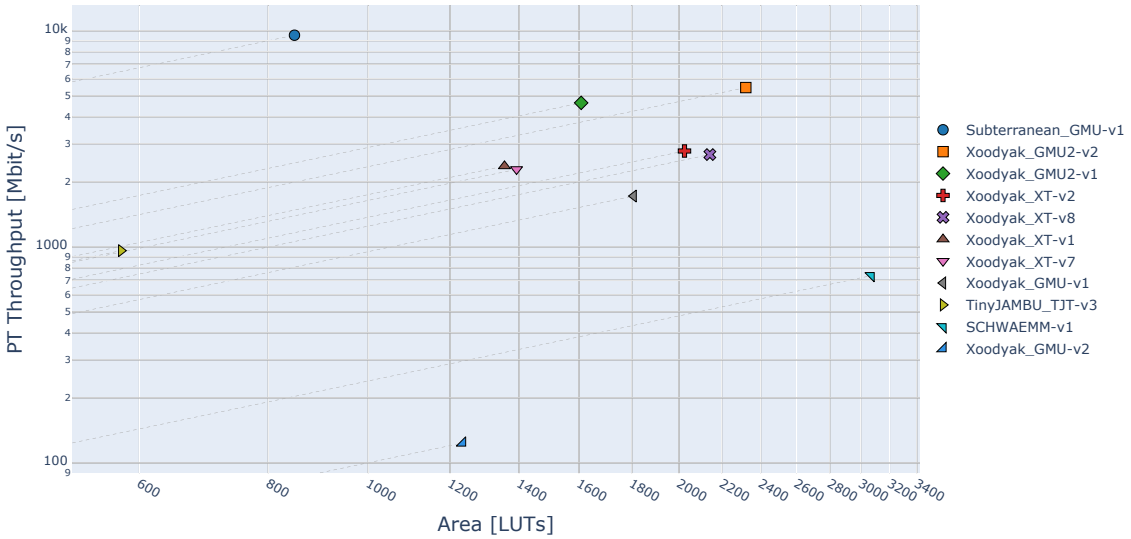


Figure 4.6: Artix-7 Xoodyak Variants Throughput PT vs LUTs

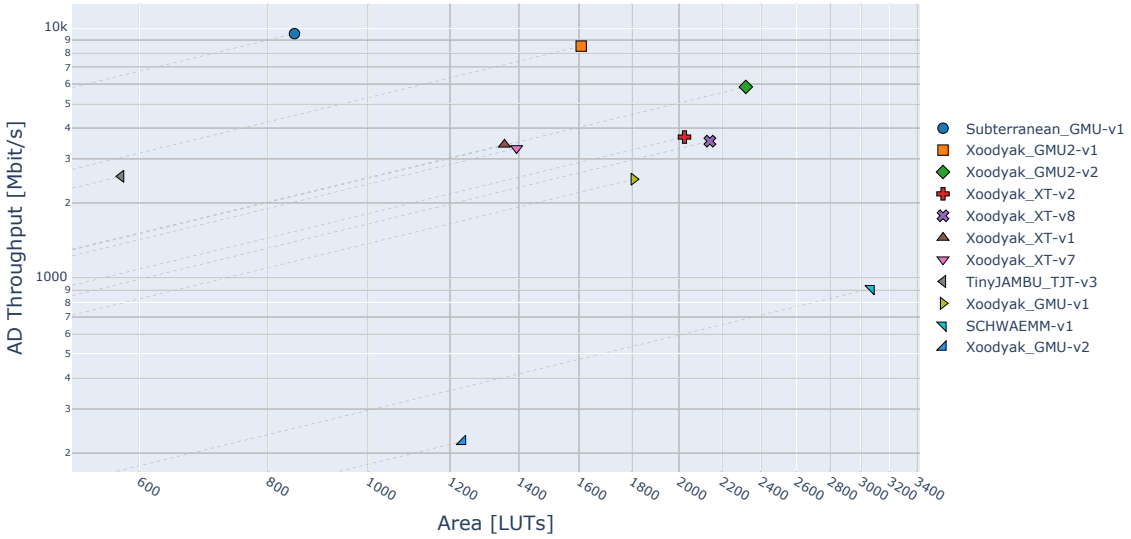


Figure 4.7: Artix-7 Xoodyak Variants Throughput AD vs LUTs

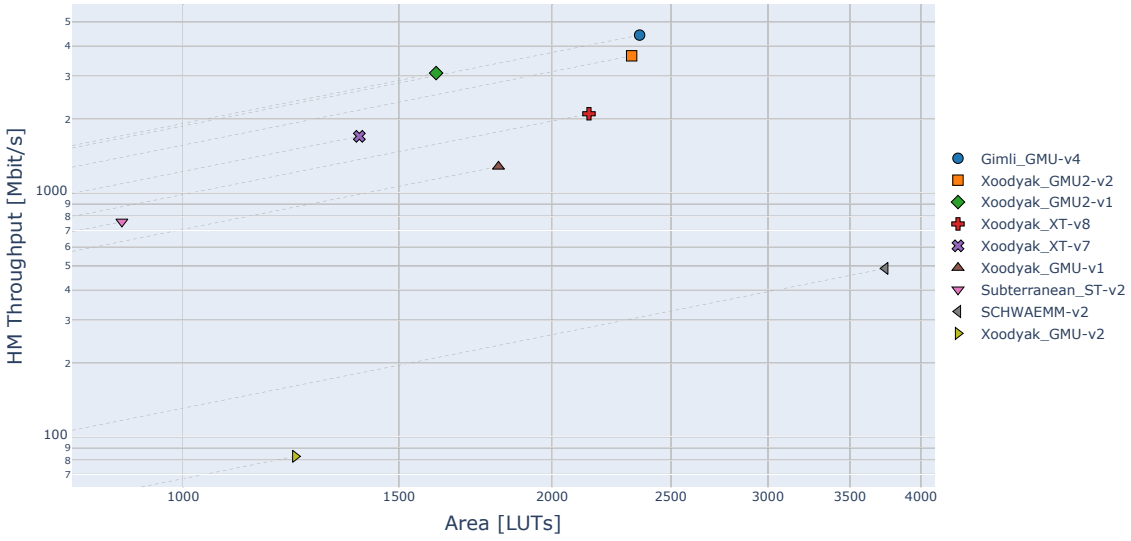


Figure 4.8: Artix-7 Xoodyak Variants Throughput Hash vs LUTs

The final topic of analysis that CERG reports in their FPGA results [32] is power and energy. Table 4.3 contains the power results for the benchmarked Elephant and Xoodyak variants. Each column's header includes a value or values to indicate the size and type of the message. For example, Enc 1536,0 represents encryption with 1536 bytes of PT and zero bytes of AD. CERG's analysis of these results points out that as the designs' unrolling factor increases, there is a superlinear increase in power. The results for Elephant are a prime example of this superlinear increase in power. Glitches in the circuit significantly increase power consumption when unrolling from a factor of four to five, causing a nearly two-fold power increase between Elephant-v3 and Elephant-v2. These results also show that adding the second permutation block doubles power consumption as expected. Elephant-v1 was one of the best candidates for power consumption at approximately 75mW. For comparison, the lowest power candidate was TinyJAMBU_TJT-v1 at 64mW.

The most surprising result from the data in Table 4.3 is the ten fold increase from Xoodyak_GMU2-v1 to Xoodyak_GMU2-v2 while only going from an unrolling factor of one to two. Interestingly, in the Xoodyak_XT designs for the same unrolling factor, power only

increased by a factor of less than five. Xoodyak_GMU-v2 is the best performer in terms of power but not by a significant factor. The b asic iterative design of Xoodyak_GMU-v1 fell in between the other two basic iterative versions because it does not have the power usage from the additional FFs that the SIPO and PISO add to Xoodyak_GMU2-v1.

Table 4.3: Estimated Total Power (mW) at 75MHz for Encryption, Decryption, and Hashing on Xilinx Artix-7

Submission	**Enc 1536,0**	**Enc 0,1536**	**Dec 1536,0**	**Enc 16,0**	**Enc 0,16**	**Dec 16,0**	**Hash 1536**	**Hash 16**
Elephant-v1	76	75	76	76	75	76		
Elephant-v2	461	421	461	427	423	416		
Elephant-v3	257	236	257	242	237	237		
Elephant-v4	118	113	117	112	112	108		
Elephant-v5	392	379	388	349	352	302		
Xoodyak_GMU-v1	169	148	168	170	168	165	169	162
Xoodyak_GMU-v2	117	115	117	116	116	116	115	114
Xoodyak_GMU2-v1	172	164	172	160	158	152	158	160
Xoodyak_GMU2-v2	1,011	669	1,001	795	792	722	990	919
Xoodyak_XT-v1	130	115	131	127	126	123		
Xoodyak_XT-v2	572	455	569	615	625	580		
Xoodyak_XT-v7	130	116	128	128	127	124	127	124
Xoodyak_XT-v8	583	464	584	625	635	592	641	549
Xoodyak_XT-v9	4,784	3,732	4,797	5,341	5,468	4,966	5,534	4,463

Figure 4.9 contains a comparison between power verses throughput for messages of size 1536 bytes. Elephant-v1 performs very competitively against the other designs for these metrics. As stated earlier, it has one of the lowest power values and provides one of the largest throughputs compared to designs on the same order of magnitude of power.

