The Thesis Committee for Gokhan Sayilar
Certifies that this is the approved version of the following thesis:

Cryptoraptor: High Throughput Reconfigurable Cryptographic Processor for Symmetric Key Encryption and Cryptographic Hash Functions

APPROVED BY

SUPERVISING COMMITTEE:

Derek Chiou, Supervisor

Mohit Tiwari

Cryptoraptor: High Throughput Reconfigurable Cryptographic Processor for Symmetric Key Encryption and Cryptographic Hash Functions

by

Gokhan Sayilar, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2014

To my family and many friends...

Acknowledgments

A major research project like this is never the work of anyone alone. I would like to extend my appreciation especially to the following.

First and foremost I offer my sincerest gratitude to my supervisor, Dr. Derek Chiou, for his excellent guidance, caring, and patience. I would also like to thank him for being an open person to ideas, encouraging and helping me to shape my interest and ideas, and giving me the freedom to work in my own way. He's the funniest advisor and one of the smartest people I know.

Besides my advisor, I would like to thank to my second reader, Dr. Mohit Tiwari, for his advises and insightful comments.

I am also thankful to my friends in US, Turkey, and other parts of the World for being sources of laughter, joy, and support.

Last but not least, I would like to thank my parents and my brother for their continuous love and unconditional support in any decision that I make.

I also want to thank to Semiconductor Research Corporation and Freescale Semiconductor, Inc for their financial support which allowed me to undertake this research

For any errors or inadequacies that may remain in this work, of course, the responsibility is entirely my own.

Cryptoraptor: High Throughput Reconfigurable Cryptographic Processor for Symmetric Key Encryption and Cryptographic Hash Functions

Gokhan Sayilar, M.S.E
The University of Texas at Austin, 2014

Supervisor: Derek Chiou

In cryptographic processor design, the selection of functional primitives and connection structures between these primitives are extremely crucial to maximize throughput and flexibility. Hence, detailed analysis on the specifications and requirements of existing crypto-systems plays a crucial role in cryptographic processor design. This thesis provides the most comprehensive literature review that we are aware of on the widest range of existing cryptographic algorithms, their specifications, requirements, and hardware structures. In the light of this analysis, it also describes a high performance, low power, and highly flexible cryptographic processor, Cryptoraptor, that is designed to support both today's and tomorrow's encryption standards. To the best of our knowledge, the proposed cryptographic processor supports the widest range of cryptographic algorithms compared to other solutions in the literature and is the only crypto-specific processor targeting the future standards as well. Unlike previous work, we aim for maximum throughput for all

known encryption standards, and to support future standards as well. Our $1GHz$ design achieves a peak throughput of $128Gbps$ for AES-128 which is competitive with ASIC designs and has 25X and 160X higher throughput per area than CPU and GPU solutions, respectively.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

1.1 Introduction

As the demand for secure communication bandwidth is growing at an unprecedented pace, efficient and high throughput cryptographic processing becomes increasingly critical for overall system performance. Besides high performance computing, the flexibility also becomes an essential feature of cryptographic processors because of the numerous cryptographic algorithms and security standards. New cryptographic algorithms are continuously being developed, which makes existing hardware inadequate to satisfy new requirements. Thus, it becomes more desirable for cryptographic processors to support existing crypto-systems as well as having the potential to support future standards.

To be to cover all cryptography domain, one must understand the whole domain and existing requirements first. In current cryptography standards, there are three types of algorithms: (i) symmetric-key encryption and (ii) cryptographic hash functions, and (iii) public-key encryption.

a. Symmetric-key encryption: Symmetric-key encryption refers to a class of cryptography algorithms where a sequence of operations is repeatedly

applied to the blocks of data using a single shared key to encrypt and decrypt. Since both parties have to share and use a single "secret" or "key" for encryption and decryption, symmetric-key encryption is also known as "shared key encryption" or "private key encryption". Symmetric-key encryption algorithms exist as block and stream ciphers. While block ciphers tend to be used for higher security, stream ciphers are known to be fast, secure enough, easier to implement, and requires less processing power. Advanced Encryption Standard (AES) [58], and Rivest Cipher 4 (RC4) [213] are often used examples of this algorithm class.

b. Cryptographic hash functions: Cryptographic hash functions refer to a class of irreversible one-way functions that take an arbitrary length message and generate a fixed-size bit sequence, message digest. Cryptographic hash functions are widely used in the form of authentication such as digital signatures and message authentication codes. Secure Hash Algorithm-1 (SHA1) [74] and Secure Hash Algorithm-2 (SHA2) [74] are widely used examples of this class.

c. Public-key encryption: Public-key encryption is a class of cryptography algorithms that requires a pair of keys; one public and one private for each user. The private key is always in the possession of the owner, while public key is sent along with the message or publicly available. Even though the public and private keys are entirely different, they are mathematically linked to each other as specified in the algorithm. Since

the keys are used for performing opposite operations, public-key encryption is also known as "asymmetric cryptography". The security of a public key encryption depends on the computational infeasibility of the algorithm which generally involves (i) computing the factors of a gigantic number (300 decimal digits or more) that is the product of two large prime numbers or (ii) exponentiation of a significantly large number over a significantly large another number in modulo p where p is a large prime. The widely known public-key encryption algorithms are Diffie-Hellman key exchange [98] and RSA [112].

While symmetric-key encryption algorithms and hash functions mostly rely on primitive logical and arithmetic operations that can be computed efficiently, modular exponentiation and modular multiplication are the most frequent operations in public-key encryption. Public-key encryption tends to be very slow and resource intensive. They require special hardware support for high performance since they have to deal with very large numbers (up to 2048 bits). Public-key encryption algorithms are relatively computationally expensive compared to the most, if not all, symmetric-key encryption algorithms and cryptographic hash functions; therefore, they are expensive in terms of time, area, and power.

In this project, our studies and the proposed cryptographic processor focus on symmetric-key encryption algorithms and cryptographic hash functions only since they rely on common structures and completely different computa-

tional primitives than public-key cryptography. Thus, public-key encryption is currently beyond the scope of this work.

1.2 Our Contributions

New cryptography standards and fast implementation of existing ones are continuously being developed. Implementations are ranging from application-specific integrated circuits (ASICs), which are fast but inflexible, to general purpose processor (GPP) based software, which are flexible, but slow. In this thesis, we address the problem of having highly flexible and yet high performance cryptographic processor.

To design a high performance configurable crypto processor, one must first understand which functionalities must be implemented by that processor in order to support current cryptographic algorithms while providing the capability of implementing future algorithms as required.

Our first contribution is the comprehensive literature review on cryptographic algorithms and the detailed analysis on the specifications and requirements of various crypto-systems. To the best of our knowledge, our algorithm analysis of 148 existing symmetric-key encryption algorithms and hash functions is the first and only work that provides comprehensive information about the algorithms and hardware structures that can efficiently implement them. During our study, we focused on the architectural structure of cryptographic algorithms to bridge the gap between hardware designers and cryptographic algorithm developers. Even though each algorithm has different structures

and characteristics, we focused both on finding common patterns and characteristics, as well as the features that differ between algorithms. Unlike other research projects that focus on a limited set of currently popular algorithms, our analysis relies on a wide range of cryptographic algorithms.

As the second and main contribution, we propose a high performance and highly flexible cryptographic processor based on our analysis. It supports a wide range of existing ciphers and cryptographic hash functions and has high potential to support future algorithms. The proposed architecture with its reconfigurable substrate provides a high degree of flexibility even when implemented in an ASIC. Besides its flexibility, our design operating at $1GHz$ achieves a peak throughput of $128Gbps$ on CTR AES-128 encryption which is highly competitive with fully-optimized AES cores described in the literature.

Lastly, we provide a detailed timing, power, and area analysis on functional primitives of cryptographic algorithms and our processor. We believe that such analysis combined with our comprehensive literature survey on symmetric-key encryption algorithms and hash functions would help both cryptographic algorithm developers and hardware designers to evaluate design trade-offs during the design and implementation.

1.3 Thesis Outline

The rest of this thesis is organized as follows. In chapter 2, we briefly describe related research projects and other solutions found in the literature. The chapter 3 and 4 present our detailed analysis on existing cryptographic

algorithms. The specifications and design rationales of our proposed cryptographic processor are given in chapter 5. In chapter 6, we provide detailed analysis of our processor in terms of performance, timing, area, power, and algorithm coverage. The chapter 7 provides brief description about a subset of algorithms and explains how these algorithms are mapped onto our processor. Finally, we discuss possible future research questions and extensions to the proposed processor in chapter 8 and 9.

Chapter 2

Related Work

Once a cryptographic algorithm is designed, it is relatively straightforward to implement that algorithm in software to run on a general purpose processor (GPP). Such an implementation, however, may not provide the desired performance at the desired power and computing resources. For example, encrypting 10Gb/s of data using an Intel processor with AES instructions today takes roughly one processing core, which may not be acceptable.

To drastically reduce or eliminate the processing overhead from the GPP, one could implement encryption algorithms in a hard-wired application-specific integrated circuit (ASIC). Doing so potentially offers the highest performance at the lowest power, but requires hardware design that is expensive and the final product is inflexible. If the ASIC does not support any of the currently used algorithms, it is worthless and would have to be replaced, at potentially great cost and effort. Even if a subset of the algorithms supported is not used, there is wasted silicon and the effort; therefore, the cost to build and deploy that ASIC may not be worthwhile. Moreover, changing standards and algorithms could make ASIC partially or entirely useless. Thus, the inflexibility of ASIC solutions is a significant negative. To eliminate the inflexibility,

one could implement encryption algorithms in hardware structures in a field programmable gate array (FPGA) that can be reprogrammed at will. FPGAs, however, are expensive and, at least currently, roughly the same level of difficulty to program as an ASIC (though much simpler to delay).

Another alternative is to design a special purpose programmable processor that is optimized to execute cryptographic algorithms. Due to their flexibility and high throughput, reconfigurable cryptographic processors are promising alternatives for the implementation of cryptographic algorithms. Therefore, developing a hardware architecture that provides efficient and high throughput implementations for crypto-systems has become increasingly important.

2.1 Instruction Set Architecture Extensions

The first category that tries to achieve high throughput on cryptographic applications is ISA extensions (ISEs) which are new instructions introduced to support one or more cryptographic algorithms. Intel added six SSE instructions and hardware support in their new generation CPUs to speed up AES [4]. Even though the proposed pipeline and new instructions help to achieve high performance by having the round latency of six cycles for one stream in serial mode, they are useful to accelerate AES only. IBM and Oracle also introduced new cryptographic instructions and hardware support in their high-end processors. IBM provided a crypto engine in IBM PowerEN^{TM} Processor Chip [40] to accelerate a predefined set of crypto-systems: AES,

ARC4, DES, Kasumi, MD5, SHA-1, and SHA-2. Likewise, Oracle followed the same strategy in the Sparc T4 Chip [84] to support a similar set of algorithms: AES, DES, Kasumi, Camellia, MD5, SHA-1, and SHA-2. Both IBM and Oracle designs consist of algorithm-specific instructions and dedicated hardware units for each algorithm, which restricts their flexibility and prevents them from supporting other existing and future algorithms.

There are several research projects [25, 44, 67, 110, 129, 193, 221] that propose new instruction extensions to existing ISAs or hardware extensions to GPPs; however, they are also restricted. Even though Parallel Table Lookup [76] and Parallel Read instructions [130] are not intended to be algorithm-specific extensions, they are not useful for the algorithms that do not have table lookup operations and not enough to implement the full functionality of the algorithms that have table lookup operations. The ISE proposed by Grabher [89] accelerates a wider range of cryptographic algorithms compared to other ISE solutions. Nevertheless, it is also limited to a subset of crypto-systems, specifically the ones that operate on data in a bit-oriented manner rather than word-oriented.

Even though ISE proposals improve software performance of cryptographic algorithms, most of the added functionalities is limited to speeding up AES only or a limited subset of crypto-systems. However, more generic crypto-specific ISEs are desirable to support a wide range of algorithms and achieve higher throughput.

2.2 Algorithm Specific Hardware

In addition to ISE solutions introduced in literature and commercial products, there are countless algorithm-specific hardware implementations for both ASIC and FPGA intended to achieve high throughput and better area and power efficiency.

Due to large development and manufacture cost of cell-based and full custom hardware, ASIC solutions become less attractive then FPGA-based solutions. FPGA-based designs also have a quicker time-to-market cycle than ASICs. Therefore, FPGA generally seems to be the ideal candidate for reconfigurable yet high-performance implementation of cryptography algorithms. FPGA-based designs often operate efficiently when highly pipelined. Most of the optimized hardware implementations of cryptographic algorithms use pipelined approaches with varying number of stages, where inner-round functions are duplicated. Doing so allows to achieve a higher maximum frequency, higher throughput, and more efficient use of hardware resources. There has been significant amount of work done in the area of high performance implementation of crypto-systems, specifically for AES [5, 31, 49, 70, 86, 87, 99, 100, 102, 107, 109, 132, 139, 151, 171, 181, 186, 187, 204, 212, 228]. Optimized hardware implementations have also been described for Camellia [60, 223], DES [144], Twofish [124], Blowfish [70], RC4 [78], SHA-1 and SHA-2 [48].

Although hardware solutions optimized for particular algorithms do not have the same objective as our work, they inspired us to design a more optimized processor and helped us to map algorithms onto our processor more

effectively rather than following the traditional structures as described in their specifications.

2.3 Domain Independent Configurable Processors

Besides application and domain-specific reconfigurable computing solutions, there also exists research projects [126, 170] on domain independent configurable processors to lower design effort and eliminate hardware modifications when requirements change. The main purpose of such systems is to enable efficient high performance computing. Despite the fact that flexibility and ease of design are listed as the key benefits of these systems, the applicability of the proposed techniques has not been evaluated on more than one application or domain. The reconfigurability of such systems only allows the processor to integrate different hardware accelerators or configure different functional units in execute stage based on the input.

One serious drawback of the proposed techniques is that they rely on traditional instruction fetch and decode structures to make control decisions, which increases the complexity of hardware, requires more area, and consumes the majority of total energy used in the whole processor (Figure 2.1). In an in-order Reduced Instruction Set Computer (RISC), a large fraction of energy dissipation can be attributed to the instruction supply; 37% for fetching, 18% for decoding, and 14% for issuing an instruction [92]. With our processor architecture, we aim to increase energy efficiency by simplifying the front-end structure of a conventional processor. We use a compact finite state machine

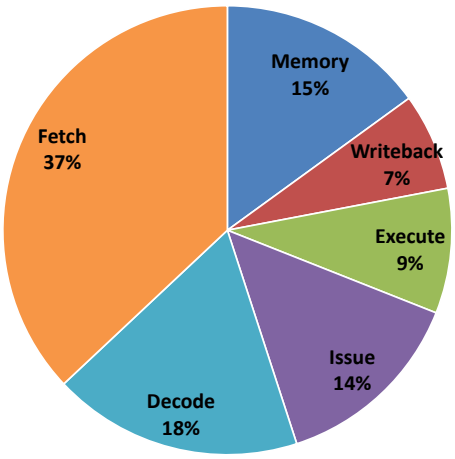


Figure 2.1: The distribution of energy dissipation in an in-order RISC processor [92]

representation for control flow of algorithms to reduce the energy consumption and area requirements.

Our proposed architecture is not the first attempt to simplify the front end of the system. BERET [92] is an energy efficient general purpose coprocessor that can be configured to benefit a wide range of applications. The proposed approach maps users' application to predefined sub-graphs and partially eliminates "fetch-decode-issue" stages using trace cache for those sub-graphs.

Besides high power consumption and area requirements, generic reconfigurable processors generally fail to achieve very high throughput due to the lack of cryptographic algorithm-specific instructions, or they require recompilation process to adapt their internal structures for a specific algorithm.

To the best of our knowledge, ProDFA [224] is the only domain independent runtime reconfigurable architecture that is evaluated on symmetric-key encryption algorithms and does not rely on traditional "fetch-decode-issue" structure. The proposed architecture consists of several reconfigurable processing units, memory units, and interconnects. Each sub-unit is self-controlled using a finite state machine. Even though overall architecture of ProDFA is domain independent, the functional units need to be recompiled for different application domains.

2.4 Configurable Cryptographic Processors

Even though there are various application-specific coprocessors and algorithm-specific hardware implementations, there are very limited attempts to build configurable cryptographic processors with generic modules suitable for a large set of cryptographic algorithms.

CRYPTONITE [43], a Very Long Instruction Word (VLIW) architecture, is a cryptographic processor that supports various encryption and hashing standards, e.g. AES, DES, MD5, and SHA-1. The proposed processor is a two-cluster architecture where each bank consists of a crypto-specific arithmetic logic unit and dedicated memory structure with vector memory addressing mode optimized for table-based encryption functions. Even though the proposed vector memory addressing scheme provides flexibility on permutations and table lookup operations, only per-byte or smaller granularity operations are supported.

CCProc [208] is a flexible cryptography co-processor for symmetric-key encryptions. The proposed coprocessor has its own instruction set tailored to symmetric-key encryption algorithms and an extended VLIW RISC-like data-path structure. The design was aimed to support a wide range of symmetric-key encryption algorithms, but only tested on AES round 2 finalists; Rijndael(AES), MARS, RC6, Serpent, and Twofish. Support for other ciphers and cryptographic hash functions has not been evaluated.

Multi-Core Crypto-Processor (MCCP) [90] is an FPGA-based reconfigurable and high throughput cryptographic processor to secure multi-channel and multi-standard communication systems. It is designed as loosely coupled multi-core system with its own crypto-specific ISA to provide a flexible and high performance cryptography solution. However, the proposed structure is designed to support only 128-bit block cipher algorithms. Thus, it fails to be generic for both symmetric-key encryptions and cryptographic hash functions.

Celator [77] is another cryptographic coprocessor that supports multiple block ciphers and cryptographic hash functions. The proposed architecture consists of 4x4 identical processing elements, each of which can be configured independently. A processing element is capable of performing XOR, AND, NOT, modular arithmetic, right shift, and one AES-specific operation; *xtime*. Even though proposed processor is designed for multi-algorithm support, it has not been evaluated on algorithms other than AES, DES, SHA-1, and SHA-2, and the processing element structure is not powerful enough to support a wide range of algorithms efficiently.

Zodiac [93] is a Network Security Processor designed to provide high performance for network security protocols; IPsec and SSL. Even though it seems to be an application-specific processor, its architecture allows to perform different algorithms and applications; DES, 3DES, AES, RSA, ECC, SHA-1, pseudo random number generation, IPsec, and SSL. Like other alternatives, the main drawback of the proposed processor is that it is restricted to a predefined set of cryptographic applications due to having dedicated hardware for each algorithm.

Cryptographic (Optimized for Block Ciphers) Reconfigurable Architecture (COBRA) [66, 68] is a reconfigurable array structure for efficient block cipher implementations. The proposed architecture is designed after a detailed analysis of 41 block ciphers. However, the algorithm analysis is restricted to block ciphers that operate on plaintext with block sizes of 64 and 128 bits. Even though it aims to support wide range of block ciphers, it fails to efficiently support some most commonly known algorithms (i.e DES and IDEA); thus, it fails to be generic even for block ciphers. The main reasons of not being able to generic are (i) limited block size support, (ii) insufficient lookup table structure, (iii) insufficient bit-wise permutations, and (iv) fixed modulus in modular arithmetic units. While bitwise shifts and rotations are possible on COBRA, bit-wise permutations are extremely difficult to implement. The main difference between COBRA and other configurable cryptographic processors is that the datapath needs to be recompiled for each algorithm separately, resulting different clock frequency and area usage for each algorithm.

However, our project focuses on designing a cryptographic processor with fixed hardware that can be reconfigurable for a wide range of existing cryptographic processor.

Besides research projects, there exists one commercial processor introduced by IBM, called IBM PCIe Cryptographic Coprocessor [103], which provides a high-security and high throughput cryptographic subsystem with specialized hardware to perform AES, DES, 3DES, RSA, SHA-1, and SHA-2. The coprocessor consists of secured sub-system modules which are controlled using sub-system control program and a cryptographic application programming interface (API).

The advantages of alternative solutions described above include better area, power and cost efficiencies, flexibility, algorithm upgradability, and higher performance. However, existing reconfigurable crypto-processors are still restricted to only a small set of symmetric-key encryption algorithms and hash functions, and are far from being generic for all existing and potential future algorithms. Moreover, the proposed solutions mostly rely on traditional instruction fetch and decode structures to make control decisions, which potentially results in high power and area consumptions.

Chapter 3

Cryptographic Algorithm Analysis

In this chapter, we describe our algorithm selection process and analysis methodology, and provide a detailed analysis on existing symmetric-key encryption algorithms and cryptographic hash functions.

3.1 Existing Workload Characterizations

Designing a flexible, high performance, and resource efficient solution for cryptographic applications requires a comprehensive literature review on existing symmetric-key encryption algorithms and hash functions. Besides detailed information on existing algorithms, such study also gives an insight about potential requirements and specifications of future cryptographic algorithms.

Even though there are numerous attempts to speed up cryptographic applications, there are only few studies [77, 206, 208, 224] that present analysis on functional and hardware structure of existing cryptographic algorithms. However, since they are only supportive parts of the presented work in these papers, these analyses only focus on a small set of cryptographic algorithms and only categorize the operation classes. Thus, they do not provide suffi-

ciently detailed information about common hardware structures of existing crypto-systems to enable the design of high performance configurable crypto processor. On the other hand, there are some attempts [45, 47, 75] to study workload characteristics of a set of cryptographic algorithms and profile their software implementation. They give a good idea about the operations classes, their usage frequencies, and required instructions. However, they are also limited to a small set of algorithms, hence not sufficient enough to design a highly configurable cryptographic processor. With its analysis on 41 block ciphers, Elbirt [68] provides detailed information about their functional primitives and common hardware elements so far. However, the algorithm analysis is restricted to block ciphers that operate on plaintext with block sizes of 64 and 128 bits. Like other studies, it also does not present the relation between functional primitives, common patterns, and connection structures. Hence, it is far from providing a sufficiently detailed analysis.

To the best of our knowledge, our analysis on 148 existing cryptographic algorithms is the first and only work that provides a comprehensive analysis on symmetric-key encryption algorithms and cryptographic hash functions about their specifications, requirements, and hardware structures. During our analysis, we mostly focused on architectural structure cryptographic algorithms, where our aim is to bridge the gap between hardware designers and cryptographic algorithm developers. We believe that such a detailed literature survey will help both algorithm developers and researchers while designing new cryptographic algorithms and/or standards, and hardware architects to

design flexible crypto-specific processors achieving high performance. Even though each algorithm has different structures and characteristics, we focused on finding general patterns, common characteristics, and the features that create diversity among algorithms. Unlike previous work, instead of focusing on algorithms and features that suit best to our needs, we provide a broader insight about cryptographic algorithms to enable users to pick their own algorithm list and configure their environments and hardware based on their needs.

3.2 Algorithm Selection

We studied more than a hundred ciphers and hash functions from Lucifer[200] (1971) to present. Our algorithm selection process was solely based on mostly used security protocols such as IPsec, TLS/SSL, WTLS, SSH, S/MIME, and OpenPGP, and cryptographic libraries such as OpenSSL and GNU Crypto. However, common security protocols and libraries do not cover a wide range of algorithms. For that reason, we crawled the literature, patents as well as famous competitions for security standards organized by National Institute of Standards and Technology (NIST), New European Schemes for Signatures, Integrity and Encryption (NESSIE), eSTREAM, and European Network of Excellence in Cryptology (ECRYPT). We analyzed not only the winners but also all finalists and semi-finalists in these competitions. Finally, our algorithm analysis consists of 148 cryptographic algorithms including 96 block ciphers, 26 stream ciphers, and 26 cryptographic hash functions.

a. Block ciphers: A block cipher is a deterministic method of encrypting text in a way that the algorithm is applied with user's secret key to fixed-length groups of bits at once as a block rather than to one bit at a time. They rely on a fixed secret key and an unvarying transformation defined by the algorithm. Many block ciphers are characterized as a Feistel network that divides the data block into two halves where one half operates upon the other half. Block ciphers play a crucial role in the design of cryptographic protocols and are widely used to encrypt large bulk data. Due to the significant number of block ciphers in the literature, they represent a huge portion of existing algorithms used in our study. Thus, they have a serious impact on the design of our processor as well.

The list of block ciphers that we used in our analysis is as follows;

- 3WAY [55]
- AES [58]
- Akelarre [6]
- Anubis [19]
- ARIA [123]
- BaseKing [51]
- Blowfish [188]
- Camellia [10]
- CAST-128 [2]
- CAST-256 [3]
- CIKS-1 [150]
- Cipherunicorn-A [182]
- Cipherunicorn-E [183]
- CLEFIA [196]
- CMEA [172]
- COCONUT98 [210]
- Crab [115]
- Cryptomeria/C2 [37]
- CRYPTON [131]
- CS-Cipher [201]
- DEAL [118]
- DES [1]
- DESX [117]
- DFC [82]
- E2 [207]
- FEAL [194]
- FEALNX [149]
- FEA-M [226]
- FOX [114]
- FROG [81]
- GOST [169]
- Grand Cru [154]
- Hasty Pudding cipher [190]
- Hierocrypt-3 [166]
- Hierocrypt-L1 [165]
- ICE [122]
- IDEA [125]
- Intel Cascade Cipher [39]
- KeeLoq [42]
- KHAZAD [20]

- Khufu and Khafre [30]
- KLEIN [85]
- KN-Cipher [162]
- Ladder-DES [174]
- LED [91]
- LOKI97 [41]
- LUCIFER [200]
- M6 [116]
- M8 [164]
- MacGuffin [32]
- Madryga [135]
- MAGENTA [108]
- MARS [46]
- MBAL [120]
- Mercy [50]
- MESH [155]
- Kasumi [140]
- MMB [54]
- MULTI2 [15]
- MultiSwap [192]
- New Data Seal [22]
- NewDES [191]
- Nimbus [134]
- Noekeon [57]
- NUSH [222]
- NXT [113]
- PRESENT [35]
- PRINCE [36]
- Q [142]
- RC2 [119]
- RC5 [178]
- RC6 [180]
- REDOC III [197]
- SAFER K-128 [137]
- SAFER K-64 [136]
- SAFER+ [138]
- SC2000 [195]
- SEED [128]
- Serpent [9]
- SHACAL [95]
- SHACAL-2 [143]
- Shark [173]
- Skipjack [159]
- SMS4 [61]
- Spectr-H64 [88]
- Square [56]
- SXAL [163]
- TEA [216]
- Threefish [72]
- Twofish [189]
- UES [96]
- Xenon [209]
- Xmx [152]
- XTEA [157]
- XXTEA [225]
- Zodiac [127]

b. Stream Ciphers: A stream cipher is a deterministic method of encrypting text in which plaintext digits are combined with a pseudorandom cipher key stream. Unlike block ciphers, stream ciphers work on smaller chunks of data (usually one byte at a time), keep some sort of memory (called "state") while processing the plaintext, and use this state as an input on the next stages. Stream ciphers are often used for their speed and simplicity in applications where plaintext comes in quantities of unknowable length like a secure wireless connection.