All Elephant variants are shown in Figure 4.10 where power usage to throughput are compared. Using the results from this figure indicates that the Elephant-v4 variant would compare favorably to the other designs shown in Figure 4.9 as well. For less than a three fold increase in power, Elephant-v4 would provide five times the throughput for processing PT-CT data. Although not benchmarked, a basic iterative version of Elephant with an unrolling factor of two would also be very competitive.

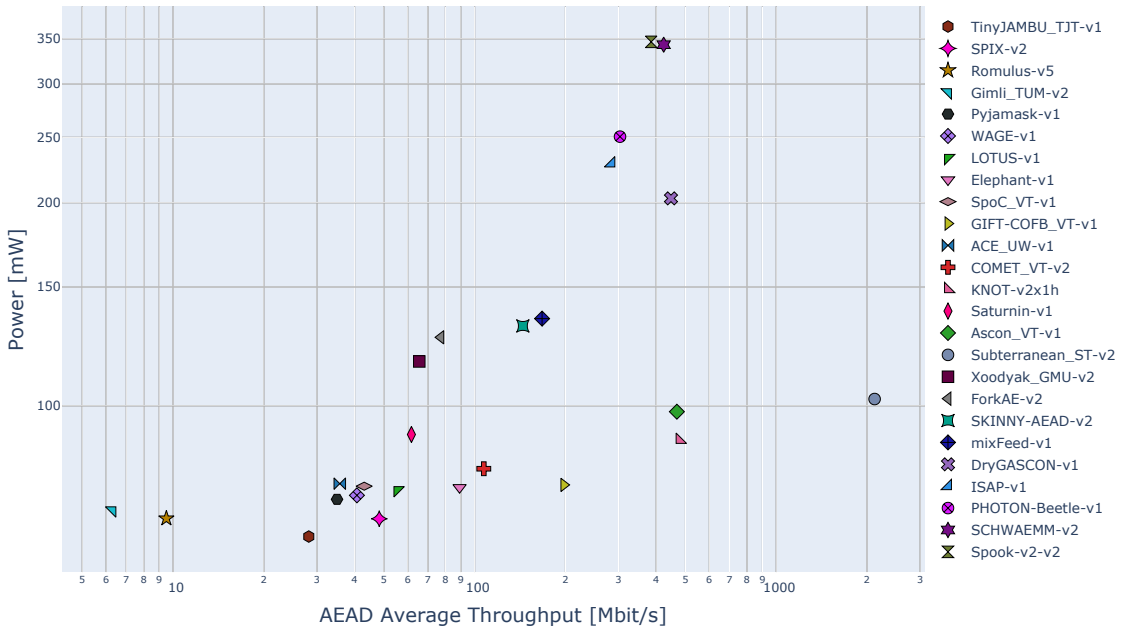


Figure 4.9: Average Power vs Average Throughput for AEAD of 1536-Byte at 75MHz

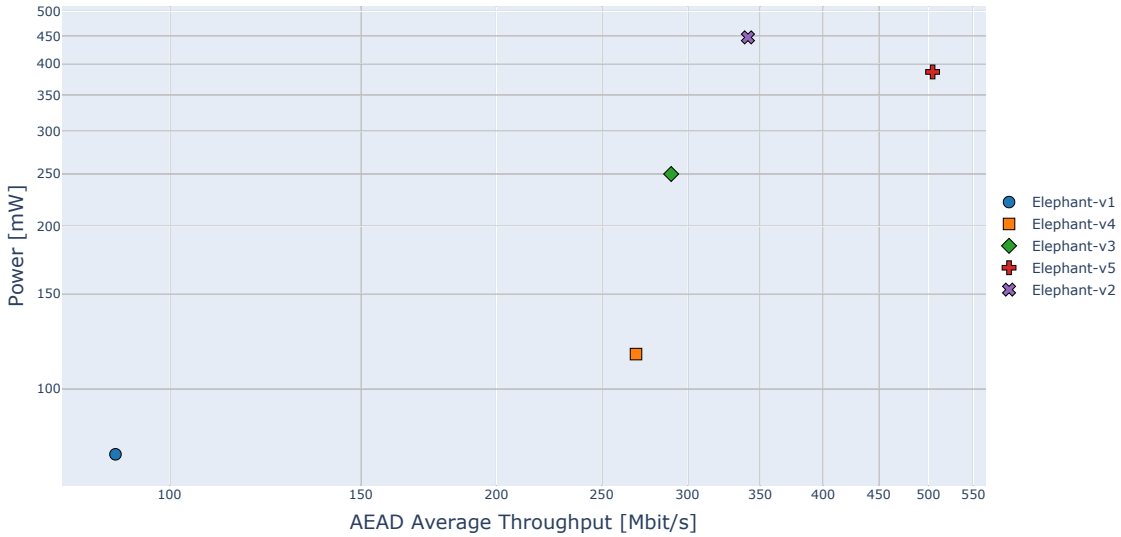


Figure 4.10: Elephant Variants Average Power vs Throughput of 1536 Bytes

CERG's energy results are given in Table 4.4 for the Elephant and Xoodyak variants. Reported results are in pico-joules per bit, for message sizes of 16 bytes and 1536 bytes, for the different message types. Elephant-v1 and Elephant-v3, which are one and four times unrolled respectively, reported nearly identical energy consumption per bit. Elephant-v2, on the other hand, had a large increase in energy consumption per bit due to glitches. Elephant versions v1-v3 required half the energy to process AD compared to PT-CT for long messages since pipelining provides no benefits. Interestingly when processing the short AD messages, more energy is required. This result is expected because when processing AD, the block processing is performed by the following equation $A_1...A_{l_A} \xleftarrow{n} N_{pub}||AD||1$. The addition of N_{pub} causes this small message size to produce two AD blocks. Since the processing of PT or CT also requires two times through the permutation, the power levels are relatively similar. This same feature was present in the Elephant-v4 and Elephant-v5 variants as well. As expected, the pipelined variants used the same amount of energy to process both the AD and PT-CT blocks for large message sizes.

Analyzing the energy results for the Xoodyak variants brings out a strong use cause for Xoodyak_GMU-v1. During the power tests, a new key is sent with every message. Therefore, designs are penalized for storing the key. If the test vectors used in the analysis allowed key reuse, Xoodyak_GMU-v1 could leverage the fact that the key was stored and not perform an additional permutation. Therefore, Xoodyak_GMU-v1 would likely be more efficient at processing short messages compared to the other Xoodyak candidates.

Table 4.4: Estimated Energy-per-bit (pJ/bit) at 75MHz for Encryption, Decryption, and Hashing on Xilinx Artix-7

Variant	**Enc 1536,0**	**Enc 0,1536**	**Dec 1536,0**	**Enc 16,0**	**Enc 0,16**	**Dec 16,0**	**Hash 1536**	**Hash 16**
Elephant-v1	1,101	580	1,101	2,758	3,409	2,781		
Elephant-v2	1,682	887	1,683	4,110	5,129	4,127		
Elephant-v3	1,111	580	1,112	2,733	3,367	2,746		
Elephant-v4	432	428	434	2,093	2,595	1,702		
Elephant-v5	754	770	775	3,614	4,488	2,871		
Xoodyak_GMU-v1	233	144	232	1,247	1,232	3,840	304	664
Xoodyak_GMU-v2	2,221	1,260	2,221	12,656	12,656	12,691	3,137	6,357

Table 4.4: Continued From Previous Page

Variant	Enc 1536,0	Enc 0,1536	Dec 1536,0	Enc 16,0	Enc 0,16	Dec 16,0	Hash 1536	Hash 16
Xoodyak_GMU2-v1	165	91	166	1,092	1,079	1,113	219	658
Xoodyak_GMU2-v2	534	330	534	3,689	3,675	3,708	740	2,154
Xoodyak_XT-v1	179	111	181	930	910	938		
Xoodyak_XT-v2	539	328	538	2,966	2,949	2,972		
Xoodyak_XT-v7	179	112	177	937	917	946	228	507
Xoodyak_XT-v8	549	334	552	3,014	2,996	3,034	746	1,558
Xoodyak_XT-v9	3,813	2,384	3,837	21,364	21,302	21,364	5,281	10,855

Figure 4.11 shows the average energy per bit for the different candidates. Elephant-v4 falls in the middle of the pack for processing long messages, but moves towards the bottom for smaller messages due to processing two blocks. Xoodyak_GMU2-v1 is the variant present in this figure and is one of the most energy-efficient candidates for processing long messages. When it comes to processing short messages, it is slightly less efficient compared to its peers. Again, I wonder what the effects of key reuse would have on these results.

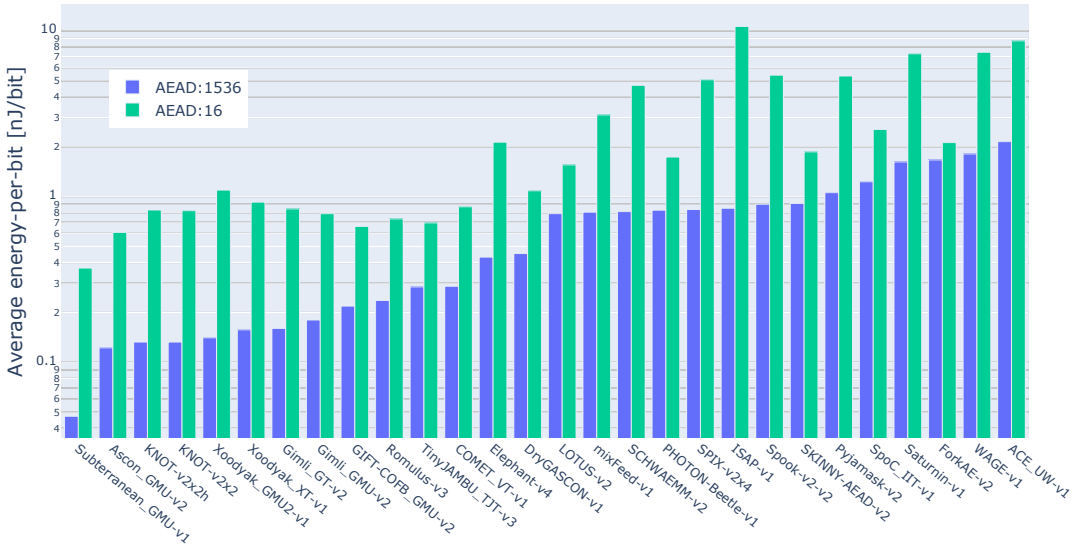


Figure 4.11: Energy-per-bit of AEAD of Long and Short Messages at 75 MHz

4.6 ASIC Benchmarking Results

Nanyang Technological University published the first ASIC benchmarking results of Round 2 Candidates [34], and a second report was published by the University of Waterloo [35]. The Nanyang report had fewer candidates and variants than the Waterloo report due to when it was published. At the time of the publishing of the Nanyang report, CERG had benchmarked few candidates as well. The Nanyang team received the following candidates for benchmarking from this work: Xoodyak_GMU-v1, Xoodyak_GMU-v2, Elephant-v1, and Elephant-v2. Each candidate was synthesized for a target throughput of 3 Mbps to accommodate typical Bluetooth data rates using the TSMC 65nm library.

Unfortunately, both of the Xoodyak_GMU submissions were not included in the report. The author's reason for not including these results was because Xoodyak_GMU-v2 was too large and slow, and Xoodyak_GMU-v1 results were too similar to the designs from the Xoodyak teams submission. The RAMs used in the Xoodyak_GMU-v2 design did not translate well into ASIC designs in a similar fashion to the results from Cyclone 10 LP. This report highlights Xoodyak's poor performance with short messages. One way to improve Xoodyak's performance with short messages is to save off and reuse the key as done in Xoodyak_GMU-v1.

The Elephant variants did not perform well compared to the other benchmarked candidates. Its poor performance was mainly due to competing against some of the best Round 2 candidates. Unfortunately, it is difficult to determine whether Elephant-v1 or Elephant-v2 was used in the results, and therefore it is hard to draw comparisons of how the new variants of Elephant might perform. This report focused primarily on processing PT data and, as already pointed out, Elephant-v4 and Elephant-v5 with the pipeline are more efficient at processing PT.

The University of Waterloo's results [35] built upon the work of the CERG team and contained a larger number of candidates and variants than the Nanyang report. Submissions were benchmarked with the following ASIC cell libraries:

- CORE65LPLVT

- tpfn65gpgv2od3 200 and tcbn65gplus 200a
- CORE90GPLVT and CORX90GPLVT
- tcbn90ghp210a
- CMRF8SF LPVT

The authors claim that they benchmarked designs at 5 MHz, 20 MHz, 50 MHz, and 100 MHz, but appear to have only reported results for 50 MHz. The authors provide graphs for each library comparing area vs. throughput and energy vs. throughput. It seems from their report that most of the designs performed similarly across the different cell libraries.

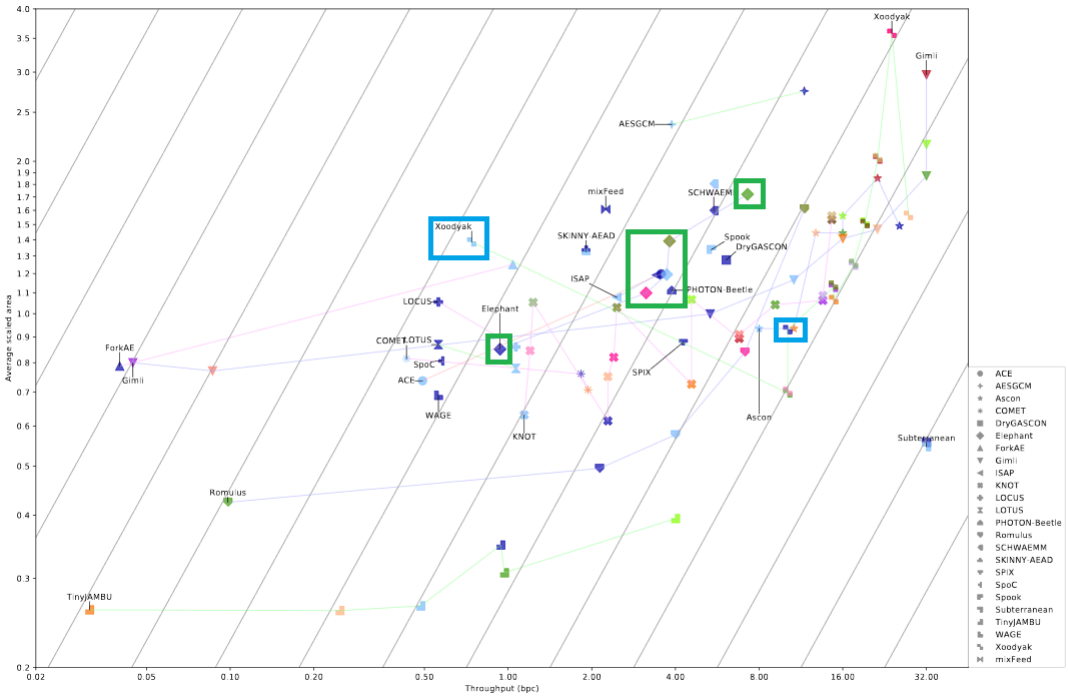


Figure 4.12: ASIC Results of Average Scaled Area vs Throughput

Figures 4.12 and 4.13 originally come from Waterloo report [35] and have been modified to highlight the candidates that were developed as part of this work. Elephant variants are represented by a diamond shape and Xoodyak designs by two small connected squares. A

unique color is used to represent each variant of a candidate. The following list contains the corresponding color for each of the variants completed in this work:

- Elephant-v1 dark blue
- Elephant-v2 light blue
- Elephant-v3 pink
- Elephant-v4 olive
- Elephant-v5 green
- Xoodyak_GMU-v1 dark blue
- Xoodyak_GMU-v2 light blue.

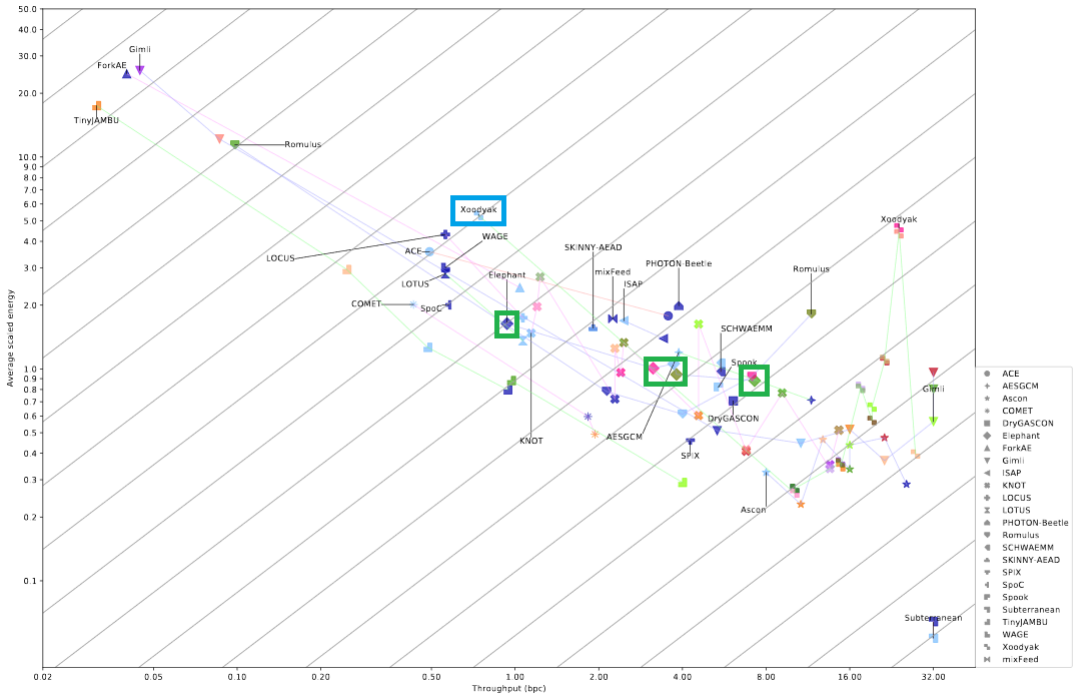


Figure 4.13: ASIC Results of Average Scaled Energy vs Throughput

When focusing on area versus throughput in Figure 4.12, the folded design of Xoodyak_GMU-v2 performs even less desirable in the ASIC benchmarking than it did in FPGA benchmarking. It was much larger and slower than the unrolled 1x variants of Xoodyak. Xoodyak_GMU-v1 was larger than the Xoodyak Team's unrolled variant but smaller than the Xoodyak_GMU2-v1 design. The reason for this placement is Xoodyak_GMU2-v1 has significantly more FFs in its design. Xoodyak_GMU-v1 was larger than the Xoodyak Team's design because of the additional FFs to store the key and the poor implementation of Cyclist. The ASIC results bring to light all the extra FFs used in the Elephant-v4 and Elephant-v5 designs which are not highlighted in the FPGA results. If the unrolling factor dominated the results, one would expect Elephant-v2, which is unrolled 5 times, to be larger than Elephant-v4, which has 2 Spongent permutation blocks each unrolled 2 times, but this is not the case.