The list of stream ciphers that we used in our analysis is as follows;

- A5/1 [83]
- A5/2 [168]
- Achterbahn [79]
- DECIM [23]
- FFCSR [11]
- FISH [33]
- GRAIN [97]
- HC256 [218]
- ISAAC [111]
- MICKEY [17]
- MUGI [214]
- PANAMA [52]
- Phelix [217]
- Py [29]
- Rabbit [34]
- RC4 [213]
- Salsa20 [24]
- Scream [94]
- SEAL [184]
- Sfinks [38]
- SNOW [65]
- Trivium [59]
- Turing [185]
- VEST [167]
- WAKE [215]
- Yamb [220]

c. Cryptographic Hash Functions: Cryptographic hash functions processes an arbitrary finite length input message to a fixed length output referred to as the hash value. Any changes in message or data (even slight ones) potentially result in entirely different cryptographic hash value due to the avalanche effect that is intentionally designed in the algorithm. The desired security level for cryptographic hash functions is that it should be impossible (i) to find two messages with substantially similar digests, and (ii) to infer any useful information about the data using its digest. Secure hash functions serve data integrity, non-repudiation, and authenticity of the source in conjunction with the digital signature schemes. For that reason, an ideal cryptographic hash function should be injective. Besides cryptographic hash functions already in use, we also included finalists and semi-finalists proposals in recent SHA3 competition into our algorithm analysis to cover recent algorithms as well.

The list of cryptographic hash functions that we used in our analysis is as follows;

- BLAKE [14]
- GOST [148]
- Groestl [80]
- HAS-160 [12]
- Haval [229]
- Hamsi [121]
- JH [219]
- Keccak [27]
- MD2 [177]
- MD4 [176]
- MD5 [175]
- MD6 [179]
- PANAMA [53]
- RadioGatÃžn [26]
- RIPEMD [62]
- RIPEMD-160 [63]
- SHA-0 [160]
- SHA-1 [64]
- SHA-2 [74]
- SHAvite3 [28]
- SipHash [13]
- Skein [73]
- Snefru [146]
- SWIFFT [133]
- TIGER [8]
- Whirlpool [21]

3.3 Analysis Methodology

We studied specifications of each cryptographic algorithm and manually gathered detailed information about all aspects of these algorithms. During our studies, we mainly focused on analyzing table sizes, addressing schemes, operation classes, high-level sequence of operations, operation widths, and connection structures of the 148 algorithms. Even though each algorithm has different structures and characteristics, we focused on finding general patterns, common characteristics, and the features that create diversity among algorithms. The detailed results and discussions are presented in following sections.

Besides manual analysis of 148 cryptographic algorithms, we created a simple cryptography programming language and implemented a toolchain that takes an algorithm, optimizes its control and data flow, generates a dataflow graph, and cross compare of dataflows with other algorithms' dataflow graphs to extract common patterns and structures. Using our tools allowed us to

extract pure data dependencies defined by the algorithm itself rather than tool-specific or language-specific optimizations that would arise if we started with a high-level language like C/C++. Due to the excessive amount of time required to implement and process the dataflow analysis of each algorithm, we limited our analysis to the most widely used algorithms in security protocols and cryptographic libraries. Our dataflow graph analysis included not only individual analysis for each algorithm but also cross comparisons of sub-graphs between algorithms' dataflow graphs.

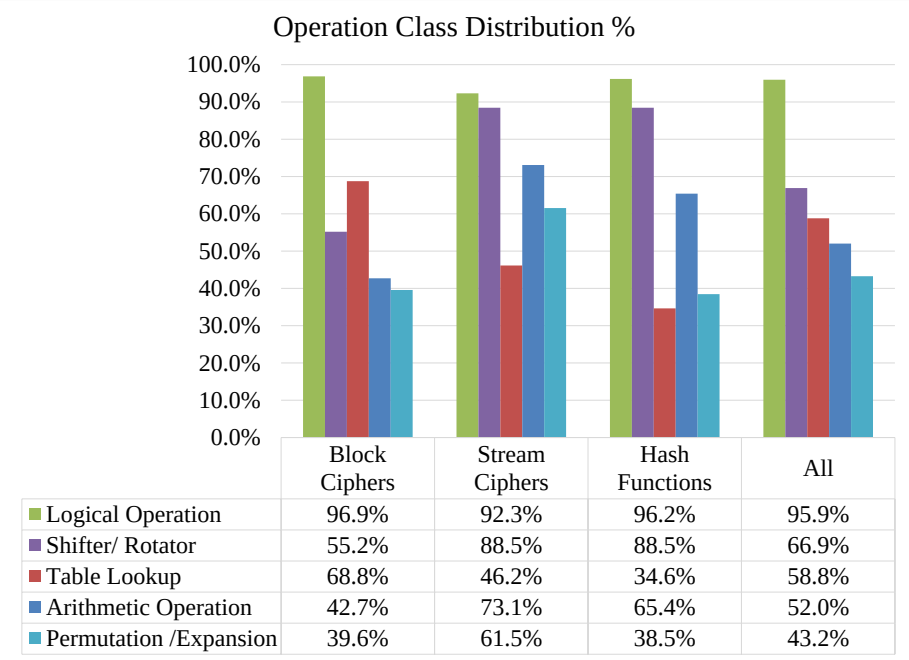
3.4 Detailed Analysis

In this section, we provide a comprehensive analysis on each aspect of cryptographic algorithms' specifications and requirements. In following sections, we describe common operation classes in cryptographic algorithms, special requirements in these operation classes, relations between other operations, and some special needs of particular algorithms. Besides primitive operations used in cryptographic algorithms, we also give a detailed analysis on essential requirements for a good cryptographic processor such as parallel functional units, their connections, and storage requirements.

3.4.1 Operation Classes

Our analysis suggested that primitive operations used in studied cryptographic algorithms can be clustered in 5 operation classes: (i) arithmetic, (ii) logical, (iii) table lookup, (iv) shift/rotate, and (v) permutation/expan-

sion. The classes were determined based on which functional primitives are used most frequently. There exists a few algorithms that require special functional units; however, we didn't include them in common operation classes since they require special hardware and consideration and those algorithms are not widely used. We discuss special functional units in following sections separately. Primitive operations in each class can be summarized as follows;



	Block Ciphers	Stream Ciphers	Hash Functions	All
Logical Operation	96.9%	92.3%	96.2%	95.9%
Shifter/ Rotator	55.2%	88.5%	88.5%	66.9%
Table Lookup	68.8%	46.2%	34.6%	58.8%
Arithmetic Operation	42.7%	73.1%	65.4%	52.0%
Permutation /Expansion	39.6%	61.5%	38.5%	43.2%

Figure 3.1: The use of operation classes in cryptographic algorithm classes

a. Arithmetic operation: The arithmetic operation class includes scalar addition and subtraction over varying lengths. Since floating point numbers and operations are not used in cryptographic algorithms, hardware support for floating point operations is not required. Even though arith-

metic operations can be found in all cryptographic algorithm classes, they are mostly used in stream ciphers. While there is no division operation in cryptographic algorithms, the multiplication will be analyzed separately as a special operation in following sections.

b. Logical operations: The logical operation class consists of bitwise primitive operations; XOR, AND, OR, and NOT. As shown in Figure 3.1 even though the usage frequency varies among algorithm classes, logical operations are the top most used functions in cryptographic algorithms. More than 95% use one or more logical operations in their datapath. In fact, cryptographic algorithms tend to perform a sequence of logical operations; however, we will present more detailed analysis on operation patterns in following sections. Even though any function can be represented as a sequence logical operations, the algorithms that do not require logical operations in their traditional implementations are KLEIN, MultiSwap, PRESENT, SWIFFT, RC4, and Turing.

c. Table Lookup: The table lookup operation, also known as SBOX lookup, replaces runtime computation with a simpler array indexing operation. The table lookup operation is one of the most commonly used operations in block ciphers and provides non-linearity during the encryption process. There is a literature [101, 109, 151] that provides computational representations instead of table lookup operations to eliminate the use of memory, but we consider them to be table lookup operations. This

structure is not limited to explicit table lookup operations defined in algorithm specifications. Some functional operations like matrix multiplication with a constant matrix can also be implemented as table lookup operation using a precomputed table.

d. Shift/Rotate: Variable amount shift and rotation in both direction are clustered in this class. Since shift and rotation operations enable changing the order of the bits in a reversible way, it is the second most commonly used operation class in all types of cryptographic algorithms.

e. Permutation/Expansion: Permutation/Expansion class is responsible for any bit manipulation on up to 64-bit data. Since permutation and expansion operations require an excessive amount of control signals, they are not widely used. However, some portion of each algorithm class still rely on this operation class. Some permutations, especially byte-wise, can also be represented as a table lookup operation.

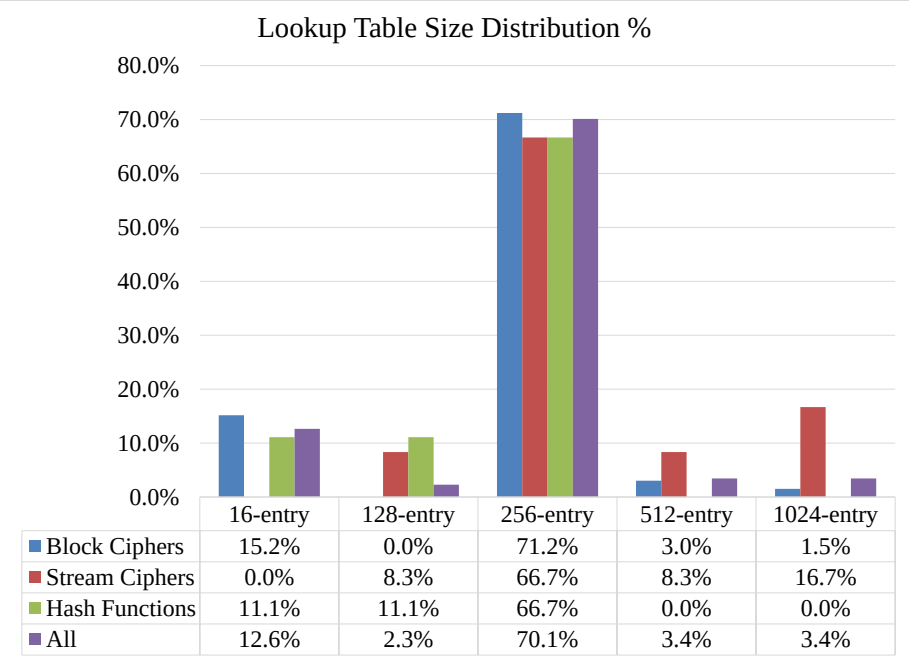
Our analysis on the use of each operation class in each crypto-system class as well as overall distributions is summarized in Figure 3.1. A detailed analysis on the use of functional units in cryptographic algorithms can be found in Appendix A. More detailed information about operation clusters is presented in Appendix B. Based on targeted cryptographic algorithm classes, the structure and amount of functional units can be changed while designing cryptographic processor.

3.4.2 Table Lookup Structure

Since table lookup is one of the most common operations, we examined each aspect of the table structure of cryptographic algorithms in detail. We analyzed table sizes, entry widths, addressing schemes, number of different tables, and the number of parallel tables in each algorithm. With a table structure that is too wide, resources are wasted. Additional lookups are required for a table structure that is too narrow. Therefore, table size and entry wide are crucial elements in the design of cryptographic processor. Our studies show that table sizes and addressing schemes greatly vary among crypto-systems. The table sizes used in cryptographic algorithms vary from 16 to 1024 entries while the entry width starts from 4-bit and goes up to 64-bit. Figure 3.2 shows that more than 70 percent of the algorithms using table lookup consist of tables with 256-entry. However, an ideal generic cryptographic processor should support as many algorithms as possible. Thus, in the light of studied algorithms, table lookup unit structures should be mostly 256-entry tables with support for any table size up to 1024 entries.

The width of the table entry is another important consideration on lookup table structure, since unnecessarily large entry width may cause waste of resources while insufficient entry width may result in loss of performance. Figure 3.3 shows that 8-bit and 32-bit are the most common entry widths among the algorithms that use table lookup operation; 51.3% and 23.3% respectively.

Our algorithm analysis shows that even though there are some outlier

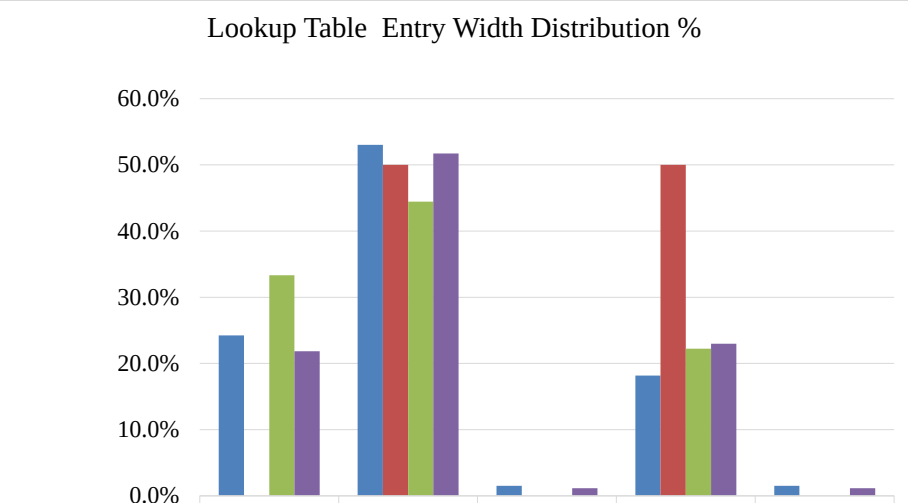


	16-entry	128-entry	256-entry	512-entry	1024-entry
Block Ciphers	15.2%	0.0%	71.2%	3.0%	1.5%
Stream Ciphers	0.0%	8.3%	66.7%	8.3%	16.7%
Hash Functions	11.1%	11.1%	66.7%	0.0%	0.0%
All	12.6%	2.3%	70.1%	3.4%	3.4%

Figure 3.2: The ratio of different table sizes used in cryptographic algorithms

algorithms with different table sizes, the most common table structures are 256x32-bit and 256x8-bit. Since there is only one algorithm, KHAZAD [20], which stores 64-bit data in the table, a reasonable table entry size is 32-bit, since any table with 64-bit data entries can be divided into two parallel tables and outputs of both lookup operations can be combined.

Moreover, our algorithm analysis suggests that there are maximum of four parallel 1024-entry, eight parallel 512-entry, and sixteen parallel 256-entry tables in any specific cryptographic algorithm. Therefore, the table lookup structure of reconfigurable cryptographic processor should ideally be capable of supporting the table size requirements of all existing algorithms



	4-bit	8-bit	24-bit	32-bit	64-bit
Block Ciphers	24.2%	53.0%	1.5%	18.2%	1.5%
Stream Ciphers	0.0%	50.0%	0.0%	50.0%	0.0%
Hash Functions	33.3%	44.4%	0.0%	22.2%	0.0%
All	21.8%	51.7%	1.1%	23.0%	1.1%

Figure 3.3: The ratio of different table entry widths used in cryptographic algorithms

for both the size of one table (4 KB), and the total size of parallel tables in algorithm (16 KB) as well as maximum number of parallel lookup operations (16 operations). Any structure that does not meet these requirements may cause lower performance even though they might save some other resources like area and power.

Table 3.1 shows the distribution of number of parallel lookup operations in cryptographic algorithms. Even though some of the the SHA-3 candidates such as Groestl, Hamsi, and JH use 128 and 256 parallel lookup operations in their bit-slice implementation, we do not include them as maximum number of parallel lookup operations.

Table 3.1: The distribution number of parallel lookup operation in cryptographic algorithms

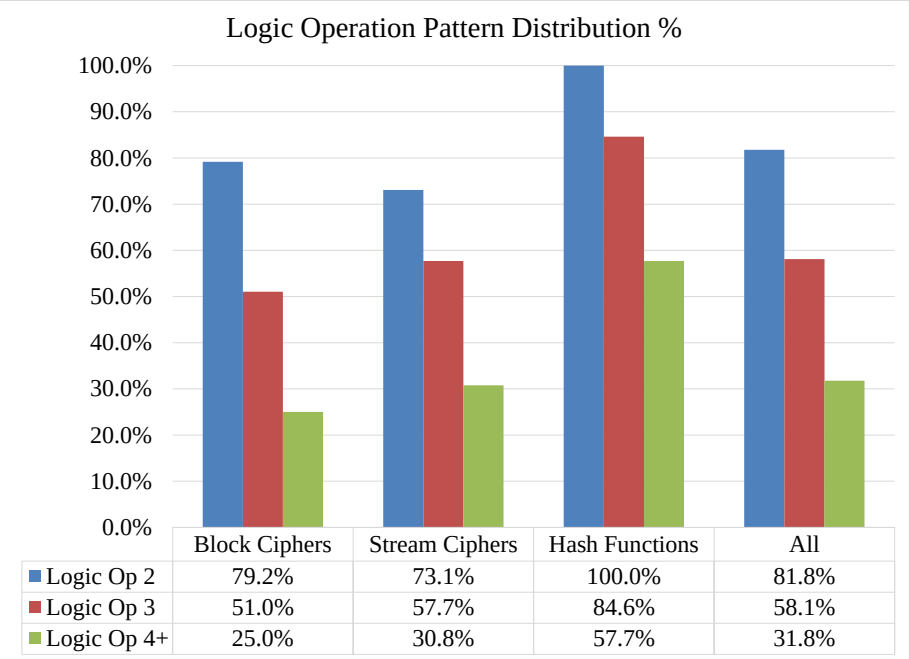
	1	**2**	**4**	**8**	**16+**
Block Ciphers	3.0%	3.0%	22.7%	43.9%	27.3%
Stream Ciphers	16.7%	33.3%	33.3%	16.7%	0.0%
Hash Functions	0.0%	0.0%	11.1%	22.2%	66.7%
All	4.6%	6.9%	23.0%	37.9%	27.6%

3.4.3 Bundled Operation Patterns

Cryptographic algorithms process a sequence of operations on a fix-sized block of data. Their fairly regular structures enable bundling commonly executed sequences of operations as a single big operation block. We studied the possibility of such bundles and examined the ratio of algorithms that use these bundles. The result of our study gave us better insight about general trends in cryptographic algorithms, enabling to design a better cryptographic processors.

As we mentioned earlier, logical operators are the most commonly used operations, and cryptographic algorithms tend to perform a sequence of logical operations back to back. Our studies show that more than half of all cryptographic algorithms process three consecutive logical operations. As shown in Figure 3.4, 82.4% of the algorithms that we studied process two consecutive logical operations while 58.8% process three, and 57.7% of the cryptographic hash functions process four or more in a row.

Our algorithm and dataflow graph analysis suggest that in most of the cryptographic algorithms table lookup operations are preceded and/or followed



	Block Ciphers	Stream Ciphers	Hash Functions	All
Logic Op 2	79.2%	73.1%	100.0%	81.8%
Logic Op 3	51.0%	57.7%	84.6%	58.1%
Logic Op 4+	25.0%	30.8%	57.7%	31.8%

Figure 3.4: The distribution of logical operation patterns in cryptographic algorithms

by an XOR operation. Table 3.2 shows that XOR-SBOX, SBOX-XOR and XOR-SBOX-XOR operation bundles are processed in 69.0%, 71.4%, and 59.5% of the algorithms that have table lookup operations, respectively.

Besides the pattern of XOR and SBOX operations, we also analyzed the possibility of other bundles between other operation classes. The second most common relation between operation classes is the pattern of shift/rotate and logical operations. Table 3.3 summarizes that Logic-Shift/rotate, Shift/rotate-Logic, and Logic-Shift/rotate-Logic operation bundles are processed in 44.4%, 49.5%, and 34.3% of algorithms using shift/rotate operations, respectively.

Table 3.2: The distribution of XOR and SBOX patterns in cryptographic algorithms

	XOR-SBOX	**SBOX-XOR**	**XOR-SBOX-XOR**
Block Ciphers	76.9%	76.9%	66.2%
Stream Ciphers	27.3%	36.4%	18.2%
Hash Functions	62.5%	75.0%	62.5%
All	69.0%	71.4%	59.5%

Table 3.3: The distribution of Shift/rotate and logic operation patterns in cryptographic algorithms

	Logic Op. - Shift/rotate	**Shift/rotate - Logic Op.**	**Logic Op. - Shift/rotate - Logic Op.**
Block Ciphers	56.6%	50.9%	39.6%
Stream Ciphers	21.7%	60.9%	21.7%
Hash Functions	39.1%	34.8%	34.8%
All	44.4%	49.5%	34.3%

Due to the significant amount of XOR and arithmetic operations in cryptographic algorithms, we analyzed the frequency of XOR and arithmetic operation patterns. Table 3.4 shows that XOR-Arithmetic, Arithmetic-XOR, and XOR-Arithmetic-XOR operation bundles are processed in 43.4%, 38.2%, and 35.5% of algorithms using arithmetic operations, respectively.

3.4.4 Special Functional Units

As mentioned above, there are some cryptographic algorithms that require special functional units to achieve higher performance or even to be supported. Even though some of those special functions can be realized by

Table 3.4: The distribution of XOR and Arithmetic operation patterns in cryptographic algorithms

	XOR - Arithmetic Op.	**Arithmetic Op. - XOR**	**XOR - Arithmetic Op. - XOR**
Block Ciphers	51.2%	46.3%	41.5%
Stream Ciphers	15.8%	10.5%	10.5%
Hash Functions	56.3%	50.0%	50.0%
All	43.4%	38.2%	35.5%

other operation classes or by a logical combination of those other operation classes, they may also require special logic or dedicated hardware. Our analysis shows that the most commonly required special operations are integer multiplication, byte-wise rotation, and modular arithmetic.

Table 3.5: The special functional unit requirements in cryptographic algorithms

	Byte Rotator	**Multiplication**	**Modular Arithmetic**
Block Ciphers	24.0%	13.5%	43.8%
Stream Ciphers	19.2%	3.8%	73.1%
Hash Functions	34.6%	11.5%	61.5%
All	25.0%	11.5%	52.0%

Our studies indicate that only 11.5 percent of cryptographic algorithms that we analyzed use integer multiplication (Table 3.5). In fact, these 17 out of 148 crypto-systems are not common and not included in mostly used security protocols or cryptographic libraries. Therefore, dedicated multiplication hardware may or may not be a necessary component of cryptographic processor

depending on target workload. The list of algorithm that explicitly requires multiplication is as follows;

Block Ciphers: Cipherunicorn-A [182], CLEFIA [196], DFC [82], FEA-M [226], IDEA [125], KN-cipher [162], MESH [155], MMB [54], MultiSwap [192], Nimbus [134], RC6 [180], SC2000 [195], and Xenon [209]

Stream Ciphers: Rabbit [34]

Hash Functions: PANAMA [53], SWIFFT [133], and TIGER [8]

As shown in Table 3.5, one-fourth of all algorithms use simple byte-wise rotation on 32-bit data instead of a variable amount. Although byte-wise rotation is a part of the shift/rotate operation class, hardware implementation is less expensive and can be affordably combined with other classes. On the other hand, a shift/rotate unit working on varying granularities is not necessary since only 1% of all cryptographic algorithms require them, and any granularity can be easily implemented using permutation/expansion class.

Our algorithm analysis suggests that modular arithmetic is one of the most common special operation used in cryptographic algorithms, especially among stream ciphers. Theoretically, the simple form of modular arithmetic (addition, subtraction, and multiplication) is not a special function since it just allows the values always staying less than a fixed number, called base or modulus. As a general rule, we do not want the encryption process to have a

big affect on the size of a message. Thus, modular arithmetic allows to keep the operation size within a chosen range. Since modular arithmetic is very well understood in terms of algorithms over various basic operations, it became the primary choice of operation for cryptographic algorithm developers and included in 52% of all cryptographic algorithms (Table 3.5).

Table 3.6: The modular arithmetic base distribution in cryptographic algorithms

	mod(2^8)	**mod**(2^{16})	**mod**(2^{32})	**mod**(2^{64})	**Others**
Block Ciphers	9.5%	2.4%	73.8%	7.1%	7.1%
Stream Ciphers	26.3%	5.3%	63.2%	0.0%	5.3%
Hash Functions	0.0%	0.0%	93.8%	6.3%	0.0%
All	11.7%	2.6%	75.3%	5.2%	5.2%

Any crypto-system that uses modular arithmetic can be constructed in an analogous way with a group having certain properties under associated group of operations. Therefore, due to the choice of the base value, a significant portion of modular arithmetic operations can be implemented using existing operation classes or with a logical combination of them. Table 3.6 shows that 75.3 percent of modular arithmetic operations in studied cryptographic algorithms use 2^{32} as the base value, while more than 90 percent of modular arithmetic operations can be realized using traditional integer arithmetic combined with an AND operation.