In the energy results from Figure 4.13 Xoodyak_GMU-v2 again performs the worst of all of the Xoodyak variants. Xoodyak_GMU-v1 appears to have been left of this graph entirely or scores the same as the Xoodyak teams variants. Xoodyak_GMU-v1 should have the same throughput as the lowest energy Xoodyak Team's design, but would have a slightly larger average scaled energy value. The Elephant results show there might be some room left for improvement of the Elephant design in terms of throughput and scaled energy. Currently, the line between Elephant-v4 and Elephant-v5 is still decreasing. It is possible that in the ASIC results, a larger unrolling factor is acceptable before the effects of the glitches become too detrimental.

In Appendix-D of the University of Waterloo's report [35], each of the variants submitted is ranked based on area, energy, and area x energy and then ranked. The area rank order is determined by taking the candidates' throughput and dividing it by the area. The same goes for energy and area x energy. Each candidate's rank in all three of these categories is averaged to give a final score. In my opinion, it seems that more weight should be placed on the energy parameter than the area parameter. The cost of the area is paid for once, when the ASIC is created, while the cost of energy required is continuous. Xoodyak_GMU-v1 was the lowest ranked of the basic-iterative 1x unrolled Xoodyak variants even though it used identical energy as some of the other variants. As stated several times, storing the key

reduces the energy required if the key is reused. The Elephant designs did not appear to do that well due to the large number of Xoodyak submissions that pushed other candidates down the ranking.

4.7 Comparison of Basic Iterative Xoodyak Versions

One of the best ways to improve as an engineer is to review how others solved similar problems to see what can be learned from their approach. In this case, the additional Xoodyak designs provide the opportunity to improve. One of the most significant differences between the designs is that the Xoodyak_GMU2 is written in Bluespec SystemVerilog and uses Bluespec LWC instead of the topical VHDL LWC provided by CERG. Bluespec LWC is fully compliant with the LWC API, and therefore, it is fair to compare to the other designs [36].

All three basic iterative designs of Xoodyak (Xoodyak_GMU-v1, Xoodyak_GMU2-v1, Xoodyak_XT-v1) all perform the Xoodoo permutation in a similar manner which should lead to almost identical hardware. The Xoodyak_GMU2-v1 and Xoodyak_XT-v1 designs use for-loops to write their code, so it is more compact. One notable difference between the designs is that Xoodyak_GMU-v1 and Xoodyak_GMU2-v1 implemented the Round Constant with a look-up table while Xoodyak_XT-v1 version calculated the Round Constant, likely increasing the size of the design. The Xoodyak_XT code is available on GitHub [37].

As already stated, Xoodyak_GMU-v1 and Xoodyak_GMU-v2 both have an inefficient implementation of the Xoodyak Cyclist operations. The Xoodyak_GMU-v1's Cyclist implementation is shown in Figure 3.4. A basic version of Xoodyak_XT and Xoodyak_GMU2-v2 are shown in figures 4.15 and 4.14 respectively. One quick improvement that can be made to Xoodyak_GMU-v1 is to control the enables of the FFs to determine which portion of the state is updated instead of the muxes currently used. Another improvement would be to remove the muxes selecting a 32-bit chunk of the state. Instead, the XORs used could be duplicated for each section of the state register.

Both of the Xoodyak_GMU implementations could be improved based on techniques

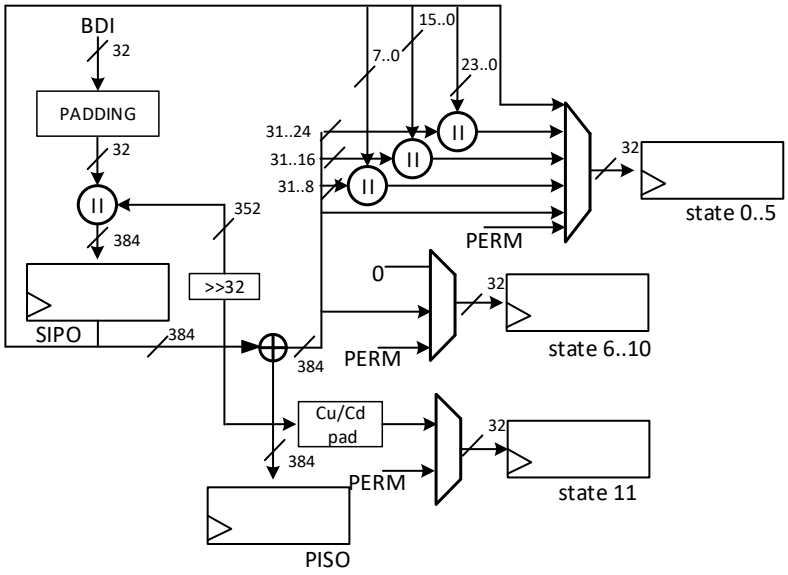


Figure 4.14: Xoodyak_GMU2 Cyclist

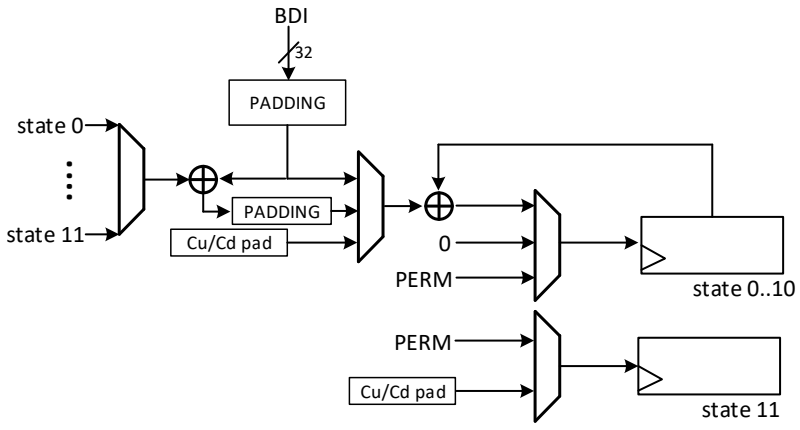


Figure 4.15: Xoodyak_XT Cyclist

used in the other designs. The Xoodyak_GMU designs should follow the methods used in the Xoodyak_XT designs. Neither of the implementations is using a SIPO, and PISO like the Xoodyak_GMU2 design is. One improvement over the Xoodyak_XT implementation that could be added is the finer gain control of the different FFs that the Xoodyak_GMU2 design uses. In the Xoodyak_XT implementation, padding logic required for decryption is added to state registers that should not need it.

Chapter 5: Protection of Designs

As NIST moves into the final round of the LWC standardization, the cryptographic community will begin focusing on the finalists' protected implementations. All of the protected implementations in this work use DOM to achieve first-order protection and are based on the unprotected designs covered in Chapter 3. An overview of DOM protection is covered in Chapter 2. Since the unprotected designs are compliant with the LWC Hardware API, using the extension proposed in [38] and shown in Figure 5.1 facilitated rapid development of these protected designs.