On the other hand, there are 5 different cryptographic algorithms that require special hardware or logic to be supported due to their unorthodox base choice for modular arithmetic; specifically 2^{33} for KN-cipher [162], $2^{32}-1$

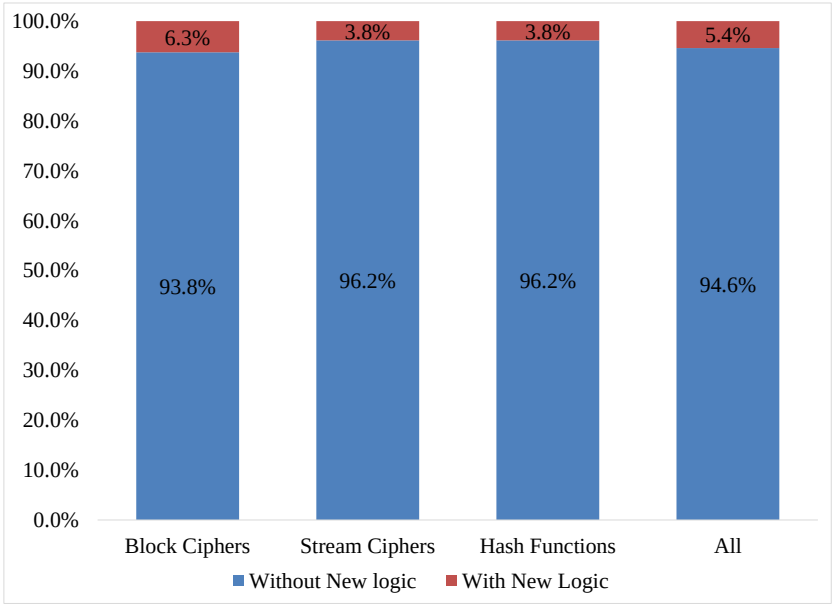


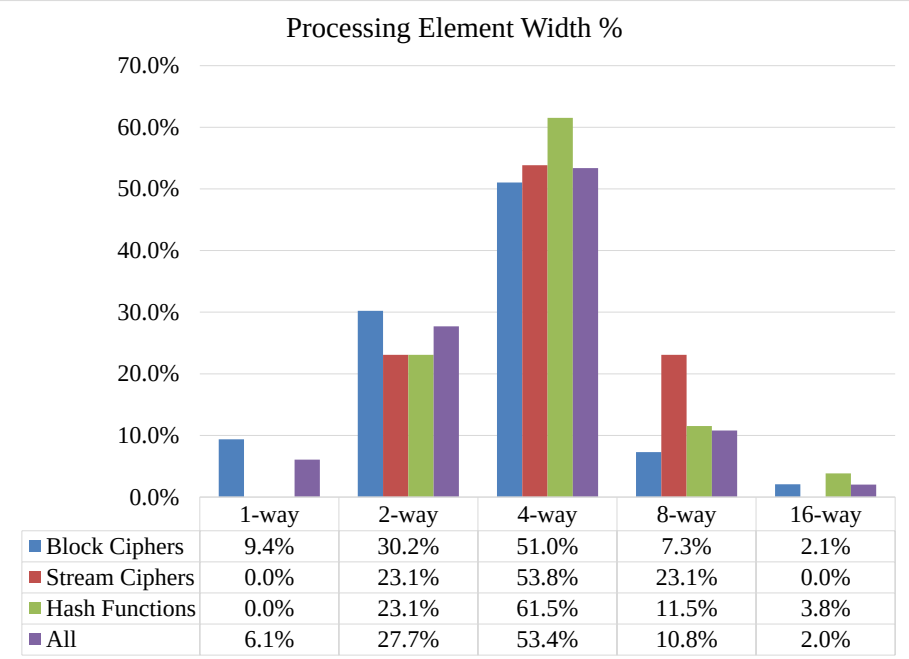
Figure 3.5: The coverage ratio of algorithms that require modular arithmetic

for MMB [54], $2^{64} + 13$ for DFC [82], 17 for PANAMA [52], and 2^{256} for GOST [148]. Therefore, 94.6 percent of all cryptographic algorithms that we analyzed can be implemented without requiring any complicated logic to combine existing operation classes while only small portion of each algorithm classes does require special consideration, specifically 6.3% of block ciphers, 3.8% of stream ciphers, and 3.8% of hash functions. (Figure 3.5). That is; the choice of target algorithms has a significant impact on designing special hardware for cryptography.

3.4.5 Processing Element Width

After examining operation classes, required special functional units, detailed internal structure of operation classes, and their relations with each

other we focused on finding the optimal processor width for cryptographic algorithms. Even though all algorithms can be implemented by one repeatedly used set of functional units (FUs), where each FU implements one operation class, multiple sets of FUs can often improve performance by exploiting parallelism inherent in most of the algorithms. In this thesis, each set of FUs is called a *processing element (PE)*. We examined how much performance gains there would be at various numbers of PEs.



	1-way	2-way	4-way	8-way	16-way
Block Ciphers	9.4%	30.2%	51.0%	7.3%	2.1%
Stream Ciphers	0.0%	23.1%	53.8%	23.1%	0.0%
Hash Functions	0.0%	23.1%	61.5%	11.5%	3.8%
All	6.1%	27.7%	53.4%	10.8%	2.0%

Figure 3.6: The distribution of algorithms requires 1, 2, 4, 8 and 16-way processing elements

The distribution of algorithms that require 1-, 2-, 4-, 8- and 16-way PEs is summarized in Figure 3.6. More detailed parallel processing element requirements of each algorithm can be found in Appendix D. Our analysis

shows that the majority of algorithms fully utilize four-way PEs while 87.2% of cryptographic algorithms requires four or less parallel PEs for maximum performance. Although 19 out of 148 algorithms benefit from eight- or sixteen-way PEs, these algorithms can still be implemented on a four-way processor. Therefore, only a small portion of algorithms can get a performance benefit out of the hardware with 8- or 16-way. On the other hand, designing a 8- or 16-way crypto-processor will potentially result in underutilization of resources for the most of the algorithms, and increase the complexity and cost of the communication among PEs.

3.4.6 Connection Structures

As we discussed in the previous section, a significant portion of cryptographic algorithms that we studied requires multiple PEs running in parallel. Multiple parallel PEs create a need of the connection structure, which drives us to examine the data and control flow of algorithms that use multiple parallel PEs.

Our studies show that cryptographic algorithms tend to have regular or slightly complex connection structure between parallel operations. In general, algorithms work on different blocks of actual data or different bytes of one data block. However, there exists some algorithms that have very complex or control intensive structures due to their characteristics. As shown in Figure 3.7, more than 80 percent of the cryptographic algorithms has fairly regular control structure, while more than 30 percent of stream ciphers require a more

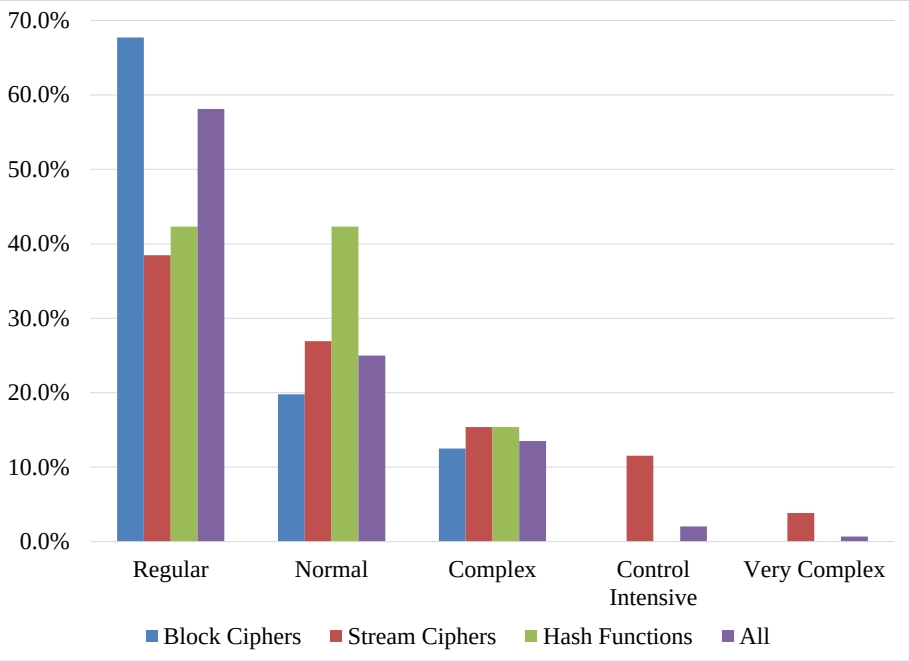


Figure 3.7: The trend of connection structure among processing elements used for implementing algorithms

complex structure to control their dataflow. The main reason that stream ciphers have a more complex control structure is that they generally apply a complex sequence of operations to the same block in various ways. Some examples are covered in following chapters.

For a moderate reconfigurable crypto-processor, having a set of connection schemes based on analysis of existing connection structure would be sufficient. However, a generic cryptographic processor requires more flexibility to support future standards and, therefore, needs higher connectivity.

3.4.7 Storage Requirements

The complex and control intensive dataflow of stream ciphers requires frequent control switches throughout the execution of the algorithm. Our algorithm analysis shows that more than half of stream ciphers and 20.9% of all cryptographic algorithms require a structure that can control more than 40 consecutive operations for their round functions due to changing round structures or changing function length. On the other hand, most of the block ciphers and cryptographic hash functions have very regular structures as shown in Figure 3.7. Due to their unvarying transformation structures, control signals are expected to remain mostly constant throughout the execution of whole algorithm. Since our project aims to achieve highest performance and flexibility for a wide range of algorithms, we believe that an ideal cryptographic processor should accommodates at least the largest of control structure required for any existing algorithm. It is obvious that analyzing existing algorithms does not provide the information that will definitely be valid for future algorithms as well. However, existing algorithms can give an approximate idea about potential lengths of future algorithms.

Besides control structures, we also analyzed the register file usage of cryptographic algorithms. Since it is expected that most algorithms have fairly regular dataflow between pipeline stages, we only examined the explicit register usage of the algorithms. Our studies indicate that an ideal reconfigurable cryptographic processor should have a register file structure with 256 32-bit entries, and should be capable of updating at least four registers in each cycle.

Such register file represents the smallest possible structure that will support all the algorithms that we studied. Future standards may or may not require more registers and/or more parallel updates.

Chapter 4

Cryptographic Algorithm Instrumentation

In this chapter, we describe our cryptographic algorithm implementation/verification process and how we profiled the algorithms, and provide a detailed instrumentation results on implemented cryptographic algorithms.

4.1 Existing Binary Instrumentation of Cryptographic Algorithms

Even though there have been numerous attempts to improve the performance of cryptographic algorithms, only a few papers analyze cryptographic algorithms and present a hardware-focused analysis of such algorithms [77, 206, 208, 224]. In those papers, however, algorithm analysis is limited to a small set of cryptographic algorithms and, even then, only determine which classes of operations should be supported. However, a detailed profiling and manual analysis is crucial to extend existing instruction sets and more importantly to design a high performance reconfigurable cryptographic processor.

To achieve that, there are papers that profiled the software implementations of a set of cryptographic algorithms [44, 47, 75]. Those papers give detailed information about cryptographic operation classes, how frequently

each is used, and the instructions used. They are, however, also restricted to a small set of algorithms.

Burke et al. [44] profiled eight benchmarks: Blowfish, 3DES, Mars, RC4, RC6, IDEA, Rijndael, and Twofish to show possible hardware factors for the performance bottlenecks such as the issue width, the number of function units, and the computation and memory access intensive parts. They used their analysis results to propose extended instructions such as 16-bit modular multiplication, bit permutation, rotation, and memory table lookup.

Fiskiran [75] used the PLX toolset to perform workload analysis of the cipher suite including DES, 3DES, RC4, Blowfish, AES, Twofish, and MARS using RISC-like instruction set. They used the PLX RISC architecture and divided the instructions into seven classes: Store, Load, Arithmetic, Logical, Shift, and Branch (conditional and unconditional) and reported the ratio of these instruction classes. Their analysis provides (i) the execution cycles used per block of encryption, (ii) the round operations in each cipher, and (iii) the fraction of the execution time consumed by round operations. However, their analysis on a very limited algorithm set fails to provide sufficient insight for a reconfigurable high performance crypto-hardware.

Chang et al. [47] performed a profiling and performance analysis on AES, 3DES, Blowfish, IDEA, RC5, MD5, SHA1, ECC, and RSA using Intel's commercial products VTune and PIN tool on Intel Core i7 processor. They clustered the instructions to seven class (binary arithmetic, bit&byte, control transfer, data transfer, logical, shift&rotation, miscellaneous) and provided fre-

quency of each group in each algorithm, as well as presenting memory readwrite accesses. They also defined and analyzed a concept called "Load-Store Block" (LSB), which is a single basic block starting with a LOAD instruction and ending with a STORE instruction. Such analysis provides a good insight about how to construct Processor-in-Memory (PiM) architectures. Even though the provided information is useful for cryptography community, the analysis is still limited to nine algorithms and lacks deeper understanding and information about useful FUs for a flexible high performance crypto-processor.

4.2 Instrumentation Methodology

Even though we analyzed the usage of operation classes in cryptographic algorithms, we believe that the usage frequency of each operation class in algorithms is an important information that may greatly help designing cryptographic processor. To achieve more accurate analysis, we choose to profile software implementation of algorithms automatically using binary instrumentation tool.

We used Intel's PIN [104] as our primary profiling tool. PIN is a dynamic binary instrumentation tool that instrument compiled code to collect data such as instruction mix, instruction address trace, memory reference trace, load-store trace, etc. while the executable is running. Such information may help us better understand cryptographic algorithms' behaviours and the frequency of each operation class used in those algorithms.

Due to the excessive amount of time required to implement and verify

each algorithm, we implemented only 79 of the 148 algorithms, consisting of 48 block ciphers, 13 stream ciphers, and 18 cryptographic hash functions. Each algorithm is implemented in C using reference implementation or description and verified using reference test vectors. To get more accurate results about algorithms' encryption structure, we profiled the algorithms by isolating the encryption function from other structures. Doing so eliminate the bias and incorrect information caused by data preparation, initialization, statistic printing, function calls, etc. To the best of our knowledge, our study is the most comprehensive profiling analysis on the largest algorithm set.

Table 4.1: Instruction Classes

Class	**Instructions**
BA	ADD, ADC, SUB, SBB, INC, DEC, NEG
BB	SETB, SETNBE, SETNLE, SETNZ, SETZ, TEST
L	AND, NOT, OR, XOR
SR	ROL, ROR, SAR, SHL, SHR, SHRD, BSWAP
CT	CALL_NEAR, INT, JB, JBE, JL, JLE, JMP, JNB, JNBE, JNL, JNLE, JNS, JNZ, JP, JRCXZ, JS, JZ, LEAVE, RET_NEAR
DT	CDQE, CMOVS, CMP, CMOVB, CMOVBE, CMOVNB, CMOVNBE, CMOVNS, CMOVNZ, CMOVZ, CMPXCHG, CLD, CWDE, MOV, MOVSB, MOVSD, MOVSW, MOVSX, MOVZX, POP, PUSH, XADD, XCHG, CMPSB, SCASB, STOSB, STOSD
SF	DIV, IDIV, MUL, IMUL
M	LEA, NOP, RDTSC

We compiled all the algorithms into IA-32 assembly instructions with x64 using GCC v4.8.2. As shown in Table 7.2, the instructions found in the outputs are grouped into eight classes based on their functionality: Binary Arithmetic instructions (BA), Bit and Byte instructions (BB), Control Transfer instructions (CT), Data Transfer instructions (DT), Logical instructions

(L), Shift and Rotate instructions (SR), Special Functional instructions (SF), and Miscellaneous instructions (M) [203].

4.3 Detailed Analysis

In this section, we give a comprehensive analysis on different aspects of cryptographic algorithms that we gathered from dynamic instrumentation of cryptographic algorithms.

Table 4.2: Instruction Class Frequencies

	BA	**BB**	**L**	**SR**	**CT**	**DT**	**SF**	**M**
Block Ciphers	10.6%	0.8%	12.8%	7.5%	3.3%	61.6%	0.3%	2.6%
Stream Ciphers	13.6%	0.5%	7.2%	6.7%	3.2%	63.7%	0.0%	4.5%
Hash Functions	11.0%	0.2%	12.8%	7.2%	2.6%	63.0%	0.3%	1.6%
All	11.0%	0.6%	12.4%	7.3%	3.1%	62.2%	0.3%	2.5%

Table 4.3: Operation Class Frequencies

	Arithmetic	**Logical**	**Shift Rotate**	**Permutation**
Block Ciphers	36.0%	38.5%	22.4%	3.1%
Stream Ciphers	49.9%	26.7%	21.6%	1.8%
Hash Functions	37.9%	41.9%	19.3%	0.8%
All	38.0%	38.1%	21.6%	2.4%

Table 4.2 and Table 4.3 shows the average usage frequency of each instruction class as well as the usage frequency of operation classes that we described before while Figure 4.1 shows the distribution of each class with minimum, maximum, and the distribution around the median.

Due to profiling methodology of PIN, it is not possible to collect array

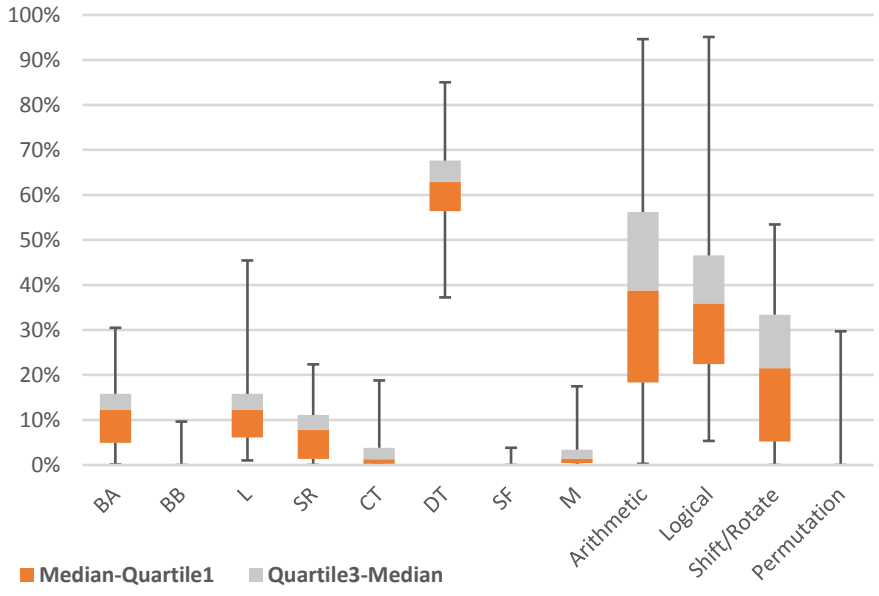


Figure 4.1: Instruction and Operation Class Distribution

and table accesses through binary instrumentation. Thus, operation class usage frequencies do not include table lookup operations. Since permutation/expansion operations can also be implemented as table lookup; our results only include explicit bit/byte instructions as permutation/expansion operations. Our analysis shows that Data Transfer class is mostly used instruction class in all algorithms. Binary Arithmetic and Logical instruction classes, which defines the actual algorithm path, account for the majority of remaining instruction count; averaging 11.0% and 12.4% with the maximum of 30.48% and 45.45% respectively. The Control Transfer class accounts for only 3% of the instruction count on the average and used mainly in fixed loops for

round structure rather than data-dependent control flow. As mentioned before, multiplication is not a common operation in cryptographic algorithms. Our analysis shows that Special Functional instructions, such as multiply, are used 0.3% on the average and 3.83% of the time at most.

Table 4.4: Distribution of Memory Accesses

	Data read	**Data write**	**IP-relative read**	**IP-relative write**	**Stack read**	**Stack write**
Block Ciphers	69.23%	88.49%	0.51%	0.12%	30.26%	11.39%
Stream Ciphers	65.51%	92.08%	0.98%	0.10%	33.51%	7.82%
Hash Functions	66.61%	89.09%	0.08%	0.00%	33.32%	10.91%
All	68.00%	89.23%	0.49%	0.09%	31.51%	10.68%

Table 4.5: Distribution of Data Read/Write Granularities

	Data read (1B)	**Data read (2B)**	**Data read (4B)**	**Data read (8B)**	**Data write (1B)**	**Data write (2B)**	**Data write (4B)**	**Data write (8B)**
Block Ciphers	8.38%	0.61%	19.68%	15.80%	2.65%	0.32%	7.08%	3.23%
Stream Ciphers	8.04%	0.21%	18.25%	19.14%	3.47%	0.12%	6.04%	2.31%
Hash Functions	3.97%	0.14%	19.65%	20.01%	0.69%	0.07%	5.82%	5.35%
All	7.30%	0.43%	19.43%	17.33%	2.33%	0.23%	6.61%	3.56%

Even though we could not accurately analyze the table lookup frequency, we analyzed the memory access types and access granularities to provide better insight about memory accesses. PIN categorizes memory accesses as Instruction Pointer Relative (IP-relative), Stack, and Data read/write accesses, as well as 1B, 2B, 4B, and 8B granularities of data accesses. Our analysis on cryptographic algorithms shows that while Data reads/writes occupy 57.23% of memory access on the average, Stack accesses is the second heavily

used access type with 42.19%. The detailed distribution of access types and access granularities is shown in Table 4.4 and Table 4.5, respectively. Even though such information does not help us to understand table lookups better, it may give insight for designing memory sub-structure of the processor.

Chapter 5

Cryptoraptor: Reconfigurable Cryptographic Processor

In this chapter, we describe a high performance, power efficient, and highly flexible cryptographic processor, *Cryptoraptor*, that supports a wide range of existing ciphers and cryptographic hash functions as well as potential future ones. The name of our processor is inspired by a genus of dinosaur; meaning "secret thief". The fast and flexible nature of Cryptoraptor represents our high throughput processor that is highly flexible for both existing and future cryptographic algorithms. In following sections, we explain our design methodology and provide detailed information about different aspects of our processor.

5.1 Design Methodology

As stated earlier, the primary goal of our project is to have a complete, flexible, and high performance cryptographic processor that can support a wide range of symmetric-key encryption algorithms and cryptographic hash functions. To achieve that, we started our design process by analyzing existing cryptographic algorithms as we described in previous chapters. We believe that

designing high performance reconfigurable cryptographic processor depends on such a detailed analysis of existing algorithms, which may also give an idea about future requirements as well. For that reason, we combined the results of both algorithm and dataflow analysis as well as our algorithm profiling results to design our cryptographic processor.

During the design of our processor, we focused on having a flexible architecture that can be generic for ciphers and hash functions instead of an optimized processor that achieves higher throughput on a particular algorithm or a predefined set of algorithms only, which makes our cryptographic processor unique compared to related work.

We put our effort to achieve high throughput and flexibility without optimizing our processor for area or power. The performance and reconfigurability of our processor are analyzed and discussed in the following chapters.

Since achieving high performance while having high degree of flexibility requires special considerations, we also considered both FGPA and ASIC as implementation platforms. Even though flexibility is the key component of FPGAs, we focused on ASIC implementations for performance while providing flexibility within our design. The following sections provide more detailed information on our implementation process.

5.2 Cryptoraptor

At a high level, the Cryptoraptor architecture consists of an *Execution Tile* as the functional part that performs the bulk computation, a *State Engine (SE)* that controls the Execution Tile, and a 256-entry register file. The high-level structure is shown in Figure 5.1.

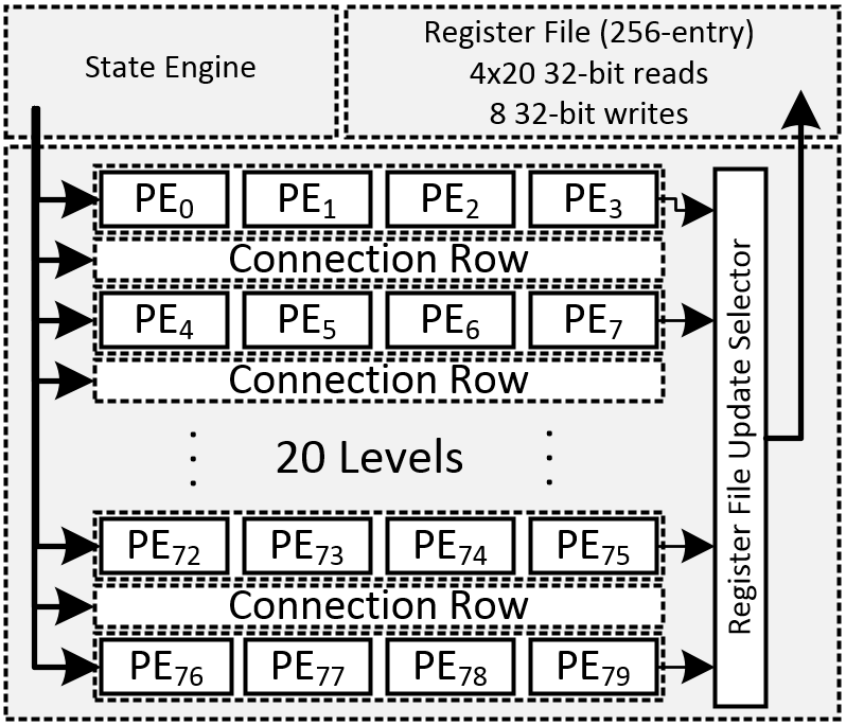


Figure 5.1: The internal structure of Cryptoraptor

The SE is a hardware state machine that is configured as part of the initial setup and remains constant as long as the algorithm being executed does not change. The SE consists of a state counter and a small control memory block. By eliminating fetch and decode stages of conventional processors, the

majority of the area and power is consumed by the Execution Tile, yielding higher area and power efficiency.