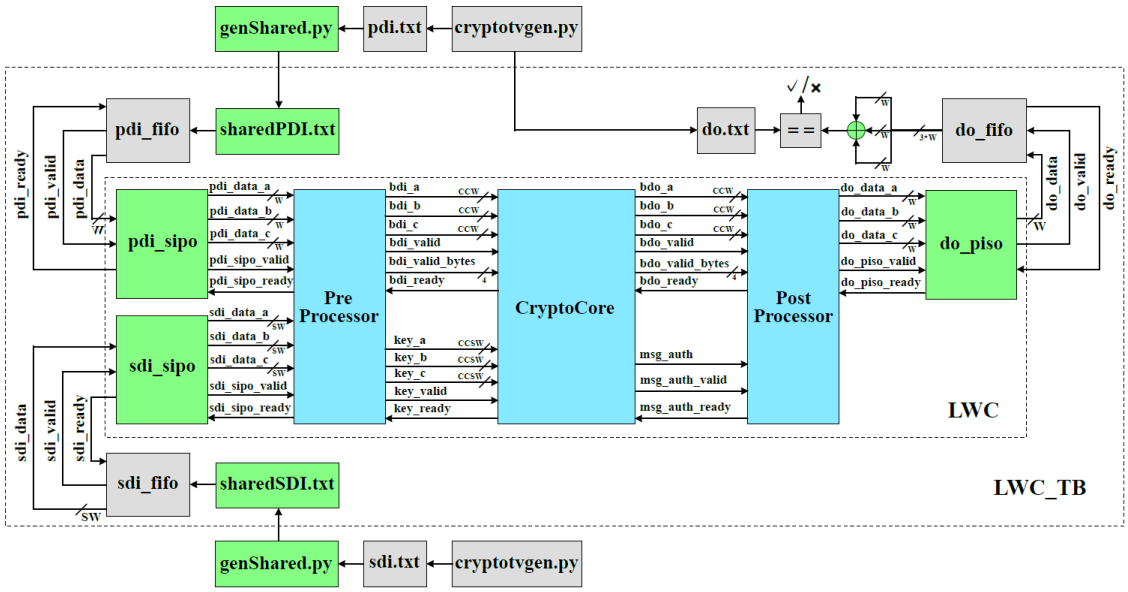


Figure 5.1: Extension Development Packet for Hardware API

To provide first-order protection with DOM only requires two shares of information to create two different domains, $Domain_0$ and $Domain_1$. The creation of the two shares

requires all linear operations from the original design to be duplicated such that each domain has the operation. Operations performed with constants must only occur in one of the domains to ensure correct results when the shares are recombined. Nonlinear operations within the circuit are the most complicated parts of the design to protect. The protection of each candidates' nonlinear operations is covered in the appropriate corresponding sections.

One common nonlinear function between the designs is the two-input AND gate. Since protected designs split data across two domains, the calculation of the AND gate is as follows:

$$= x \cdot y$$

$$= (x_0 + x_1)(y_0 + y_1)$$

$$= (x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1)$$

During these calculations, at no point in time are all shares of the same data combined. For instance, if x_0x_1 were combined, it would leak the current value of x. Figure 5.2 contains the DOM protection for the two-input AND gate. x_0y_0 and x_1y_1 do not require the XOR with the random data and have an optional register since there is no cross-domain contribution. The two gates that share information across domains must add the random data and store the data in a register to prevent leakages from occurring due to glitches.

In Figure 5.2, $r0$ represents a random bit of data. Trivium [39] Pseudo-Random Number Generators (PRNG) are used to generate the random bits. Multiple Trivium instances are required for each design to obtain the number of random bits required for each clock cycle.

5.1 Xoodyak DOM Protection

Both of the Xoodyak designs discussed in Chapter 3 have been modified and protected. The only part of Xoodyak that requires DOM protection is the χ operation. In the χ operation, the AND gates are nonlinear and are protected using the two-input AND gate previously discussed. Constant additions, such as the Round Constant and Cyclist UP

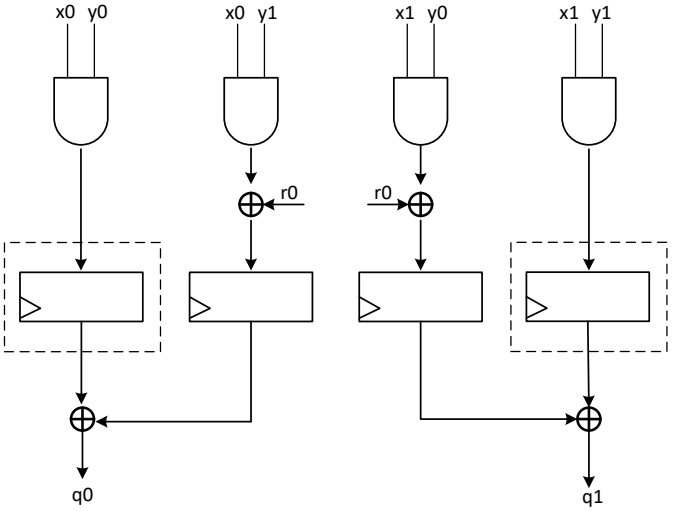


Figure 5.2: DOM Protected Two Input AND Gate

$DOWN$ operations, occur in domain0.

$$B_0 \leftarrow \overline{A_1} \cdot A_2$$

$$B_1 \leftarrow \overline{A_2} \cdot A_0$$

$$B_2 \leftarrow \overline{A_0} \cdot A_1$$

$$A_y \leftarrow A_y + B_y \quad for\ y \in \{0, 1, 2\}$$

The folded version of Xoodyak uses an ALU to perform the χ operation. Figure 5.3 contains the internals of the ALU for the protected implementation. The red block labeled χ contains the 128-bit two-input DOM protected AND gates. The AND gates require 1-bit of random data per gate, and therefore, two instances of Trivium are needed to generate the necessary 128-bits of random data. The input registers to the χ block prevent unintended inputs to the AND gates during other ALU operations.

The full-width protected datapath of the Xoodoo permutation is shown in Figure 5.4.