The Execution Tile consists of multiple identical *stages*, each containing a number of *Processing Elements (PEs)* connected to the next stage by *Connection Row (CR)*. As shown in Figure 5.1, it consists of a number of PEs and CRs, and loopback connections from each stage to a register file.

The implementation details, as well as timing, area, power, and power analysis of Cryptoraptor will be analyzed and discussed in following chapters. Chapter 6 also discusses the algorithm coverage of our processor while example algorithm mappings are provided in Chapter 7.

5.3 Execution Tile

As shown in Figure 5.2, the Execution Tile consists of varying number of rows of PEs and CRs as well as loopback connections from each stage to a register file.

Our analysis suggests that 87.2% of cryptographic algorithms require four or less parallel PEs for maximum performance while only 19 out of 148 algorithms benefit from eight-way and sixteen-way PEs. An eight-way or sixteen-way crypto-processor will potentially result in underutilization of resources for the most of the algorithms and increase the complexity and cost of the communication across PEs. Therefore, Cryptoraptor consists of four parallel and independently configurable PEs in each stage, called *PE row*.

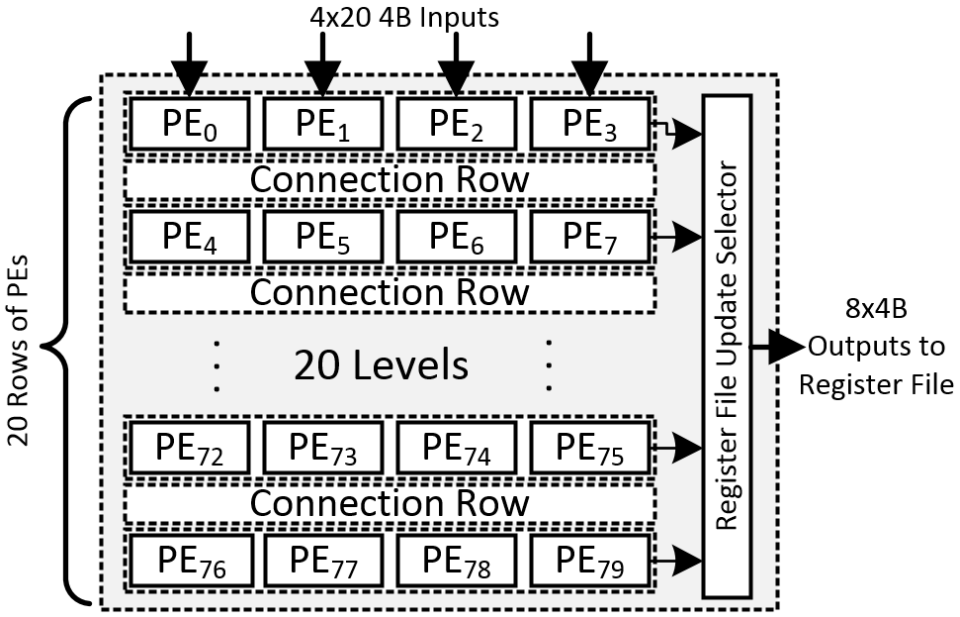


Figure 5.2: The high level structure of Execution Tile

In the light of our algorithm analysis, we decided that 20-stages of PE rows and CRs with a 256-entry register file is the optimal configuration in the Execution Tile . Therefore, the processor can provide up to 80 logical stages by fully utilizing the control memory that provides four distinct stage images. Smaller numbers of pipeline stages can be realized using register file loopbacks and partial configuration of PEs in each stage. The flexibility of storing intermediate results from any stage to a register file enables us to utilize any portion of the processor for computation and to have varying number of pipeline stages for different algorithms. Due to its independently configurable structure, any PE individually and/or a PE row as a whole can be turned on/off depending on the algorithm to have a better fit and further

power efficiency.

The Execution Tile takes 4x20 4B inputs, processes through N level of PE and CRs (N=20 for Cryptoraptor), and update up to eight 4B registers per cycle. As mentioned in previous sections, the cryptographic algorithms may need up to four register updates per cycle, but we allow each PE row to generate up to eight register update candidates to achieve a higher degree of flexibility.

5.4 Connection Row

The Connection Row (CR) is a crossbar that connects any PE from the previous stage to any PE in the next stage. That is, it is capable of connecting 12xN outputs in stage i to 20xN inputs in stage j, where N is the number of parallel PEs in each stage (N=20 for Cryptoraptor). Although full crossbar topology has a significant impact on the cycle time of our processor (analyzed and discussed in following chapters), it increases the flexibility of our design and allows us to support more algorithms efficiently. This unit can also be structured with a predefined set of connection schemes between functional units based on pattern analysis on existing algorithms. However, doing so will potentially limit the capabilities of our processor for future algorithms.

A CR consists of four parallel PE connectors, each of which prepares all inputs of corresponding PE in the next stage. Each PE connector is configured using a 120-bit control signal; 6 bits for each PE input in the next stage. The control signal structure of a PE connector is summarized in Table 5.1. Most

Table 5.1: The control structure of one PE connector

Control index #	**Output destination**
control[5:0]	AU operand #0
control[11:6]	AU operand #1
control[17:12]	AU operand #2
control[23:18]	AU operand #3
control[29:24]	LOU operand #0
control[35:30]	LOU operand #1
control[41:36]	LOU operand #2
control[47:42]	LOU operand #3
control[53:48]	LOU operand #4
control[59:54]	LOU operand #5
control[65:60]	TLU operand #0
control[71:66]	TLU operand #1
control[77:72]	TLU operand #2
control[83:78]	TLU operand #3
control[89:84]	SRU operand #0
control[95:90]	SRU operand #1
control[101:96]	SRU operand #3
control[107:102]	SRU operand #4
control[113:108]	PEU operand #0
control[119:114]	PEU operand #1

significant two bits of six selection bits per PE connector to select one of the 4 PEs in the previous stage. Table 5.2 describes how the remaining four bits select one among twelve 32-bit outputs of the selected PE in the previous stage.

Our algorithm analysis suggests that the cryptographic algorithms may need up to four register updates per cycle. We extended this requirement for each PE row to provide a high degree of flexibility. Besides controlling connections between PEs, a CR is also responsible for controlling up to eight outputs

Table 5.2: The input selection structure of PE connector (least significant 4 bits of 6 selection bits)

CONNcontrol[3:0]	**Functional Unit**
0000	AU out
0001	AU out (shifted)
0010	LOU out
0011	LOU out (shifted)
0100	TLU out #0
0101	TLU out #0 (shifted)
0110	TLU out #1
0111	TLU out #2
1000	TLU out #3
1001	SRU out
1010	PEU out #0
1011	PEU out #1

from all stages to be stored to the register file. By providing eight parallel register file write port, we doubled the requirements of existing symmetric-key encryption algorithms and hash functions. The flexibility of storing intermediate results from any stage enables us to utilize any portion of Cryptoraptor and to have a different number of pipeline stages for each algorithm. Thus, any PE individually and/or a PE row can be turned on/off as needed by the algorithm. The control structure enables users to configure only required PEs and corresponding PE connectors or even a small portion of a single PE and PE connector. Since selecting 8 outputs requires 8x6-bit control signal, the total number bits of control in CR is (8x6)+(4x120) bits, and organized as

follows;

$$\begin{aligned} control &= \{RegOutSLCT \,||\, PEconnectorCTRL\} \;\; where \\ RegOutSLCT &= \{OUTslct_7 \,||\, OUTslct_6 \,||\, \ldots \,||\, OUTslct_1 \,||\, OUTslct_0\} \end{aligned} \tag{5.1}$$

A control memory block associated with each CR. Each CR control memory can hold up to 4 sets of control signals that are loaded at the beginning and controlled by the central state machine of the processor. The control interface of a CR consists of one 32-bit input for control load and a 7-bit control signal defined as follows;

$$CONNcontrol = \{WriteEnable \,||\, State \,||\, WordAddress\} \tag{5.2}$$

The control memory of a CR is loaded by 32-bit data chunks, and address of each word is controlled by 2-bit $State$ and 4-bit $WordAddress$. Throughout the control loading process, the $WriteEnable$ bit of CR state control signal must be asserted. All other times control memory is in read-only mode. Loading the control signals of single CR may take up to 17 cycles depending on the amount of control signals needed to implement an algorithm. The control structure and addressing flexibility enable users to configure only corresponding PE connectors or even small portion of one PE connector.

5.5 Processing Element Row

The Processing Element Row is encapsulation module to organize parallel PEs in each level. The proposed structure contains four parallel and

independently configured PEs in a row. Thus, one PE row takes four sets of 20x4B as inputs and 4x7-bit control signals to produce four sets of 12x4B outputs. Each PE has its own control memory and is configured separately using the same interface.

5.6 Processing Element

A Processing Element (PE) is the smallest execution element of our cryptographic processor which includes one of each functional units: an Arithmetic Unit (AU), a Logical Operation Unit (LOU), a Table Lookup Unit (TLU), a Shifter-Rotator Unit (SRU), and a Permutation-Expansion Unit (PEU).

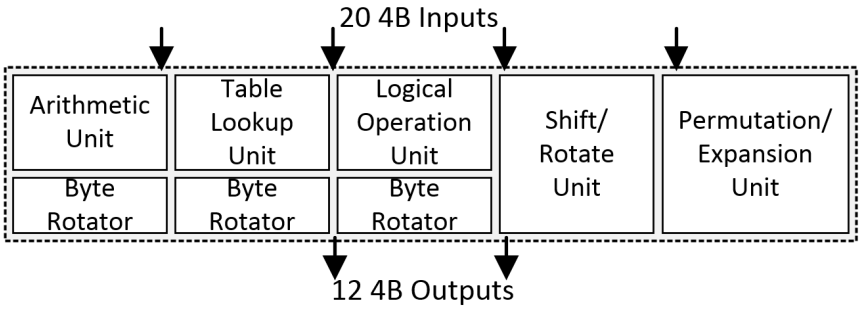


Figure 5.3: High level unit structure of a Processing Element

Our analysis on cryptographic algorithms suggests that even though variable amount shift and rotate operations are needed in crypto-systems, one-fourth of the algorithms use byte-wise rotation instead. This ratio becomes as high as 34.6 percent in cryptographic hash functions. Thus, we integrated byte-wise rotation operation to the output of AU, LOU, and TLU without

adding significant overhead to the cycle time of our processor. The overhead of having byte-wise rotation unit is further analyzed and discussed in following chapters. Having such byte-wise rotation unit allows us to have both original and 1, 2, or 3-byte rotated outputs, which is very desirable for some algorithms to reduce the total number of cycles and eliminate redundant and/or duplicate operations. Figure 5.3 shows high-level unit structure of a PE.

Table 5.3: The input structure of PE

Input #	**Functional Unit**
0	AU operand #0
1	AU operand #1
2	AU operand #2
3	AU operand #3
4	LOU operand #0
5	LOU operand #1
6	LOU operand #2
7	LOU operand #3
8	LOU operand #4
9	LOU operand #5
10	TLU operand #0
11	TLU operand #1
12	TLU operand #2
13	TLU operand #3
14	SRU operand #0
15	SRU operand #1
16	SRU operand #2
17	SRU operand #3
18	PEU operand #0
19	PEU operand #1

Since each of these functional units can work concurrently on different inputs, a PE takes twenty 4B inputs and produces twelve 4B outputs. Thus,

Table 5.4: The output structure of PE

Output #	**Functional Unit**
0	AU out
1	AU out (shifted)
2	LOU out
3	LOU out (shifted)
4	TLU out #0
5	TLU out #0 (shifted)
6	TLU out #1
7	TLU out #2
8	TLU out #3
9	SRU out
10	PEU out #0
11	PEU out #1

the proposed PE is capable of utilizing all functional units in parallel to generate up to twelve outputs. Since each unit is independently bypassable, PEs can also forward six different inputs to next stages. Table 5.3 and 5.4 provide detailed information about inputs and outputs respectively.

Table 5.5: The control signal structure of PE

Control Signal	**Amount**	**Functional Unit**
control[3:0]	4	Arithmetic Unit
control[27:4]	24	Logical Operation Unit
control[35:28]	8	Table Lookup Unit
control[43:36]	8	Shifter/Rotator Unit
control[427:44]	384	Permutation/Expansion Unit
control[429:428]	2	Byte-wise rotator for AU
control[431:430]	2	Byte-wise rotator for LOU
control[433:432]	2	Byte-wise rotator for TLU

Each PE is configured using one 434-bit control signal; 4 bits for AU,

24 bits for LOU, 8 bits for TLU, 8 bits for SRU, 384 bits for PEU, and 3x2 bits for byte-wise rotators for AU, LOU, and TLU. The control signal structure of a PE is summarized in Table 5.5.

We placed a small memory block dedicated to each PE to store the required control signals. Even though we expect those signals to remain mostly constant for a given algorithm, such system gives the flexibility of storing multiple sets of control signals that can be easily controlled by state machine of the processor. Each PE can hold up to four sets of control signals that are loaded in the beginning and controlled by the central state machine.

The control interface of a PE is one 32-bit input for control load and a 7-bit control signal defined as follows;

$$PEcontrol \quad = \quad \{WriteEnable \,||\, State \,||\, WordAddress\} \tag{5.3}$$

The control memory of a PE is loaded by 4B data chunks, and address of each word is controlled by 2-bit $State$ and 4-bit $WordAddress$. Throughout the control loading process, the $WriteEnable$ bit of PE state control signal must be asserted. All other times control memory is in read-only mode. Loading the control signals of a PE may take up to 56 cycles depending on the amount of control signals needed to implement an algorithm. The control structure and flexibility on addressing enable users to load only required PEs and functional units; yielding decrease in setup time.

5.7 Functional Units

Our processor design relies on five bypassable and independently configurable functional unit structures that can work concurrently: Arithmetic Unit (AU), Logical Operation Unit (LOU), Table Lookup Unit (TLU), Shifter-Rotator Unit (SRU), and Permutation-Expansion Unit (PEU). The following sections provide detailed information about the structure of each functional unit.

5.7.1 Logical Operation Unit (LOU)

Our studies show that logical operators are the most commonly used operations, and cryptographic algorithms tend to perform a sequence of logical operations. More than 80 percent of existing cryptographic algorithms perform two consecutive logical operations while 58.8% process three logical operations back to back and 31.8% of the algorithms perform four or more consecutive logical operations. That finding inspired us to separate logical operations from the AU to support sequences of logical operations and to process more operations concurrently, which enables us to achieve better utilization of hardware while mapping the cryptographic algorithms and to get higher throughput.

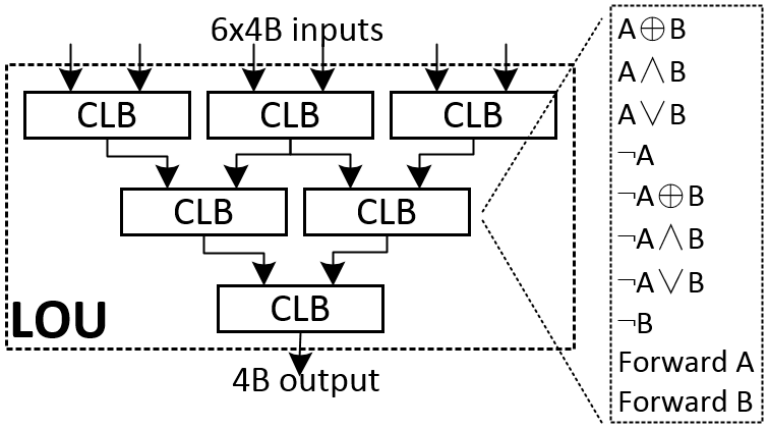


Figure 5.4: The internal structure of LOU

The proposed LOU structure consists of three levels of operation reduction tree with six independently configurable logic blocks (CLBs) as shown in Figure 5.4.

Each CLB is capable of performing four logic primitives (AND, OR, NOT, and XOR) on its operands as well as applying bitwise inversion to any operand (i.e. ¬A⊕B). Each CLB is configured by using one 4-bit control signal. The TLU operates as detailed in Table 5.6.

The LOU takes six 4B inputs, two for each CLB in the first level, produces one 4B output by applying an user-defined logical function and is configured by a 24-bit control signal where;

$$LOUcontrol \quad = \quad \{CLB_5||CLB_4||CLB_3||CLB_2||CLB_1||CLB_0\} \qquad (5.4)$$

The proposed LOU can support all functions found in common cryptographic hash functions and complex logic reduction functions with up to six

Table 5.6: The Configurable Logic Block functionality

Control Signal	**Operation**
0000	Forward operand #1 (A)
0001	Forward operand #1 (A)
0010	Forward operand #1 (A)
0011	Forward operand #1 (A)
0100	Forward operand #2 (B)
0101	Forward operand #2 (B)
0110	Forward operand #2 (B)
0111	Forward operand #2 (B)
1000	A $\oplus$ B
1001	A $\wedge$ B
1010	A $\vee$ B
1011	$\neg$A
1100	$\neg$A $\oplus$ B
1101	$\neg$A $\wedge$ B
1110	$\neg$A $\vee$ B
1111	$\neg$B

inputs. Let's consider the following real life example:

$$F(X,Y,Z) \quad = \quad (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge z) \tag{5.5}$$

The Equation 5.5 is a 6-input function that is used in MD4, SHA-1, and SHA-2. Such a simple function requires five iterations to produce the result using a simple arithmetic logic unit. However, we can easily implement it in a single cycle with the three-level reduction tree structure in our LOU. Besides its functional capacity, the proposed LOU is also capable of forwarding one output among any of its operands.

5.7.2 Table Lookup Unit (TLU)

The table lookup operation is one of the most commonly used operations in cryptographic algorithms to provide non-linearity to ciphers. Our studies on cryptographic algorithms show that table sizes and addressing schemes greatly vary among algorithms. Since more than 70 percent of the algorithms that use table lookup use tables with 256 entries, we use that structure as a general trend. However, there are algorithms that require larger tables. Our algorithm analysis shows that the largest table that is used in existing cryptographic algorithms has 1024 entries. Thus, our table lookup unit structure mostly relies on 256-entry tables while supporting any table size up to 1024-entry.

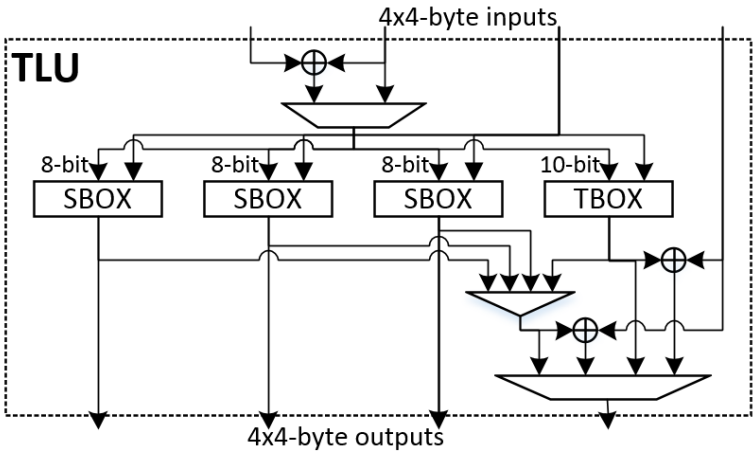


Figure 5.5: The internal structure of TLU

To support a wider range of cryptographic algorithms, three 256-entry tables and one 1024-entry lookup table (named as SBOX and TBOX, respectively) are included in each TLU as shown in Figure 5.5. The current TLU

structure offers three different ways of processing a table lookup: (i) one table lookup with up to a 10-bit address that returns a 4B output, (ii) four table lookups with each byte of a 4B input used as one address that returns four 4B outputs, and (iii) one table lookup with each byte of a 4B input as addresses that outputs each byte of one 4B output. Our TLU structure and its current limitations are solely based on our analysis of studied algorithms; however, larger lookup tables can be supported by splitting them into multiple tables, and using multi-stage lookup operations.

Table 5.7: The Table Lookup Unit functionality

Control Signal	**Operation**
control[0]	TBOX write enable
control[1]	SBOX #0 write enable
control[2]	SBOX #1 write enable
control[3]	SBOX #2 write enable
control[4]	XOR level 1 enable
control[5]	XOR level 2 enable
control[6]	TBOX out / Merge 4 lookup
control[7]	Table lookup enable

Our dataflow graph analysis suggests that in most cryptographic algorithms, table lookup operations are preceded and/or followed by an XOR operation. We include by-passable XOR operations before and after each memory lookup block as shown in Figure 5.5. Doing so enables us to implement a sequence of XOR and SBOX lookup operations in one cycle while it a take multiple cycles without such structure.

The current TLU structure takes four 4B inputs (one for writing data

to tables, three for XOR and lookup operations) and generates up to four 4B outputs, which enables one PE to process four parallel lookups. The TLU is configured by using one 8-bit control signal and operates as summarized in Table 5.7.

5.7.3 Arithmetic Unit (AU)

We describe the arithmetic operation class as integer addition and subtraction operations over varying lengths. Our analysis shows that more than half of the algorithms uses arithmetic operations. While 73.1% of stream cipher algorithms require arithmetic operations, they are also used in 42.7% of block ciphers and 61.5% of cryptographic hash functions.

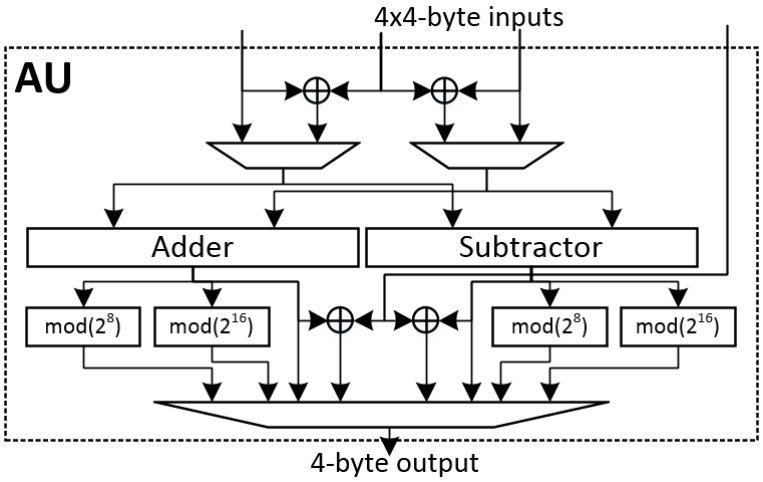


Figure 5.6: The internal structure of AU

We initially designed our AU as being capable of doing only 32-bit addition and subtraction. However, we later decided to extend our AU structure

to support 8 and 16-bit operation granularities due to the high demand on modular arithmetic operations in cryptographic algorithms (Figure 5.6). As we mentioned in algorithm analysis chapter, 75.3% of modular arithmetic operations in cryptographic algorithms use 2^{32} as the base value, while more than 90% of modular arithmetic operations use either 2^8, 2^{16}, or 2^{32} as base, which can be realized by masking the result of standard integer arithmetic with an AND operation. Thus, we provided the flexibility of performing operations in modulo 2^8 and 2^{16} within the current AU structure as well.

Table 5.8: The Arithmetic Unit functionality

Control Signal	**Operation**
0000	Addition (32-bit)
0001	Addition in modulo 2^8
0010	Addition in modulo 2^{16}
0011	Forward the 1^{st} operand
0100	Subtraction (32-bit)
0101	Subtraction in modulo 2^8
0110	Subtraction in modulo 2^{16}
0111	Forward the 2^{nd} operand
1000	(A $\oplus$ B) + C in modulo 2^{32}
1001	A + (B $\oplus$ C) in modulo 2^{32}
1010	(Output of first level) $\oplus$ D in modulo 2^{32}
1011	(Output of first level) $\oplus$ D in modulo 2^{32}
1100	(A $\oplus$ B) - C in modulo 2^{32}
1101	A - (B $\oplus$ C) in modulo 2^{32}
1110	(Output of first level) $\oplus$ D in modulo 2^{32}
1111	(Output of first level) $\oplus$ D in modulo 2^{32}

Beside modular arithmetic support for modulo bases 2^8 and 2^{16}, we also introduced XOR bundles before and after performing arithmetic operations.

Our algorithm analysis suggests that a good fraction of algorithms use XOR and arithmetic operations back to back. We also analyzed the timing, area, and power overhead of these bundles and decided to integrate it into our AU structure since it does not affect the critical path of our processor. Following chapters provides more detailed information about or timing, area, and power analysis. The current AU structure is configured by using one 4-bit control signal and operates as detailed in Table 5.8.

5.7.4 Permutation/Expansion Unit (PEU)

Bit manipulation is used as the main operation in some cryptographic algorithms like DES to provide non-linearity. The PEU provides a capability of merging, manipulating, and expanding bits from two 4B inputs to generate up to an 8B output or two 4B outputs. Even though it is possible to predefine some bit manipulation schemes based on current cryptographic algorithms and then select among them using control signals, we chose to provide full capability in our design to support not only existing algorithms but also potential future algorithms. We believe that PEU will also help to support algorithms that are not easily mapped onto our current design since bit manipulation can implement complex logical operations very easily.

Each bit of the outputs is connected to an independently configurable unit called Bit Selector (BS). Thus, PEU consists of 64 BS, each of which is responsible for generating one particular bit of outputs. Each BS is configured by using one 6-bit control signal. The BS operates as detailed in Table 5.9.

Table 5.9: The Bit Selector control structure

Control Signal	**Selected Bit**
000000	operandA[0]
000001	operandA[1]
000010	operandA[2]
. .	. .
000011	operandA[31]
100000	operandB[0]
100001	operandB[1]
. .	. .
111101	operandB[29]
111110	operandB[30]
111111	operandB[31]

The PEU takes two 4B inputs and produces two 4B outputs by manipulating the bits of the inputs according to user-defined permutation. Since each bit of two outputs may come from any bit of inputs, the PEU is configured by using one 384-bit control signal where;

$$PEUcontrol \quad = \quad \{BS_{63}||BS_{62}|| \ldots ||BS_2||BS_1||BS_0\} \tag{5.6}$$

5.7.5 Shifter/Rotator Unit (SRU)

The Shifter/Rotator Unit supports a variable amount shift/rotate operation on 32-bit data for both ways. Since shift/rotation on 8 or 16-bit data is not common among studied cryptographic algorithms, and it can be implemented within PEU, smaller granularity rotations are not supported by this unit.

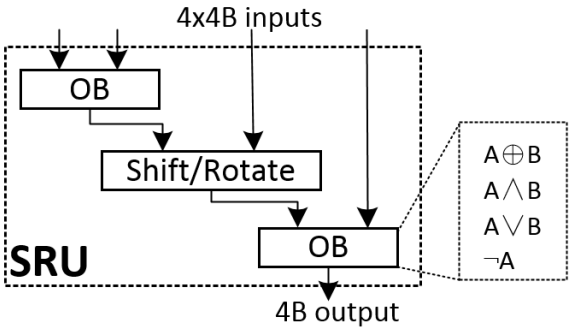


Figure 5.7: The internal structure of SRU

The SRU takes two 4B inputs, one as an operand and the other as shift/rotate amount, and is controlled by one 8-bit control signal where;

$$LOUcontrol = \{enOp2||ctrlOp2||ctrlSHFT||enOp1||ctrlOp1\} \quad (5.7)$$

Our algorithm analysis suggests that more than 40% of algorithms using shift/rotate operations also perform a logical operation before and/or after shift/rotate operations. Thus, we introduced this flexibility into our SRU structure, where user can specify logical operations (AND, OR, NOT, and XOR) before and/or after shift/rotate operation (Figure 5.7).

Table 5.10: The Shifter/Rotator Unit functionality

Control Signal	**Operation**
00	Left shift
01	Left rotate
10	Right shift
11	Right rotate

The shift/rotate functionality is controlled by 2-bit control signal and operates as detailed in Table 5.10. Each operation block is controlled by 3-

Table 5.11: The Operation Block functionality

Control Signal	**Operation**
00	XOR
01	AND
10	OR
11	NOT

bit control signal (1 bit enable and 2 bits for operation configuration). The functional configuration of the operation block is shown in Table 5.11.

Chapter 6

Processor Analysis

In this chapter, we provide detailed timing, power, and area analysis of our processor. We believe such detailed analysis on functional units might give deeper insight about trade-offs between these metrics to both cryptographic algorithm developers and hardware architects. We also analyzed the current algorithm coverage of Cryptoraptor and its limitations.

6.1 Implementation

We have developed a highly modular fully configurable architecture in Verilog HDL. Our parametrized architecture enables us to explore the design space and study the trade-offs by changing the width and height of Execution Tile, the number of PE rows that will fit into a pipeline stage, and even the configuration of each functional units.

Synthesizing our RTL code into a gate level structure was done using Synopsys Design Compiler (DC) and FreePDK45^{TM} v1.4 45nm standard-cell CMOS technology [202]. However, due to lack of a memory compiler, we used register blocks to mimic the functionality of SBOX in TLU while analyzing the maximum frequency of our design. We used "-uniquify", "-ungroup", and

"-flatten" DC optimizations, which removed module boundaries and synthesized the design as a whole block. Our processor design achieves a maximum frequency of $1GHz$, where each pipeline stage consists of one pair of a PE row and Connection Row. More detailed timing analysis and discussions on sub-modules are provided in following sections. We have not yet optimized for frequency by introducing design-related optimizations to our datapath. We leave further optimizations and design exploration as future work. Since compiling memory blocks using registers requires unreasonably large area and consumes high power, we used CACTI 6.5 Memory Model [153] to get more accurate results for our memory blocks. To get more accurate estimate for the overall die area and the power consumption of our processor, we combined the results from DC and the memory blocks from CACTI with an additional 10% error to derive our results. The following sections provide more detailed analysis and discussion on area requirements and differences between DC's and CACTI's area results.

Even though we are not planning to use FPGA as the platform for our processor, we also synthesized our RTL code to Xilinx Virtex-6 FPGA using ISE Design Suite 14.6 without introducing any FPGA-related optimizations to the system. The only thing that we had to change was the memory block used in TLU as Block RAM blocks (BRAMs) in FPGA. Even though the FPGA still can be reconfigured in real time when the algorithm is changed, we keep our design the same to enable reconfigurability without having to recompile FPGA. Despite the fact that the excessive use of multiplexers in our design

has a negative impact on our cycle time on FPGA, we still managed to achieve 203.80MHz with the same configuration used in ASIC version.

6.2 Timing Analysis

We synthesized functional units in our PE separately to help determine which functional unit or structure defines the critical path of our processor. Such analysis also allows us to further improve the design of the functional units which are not on the critical path.

Table 6.1: The cycle time of functional units in PE

	Cycle time (ns)	**Cycle time with byte-rotator** (ns)
AU	0.51	0.56
LOU	0.48	0.53
TLU	0.58	0.63
SRU	0.55	-
PEU	0.23	-
Byte rotator	0.07	-
Multiplication	0.91	-

The cycle time of each functional unit in PE and their combined versions with byte rotator are summarized in Table 6.1. Due to its memory operations and complex structure, TLU is the critical path in our PE design. Since we cannot make the PE faster than the slowest component, we augmented the functionalities of other functional unit; yielding higher compute capabilities. As we described in previous chapters, we bundled arithmetic operations with XOR and shift/rotate operations with logical operation blocks. Table 6.2 summarizes the effects of these bundles on the cycle time our functional units.

Table 6.2: The cycle time comparison of functional units with bundles

	Cycle time (ns)	**Cycle time with byte-shifter** (ns)
AU	0.42	0.46
AU (bundled)	0.51	0.56
SRU	0.30	-
SRU (bundled)	0.55	-

Considering one-fourth of the algorithms use byte-wise rotations, integrating byte rotators to AU, LOU, and TLU has an insignificant effect (0.04-0.05ns when merged) on overall cycle time. As shown in Table 6.1, a single stage multiplication unit takes 0.91ns to multiply two 32-bit integer, which would significantly increase the cycle time of PE; yielding lower clock frequency for the processor. Our algorithm analysis shows that supporting multiplication would increase the algorithm coverage by 8.1 percent, but results in 25% decrease in clock frequency. Thus, we chose not to include multiplication in our current datapath, yielding 0.65ns cycle time for our PE structure.

Table 6.3 shows the cycle times required for each sub-structure in our processor as well as overall cycle time of Cyrptoraptor. As we discussed before, a full crossbar structure between levels has a significant impact on the cycle time of our processor. The functional part of a full row accounts for only two third of the overall cycle time. Even though one full row, which is a merged version of one PE and Connection row, requires 0.95ns, the register file connections and capacitance effects of all connections yields us to have 1.00ns as the cycle time of our processor.

Table 6.3: The cycle time of sub-modules in Cryptoraptor

	Cycle time (ns)
PE	0.63
PE Row	0.63
Connection Row	0.47
Pipeline Register	0.09
One Full Row	0.95
Execute Tile	1.00
Cryptoraptor	1.00

Such a detailed analysis also shows the affect of synthesizing modules with "-uniquify", "-ungroup", and "-flatten" optimizations. These optimizations allow DC to flat out all design, remove module boundaries, and synthesize it as a whole block. Thus, it makes impossible to predict the cycle time of larger modules by just combining the cycle time of smaller ones. We will discuss similar affects on area and power analysis in later sections.

6.3 Area Analysis

We analyzed area requirements of each functional unit and sub-modules in our processor. Even though we used registers while doing timing analysis, we used the CACTI 6.5 Memory Model [153] to get more accurate area analysis because compiling memory blocks using registers requires unreasonably large area, resulting in misleading information. Table 6.4 shows the difference between compiling memory blocks with DC and CACTI.

Ideally, the memory blocks should be compiled using Memory Compiler and integrated into DC. However, we tried to estimate the area requirements

Table 6.4: The area comparison between Design Compiler and CACTI

	Entry size	**Design Compiler** (mm^2)	**CACTI** (mm^2)	**Ratio**
SBOX	256	0.1582	0.0017	93.0
TBOX	1024	0.5731	0.0031	184.8

as accurate as possible by using CACTI.

Table 6.5: The area of functional units in PE

	Area (mm^2)	**Area with byte-rotator** (mm^2)
AU	0.0107	0.0117
LOU	0.0080	0.0087
TLU	0.0110	0.0115
SRU	0.0093	-
PEU	0.0156	-
Byte rotator	0.0011	-
Multiplication	0.0075	-

Table 6.5 shows the area requirements of each functional unit in PE and their combined versions with byte rotators. As opposed to discussion that we did for timing analysis, the functional unit that has the largest area requirement is PEU instead of TLU. On the other hand, Table 6.6 summarizes the effects of our bundles (arithmetic operations with XOR, and shift/rotate operations with logical operation blocks) on the area requirements of AU and SRU. Even though the improved functionality increases the area, we concentrate more on performance and flexibility for this thesis.

As seen in Table 6.5, merging functional units with byte rotator does not have a significant impact on the overall die area. Unlike its effects on

Table 6.6: The area comparison of functional units with bundles

	Area (mm^2)	**Area with byte-shifter** (mm^2)
AU	0.0035	0.0042
AU (bundled)	0.0107	0.0117
SRU	0.0063	-
SRU (bundled)	0.0093	-

cycle time, a multiplication unit does not have a serious impact on the area. However, since our main aim is to design a "high performance" cryptographic processor, its effects on cycle time are the determining cause for not having a multiplication unit in our datapath.

Table 6.7: The area of sub-modules in Cryptoraptor

	Area (mm^2)
PE	0.0344
PE Row	0.1448
Connection Row	0.0627
Pipeline Register	0.0221
One Full Row	0.2115
Execute Tile	4.5428
Register File	1.7816
Cryptoraptor	6.3244

The area requirements of each sub-structure in our processor, as well as overall cycle time of Cyrptoraptor, are summarized in Table 6.7. The functional and execution parts of our processor use 71.8 percent of overall die area. With a 256-entry register file ($1.78mm^2$), Cryptoraptor requires only $6.32mm^2$, which is approximately 34X and 78X smaller than existing CPUs and GPUs, respectively.

6.4 Power Analysis

Since comparing the power usage of our processor with AES-specific cores or existing GPPs would be unfair, we provide detailed analysis for each functional unit instead. Besides fairness, providing such comparison is quite hard, if not impossible, due to the lack of data provided by chip vendors and in research papers.

We believe that a detailed analysis on power usage of functional units may give more insight to cryptographic algorithm developers about hardware implementation of their algorithms. Combination of timing, area, and power analysis enables both cryptographic algorithm developers to optimize their designs in the light of these trade-offs and hardware designers to choose more appropriate hardware structures while implementing the algorithms.

Table 6.8: The power usage comparison for memory blocks

	Internal Power (mW)	**Switching Power** (mW)	**Leakage Power** (mW)	**Dynamic Power** (mW)
SBOX (DC)	260.4114	7.6365	0.83	268.8782
SBOX (CACTI)	11.62	$3.48\text{x}10^{-6}$	0.15	11.77
TBOX (DC)	774.8735	20.2947	3.03	798.2086
TBOX (CACTI)	14.41	$1.31\text{x}10^{-5}$	0.18	14.59

While providing power dissipations, we applied the same strategy that we did for the area in the previous section. Thus, we combined the power usage of functional parts from DC and memory blocks from CACTI with an additional 10% for error to derive power usage of our functional units, sub-modules, and our processor. The power usage difference for memory blocks

synthesized using DC and CACTI is summarized in Table 6.8.

Table 6.9: The power usage of functional units in PE

	Internal Power (mW)	**Switching Power** (mW)	**Leakage Power** (mW)	**Dynamic Power** (mW)
AU	3.97	3.07	0.0479	7.08
LOU	2.49	2.30	0.0269	4.81
TLU	51.54	1.35	0.0399	53.55
SRU	1.90	1.84	0.0303	3.78
PEU	14.35	4.07	0.0748	18.50
AU (with BR)	4.06	3.22	0.0513	7.33
LOU (with BR)	2.62	2.41	0.0282	5.06
TLU (with BR)	51.75	1.48	0.0427	53.89
Byte rotator	0.58	0.52	0.0021	1.10
Multiplication	4.09	3.06	0.0374	7.18

Synopsys Design Compiler divides the power dissipated in a design into four categories: leakage, dynamic, internal, and switching. The leakage (static) power is the power consumed by a gate when it is not switching, and caused by currents that flow through the transistors even when they are turned off. Dynamic power is the power dissipated when the circuit is active i.e. performing some function. Dynamic power consists of internal and switching components. Internal power is consumed within a cell for charging and discharging internal cell capacitances, while switching power is dissipated when charging and discharging the load capacitance at the cell output.

Table 6.9 provides a detailed power usage analysis of each functional unit in PE and their combined versions with byte rotator. Our power analysis provides not only total dynamic power per functional unit, but also internal, switching, and leakage powers separately. As shown in Table 6.9, TLU requires

Table 6.10: The power usage comparison of functional units with bundles

	Internal Power (mW)	**Switching Power** (mW)	**Leakage Power** (mW)	**Dynamic Power** (mW)
AU	1.23	0.87	0.0147	2.12
AU (bundled)	3.97	3.07	0.0479	7.08
SRU	1.86	1.82	0.0220	3.71
SRU (bundled)	1.90	1.84	0.0303	3.78

the highest dynamic power by far with $53.5mW$ due to its memory blocks. However, there is also a huge gap between the power usage of PEU and other functional units. This finding clearly shows the trade-off between the flexibility and power usage. We also analyzed the effects of our bundled operations to the power usage of our functional units. Table 6.10 provides a detailed analysis on the effects of these bundles on the power usage. Even though it seems like the bundled arithmetic and XOR operation structure has a great impact on power usage of AU, it is negligible compared to the power usage of other functional units and the overall system.

Table 6.11: The power usage of modules in Cryptoraptor

	Internal Power (mW)	**Switching Power** (mW)	**Leakage Power** (mW)	**Dynamic Power** (mW)
PE	63.42	4.21	0.15	68.40
PE Row	254.37	18.35	0.60	277.17
Connection Row	1.58	0.86	0.17	2.60
Pipeline Register	288.74	3.71	$1.20\text{x}10^{-7}$	292.57
One Full Row	336.20	19.35	0.96	360.37
Execute Tile	5229.32	303.75	19.96	5602.65
Register File	573.49	$4.68\text{x}10^{-5}$	3.40	576.89
Cryptoraptor	5802.80	303.75	19.96	6207.04

The detailed power usage of each sub-structure in our processor are summarized in Table 6.11. Our studies shows that CR between levels does not have significant impact on power usage of the processor and majority of total power is used by the functional parts of our processor. We realized that DC and optimization flags make power usage unpredictable and more complicated than a simple calculation of combining sub-modules. For example; one PE row, CR, and pipeline registers require $277.17mW$, $2.60mW$, and $292.57mW$ dynamic power respectively when they are synthesized separately. However, DC reports $360.37mW$ dynamic power usage for one Full Row (which consists of one pair of a PE row and CR with pipeline registers), where only $119.39mW$ is attributed to pipeline registers.

Table 6.12: Power usage comparison of GPPs

	Lith. (nm)	**Area** (mm^2)	**Max Power** (W)	**Power per area** (W/mm^2)
Core i7-2600K	32nm	216	95	0.4398
Core i7-980X	32nm	239	130	0.5439
VIA C7	90nm	30	20	0.6666
GTX 260	55nm	576	182	0.3159
GTX 285	55nm	490	204	0.4163
GTX 295	55nm	470	289	0.6148
Tesla C2050	40nm	539	238	0.4415
Cryptoraptor	45nm	6.32	6207.04	0.9777

Even though we avoid to compare the power consumption of our processor with existing GPPs, we would like to analyze power per area (W/mm^2) metric of our processor (shown in Table 6.12). In previous chapters, we mentioned that a large fraction of energy dissipation can be attributed to the

instruction supply in an in-order RISC machine [92]. Table 6.11 shows that 90.6 percent of total power is consumed by functional and execution part of our processor. Therefore, we can state that using a compact finite state machine representation for control flow in algorithms increase energy efficiency by simplifying the front-end structure of a traditional processor. We can also improve this idea by claiming such control mechanism reduces not only the energy consumption but also area requirements of the processor.

6.5 Performance Analysis

Since we are unique in supporting a wide range of cryptographic algorithms on a single cryptographic processor, there is no directly comparable related work. As discussed earlier, most of the existing solutions are hard-coded and hand-tuned, usually not even parameterized, for a particular cryptographic algorithm.

Even though we aim to achieve high throughput for all algorithms, we can only compare our performance on AES due to the lack of published results for any other algorithm. The main purpose of our performance comparisons is not to show that we achieve the highest throughput but to emphasize that our proposed architecture has competitive performance results compared to existing solutions. However, we should re-emphasize that we compare our flexible processor against to the designs that are hand-tuned and specialized specifically for AES.

The one of the biggest challenges on performance analysis is the per-

formance data provided by chip vendors and in research papers. Hardly any company provides detailed performance data nor an insight into the architecture itself such as chip architecture and pipeline depth. The performance numbers and architecture descriptions are not complete, even in resesearch papers; making comparisons even harder.

Another obstacle for having a fair comparison is the selection of base settings for AES. Our literature study on AES shows that there is no uniformity among research papers while presenting the performance results. The first challenge is to find a base performance metric to compare a variety of solutions effectively. Among many other alternatives used in literature such as bytes per cycle and bytes per area, we choose our throughput metric as gigabit per second (*Gbps*) to incorporate characteristic of the algorithm (block size), hardware design performance (frequency), and hardware design architecture (latency) into a single metric. We studied each proposed solution in detail to generate their throughput metrics as *Gbps* if they have not presented in that form.

Our initial studies also indicate that there is no agreed mode of operation in the literature to study the performance of AES solutions. The choice of feedback (CBC, CFB, and OFB) or non-feedback (ECB and CTR) operation mode plays a crucial role on throughput of the design. In general, AES cores are designed to achieve high throughput on non-feedback modes with a deep pipeline structure and have a dramatic decrease in performance on feedback modes.

Even though we believe that feedback mode throughput of an AES core provides a better insight about the performance of a design, we find it unfair to normalize the performance results of a core optimized for non-feedback mode to feedback ones. For that reason, we present throughput of existing solutions for both as well as their clock frequencies, lithographies, and pipeline depths. We prefer not to scale the presented results to a particular technology or device to avoid unrealistic advantages/disadvantages created by mathematical formulas used in [132] and [126]. A comprehensive comparison of the state-of-the-art hardware implementations of AES is summarized in Tables 6.13, 6.14, and 6.15 for ASIC, FPGA, and GPP solutions, respectively.

Table 6.13: AES Performance comparison of ASIC solutions

	Lith. (nm)	**Parallel Stream**	**# of Cycles**	**Freq.** (MHz)	**CBC Through-put** ($Gbps$)	**CTR Through-put** ($Gbps$)
Saravanan [187]	180nm	1	80	333.0	0.53	10.66
Amphion [7]	180nm	1	1	200.0	25.60	25.60
EnSilica eSi-8110 [69]	65nm	1	11	500.0	5.82	64.00
Mathew [139]	45nm	4	20	2615.0	16.74	66.94
Hodjat [99]	180nm	1	41	606.0	1.89	77.57
Swankoski [205]	160nm	1	50	680.3	1.74	87.07
Ip Cores, Inc [105]	90nm	1	10	824.0	10.55	105.47
Morioka [151]	130nm	1	10	909.0	11.64	116.35
Ali [5]	180nm	1	21	1015.0	6.19	129.92
Liu [132]	65nm	1	152	1210.0	1.02	153.70
Our processor	45nm	1	20	1000.0	6.40	128.00

Table 6.13 shows that an ASIC implementation of our cryptographic processor achieves a competitive throughput result compared to AES-specific ASIC cores. With its outer-round pipelined structure and highly tuned data-path, Morioka's AES-only implementation [151] on an old $130nm$ technology

achieves a high throughput on both AES-CBC and AES-CTR, 11.64$Gbps$ and 116.35$Gbps$ respectively. Besides its reconfigurability, our processor achieves peak AES throughputs of 6.40$Gbps$ and 128$Gbps$, respectively. Since the number of pipeline stages has a negative impact on CBC throughput, our throughput is roughly half of Morioka's.

With its higher frequency, 152 pipeline stages, and its many core structure, Liu's AES processor [132] achieves the highest AES-128-CTR throughput of 154.88$Gbps$. Due to its very deep pipeline, however, it is not able to provide high performance on feedback modes. Liu's AES processor ($6.63mm^2$) only supports AES while our processor ($6.32mm^2$) is capable of supporting a wide range of algorithms. Our processor design achieves similar throughput per area as Liu's many core solution, $20.25Gbps/mm^2$ and $23.18Gbps/mm^2$ respectively.

Amphion's high performance AES core [7] running at $200MHz$ achieves the highest AES-CBC throughput by processing ten rounds in a single cycle. Even though its internal structure is not publicly available, it is obvious that the core cannot achieve a high CBC throughput due its low clock frequency. The 10-stage pipelined $824MHz$ AES core on 90nm technology introduced by Ip Cores presents a balanced performance on both CBC and CTR mode. Both designs clearly show the importance of the balance between the pipeline depth and the clock frequency. Even though we are slower then the highest throughput AES-specific ASIC cores, our highly configurable cryptographic processor is competitive to them.

Table 6.14: AES Performance comparison of FPGA solutions

	FPGA Family	**Lith.** (nm)	**Num. of Cycles**	**Freq.** (MHz)	**CBC Throughput** ($Gbps$)	**CTR Throughput** ($Gbps$)
Jarvinen [109]	Virtex 2	130nm	41	139.10	0.43	17.80
Swankoski [204]	Virtex 2	130nm	10	147.00	1.88	18.82
Hodjat [100]	Virtex 2	130nm	41	168.30	0.53	21.54
Zhang [228]	Virtex 1	130nm	70	168.40	0.31	21.56
Good [87]	Spartan 3	90nm	70	196.10	0.36	25.10
Iyer [107]	Virtex 2	130nm	50	206.84	0.53	26.48
Good [86]	Virtex 2	130nm	240	222.90	0.12	28.53
Rizk [181]	Virtex 4	90nm	20	223.00	1.43	28.54
Yoo [227]	Virtex 2	130nm	30	232.60	0.99	29.77
Good [86]	Virtex 3	90nm	120	240.90	0.26	30.84
Fan [70]	Virtex 2	90nm	50	250.00	0.64	32.00
EnSilica [69]	Virtex 6	40nm	11	275.00	3.20	35.20
Ali [5]	Stratix II GX	180nm	21	282.50	1.72	36.16
Wang [212]	Virtex 6	40nm	66	344.12	0.67	44.05
Mercoratech [145]	Virtex 6	40nm	10	357.00	4.57	45.70
Deshpande [186]	Spartan 6	45nm	80	430.00	0.69	55.04
Hossain [102]	Stratix II GX	90nm	23	450.05	2.50	57.61
Swankoski [205]	Virtex 4	90nm	50	519.18	1.33	66.46
Soliman [198, 199]	Virtex 5	65nm	40	557.00	1.78	71.30
Qu [171]	Virtex 5	65nm	10	576.07	7.37	73.74
Chen [49]	Virtex 4	90nm	10	645.70	8.26	82.65
Our processor	Virtex 6	40nm	20	203.80	1.30	26.09

There exists a rich literature on high performance AES hardware architectures targeting FPGAs as summarized in Table 6.14. The AES core proposed by Chen [49] adapts outer-round pipelining scheme and is stated to achieve a very high frequency of $645.70MHz$ on a Xilinx Virtex-4 FPGA, producing the highest AES-CTR throughput of $82.65Gbps$. We tried to replicate the work using the source code provided by authors and synthesized to same FPGA family with the highest speed grade with all optimizations on, using ISE Design Suite 14.6. However, the highest clock frequency that

we were able to produce was 284.43MHz, yielding 36.41$Gbps$ as opposed to 82.65$Gbps$. Similarly, 10-staged AES core reported by Qu [171] claims an astoundingly high clock frequency of 576.07MHz on Virtex-5 while Soliman's solution [199] achieves 557MHz clock frequency on the same FPGA even with its deep inner-round pipelined structure.