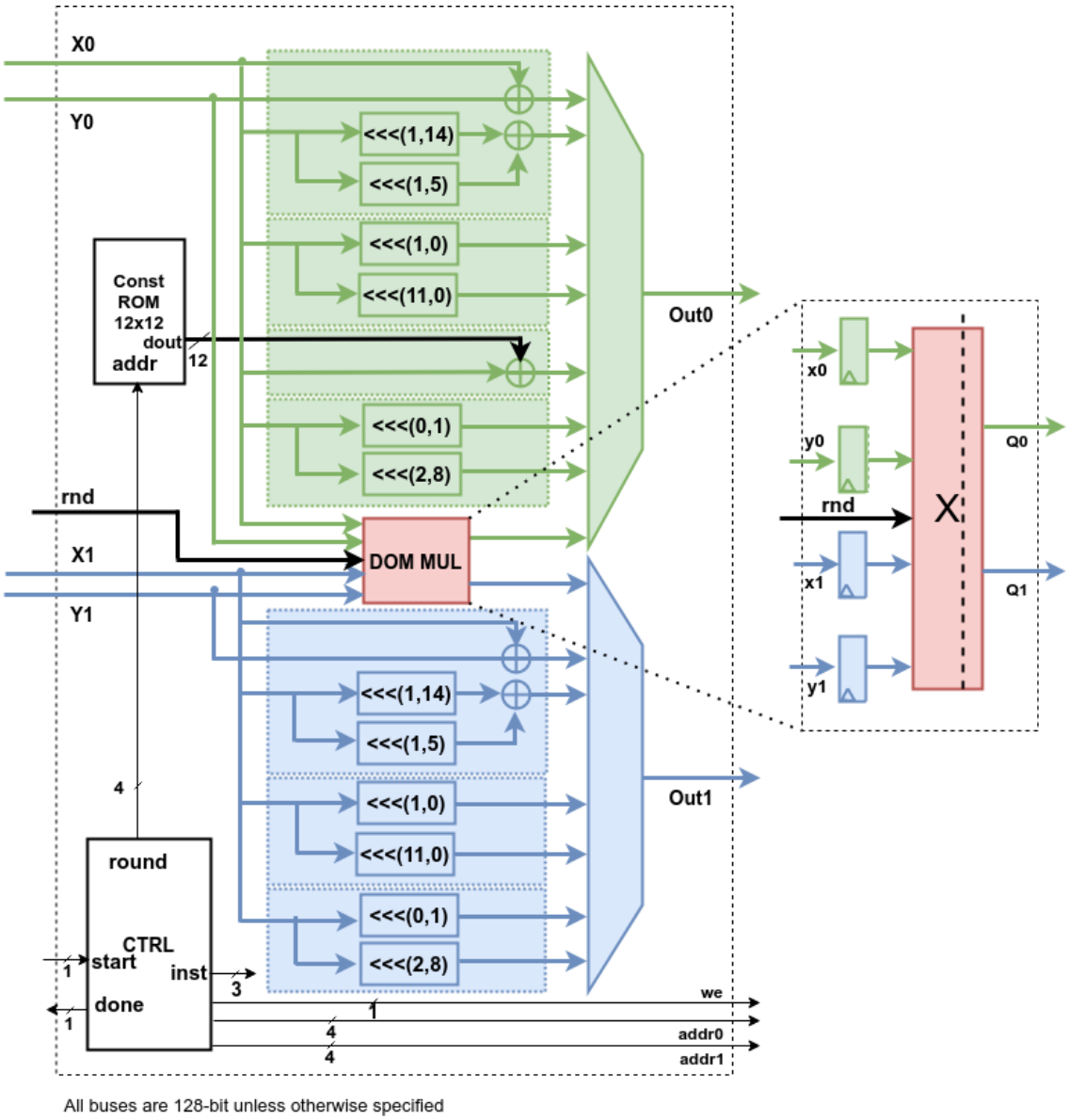


Figure 5.3: Xoodoo 128-bit Protected

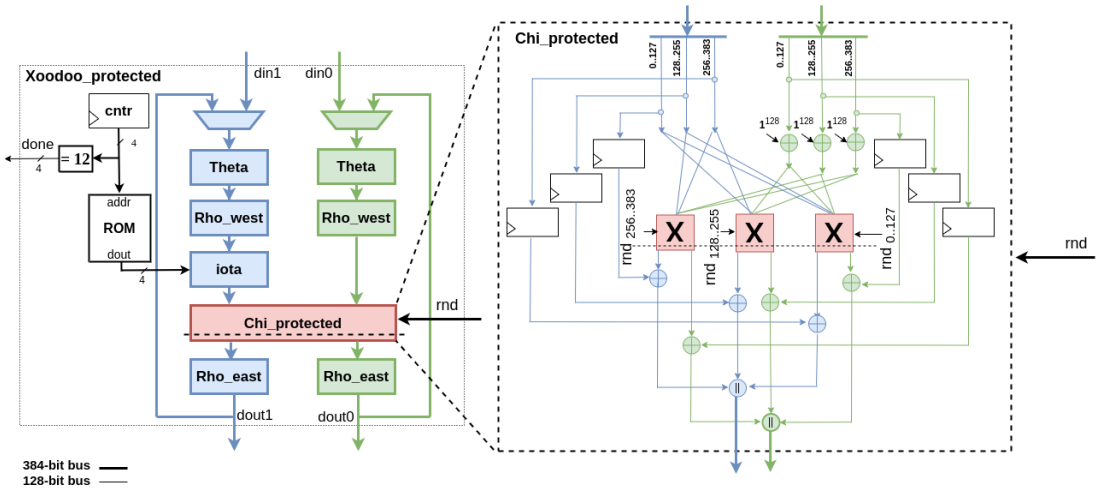


Figure 5.4: Xoodoo 384-bit Protected

This version creates 384 DOM-protected AND gates requiring 384 bits of randomness per clock cycle. Six instances of Trivium are instantiated to achieve the required number of random bits. The addition of the mux at the top of the circuit allows for a pipelined construction of the permutation. This design allows the permutation to be performed with only one more additional clock cycle than before.

5.2 Elephant DOM Protection

Only the basic iterative architecture of Elephant-v1 is protected as part of this thesis. The S-box operations within the Spongent permutation are nonlinear. Protecting the S-box using DOM requires that the S-box be reduced to its algebraic normal form (ANF). The ANF of the Elephant S-box was obtained using software developed for decomposition of

S-boxes [40, 41]. The S-box's ANF is as follows:

$$F[x, w, v, u] = [EDB0214F7A859C36]$$

$$F[0] = 0 + u + v + w \cdot v + x$$

$$F[1] = 1 + u + w \cdot v + x \cdot u + x \cdot v + x \cdot w + x \cdot w \cdot v$$

$$F[2] = 1 + v + w + x \cdot u + x \cdot w \cdot v$$

$$F[3] = 1 + v \cdot u + w + x + x \cdot u + x \cdot v + x \cdot v \cdot u + x \cdot w \cdot u$$

DOM protection of a three-input AND gate is required to implement this ANF. Figure 5.5 contains the protected implementation of the three-input AND gate, which was derived as follows:

$$= x \cdot y \cdot z$$

$$= (x_0 + x_1)(y_0 + y_1)(z_0 + z_1)$$

$$= (x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1)(z_0 + z_1)$$

$$= x_0y_0z_0 + x_0y_1z_0 + x_1y_0z_0 + x_1y_1z_0 + x_0y_0z_1 + x_0y_1z_1 + x_1y_0z_1 + x_1y_1z_1$$

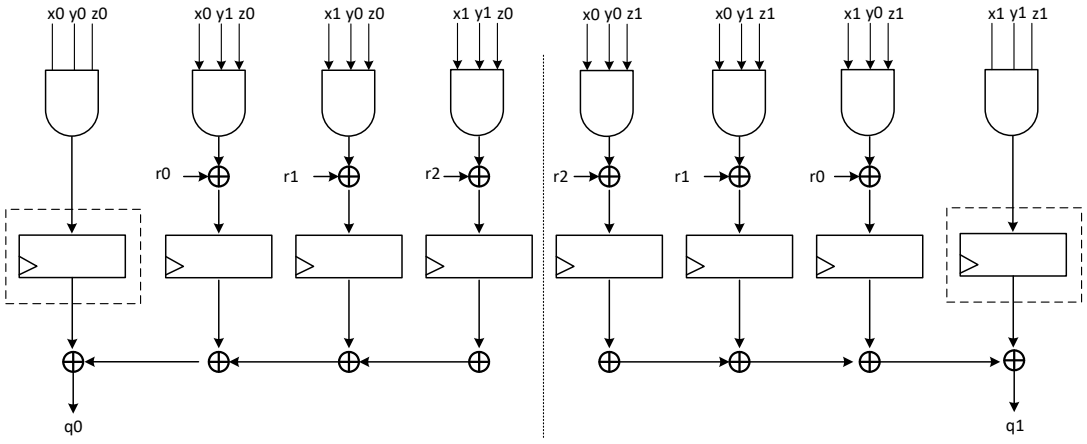


Figure 5.5: DOM Protected Three Input AND Gate

The S-box's ANF requires five instances of two-input AND gates and three cases of three-input AND gates. With these gates, 14-bits of randomness are needed per S-box. The Spongent permutation requires 40 S-boxes and therefore 560-bits of random data per cycle. To determine the most efficient way to express the same logic, a Satisfiability Modulo Theories (SMT) solver was used to minimize the number of AND gates. The resulting expression is more complex but more efficient using only three two-input AND gates and two three-input AND gates. Due to the expression's complexity, bold text is added after each instance of an AND gate. The first number in the text represents the number of input bits to the AND gate, and the second number identifies the gate's uniqueness. This expression reduces the number of bits required per S-box to nine and a total of bits to 360-bits per cycle for the Spongent permutation.

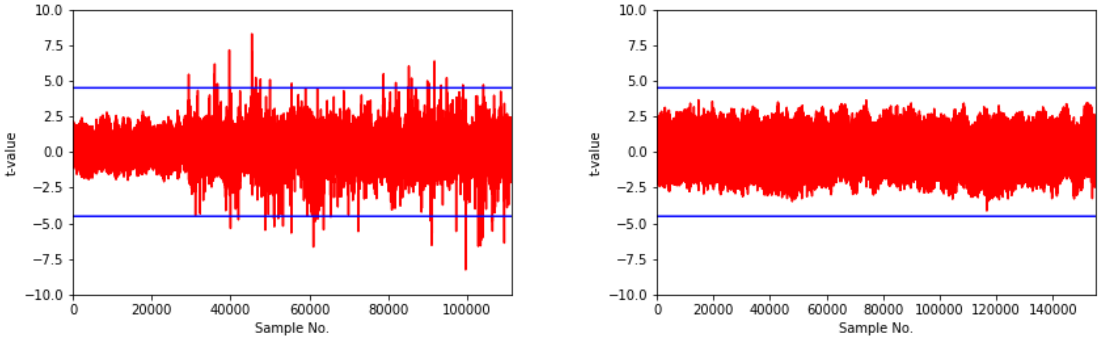
$$
\begin{aligned}
F[s3, s2, s1, s0] =& [EDB0214F7A859C36] \\
F[0] =& s0 + s3 + \\
& ((s1 + s2) \cdot s1)\mathbf{(2,1)} \\
F[1] =& not(s0 + s1 + \\
& ((s1 + s2) \cdot s1)\mathbf{(2,1)} + \\
& (s3 \cdot (s0 + s1) \cdot (s0 + s2))\mathbf{(3,1)} + \\
& ((s1 + s2) \cdot (s0 + s3) \cdot s3)\mathbf{(3,2)}) \\
F[2] =& not(s1 + s2 + \\
& ((s1 + s2) \cdot (s1 + s3))\mathbf{(2,2)} + \\
& ((s1 + s2) \cdot s1)\mathbf{(2,1)} + \\
& (s3 \cdot (s0 + s1) \cdot (s0 + s2))\mathbf{(3,1)} + \\
& ((s1 + s2) \cdot (s0 + s3) \cdot s3)\mathbf{(3,2)}) \\
F[3] =& not(s0 + s2 + s3 + \\
& ((s1 + s2) \cdot (s1 + s3))\mathbf{(2,2)} + \\
& ((s0 + s3) \cdot (s0 + s1))\mathbf{(2,3)} + \\
& ((s1 + s2) \cdot s1)\mathbf{(2,1)} + \\
& ((s1 + s2) \cdot (s0 + s3) \cdot s3)\mathbf{(3,2)});
\end{aligned}
$$

5.3 Verification and Results

To ensure the designs are not leaking, they had to pass TVLA verification. For more information about TVLA refer back to Section 2.4. TVLA results were obtained using the Flexible Opensource workBench fOr Side-channel analysis(FOBOS) platform [42]. Cryptotvgen was used to generate test vectors, which were then converted into shares using the

LWC Hardware API extension conversion script. The device under test (DUT) was loaded on a NewAE CW305 SCA board that uses an Artix-7 (xc7a100tftg256-3) FPGA. Power was measured across the boards Low-Noise Amplifier (LNA) that amplifies the voltage drop of the 0.1Ω shunt resistor. The DUT was run at 1.25 MHz to prevent the signal distortion observed at high frequencies. Traces were collected using a Picoscope 5000 that obtains 8-bit samples at a sampling rate of 125 MS/s. With this sample rate, 100 samples were collected during the one-clock cycle of the DUT.

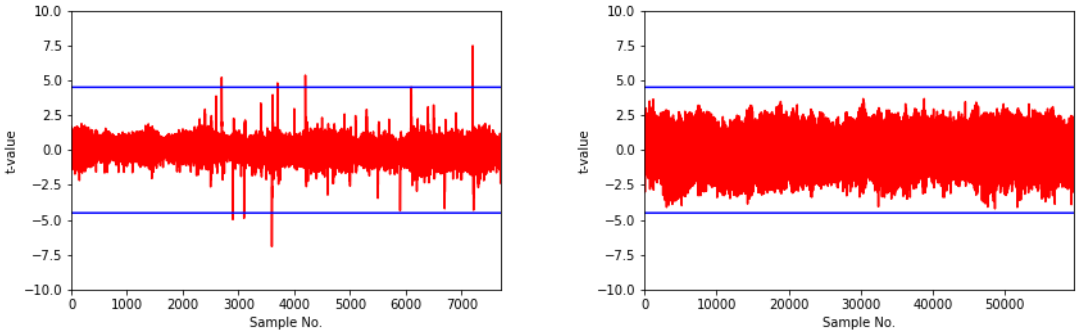
TVLA results comparing the unprotected and protected designs of Xoodyak are contained in the following figures: Figure 5.6 for Xoodyak_GMU-v2 and Figure 5.7 for Xoodyak_GMU-v1. Both of the unprotected designs exceed the 4.5 threshold with only 2,000 traces. On the other hand, the protected designs do not exceed the threshold even with 100,000 traces. These results confirm that the protected implementations improved the resistance to SCA.



(a) Unprotected 2,000 Traces (b) Protected 100,000 Traces

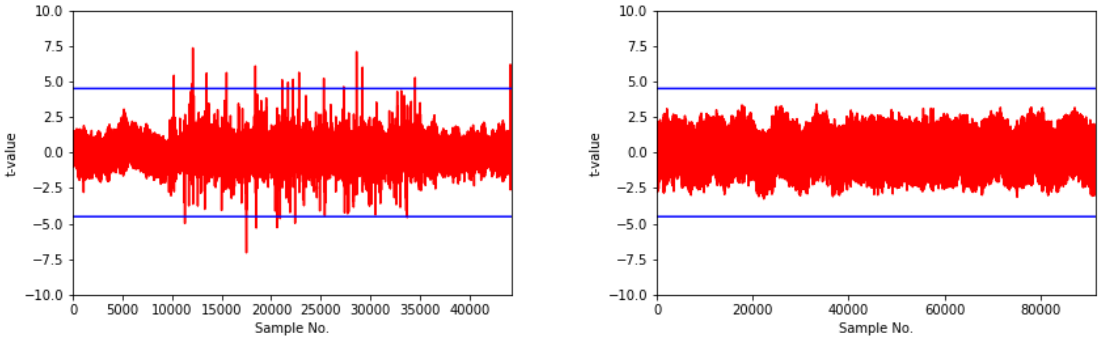
Figure 5.6: Xoodyak 128 TVLA Traces

Similar TVLA results for Elephant versions are shown in Figure 5.8. The unprotected Elephant-v1 did not pass the TVLA leakage test by exceeding 4.5 when there were 10,000 traces. The protected implementation certainly improves SCA resistance because even with 1,000,000 traces, it is not approaching the 4.5 threshold.



(a) Unprotected 2,000 Traces (b) Protected 100,000 Traces

Figure 5.7: Xoodyak 384 TVLA Traces



(a) Unprotected 10,000 Traces (b) Protected 1,000,000 Traces

Figure 5.8: Elephant TVLA Traces

Benchmarking results for area and throughput of the protected designs are contained in Table 5.1, and the results for average power are contained in Table 5.2. The protected Xoodyak implementations do not contain the area that the PRNGs adds to the circuit. This follows the same approach that the authors of [38] performed, resulting in a closer comparison. Unfortunately, the Elephant implementation is not written with this same comparison method in mind. Instead, the results for Elephant contain 6 Trivium PRNGs and, therefore, look much larger compared to the Xoodyak designs. Each instance of the Trivium PRNG takes approximately 440 LUTs and 475 FFs. Six instances of Trivium would therefore take around 2640 LUTs and 2850 FFs. Thus, the approximate size of Elephant without the PRNGs is 5451 LUTs and 2970 FFs, however, these results should be confirmed.

Table 5.1: Benchmarking Results on Xilinx Artix-7

Implementation	**Area**		**Freq.**	**TP**	**Area ratio**	**TP ratio**
	LUTs	**FFs**	**MHz**	**Mbps**		
Unprotected						
Elephant-v1	1291	910	229	214.3	1.00	1.00
Xoodyak-384	1808	851	170	1717.9	1.00	1.00
Xoodyak-128	1234	98	168	118.0	1.00	1.00
Protected						
Elephant-v1	8091	5550	200	93.5	6.21	0.43
Xoodyak-384	6431	4210	158	891.39	4.02	0.42
Xoodyak-128	3627	1753	140	81.78	2.49	0.71

The throughput of protected Elephant-v1 and Xoodyak-384 both decrease by a factor of over 1/2. Elephant reduction by this factor is because the permutation requires twice as many clock cycles to complete. Xoodyak-384's reduction is because it operates at a lower max frequency and requires an additional clock cycle to complete the permutation. The Xoodyak-128 throughput decreases by a smaller factor because the ratio of clock cycles added to the permutation compared to the original number of clock cycles to perform the permutation is smaller.

The estimated energy-per-bit of both the protected and unprotected implementations

Table 5.2: Estimated Average Power on Xilinx Artix-7 (at 75 MHz)

Implementation	Static Power (mW)	Dynamic Power (mW)	Dynamic Power Ratio	Total Power (mW)	Total Power Ratio
Unprotected					
Elephant-v1	91	76	1.00	167	1.00
Xoodyak-384	91	88	1.00	179	1.00
Xoodyak-128	91	34	1.00	125	1.00
Protected					
Elephant-v1	91	114	1.5	205	1.22
Xoodyak-384	92	217	2.46	309	1.73
Xoodyak-128	91	65	1.91	156	1.24

is reported in 5.3. The 128-bit Xoodyak implementation notices the least change in energy per bit of the protected implementations due to having the fewest components used for protection. The 384-bit implementation of Xoodyak is still efficient, requiring only 0.7nJ per bit. Due to the complexity of the Elephant protection, it is not surprising that this design requires a significant amount more energy-per-bit.

Table 5.3: Estimated Energy-per-bit on Xilinx Artix-7 (at 75 MHz)

Implementation	Throughput (Mbps)	Energy-per-bit (nj/bit)	Energy-per-bit Ratio
Unprotected			
Elephant-v1	70.2	2.3	1.00
Xoodyak-384	757.9	0.4	1.00
Xoodyak-128	55.2	2.8	1.00
Protected			
Elephant-v1	35.1	5.8	2.52
Xoodyak-384	423.2	0.7	1.75
Xoodyak-128	43.8	3.6	1.28

Based on the results, it appears that DOM protection method appears well suited for the Xoodyak permutation. Looking at Elephant results, it appears that this algorithm

should also be protected using TI to determine if that protection method is better suited. For more discussion on this topic see Chapter 7.

Chapter 6: Conclusions

Indeed, the benchmarking efforts of CERG helped NIST with their selections of LWC finalists, which were announced on March 29, 2021, to include: ASCON, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, Photon-Beetle, Romulus, Sparkle, TinyJambu, and Xoodyak. Of note, both of the candidates focused on in this thesis advanced to the final round. The Elephant variants from this work were the only known implementations of Elephant at the time of NIST's selection. In my opinion, the addition of the pipelined implementation significantly helped Elephant be a viable finalist candidate by substantially improving the throughput.

Now that the candidate pool is reduced, Elephant and Xoodyak will likely receive greater attention within the cryptographic community leading to new or improved designs. Chapter 7 contains some possible improvements that could be made to the current designs. No hardware submissions of Grain128-AEAD were benchmarked; therefore, it would also be interesting to see how it compares against the other candidates.

In the final round, having protected implementations of the candidates will be desirable by NIST. Having verified the protected implementation of these two candidates is another great success of this thesis. Improvements to these protected candidates and designs of the other candidates certainly should be attempted in the final round. CERG will undoubtedly want to perform even more intensive benchmarking of both the unprotected and protected implementations before the completion of the final round to assist NIST.

Chapter 7: Future Work

Since both Elephant and Xoodyak are finalists in NIST's LWC process, new or improved implementations of these candidates are desirable. Based on the number of full datapath submissions of Xoodyak for Round 2 benchmarking, it is unlikely that significant improvements can be made to the full datapath designs. One additional analysis feature that would be interesting to investigate in the final round is how key reuse affects the power and energy results. Currently, designs that support key reuse have a larger size and require more energy without the benefits of key reuse being measured.

Even though the 128-bit design of Xoodyak is inefficient compared to the full datapath version of Xoodyak, it may be worth revisiting. As stated in section 4.7, the Cyclist portion in the designs developed from this work are inefficient and could be improved. Additionally, the STATE RAM was not portable to ASICs. A new implementation could use FFs for the state storage to reduce the design's size. Using FFs also allows the designer to be more flexible with the number of storage elements available, allowing for a smaller or faster design.