Table 6.15: AES Performance comparison of GPP solutions

	Architecture	**Lith.** (nm)	**Config.** ($core/warp$ **x** $thread$)	**Freq.** (MHz)	**Throughput** ($Gbps$)	**Area** (mm^2)	**Throughput per area** ($Gbps/mm^2$)
Nishikawa [158]	Core i7-2600K	32nm	1 x 1	3400	1.90	216	0.009
Nishikawa [158]	Core i7-2600K	32nm	4 x 8	3400	7.50	216	0.035
Nishikawa [158]	Core i7-2600K	32nm	1 x 1	3400	25.10	216	0.116
Nishikawa [158]	Core i7-2600K	32nm	4 x 4	3400	44.20	216	0.205
Akdemir [4]	Core i7-980X	32nm	1 x 1	3300	6.30	239	0.026
Akdemir [4]	Core i7-980X	32nm	6 x 12	3300	72.30	239	0.303
VIA Tech. [211]	VIA C7	90nm	1 x 1	2000	25.00	30	0.833
Zola [161]	GTX 260	55nm	27 x 256	1242	30.00	576	0.052
Iwai [106]	GTX 285	55nm	60 x 512	1500	35.20	490	0.072
Nishikawa [158]	GTX 285	55nm	60 x 512	1242	35.20	490	0.072
Nishikawa [158]	Tesla C2050	40nm	28 x 512	1150	48.60	539	0.090
Bos [16]	GTX 295	55nm	120 x 512	1240	59.60	470	0.127
Our processor	ASIC	45nm	1 x 1	1000	128.00	6.32	20.253

On the other hand, even commercial high performance AES cores introduced by EnSilica [69] and Mercoratech [145] achieve significantly lower clock frequencies on a faster FPGA family with the same number of pipeline stages. Table 6.14 clearly shows the impact of the number of pipeline stages on AES-CTR throughput. Despite their higher frequency, while most of the AES cores targeted FPGA has poor performance in feedback mode, they achieve very high throughput on non-feedback mode due to their deep pipelines rang-

ing from 40 to 240 stages. The performance results of FPGA solutions suggest that outer-round pipelined architecture yields better overall performance for AES by balancing the pipeline depth and the clock frequency of the architecture. Because our design is tuned for an ASIC implementation, it contains an aggressive connection network and excessive use of multiplexers. Even though it only runs at $203.8MHz$ it still achieves reasonable throughput of $1.30Gpbs$ AES-CBC and $27.31Gbps$.

Table 6.16: Performance summary of algorithms on Cryptoraptor

	Parallel Stream	**# of Round**	**Block size** ($bits$)	**# of Cycles**	**CBC Throughput** ($Gbps$)	**CTR Throughput** ($Gbps$)
Blowfish	4	16	64	48	5.33	85.33
Camellia	2	16	128	73	3.51	64.00
CAST-128	4	16	64	80	3.20	64.00
DES	2	16	64	48	2.67	42.67
GOST	4	32	64	98	2.61	51.20
Kasumi	1	6	64	64	1.00	16.00
RC5	4	12	64	48	5.33	85.33
SEED	4	16	128	152	0.84	16.00
Twofish	2	16	128	80	3.20	64.00
RC4	4	4	32	32	4.00	-
Phelix	2	1	32	10	6.40	-
MD4	2	48	128	144	1.78	-
MD5	2	64	512	254	4.03	-
SHA-1	4	80	512	225	4.55	-
SHA-2	1	64	512	320	1.60	-

With their highly parallel structures and very high clock frequencies, current generation CPU and GPU solutions also generate very competitive throughput results up to $72.30Gbps$ and $59.60Gbps$ respectively. It is apparent that increasing number of threads allow to hide pipeline latency in GPPs, yielding very high throughputs. However, current generation GPPs require

a very large chip footprint up to $539mm^2$. Even though we haven't put any effort to minimize the required area of our processor, we achieve one to three order of magnitude higher throughput per area than commercial GPPs as shown in Table 6.15. We couldn't present a similar analysis for AES cores targeting ASIC and FPGA due to missing or inconsistent area constraints in the reference papers.

As we mentioned earlier, there are not many studies in the literature on hardware implementation of other algorithms that we mapped. For that reason, we are unable to provide such a comparison. However, the performance of other algorithms on our processor is summarized in Table 6.16.

6.6 Resource Utilization

To provide a high degree of flexibility, we introduced some redundant operations, units, and connections in our processor. Therefore, not all algorithms can utilize all available resources in Crytoraptor. Table 6.17 summarizes the resource utilization of the algorithms that we mapped onto Cryptoraptor, and Table 6.18 compares the resource utilization of our manual mapping and toolchain. The following chapter discusses the reasons of this difference. The utilization analysis includes only functional units and PEs that are processing some operations; thus, operand forwarding is not included in resource utilization.

Our analysis shows that, even though the average PE utilization is pretty high, utilization of individual FUs is significantly low on both manual

Table 6.17: Resource utilization summary of mapped algorithms on Cryptoraptor

	Parallel Stream	**LOU**	**TLU**	**AU**	**SRU**	**PEU**	**PE**	**Way per Stream**	**Max. Connection**
AES	1	50%	0%	50%	0%	0%	100%	4	4
Blowfish	4	33%	67%	33%	0%	0%	100%	1	1
Camellia	2	71%	0%	25%	3%	0%	71%	2	2
CAST-128	4	6%	54%	20%	20%	0%	100%	1	1
DES	2	0%	0%	33%	0%	50%	83%	2	2
GOST	4	34%	33%	33%	33%	0%	100%	1	1
Kasumi	1	25%	9%	20%	0%	5%	48%	4	4
RC5	4	0%	50%	0%	50%	0%	100%	1	1
SEED	1	14%	8%	32%	0%	0%	50%	4	4
Twofish	2	21%	40%	40%	20%	0%	81%	2	2
RC4	4	0%	38%	50%	0%	0%	100%	1	1
Phelix	2	0%	70%	0%	80%	20%	100%	2	2
MD4	2	22%	50%	0%	17%	0%	73%	2	1
MD5	2	19%	63%	0%	13%	0%	85%	2	1
SHA-1	2	18%	75%	0%	28%	0%	99%	2	2
SHA-2	1	25%	35%	0%	30%	0%	50%	3	3
Average	2.38	21%	37%	21%	18%	5%	84%	2	2

Table 6.18: Resource utilization summary of mapped algorithms on Cryptoraptor

	Manual mapping						**Automated mapping**					
	LOU	**TLU**	**AU**	**SRU**	**PEU**	**PE**	**LOU**	**TLU**	**AU**	**SRU**	**PEU**	**PE**
AES	50%	0%	50%	0%	0%	100%	50%	0%	50%	0%	0%	100%
Blowfish	0%	67%	33%	0%	0%	100%	33%	67%	33%	0%	0%	100%
Camellia	80%	0%	20%	3%	0%	60%	71%	0%	25%	3%	0%	71%
CAST128	6%	54%	20%	20%	0%	100%	6%	54%	20%	20%	0%	100%
DES	0%	0%	17%	0%	50%	50%	0%	0%	33%	0%	50%	83%
GOST	34%	33%	33%	33%	0%	100%	34%	33%	33%	33%	0%	100%
Kasumi	25%	0%	20%	9%	5%	50%	25%	9%	20%	0%	5%	48%
RC5	0%	50%	0%	50%	0%	100%	0%	50%	0%	50%	0%	100%
SEED	13%	8%	30%	0%	0%	50%	14%	8%	32%	0%	0%	50%
Twofish	20%	40%	30%	20%	0%	90%	21%	40%	40%	20%	0%	81%
RC4	0%	38%	50%	0%	0%	100%	0%	38%	50%	0%	0%	100%
Phelix	0%	70%	0%	80%	20%	100%	0%	70%	0%	80%	20%	100%
MD4	17%	50%	0%	17%	0%	66%	22%	50%	0%	17%	0%	73%
MD5	12%	50%	0%	12%	0%	62%	19%	63%	0%	13%	0%	85%
SHA-1	17%	83%	0%	33%	0%	100%	18%	75%	0%	28%	0%	99%
SHA-2	20%	40%	0%	30%	0%	55%	25%	35%	0%	30%	0%	50%
Average:	18%	36%	19%	18%	5%	80%	21%	37%	21%	18%	5%	84%

and automated mappings. The redundancy that we added for the flexibility and the unpredictability of future algorithms increase area and power consumption significantly. The average utilization of FUs and PE is summarized in Figure 6.1 with minimum, maximum values as well as 25% and 75% distributions around the median.

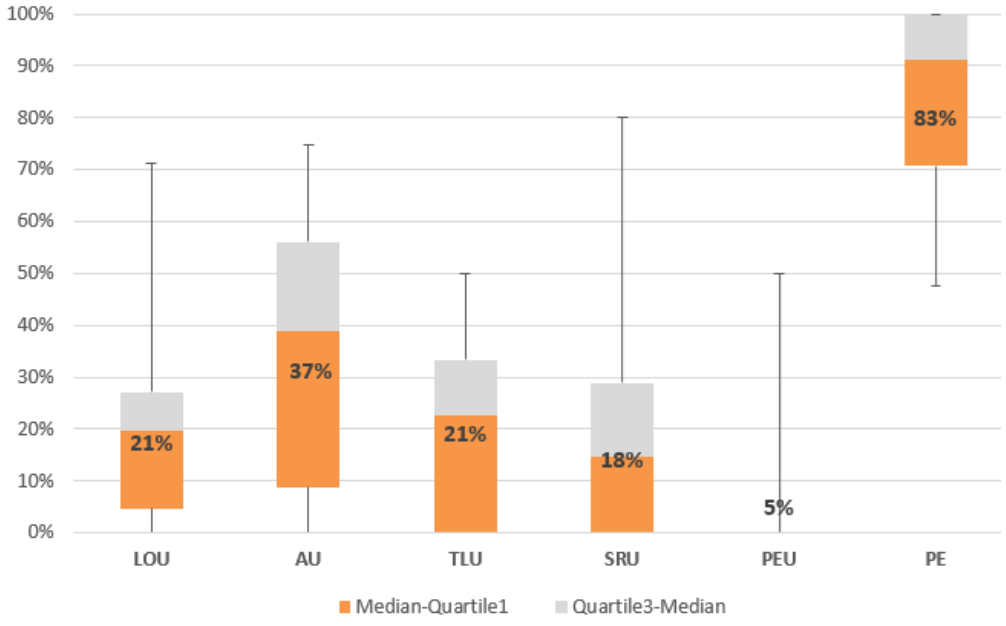


Figure 6.1: The utilization summary

6.7 Current Algorithm Coverage

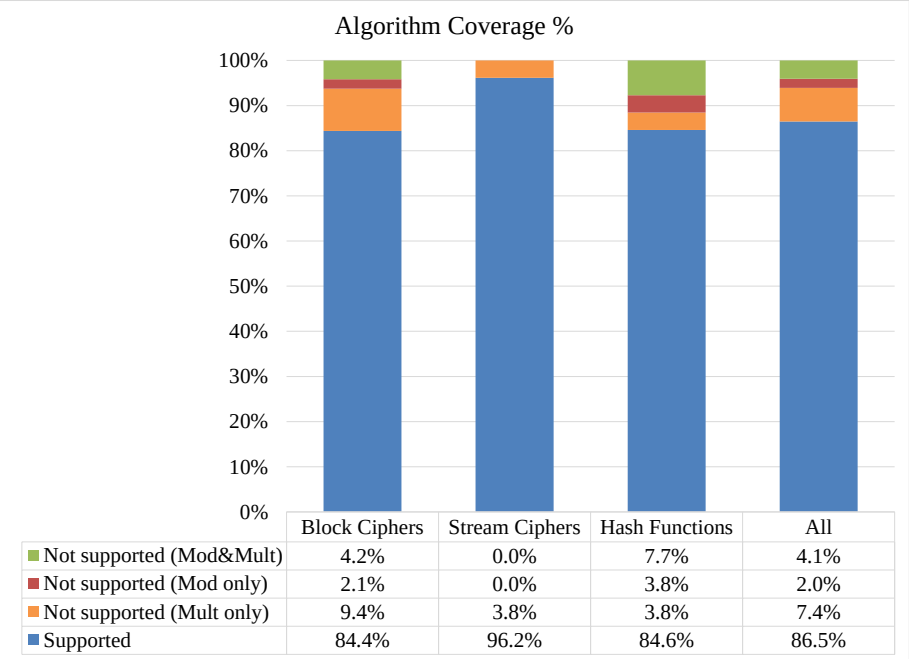
With the proposed cryptographic processor, we aim to support a wide range of existing cryptographic algorithms and future standards. Therefore, it is important to analyze the ratio of algorithms that can be supported by our processor. Table 6.19 summarizes the current algorithm coverage of our

processor. Even though we did not implement all algorithms due to excessive amount of time required to do so, our initial studies show that 86.5 percent of the algorithms that we analyzed is supported by current structure of our processor while 13.5% require additional hardware or logic to be supported.

Table 6.19: The current coverage of cryptographic algorithms

	Requires additional logic to be supported	**Supported by current structure**
Block Ciphers	15.6%	84.4%
Stream Ciphers	3.8%	96.2%
Hash Functions	15.4%	84.6%
All	13.5%	86.5%

We also analyzed the reason behind not being able to support all existing algorithms with the current structure. Our analysis shows that the lack of modular arithmetic support for arbitrary choice of modulus and the lack of dedicated multiplication units prevent us to support all algorithms. The impact of modular arithmetic and multiplication support on the algorithm coverage is illustrated in Figure 6.2. Our initial studies show that there are a few algorithms requiring both modular arithmetic and multiplication in their datapath. Therefore, extending the current structure with modular arithmetic or multiplication will not enable us to cover all algorithms. However, multiplication unit can be a good candidate for next hardware improvement since it will increase the algorithm coverage by 8.1%. To support all existing algorithms, we need to add multiplication operation as a functionality and



	Block Ciphers	Stream Ciphers	Hash Functions	All
Not supported (Mod&Mult)	4.2%	0.0%	7.7%	4.1%
Not supported (Mod only)	2.1%	0.0%	3.8%	2.0%
Not supported (Mult only)	9.4%	3.8%	3.8%	7.4%
Supported	84.4%	96.2%	84.6%	86.5%

Figure 6.2: The distribution of supported and non-supported algorithms based on limitations

integrate the flexibility of supporting any value of modular base to our AU structure.

6.8 Limitations

One of the main limitation of our current design is the lack of dedicated multiplication unit which might be useful for some crypto-systems such as IDEA and RC6. However, only 17 out of 148 cryptographic algorithms (13 block ciphers, 1 stream cipher, and 3 cryptographic hash functions) require multiplication. Despite the fact that the number of algorithms using multi-

plication is very limited, and multiplication can be realized with a set of shift operations and a sequence of additions, having a dedicated multiplication unit may speed up the algorithms that are heavily depending on multiplication and increase the flexibility of our processor for future algorithms.

Even though the most of the modular arithmetic operations can be implemented using existing functional units, current processor design can be extended to support varying and unorthodox modulo bases. Extending existing hardware for modular arithmetic may increase the algorithm coverage of our processor by 2 percent while having both multiplication unit and modular arithmetic support may increase up to 13.5 percent.

The other major limitation is limited addressing structure of our TLU. As we mentioned earlier, TLU is capable of processing one lookup with up to 10-bit addressing or four parallel 8-bit lookup. We designed our parallel lookup structure and the upper limit for addressing by solely analyzing existing algorithms. Since it is impossible to predict the requirements of future standards in advance, this structure and limitations may or may not block efficient implementation of future algorithms. Whereas, we believe that our highly flexible Connection Row structure and PEU will ease the implementation of any algorithm that does not easily fit into current table lookup limitations. In fact, table lookup operation with more than 10-bit addressing can be implemented by separating the table manually or with the help of a tool, and applying a sequence of lookup operations instead. However, providing a better flexibility within TLU may allow us to achieve high performance even on algorithms with

higher requirements. We leave further study on improving existing TLU and other functional units as future work.

Due to our initial target application set for our processor, public key encryption and elliptic curve cryptography are beyond the scope of this work.

We believe that after solving current limitations of our design, we may have a complete, flexible and high throughput cryptographic processor that can support a wide range, if not all, of symmetric key encryption algorithms and cryptographic hash functions, and ideally public key cryptography as well.

Chapter 7

Cryptographic Algorithm Mapping

We mapped a total of 16 block ciphers, stream ciphers and cryptographic hash functions to verify its flexibility and reconfigurability of our cryptographic processor. During mapping process, we focused on encryption only for each algorithm with the configurations widely used in literature or suggested by their designers.

As we did for our algorithm analysis phase, we focused on selecting algorithms based on widely used security protocols (IPsec, TLS/SSL, WTLS, SSH, S/MIME and OpenPGP) and cryptographic libraries (OpenSSL and GNU Crypto). We also mapped some algorithms that are not in existing protocols and libraries to stress the flexibility of Cryptoraptor. The list of algorithms that we mapped, the security protocols and cryptographic libraries that they are included, and the rationale behind including these 16 algorithms in the mapping process can be found in Table 7.1.

Table 7.1: Algorithm summary and selection for mapping process

	Security Protocols	**Cryptographic Libraries**	**Speciality**
AES [156]	Almost all	Almost all	Most widely used algorithm, stresses maximum parallel lookup table
Blowfish [188]	IPsec	OpenSSL	Base structure for some block ciphers, example Arithmetic-XOR pattern
Camellia [10]	IPsec, TLS	Crypto++, Cryptospecs, OpenSSL	Example algorithm that requires special attention to achieve high performance, example byte-wise rotator
CAST-128 [2]	IPsec, PGP	Crypto++, Cryptospecs, OpenSSL, GnuPG	Base structure for some block ciphers, example changing round structures, and Arithmetic-XOR pattern
DES [1]	IPsec, SSL, TLS	Almost all	Example bit permutation and unorthodox operation width
GOST [169]	-	Crypto++, Cryptospecs	Doesn't fit well, example Shift-Logic pattern
Kasumi [141]	as A5/3 in UMTS, GSM, GPRS	Cryptospecs	Data dependent and fairly complex structure due to its unbalanced rounds and unorthodox table sizes (7 and 9-bit addressing)
RC5 [178]	S/MIME	Crypto++, Cryptospecs, OpenSSL	Good on software, doesn't fit well in Cryptoraptor
SEED [128]	CMS, IPsec, SSL, S/MIME, TLS	Crypto++, Cryptospecs, OpenSSL	Recursive round structure, example algorithm that requires special attention to achieve high performance
Twofish [189]	OpenPGP	GnuPG	Complex structure and example operation patterns
RC4 [213]	TLS, WEP, WPA	Crypto++, GnuPG, Cryptospecs, OpenSSL	Most widely used stream cipher, table updates, register file usage
Phelix [217]	-	-	Very complex structure, use of patterns
MD4 [176]	PGP, S/MIME	Crypto++, Cryptospecs, OpenSSL	Base structure for several hash functions, stresses LOU structure, changing round structures
MD5 [175]	IPsec, NTLM, SSL, S/MIME, TLS	Almost all	Base structure for several hash functions, stresses LOU structure, changing round structures
SHA-1 [64]	IPsec, PGP, SSH, SSL, S/MIME, TLS	Almost all	Most widely used hash function, changing round structures
SHA-2 [74]	Bitcoin, IPsec, PGP, SSH, SSL, S/MIME, TLS	Almost all	Complex round structure, replacing SHA-1

To enable easier implementation, more efficient and optimized algorithm mapping, and higher throughput, a simple cryptography assembly language is introduced (Table 7.2). Besides operation primitives, the proposed language allows users to define and use variables, arrays, tables, constants,

and permutation tables. Instructions operate on 32-bit immediate values as well. Since the multiplication and modular arithmetic with an arbitrary modulus are currently not supported, they are not included in the language. We also implemented a custom toolchain which is fully aware of the underlying processor architecture and optimizes the input mapping for high throughput.

Table 7.2: Instruction List

Operation Class	**Instructions**
AU	ADD, ADD8, ADD16, ADDi, ADD8i, ADD16i, SUB, SUB8, SUB16, SUBi, SUB8i, SUB16i
LOU	AND, OR, XOR, NOT, ANDi, ORi, XORi
TLU	SBOX, SBOX_M, SBOX_P, STR
SRU	SHR, SHL, SHRi, SHLi, ROTR, ROTL, ROTRi, ROTLi, BROTR, BROTL
PEU	PERM, PERM32_64, PERM64_32, PERM64_64
Helper	REPEAT, MOVE, SWAP

The toolchain unrolls the loops, in which the round operations are defined, generates a dataflow graph, and optimizes the operation sequence for underlying hardware. It issues the operations to available FUs as soon as their operands are ready. Besides its own optimization process, the assembly language and toolchain also allow users to hand-tune their implementation. Since it finds implicit parallelism better, the automated toolchain enables achieving throughput and resource utilization that are greater than or equal to well-studied hand-based mapping.

Our experience leads us to believe that mapping an algorithm is straightforward as long as all of the necessary functional blocks are available. Multipli-

cation, used in 11.5% of the analyzed algorithms, is a notable exception. Our algorithm analysis suggests that 86.5% of studied algorithms can be mapped efficiently onto current Cryptoraptor architecture. Since even multiplication can be performed using existing FUs and we provided more functionality than studied algorithms require, we strongly believe that Cryptoraptor can support all existing algorithms.

7.1 Block Ciphers

Due to the large number of block ciphers in the literature and our algorithm analysis, we mapped ten block ciphers to stress the flexibility of our processor. The following sections provide brief descriptions about those algorithms and how we mapped them onto our processor. The block ciphers that we mapped are AES, Blowfish, Camellia, CAST-128, DES, GOST, Kasumi, RC5, SEED, and Twofish.

Most of the block ciphers may be characterized as the Feistel network invented by Horst Feistel in 1973 [71]. Feistel network is a transforming function structure which divides the data block into two halves and applies transformation function f on one half using sub-key derived from user's secret key. Two halves of the data block are XORed and swapped at the end of each round. The Feistel network structure may also be generalized to larger data blocks. Figure 7.1 shows the high-level block diagrams of block ciphers that use original or derived version of Feistel network.

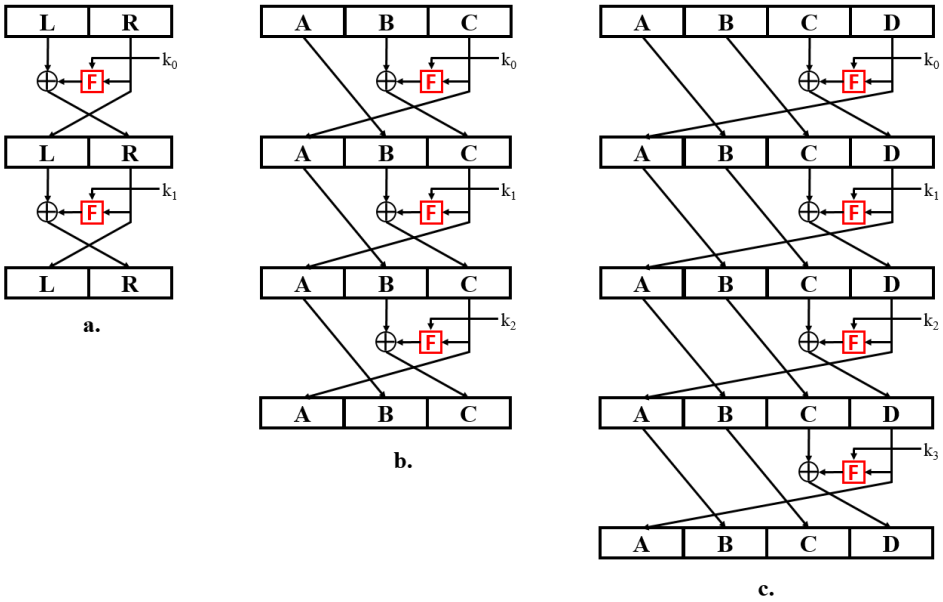


Figure 7.1: The overall structure of Feistel network and its derivations

7.1.1 Advanced Encryption Standard (AES)

AES is a NIST specification for the encryption and a widely used and studied encryption algorithm. The AES algorithm is a symmetric block cipher that processes 128-bit data blocks, called "state", using a secret key of length 128, 192, or 256 bits [58]. The cipher, which is structured as an SPN network, executes 10, 12, or 14 rounds of transformation depending on the selected key size. In this thesis, we focus our research and results on only encryption with a 128-bit key and ten rounds of operation to be consistent with other papers in the literature. In a traditional AES implementation, four operational stages are performed during each round: SubBytes, ShiftRows, MixColumns and AddRoundKey. The high-level structure of AES encryption is depicted in

Figure 7.2.

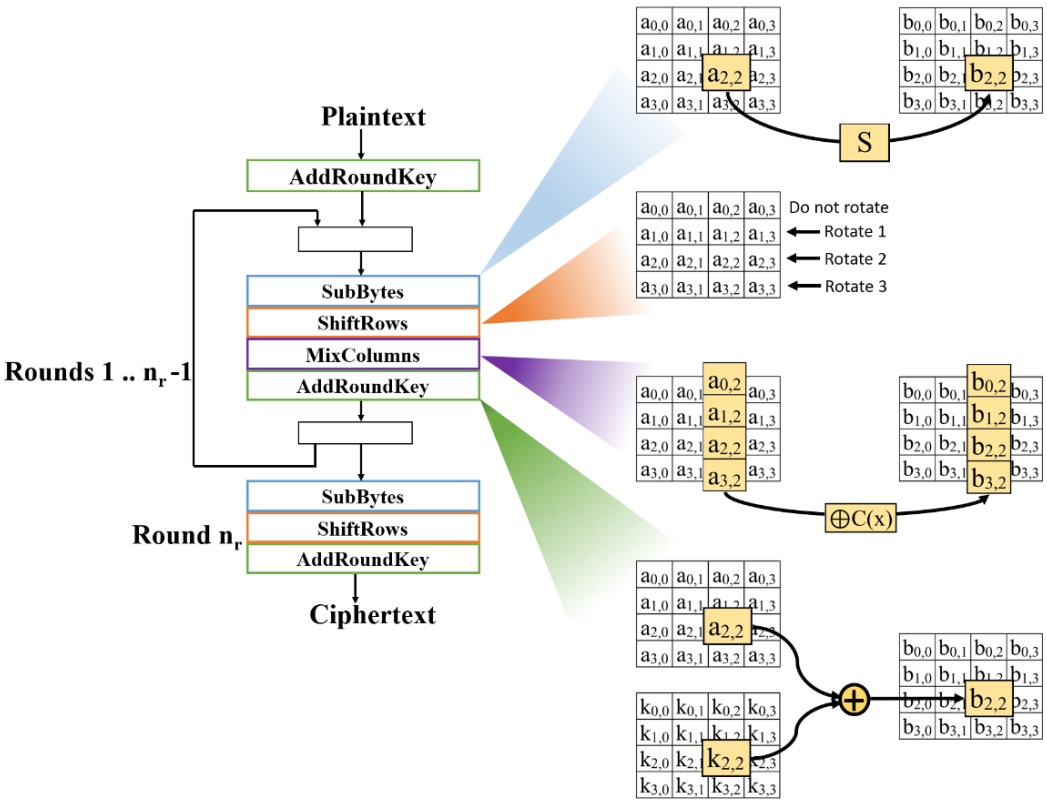


Figure 7.2: The traditional structure of AES

The SubBytes is a non-linear transformation, in which each byte from the input state is replaced by another byte according to a predefined lookup table. It can also be realized through calculation as the multiplicative inverse in the finite Galois Field (GF) on 2^8 and a bitwise affine transformation.