Since Elephant advanced to Round 3, there may be interest in seeing the Delirium variant of Elephant because it uses Keccak for the permutations. Another appealing modification would be to see how efficient Elephant can be if allowed to process AD and PT/CT simultaneously. Even though this would be a deviation from the LWC Hardware API, it would highlight a useful feature not available in other designs. Another slight modification that may make the existing Elephant designs more competitive would be to add a PISO to the original basic iterative architecture.

Although this work contains protected implementations of Xoodyak already, protection of the other Xoodyak designs submitted for Round 2 benchmarking would be attractive. Since there were no significant differences in the permutation between designs, the same

protected permutation could be used. DOM protection of Elephant resulted in a larger implementation than desired. In my opinion, another version of Elephant should also be protected using TI to see if this protection method provides more favorable results. According to the S-box's classification from [41], it should be able to be protected using only three shares and no additional clock cycles.

Bibliography

Bibliography

[1] K. McKay, L. Bassham, M. Turan, and N. Mouha, "Report on lightweight cryptography." [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf

[2] J.-P. Kaps, P. Yalla, K. K. Surapathi, S. Vadlamudi, S. Gurung, and B. Habib, "Lightweight Implementations of SHA-3 Finalists on FPGAs," 2012, p. 17.

[3] M. Tempelmeier, G. Sigl, and J.-P. Kaps, "Experimental Power and Performance Evaluation of CAESAR Hardware Finalists," in *2018 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2018*, 2018, pp. 1–6.

[4] F. Farahmand, W. Diehl, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "Improved Lightweight Implementations of CAESAR Authenticated Ciphers," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018*, 2018, pp. 29–36. [Online]. Available: https://ieeexplore.ieee.org/document/8457629/

[5] J.-P. Kaps, W. Diehl, M. Tempelmeier, F. Farahmand, E. Homsirikamol, and K. Gaj, "A Comprehensive Framework for Fair and Efficient Benchmarking of Hardware Implementations." [Online]. Available: https://eprint.iacr.org/2019/1273

[6] K. Mohajerani, R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "FPGA benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process: Methodology, metrics, tools, and results," p. 93, 2020.

[7] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, P. Yalla, J.-P. Kaps, and K. Gaj, "CAESAR Hardware API," Cryptology ePrint Archive 2016/626, 2016.

[8] D. J. Bernstein. Crypto competitions: CAESAR submissions. [Online]. Available: https://competitions.cr.yp.to/caesar-submissions.html

[9] E. Homsirikamol, P. Yalla, F. Farahmand, W. Diehl, A. Ferozpuri, J.-P. Kaps, and K. Gaj, "Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API," GMU, Fairfax, VA, GMU Report, 2016.

[10] E. Homsirikamol, P. Yalla, and F. Farahmand, "Development Package for Hardware Implementations Compliant with the CAESAR Hardware API," https://cryptography.gmu.edu/athena/index.php?id=CAESAR, 2016.

[11] "Submission requirements and evaluation criteria for the lightweight cryptography standardization process." [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf

[12] I. T. L. Computer Security Division. Lightweight cryptography | CSRC | CSRC. [Online]. Available: https://csrc.nist.gov/Projects/lightweight-cryptography

[13] S. Renner, E. Pozzobon, and J. Mottok, "Benchmarking Software Implementations of 1st Round Candidates of the NIST LWC Project on Microcontrollers," p. 27, 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2019/documents/papers/benchmarking-software-implementations-lwc2019.pdf

[14] T. Beyne, Y. L. Chen, C. Dobraunig, and B. Mennink, "Elephant v1.1," p. 48, 2019.

[15] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, "Xoodyak, a lightweight cryptographic scheme," p. 24, 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Xoodyak-spec-round2.pdf

[16] H. Gross, "Domain-oriented masking generically masked hardware implementations." [Online]. Available: https://pure.tugraz.at/ws/portalfiles/portal/19504496/thesis_main.pdf

[17] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold Implementations Against Side-Channel Attacks and Glitches," in *Information and Communications Security, ICICS 2006*, ser. LNCS, vol. 4307. Springer Berlin Heidelberg, 2006, pp. 529–545.

[18] H. Gross, S. Mangard, and T. Korak, "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order," Tech. Rep. 486, 2016.

[19] H. Gross, S. Mangard, and Korak, "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order," in *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security - TIS'16*. Vienna, Austria: ACM Press, 2016, pp. 3–3.

[20] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side-channel resistance validation," p. 15, 2011.

[21] T. Schneider, A. Moradi, T. Schneider, and A. Moradi, "Leakage Assessment Methodology: A Clear Roadmap for Side-Channel Evaluations," in *Cryptographic Hardware and Embedded Systems – CHES 2015*. Springer Berlin Heidelberg, 2015, pp. 495–513. [Online]. Available: http://link.springer.cm/10.1007/978-3-662-48324-4_25

[22] K. Gaj, J.-P. Kaps, and W. Diehl, "Why_does_hardware_api_matter," https://cryptography.gmu.edu/athena/LWC/Why_Does_Hardware_API_Matter.pdf. [Online]. Available: https://cryptography.gmu.edu/athena/LWC/Why_Does_Hardware_API_Matter.pdf

[23] Cryptographic Engineering Research Group (CERG) at George Mason University, "Hardware Benchmarking of Lightweight Cryptography," https://cryptography.gmu.edu/athena/index.php?id=LWC, 2020.

[24] J.-P. Kaps, W. Diehl, M. Tempelmeier, E. Homsirikamol, and K. Gaj, "Hardware API for Lightweight Cryptography," GMU, Fairfax, VA, GMU Report, Oct. 2019.

[25] Cryptographic Engineering Research Group (CERG) at George Mason University, "Hardware Benchmarking of CAESAR Candidates," https://cryptography.gmu.edu/athena/index.php?id=CAESAR, 2019.

[26] P. Yalla and J.-P. Kaps, "Evaluation of the CAESAR hardware API for lightweight implementations," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*, Cancun, Mexico, Dec. 2017.

[27] Cryptographic Engineering Research Group (CERG) at George Mason University, "LWC hardware API developement package," original-date: 2019-10-14T18:26:29Z. [Online]. Available: https://github.com/GMUCERG/LWC

[28] E. Homsirikamol, P. Yalla, and F. Farahmand, "Implementer's guide to hardware implementations compliant with the CAESAR hardware API." [Online]. Available: https://cryptography.gmu.edu/athena/LWC/LWC_HW_Implementers_Guide.pdf

[29] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs," in *2010 International Conference on Field Programmable Logic and Applications, FPL 2010*. Milan, Italy: IEEE, Aug. 2010, pp. 414–421.

[30] F. Farahmand, W. Diehl, and K. Gaj, "Minerva: Automated hardware optimization tool," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–8.

[31] K. Mohajerani and R. Nagpal, "Xeda." [Online]. Available: https://github.com/kammoh/xeda

[32] K. Mohajerani, R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "Fpga benchmarking of round 2 candidates in the nist lightweight cryptography standardization process: Methodology, metrics, tools, and results," Cryptology ePrint Archive, Report 2020/1207, 2020, https://eprint.iacr.org/2020/1207.

[33] K. Gaj, "Hardware benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process," DATE. [Online]. Available: https://www.date-conference.com/programme/session/2.3

[34] M. Khairallah, T. Peyrin, and A. Chattopadhyay, "Preliminary hardware benchmarking of a group of round 2 NIST lightweight AEAD candidates." [Online]. Available: https://github.com/mustafam001/lwc-aead-rtl

[35] M. D. Aagaard and N. Zidaricˇ, "ASIC benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process," p. 49, 2021. [Online]. Available: https://eprint.iacr.org/2021/049.pdf

[36] K. Mohajerani, "BlueLight and CocoLight," original-date: 2020-12-03T06:15:39Z. [Online]. Available: https://github.com/kammoh/bluelight

[37] S. Mella, "Xoodoo," original-date: 2017-11-13T12:43:42Z. [Online]. Available: https://github.com/KeccakTeam/Xoodoo

[38] F. Coleman, B. Rezvani, S. Sachin, and W. Diehl, "Side Channel Resistance at a Cost: A Comparison of ARX-based Authenticated Encryption," in *Field-Programmable Logic and Applications (FPL)*, Aug. 2020, p. 7.

[39] C. De Canni'ere and B. Preneel, "TRIVIUM Specifications," Katholieke Universiteit Leuven, Tech. Rep., 2005.

[40] B. Bilgin, S. Nikova, V. Nikov, and V. Rijmen, "TI toolkit." [Online]. Available: http://homes.esat.kuleuven.be/~snikova/ti_tools.html

[41] B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz, "Threshold implementations of all 3 $\times$3 and 4 $\times$4 s-boxes," series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-33027-8_5

[42] A. Abdulgadir, W. Diehl, and J.-P. Kaps, "An Open-Source Platform for Evaluation of Hardware Implementations of Lightweight Authenticated Ciphers," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun, Mexico: IEEE, Dec. 2019, pp. 1–5.

Curriculum Vitae

Richard Haeussler received his Bachelors of Science in Electrical Engineering from George Mason University in 2015 and has worked as a government contractor since. During his master's studies at George Mason University, Richard worked on the software implementation of the Post-quantum cryptographic algorithm DAGs and joined the Cryptographic Engineering Research Group (CERG). Richard's research for this thesis was done in collaboration with the other members of CERG as part of CERG's efforts to benchmark NIST LWC Round 2 candidates.