The ShiftRows transformation function processes the state array by circularly shifting the last three rows over different numbers of offsets.

The MixColumn transformation multiplies the columns of the data ma-

trix by a predefined matrix over GF(2^8). The column multiplication of the matrix is formulated as:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{7.1}$$

where a is the input column, a_i denotes one byte of data in the corresponding cell, and b is the output column. All multiplications and additions used in MixColumn is defined in GF and dependent upon the field polynomial given as:

$$f(x) = x^8 + x^4 + x^3 + x + 1; \tag{7.2}$$

Even though the traditional flow of the algorithm includes an explicit matrix multiplication operation in each round, there are some other realizations of this operation in the literature to get better performance or power and area efficiency. While matrix multiplication can be implemented as a set of logical operations (potentially in a pipelined structure), precalculated lookup tables are also used since one operand of the multiplication is a constant matrix.

The AddRoundKey transformation makes the relation between ciphertext and user's secret key by performing XOR operation with a set of sub keys derived from the actual key.

Besides its traditional implementation, Rijndael [58] also suggested that multiple stages of the round transformation can be combined as a single set of

table lookups, allowing very fast implementations on processors. Using these combined tables, the round transformation can be expressed as:

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}] \oplus k_j; \tag{7.3}$$

where a is the round input, $a_{i,j}$ denotes one byte of a in row i, column j, a_j denotes the column j of state a and T_0, T_1, T_2, and T_3 are the new combined lookup tables. Hence T-box implementation of AES takes only 4 table lookups and 4 XORs per column per round.

Due to the PE structure of our processor, the mapping T-box implementation onto our processor is straightforward: 16 parallel lookups in even rounds and four parallel three level XOR reductions in odd rounds. In each round, a 128-bit data block is divided into four 32-bit words, each of which are fed into TLU in four parallel PEs. In each TLU, each byte of a 32-bit input is used as an address for S-boxes to generate four 32-bit outputs. After 16 parallel table lookup operations, sixteen 32-bit outputs are routed to LOUs in the next level based on Equation 7.3. Finally, four 32-bit table lookup operation and a round key is processed through XOR tree in each LOU to generate one 32-bit word of 128-bit output. This process is repeated ten times to encrypt 128-bit plaintext; thus, AES-128 encryption process requires 20 cycles on our processor.

7.1.2 Blowfish

Blowfish is a secret-key block cipher proposed by Bruce Schneier in 1993 [188]. Blowfish operates on a block size of 64 bits while the key can be

any length from 32 bits up to 448 bits. The algorithm was proposed to be an alternative to DES and planned to be free from problems and constraints associated with other algorithms. However, it did not meet all the requirements for a new cryptographic standard at the time.

Due to the nature of the algorithm, Blowfish is only suitable for applications where the key does not change often, as a communications link or an automatic file encryptor. Even though it is not very common cryptographic algorithm, Blowfish is included in IPsec and implemented in OpenSSL.

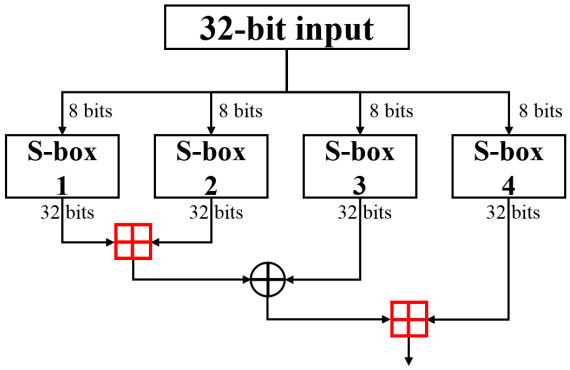


Figure 7.3: The round function of Blowfish

The algorithm iterates over a simple encryption function 16 times in Feistel network structure. The F function of Feistel network uses four different 256-entry S-boxes, XOR and addition in modulo 2^{32}. Each byte of 32-bit input is fed into four separate lookup tables as one address to produce four 32-bit outputs. The outputs of S-boxes are combined using XOR operations and an addition to generate 32-bit output as shown in Figure 7.3. The output of the round function is XORed with the second half of the previous round and

swapped its place.

Blowfish utilizes both XOR-SBOX and Arithmetic-XOR bundled operations. Due to its structure and capability of our functional units, Blowfish fits perfectly onto our processor and shows the capability of Cryptoraptor to support similar block ciphers. One round of Blowfish takes three cycles in our processor, yielding total of 48 cycles to transform 64-bit plaintext.

7.1.3 Camellia

Camellia, developed by NTT and Mitsubishi Electric Corporation, is a secret key block cipher working on 128-bit blocks of data using 128, 192, and 256-bit secret key [10]. The proposed algorithm offers the same interface specifications used in Advanced Encryption Standard. Camellia is one of the algorithms listed in TLS and IPsec, and implemented in many cryptography libraries.

The high-level structure of Camellia follows the Feistel network with either 18 or 24 rounds depending on the selected key size. The overall structure of the algorithms is shown in Figure 7.4. The round function F consists of four different 256-entry tables with 8-bit values on each entry and apply affine transformations and logical operations. As shown in Figure 7.5, the number of lookup tables can be doubled to process 64-bit data at a time.

Besides its fairly complex round structure, a logical transformation layer called "FL-function", and its inverse are applied in every six rounds. This logical transformation layer consists of a AND, OR, XOR and left rotation by

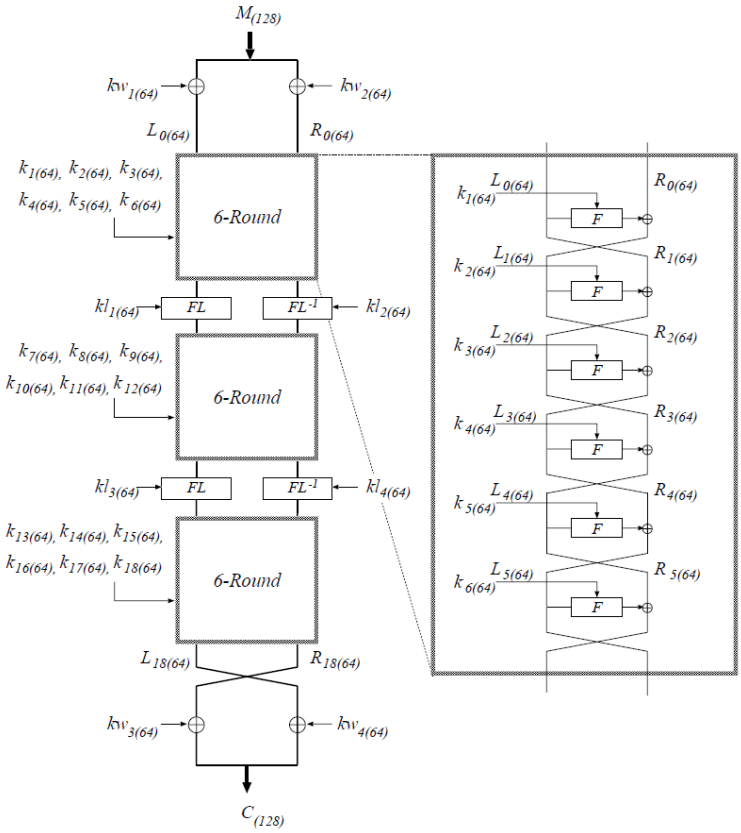


Figure 7.4: The high level structure of Camellia

1. The structure of FL-function is similar to the FL-function of Kasumi [141] that we discuss in following the section. The only difference between Kasumi and Camellia is the addition of 1-bit rotation. FL-function and FL^{-1}-function transform a 64-bit input $X_{(64)}$ to a 64-bit output $Y_{(64)}$ using 64-bit sub-key $k_{(64)}$ as follows;

$$
\begin{aligned}
&FL - function: \\
&\qquad Y_{R(32)} = ((X_{L(32)} \cap k_{L(32)}) <<< 1) \oplus X_{R(32)} \\
&\qquad Y_{L(32)} = (Y_{R(32)} \cup k_{R(32)}) \oplus X_{L(32)}
\end{aligned} \tag{7.4}
$$

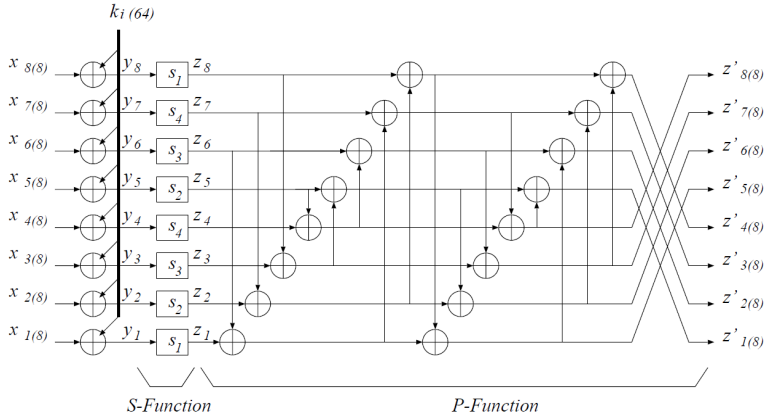


Figure 7.5: The internal structure of one Camellia round

$$FL^{-1} - function:$$
$$\begin{aligned} X_{L(32)} &= (Y_{R(32)} \cup k_{R(32)}) \oplus Y_{L(32)} \\ X_{R(32)} &= ((X_{L(32)} \cap k_{L(32)}) <<< 1) \oplus Y_{R(32)} \end{aligned} \tag{7.5}$$

Camellia is an excellent example algorithm that utilize both original and rotated versions of the output on following stages. Besides the utilization of byte shifters, it also requires changing control signal due to FL-function and FL^{-1}-function in every six rounds. Thus, it's a good example algorithm to stress the reconfigurability of our processor.

Even though the structure of Camellia algorithm seems to be fairly complex in Figure 7.4, it can be realized as 32-bit XOR operations followed by rotation instead of byte-wise XORs and complex connections among bytes of 64-bit data. If we consider operation granularity as 32-bit, then P function structure in a Camellia round becomes XOR operations between 32-bit halves of 64-bit data while applying 1-, 2-, and 3-byte rotations in each level of XOR

operation respectively.

Since FL and FL^{-1} functions consist of logical operations and a rotation, mapping them into our processor is straightforward. The longest path in each function requires three levels to perform a logical operation first, and then rotation, and finally the second logical operation. While mapping, our toolchain utilizes our independently configurable FU structure, which enables us to start to process next round as soon as one column finishes its required operations in the current round. Therefore, it enables us to start calculating FL and FL^{-1}-functions one cycle ahead. Since FL and FL^{-1} can be implemented in one cycle thanks to our SRU structure, they are processed in parallel. One round of Camellia takes five cycles in our processor, yielding total of 80 cycles to transform 128-bit plaintext.

7.1.4 CAST-128

CAST-128 is a symmetric-key block cipher that operates on 64-bit data blocks using a secret key with size of 40 to 128 bits in 8-bit increments [3]. It is a member of the CAST family of ciphers, proposed by Carlisle Adams and Stafford Tavares in 1996. CAST-128 is listed in IPsec and OpenSSL, and included in some versions of Pretty Good Privacy (PGP) and GNU Privacy Guard (GPG).

CAST-128 follows Feistel Network structure with 12 or 16-rounds depending on selected key length. Figure 7.6 summarizes the overall structure of CAST-128 round function. It consists of four 256-entry lookup tables where

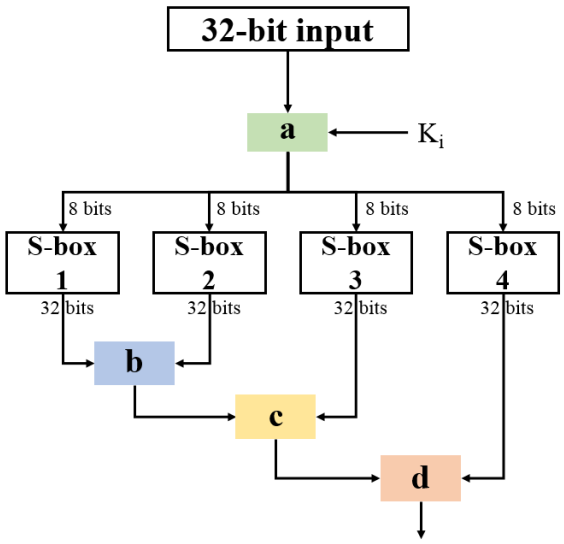


Figure 7.6: The round function template for CAST-128

each entry holds 32-bit value. The round function of CAST-128 works on 32-bit data half as input and starts with applying operation "a" using round key. After splitting the result into four 8-bit pieces, each piece is fed into a different lookup table. The output of lookup operations is combined using "b", "c", and "d" operations.

A simple way to complete the definition of the CAST-128 round function is to select all operations ("a", b", "c", and "d") as XOR operations of 32-bit quantities, although other operations or more complex structures may also be used instead. The operations used in three different rounds of traditional CAST-128 block cipher are as follows;

$$\begin{aligned} &Type\ 1: \\ &\quad I = ((Kmi + D) <<< Kri) \\ &\quad f = ((S1[I_a] \oplus S2[I_b]) - S3[I_c]) + S4[I_d] \end{aligned} \tag{7.6}$$

$$
\begin{aligned}
&Type\ 2: \\
&\quad I = ((Kmi \oplus D) <<< Kri) \\
&\quad f = ((S1[I_a] - S2[I_b]) + S3[I_c]) \oplus S4[I_d]
\end{aligned} \tag{7.7}
$$

$$
\begin{aligned}
&Type\ 3: \\
&\quad I = ((Kmi - D) <<< Kri) \\
&\quad f = ((S1[I_a] + S2[I_b]) \oplus S3[I_c]) - S4[I_d]
\end{aligned} \tag{7.8}
$$

where "D" is the data input to the round function, I_a, I_b, I_c, and I_d denotes bytes of I, $+$ and $-$ represent addition and subtraction modulo 2^{32}, $\oplus$ is bitwise XOR, and $<<<$ represents the rotation operation. Km and Kr denotes the derived sub-key and amount of rotation for each round. The f functions defined above are used in a predefined order. The function f in Type 1 is used in rounds 1, 4, 7, 10, 13, and 16 while rounds 2, 5, 8, 11, and 14 use Type 2 and rounds 3, 6, 9, 12, and 15 use Type 3.

Each of these three round functions can be implemented using only one PE column in our processor. Due to the varying round structure, control signals for both PEs and Connection Rows do not remain constant. However, control structure of our processor gives us the flexibility of changing connections and functional units used in PEs since each control memory is capable of storing four different sets of control signals. Even without such flexibility, we could map each different round structure to separate PE columns and route the inputs accordingly.

While manually mapping the algorithm, we found that Type 1, 2, and 3 functional structures require 5, 3, and 4 cycles, respectively, and initialization function I requires 2, 1, and 2 cycles in each type, respectively. However, our

toolchain extracts more parallelism between rounds after unrolling the loops and saves 7 more cycles per 64-bit data block. Thus, CAST-128 requires a total of 73 cycles to transform 64-bit data block. Since only one PE column is utilized for a given data block, Cryptoraptor enables us to process four different data streams in parallel.

7.1.5 Data Encryption Standard (DES)

DES [1] is a secret-key block cipher designed by IBM in 1977 by deriving from Lucifer and submitted to National Bureau of Standards (NBS) as a candidate for the protection of sensitive, unclassified electronic government data. It has been widely used until theoretical weaknesses in the cipher were demonstrated, and it was withdrawn as a standard by the NIST. DES and its successors has been included in SSL, TLS, IPsec and implemented in almost all security libraries.

The algorithm is designed to encipher and decipher 64-bit data blocks under control of a 64-bit key. It takes 64-bit plaintext and transforms it through a series of complicated operations through 16 rounds in Feistel network structure to produce the same length ciphertext. The round function of DES, shown in Figure 7.7, operates on one 32-bit half of data block and consists of 4 operation stages: (i) Expansion, (ii)Key mixing, (iii) Substitution, and (iv) Permutation.

The first stage, Expansion, expands 32-bit half data block to 48 bits using a predefined expansion permutation, denoted E in the diagram (Figure

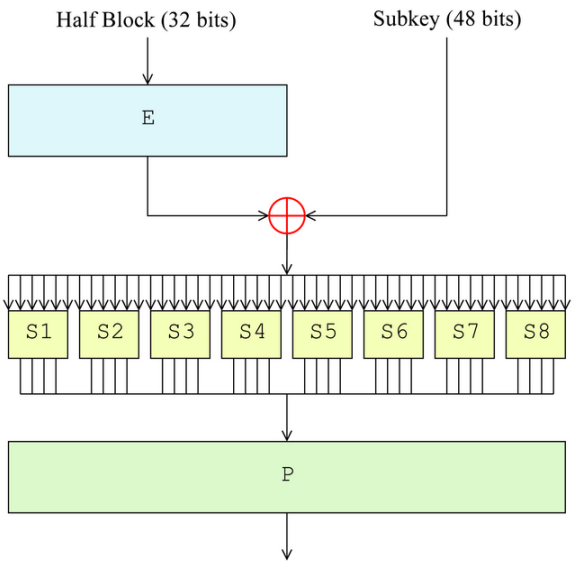


Figure 7.7: The round function f of DES

7.7). The result of Expansion stage is combined with a derived sub-key using an XOR operation, called Key mixing stage.

After making the relation with input data and user key, the output is split into eight 6-bit pieces, and each piece is fed into separate lookup table. DES consists of eight different S-boxes, each of which takes 6-bit block as input and yields a 4-bit block as output. Finally, outputs of S-boxes are rearranged using a predefined table that controls the permutation. Thus, implementation of DES highly relies on table lookup operations and bit manipulations.

Due to heavy use of bit manipulations, our mapping structure mostly relies on PEU unit. Besides bit manipulations, DES consists of unorthodox granularity for both S-box address and table lookup operations, 4-bit and 6-bit respectively. Our PEU structure helps us to handle such different granularities

as well.

To better utilize available resource in our processor, we slightly modify the traditional implementation. The first level of our mapping performs the Expansion phase of the algorithm; however, produces two 32-bit outputs instead of one 48-bit result. Doing so enables us to perform eight parallel table lookup in one cycle. Therefore, the input bits are padded in a way that each 4-bit of the data starts at byte boundaries. Since we used PEU, it's just a matter of defining the control signals in the setup phase. We also transform the round key into two 32-bit halves in the first level. The second level performs XOR operation between the output of the first level and a round key and feed each byte to S-boxes as an address. The outputs of table lookups are combined to produce two 32-bit outputs (one per TLU). In traditional implementation, output of S-boxes should be 8x6-bits, but we padded table entries in a way that each 6-bit of output starts at byte boundaries. Finally, two 32-bit output from the second level is transformed in PEU to generate one 32-bit output (Permutation phase). The last level of our mapping implements Permutation phase of the algorithm as well as remove any additional padding bits that we introduced in earlier levels. Therefore, one round of DES takes three cycles in our processor, yielding total of 48 cycles to transform 64-bit plaintext.

The mapping and implementation of DES on our processor require architecture and manual effort to achieve higher performance. The changes to traditional implementation are manually introduced to help the toolchain to map it more efficiently. However, the conventional implementation can also

be mapped in exchange of performance.

7.1.6 GOST

The GOST block cipher is a symmetric-key cipher designed and used by Soviet and Russian government for top secret information since 1970s, but declassified and released to the public in 1994 [169]. It was an alternative to the United States standard, DES, of the time.

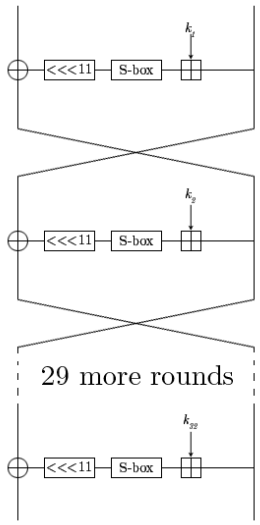


Figure 7.8: The overall structure of GOST block cipher

GOST transforms 64-bit data block using 256-bit user key through 32 rounds in Feistel network. The round function of GOST cipher is pretty straightforward, as shown in Figure 7.8. It takes one 64-bit data block as input, splits it into two halves, and add a 32-bit derived sub-key in modulo 2^{32}. After modular addition, the 32-bit output is split into eight 4 bit pieces,

each of which is fed into 4x4 S-box for substitution. Finally, the produced 32-bit value is left rotated by 11, and XORed with the other half of the input data block.

Due to its structure, GOST can utilize only one bundle in our processor. However, it enables us to show that our processor is capable of supporting algorithms even if they do not fit into mainstream operation flows. Each operation in one round of GOST can be implemented using one PE level. Thus, one GOST round takes three cycles for round function and hides the extra cycle required to establish Feistel Network, yielding total of 96 cycles to transform 64-bit data block.

7.1.7 Kasumi

The Kasumi, also known as MISTY [141], is widely used for security in many synchronous wireless standards; UMTS, GSM, and GPRS mobile communications systems. It is also used as A5/3 key stream generator.

Kasumi is a block cipher with eight Feistel rounds with a key of up to 128 bits and works on 64-bit plaintext blocks. Each round uses a set of derived round keys KLi, KOi, and KIi for each round i. It processes the 64- bit word in two 32-bit halves, and the right half is XOR'ed with the output of the round function after which the halves are swapped in each round. The input word is concatenation of the left and right halves of the first round. Details on traditional implementation of Kasumi algorithm are shown in Figure 7.9.

The structure of Kasumi consists of two slightly different round trans-

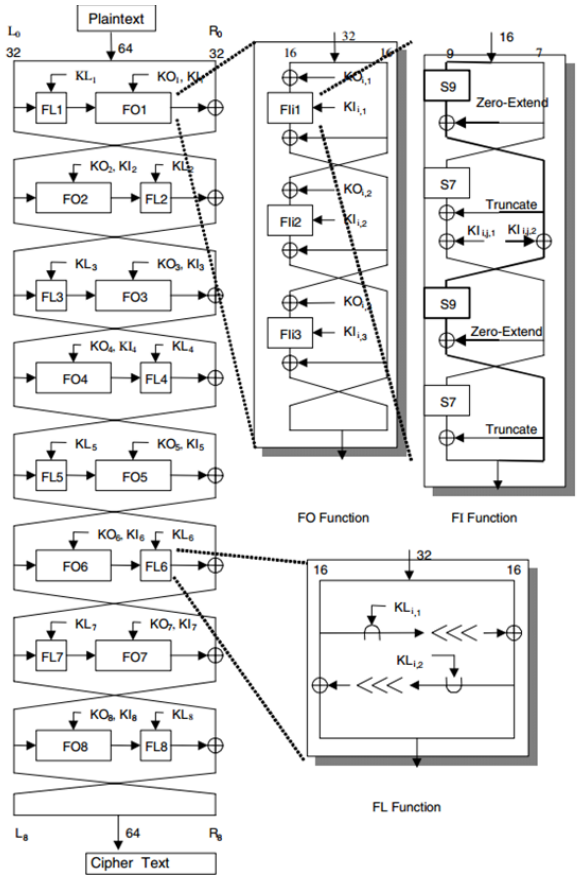


Figure 7.9: The traditional structure of Kasumi

formations: even and odd rounds. In odd rounds, the round-function is computed by applying the FL function followed by the FO function while it is in reverse order for even rounds.

We specifically included Kasumi into our algorithm mapping discussion to stress flexibility and reconfigurability of our processor with its data dependent and fairly complex structure due to its unbalanced rounds and unorthodox table sizes (7- and 9-bit addressing).

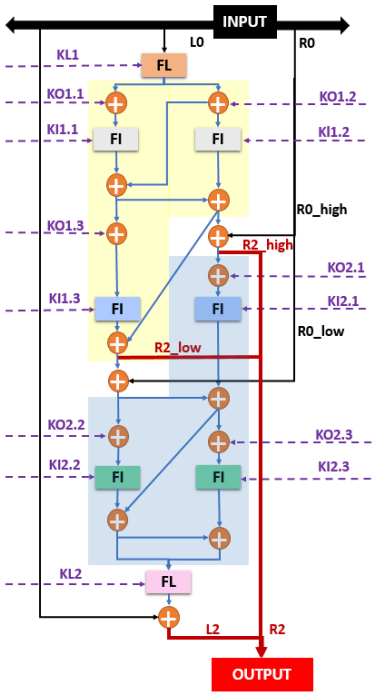


Figure 7.10: Merging one odd and one even round of Kasumi as one big operation block.

Even though the structure of Kasumi seems to be highly data dependent, it is possible to extract parallelism if we consider the size of operation as 16 bits instead. As Balderas described [18], we merged two asymmetric (odd and even) rounds as one big operation block to fully utilize the parallelism between operations. After merging one pair of odd and even rounds of Kasumi, we can extract the parallelism between the functions (FL, FI, FO) of Kasumi and XOR operations in between. Figure 7.10 shows how those functional blocks can be rearranged to remove false data dependencies caused by 32-bit operations, hence extract more parallelism.

While mapping proposed Kasumi structure onto the proposed architecture, we extract even more parallelism due to four parallel processing elements each of which has five independent highly powerful functional blocks that can work in parallel. Even though we mostly followed the order of functions as described in the proposed structure above, we introduced some further optimizations in appropriate places like redundantly calculate some operation sequence to avoid data dependencies. Doing so enable us to save couple more cycles in overall execution. Our current toolchain is not capable of doing such optimizations automatically; thus we manually introduced them.

With the proposed structure, we managed to map Kasumi onto our processor in an efficient way such that one big operation block (two rounds of traditional Kasumi) requires 16 cycles to operate, yielding 66 cycles to encrypt one block of plaintext.

7.1.8 Rivest Cipher 5 (RC5)

The RC5 is a secret-key block cipher proposed by Ron Rivest in 1994 [178]. It was a predecessor of the Advanced Encryption Standard candidate; RC6. Unlike other symmetric-key ciphers, RC5 algorithm works on variable block size (32, 64, or 128 bits) and secret key (up to 2040 bits). RC5 is implemented in the OpenSSL crypto library, used in Secure/Multipurpose Internet Mail Extensions (S/MIME) standard, and implemented in various products of RSA Security LLC.

The encryption and decryption algorithms are exceptionally simple.

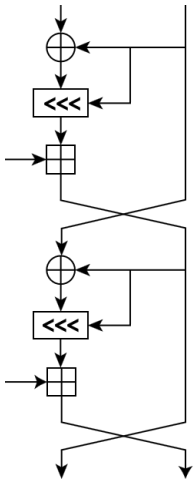


Figure 7.11: Two half rounds (one round) of RC5

Figure 7.11 shows the high-level structure of RC5 round function. It consists of a sequence of XOR operations, data-dependent left rotation, and an addition in modulo 2^{32}. Compared to other block ciphers, RC5 heavily relies on data-dependent rotations. The rotation operations are the only non-linear operator in RC5; there are no non-linear substitution tables or other non-linear operators. The goal behind heavy use of rotations in the algorithm was to prompt the study and evaluation of such operations as a cryptographic primitive.

Algorithm 7.1.1: RC5ENCRYPT($plaintext, derivedSubkeys$)

$(A||B) = plaintext$;
$A = A + derivedSubkeys[0]$;
$B = B + derivedSubkeys[1]$;
for $i \leftarrow 1$ **to** 12
do $\begin{cases} A = ((A \oplus B) <<< B) + derivedSubkeys[2 + i]; \\ B = ((B \oplus A) <<< A) + derivedSubkeys[2 + i + 1]; \end{cases}$
return ($ciphertext$)

Even though the algorithm is flexible on block size, key size, and the number of rounds, the suggested configuration uses 64-bit data blocks, 128-bit secret key, and transforms the plaintext in 12 rounds. The algorithm can be implemented using a few lines of code in software as shown in Algorithm 7.1.1. One 64-bit data block is split into two 32-bit halves; A and B, and the resulting ciphertext is returned after applying the transformation of XOR, rotation and modular addition for 12 rounds.

It's straightforward to map RC5 onto our processor, where one RC5 round requires 4 levels (4 cycles) to implement the round function and a total of 48 cycles to encrypt 64-bit data block. The mapping RC5 clearly shows the power of having logical operation units bundled with shift/rotate operations. Without having such structure one RC5 round requires six levels (6 cycles) and a total of 72 cycles. Thus, it enables us to save 24 cycles.

Similar to some previous algorithms, Cryptoraptor enables us to apply RC5 encryption to four different data block in parallel since only one PE column is utilized for a given data stream. Therefore, full utilization of the available hardware can increase the effective throughput.

7.1.9 SEED

SEED [128] is a symmetric-key cipher developed by the Korea Information Security Agency and a group of experts in 1998 and has been used since then. SEED is a national industrial association standard and is included in several versions of SSL, TSL, IPsec, S/MIME, and Cryptographic Message

Syntax (CMS).

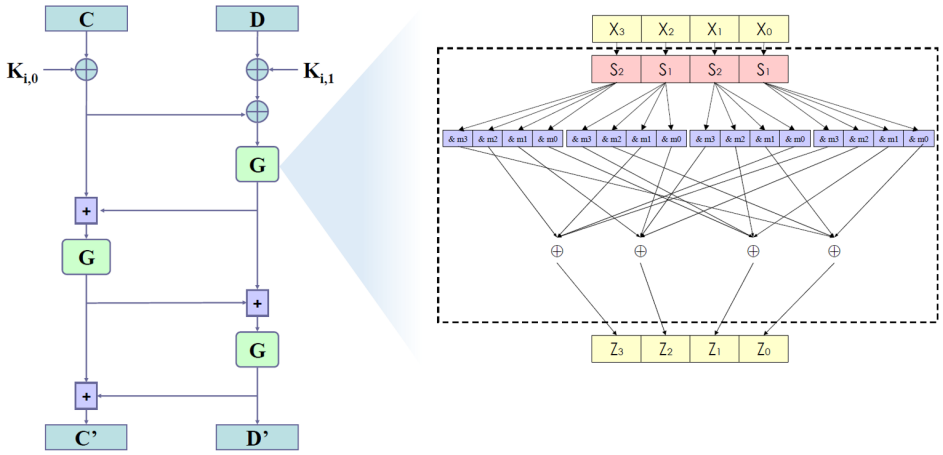


Figure 7.12: The round structure of SEED block cipher

The high-level structure of SEED block cipher follows the Feistel network through 16 rounds with 128-bit block size and a 128-bit secret key. The round structure of the algorithm is shown in Figure 7.12. The overall structure of SEED block cipher also has some resemblance to Kasumi in the recursiveness of its structure. While the overall structure is a Feistel network with an F-function operating on 64-bit halves, the F-function is also a Feistel network composed of a G-function operating on 32-bit halves as shown in Figure 7.12. The G-function takes a 32-bit half block as input, splits it into four 8-bit pieces, passes each of them through S-boxes, and finally combines byte outputs using a set of boolean functions such that each output bit depends on 3 of 4 input bytes. The algorithm consists of two 256 entry lookup tables with 8-bit entries that are derived from discrete exponentiation. However, the amount of the lookup table should be doubled to allow parallel accesses, yielding faster

implementation.

Even though G function shown in Figure 7.12 looks complex and requires complicated communication, it is actually not. When we carefully analyzed the inputs of XOR operations within G function, we realized that the byte-wise communication can be realized using byte shift operations after table lookup operations. To be more specific; if we rotate the outputs W3, W2, and W1 left by 1, 2, and three respectively, four parallel byte-wise XOR trees can be merged to one 32-bit tree. These findings enabled us to map one round of SEED takes ten cycles (1 initial + 3x3 G functions) in our processor, yielding total of 160 cycles to transform 128-bit plaintext. Our toolchain extracts even more parallelism after unrolling the loops and saves 8 cycles in total execution time of 128-bit data.

7.1.10 Twofish

Twofish [189] is a secret-key block cipher designed by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson in 1998 by deriving from Blowfish, SAFER, and Square, and submitted to Advanced Encryption Standard contest. It was selected as one of the five finalists for a new standard. Twofish is one of a few ciphers included in the OpenPGP standard and included in GNU Crypto library.

Twofish transforms 128-bit data block using a secret key up to 256 bits through 16 rounds in Feistel network with a bijective F function. The overall structure of the algorithms is shown in Figure 7.13. The F function

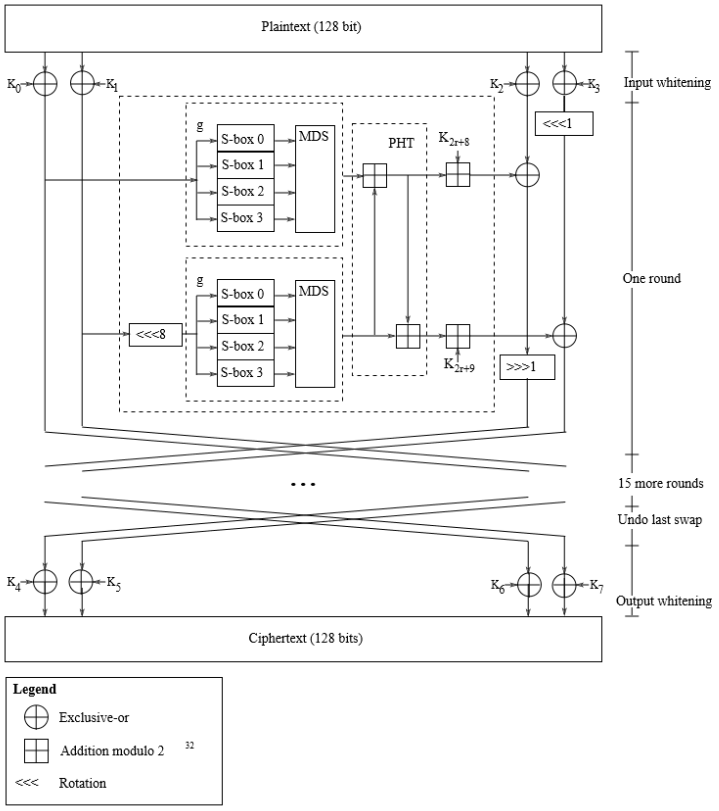


Figure 7.13: The overall structure of Twofish block cipher

consists of four key-dependent 8x8 S-boxes, a fixed 4-by-4 maximum distance separable (MDS) matrix over GF(2^8), a pseudo-Hadamard transform, bitwise rotations, and XOR operations. For faster implementation, the number of S-boxes is doubled and matrix multiplication, shown as MDS, is implemented as separate lookup table. A pseudo-Hadamard transforms (PHT) is a simple mixing operation that is defined as;

$$\begin{aligned} a^1 &= a + b \quad mod 2^{32} \\ b^1 &= a + 2b \quad mod 2^{32} \end{aligned} \tag{7.9}$$

where a and b are two 32-bit inputs. That is; PHT is two parallel modular addition, one of which is operates on left shifted operand. The Figure 7.13 shows how PHT is used within the algorithm. Due to its fairly regular and simple datapath, implementation of Twofish is straightforward on both software and hardware.

Even though Twofish can be mapped onto our processor as is, we introduced some optimizations that enable us to fit it onto Cryptoraptor more efficiently. The first thing we changed is the order of the rotation operation before table lookup operations. Since it is a byte-wise rotation and the following operation is byte-wise lookup, it is safe to change the order of S-boxes in the loading phase, and have the rotation afterwards. Doing so enables us to pack XOR, table lookup and rotation operation into TLU and save two redundant cycles per round. Since MDS is a matrix multiplication over GF(2^8), we pre-calculated a table to perform this operation in one cycle as it is done for other algorithms that have matrix multiplication. Implementing matrix multiplication using a table lookup helps us to save a couple of cycles per round. By using the advantage of having independently configurable functional units, we performed XOR and rotate-by-1 operations on the second 64-bit half (two 32-bit operations shown in the right part of Figure 7.13) in parallel, which allow us to save one extra cycle per round. Therefore, one round of Twofish takes only five cycles in Cryptoraptor, yielding total of 80 cycles to transform 128-bit plaintext. Since Twofish can be implement using only two PE columns for one data block, Cryptoraptor can be fully utilized with two parallel data

stream for higher throughput.

7.2 Stream Ciphers

The following sections briefly describe how we mapped most widely used stream cipher, RC4, and Phelix as an additional example. We already described how we mapped the most common block cipher, Kasumi (A5/3) used in UMTS, GSM, and GPRS mobile communications systems to provide confidentiality and integrity. Since A5/2 is prohibited in 2006 due to its security weakness, and straightforward structure of A5/1 (fixed sequence of shift and bit-wise XOR operation), we choose to implement more complex stream cipher; Phelix. Phelix is not used in any security standard but is a good candidate to stress the flexibility of our processor.

7.2.1 Rivest Cipher 4 (RC4)

The RC4 [178] is a secret-key stream cipher designed by Ron Rivest in 1987 and published in 1994. RC4 is the most popular stream cipher in several security protocols and standards; TLS, WEP, and WPA for wireless cards, and included in cryptographic libraries; OpenSSL and GNU Crypto.

RC4 is a binary additive stream cipher with a variable sized key that can range between 8 and 2048 bits in multiples of 8 bits. Like other ciphers, RC4 consists of two part: (i) key-scheduling and (ii) encryption. During key scheduling process, a lookup table of 256-entry is initialized and is used throughout the encryption process. However, unlike other ciphers mentioned

before, encryption phase of RC4 also make changes on the lookup table by swapping the entries. The details of encryption phase are defined as follows.

Algorithm 7.2.1: RC4ENCRYPT($plaintext, keyArray, ciphertext$)

$$
\begin{array}{l}
j = 0; \textbf{for } i \leftarrow 0 \textbf{ to } messageLength \\
\quad \textbf{do} \begin{cases}
i = (i+1) mod 256 \\
j = (j + keyArray[i]) mod 256 \\
swap(keyArray[i], keyArray[j]); \\
keyIndex = (keyArray[i] + keyArray[j]) mod 256; \\
ciphertext[i] = plaintext[i] \oplus keyArray[keyIndex];
\end{cases}
\end{array}
$$

One round of RC4 produces a word of the keystream and XOR it with the plaintext to produce the ciphertext. Therefore, each round generates one byte of keystream and ciphertext.

Compared to other algorithms, mapping RC4 onto our processor is a bit more difficult due to frequent table lookups and table updates. For that reason, RC4 algorithm significantly stresses the flexibility of Cryptoraptor and our mapping process. Due to the fact that lookup tables are being updated and indexed, it is not possible to unroll the loop over multiple PE levels, and we are restricted to use same PE rows. We naively mapped RC4 round function and tried to utilize operation bundles available in FUs as much as possible. One RC4 round requires eight levels (8 cycles) to implement the round function, and a total of (4x8x$messageLength$) cycles to encrypt a given message. Even though RC4 does not fit into our processor perfectly, we are still able to support such a complex algorithm. Thus, like Kasumi, RC4 also provides good insight about the flexibility of Cryptoraptor. Since RC4 is implemented using a single

PE column, four parallel data streams can be encrypted concurrently for higher throughput.

7.2.2 Phelix

Phelix is a high-speed stream cipher with a built-in MAC functionality, developed by Doug Whiting, Bruce Schneier, Stefan Lucks, and Frederic Muller for eSTREAM contest in 2004 [217]. The algorithm is a slightly modified form of an earlier cipher, Helix, developed by the same designers in 2003.

Even though it is not included in any security protocols or standards, we included it in our algorithm mapping process to stress our processor's flexibility with another complex algorithm that includes several parallel operations and connections.

Phelix algorithm consists of a sequence of blocks and an encryption function over each block. Figure 7.14 shows the high-level structure of Phelix one block (two-half block) encryption. The algorithm relies on only addition in modulo 2^{32}, XOR operation, and rotation by a pre-defined number of bits. A single round of Phelix consists of adding (or XORing) one active state word into the next, and rotating the first word. One full Phelix block consists of twenty simple rounds. Each block is 160-bit, divided into five 32-bit variables denoted as $Z_0^{(i)}$, $Z_1^{(i)}$, $Z_2^{(i)}$, $Z_3^{(i)}$, and $Z_4^{(i)}$. While four of these five state variables are used as input to the next block, the remaining one 32-bit variable is generated as output at the end of the current block.

Mapping Phelix onto our processor proves the degree of complexity for

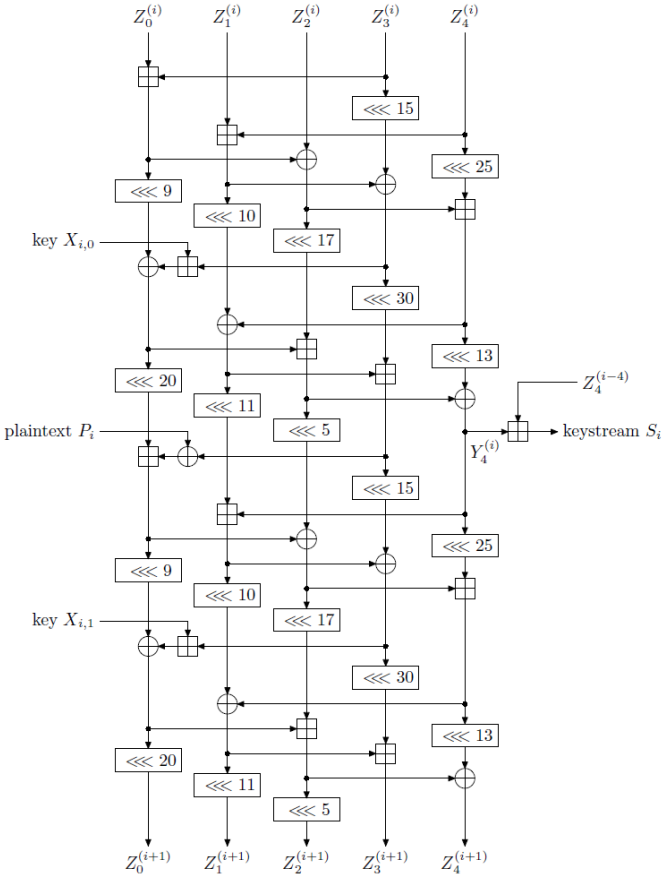


Figure 7.14: One block of Phelix encryption

connections between PEs and justifies the full crossbar connection structure between the levels. One half block of Phelix requires 2 PE columns and five levels to be implemented in our processor, yielding a total of 10 cycles to process 32-bit of the message.

7.3 Cryptographic Hash Functions

In following sections, we explain how we mapped widely used cryptographic hash functions; MD4, MD5, SHA-1, and SHA-2 onto our processor. Many popular hash algorithms may be characterized as Merkle-Damgard construction invented by Ralph Merkle in 1979 [147]. Merkle-Damgard construction is a method of building collision-resistant cryptographic hash functions using collision-resistant one-way compression functions. The Figure 7.15 shows the high-level structure of Merkle-Damgard construction.

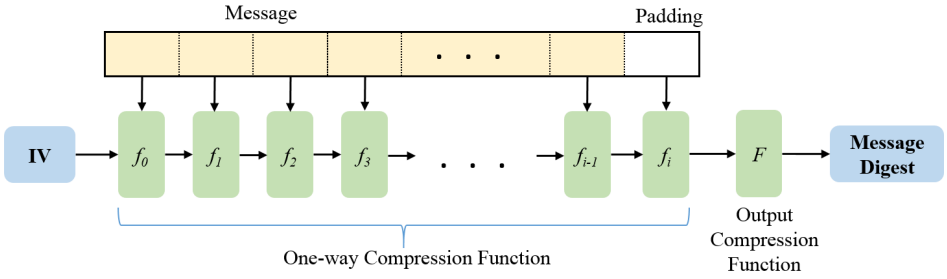


Figure 7.15: The high level structure of the Merkle-Damgard construction

The hash functions that follow the Merkle-Damgard construction take a message of arbitrary length as input, add padding to extend the message to appropriate length, iteratively transform message blocks using one-way compression function and an initialization vector, and finally produce fixed-length message digest. Therefore, a good hash function consists of: (i) a collision-resistant compression function, (ii) a padding procedure, and (iii) a good initial vector.

7.3.1 Message Digest Algorithm-4 (MD4)

MD4 is a cryptographic hash function designed by Ronald Rivest in 1990 [176]. MD4 is one of the widely known hash functions and triggered researchers to develop new attacks for cryptographic hash functions. However, it is possible to find an another message that produces the same MD4 digest as a given message without requiring a brute force search. The structure of MD4 inspired and influenced other later designs such as MD5, SHA-1, and RIPEMD, and it was replaced with its successor, MD5, due to its security issues. MD4 has been used in Microsoft security protocol suite, NT LAN Manager (NTLM), provides authentication, integrity, and confidentiality to users. It is also incorporated into PGP v1.0 and S/MIME and implemented in OpenSSL, Crypto++, and many other cryptography libraries.

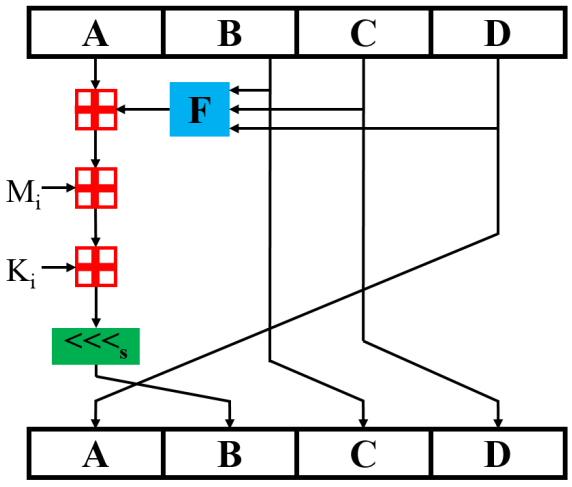


Figure 7.16: The round structure of MD4

The algorithm takes a message of arbitrary length as input, transforms

each 128-bit block of message through 3 rounds (each of which consists of 16 inner rounds), and produces a 128-bit message digest of the input as an output. The detailed structure of round operation is shown in Figure 7.16. The algorithm relies on 128-bit state, divided into four 32-bit words (A, B, C, and D) and applies a sequence of addition in modulo 2^{32}, rotation, and a logical function F. Each one of three rounds uses different F functions as defined below.

$$\begin{array}{lcl} F(B,C,D) & = & (B \wedge C) \vee ((\neg B) \wedge D) \\ G(B,C,D) & = & (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) \\ H(B,C,D) & = & B \oplus C \oplus D \end{array} \tag{7.10}$$

Thanks to our powerful LOU, which consists of six independently configurable logic blocks, we can implement any F function defined above in one cycle. Since we can store up to four sets of control signals per PE, changing round functions can be controlled by the state machine easily. Instead of following the traditional operation order, we added M_i and K_i as the first step in parallel while processing F function. Doing so enables us to save one cycle per round. Even though the first inner round still takes four cycles, the remaining inner rounds (15+2*16) will take only three cycles each. Thus, processing 128-bit block of the message will be processed in total of 145 cycles. Considering hashing process will continue for other 128-bit blocks of the message; the rounds will take only three cycles after spending four cycles at the very first 128-bit block of the message.

7.3.2 Message Digest Algorithm-5 (MD5)

MD5 [175] is a cryptographic hash function designed by Ronald Rivest in 1992 to replace prior hash function, MD4. Due to the high number of rounds and extra modular addition, MD5 is slower than MD4 but is more secure in design. MD5 has been utilized in a wide variety of cryptographic applications for a significant amount of time and has been commonly used to verify data integrity. It has been included in many standards such as SSL, TLS, IPsec, S/MIME, and NTLM, and implemented in almost all cryptography libraries.

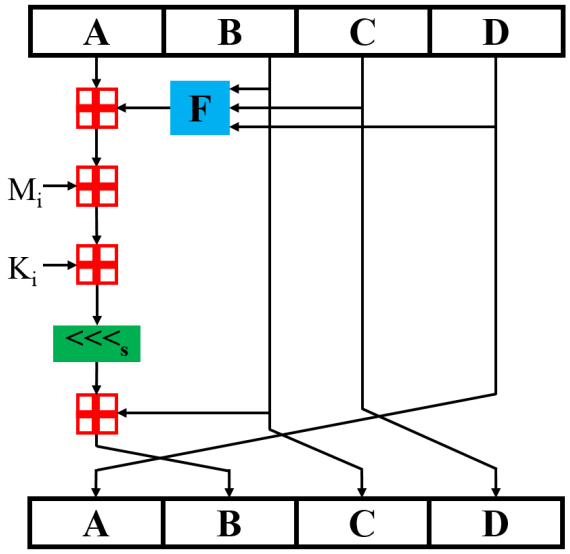


Figure 7.17: The structure of one MD5 operation

MD5 takes a message of arbitrary length as input, splits the message into 512-bit message blocks, and transforms message blocks to 128-bit message digest through 4 rounds of 16 MD5 operations. As shown in Figure 7.17, the overall structure of MD5 is very similar to MD4 with an additional modular

addition after rotation. The F functions used in each round are also slightly different to provide higher security. With a given 128-bit state, divided into four 32-bit words (A, B, C, and D), the round functions are defined as follows;

$$\begin{array}{rcl} F(B,C,D) & = & (B \wedge C) \vee ((\neg B) \wedge D) \\ G(B,C,D) & = & (B \wedge D) \vee (C \wedge (\neg D)) \\ H(B,C,D) & = & B \oplus C \oplus D \\ I(B,C,D) & = & C \oplus (B \wedge (\neg D)) \end{array} \tag{7.11}$$

Mapping process of MD5 algorithm onto our processor is very similar to the one for MD4. However, MD5 requires one more modular addition after the last rotation operation. Even though the round functions used in MD5 are quite different than the ones in MD4, they also can be implemented in one LOU. Having four different round function does not cause any issue since our control memories for PEs are capable of storing four sets of control signals and controlled by the state machine very easily.

Similar to MD4 mapping, we process addition operations for A, M_i, and K_i in advance wherever we can, which allows us to save one extra cycles per round except the very first round. Therefore, processing 512-bit message requires a total of 257 cycles (5+63x4) in our processor. Our toolchain extracts more parallelism with loop-unrolling optimization, which saves three cycles in total execution time; yielding 254 cycles to process 512-bit message. Since 2 PE columns is required to implement MD5 (one for algorithm and the other one for parallel additions), remaining 2 PE columns can be used for processing one more message stream in parallel.

7.3.3 Secure Hash Algorithm-1 (SHA1)

SHA-1 is a cryptographic hash function designed by the U.S National Security Agency in 1995 and published by the NIST as a U.S. Federal Information Processing Standard [64]. Due to the similarity of the round structure, it seems to be the successor of MD4 and MD5. However, the level of security provided by SHA-1 is significantly higher than its predecessors. SHA-1 is the world's most popular cryptographic hash function which is included in all important standards such as SSL, TLS, IPsec, S/MIME, SSH, and PGP, and implemented in all cryptography libraries.

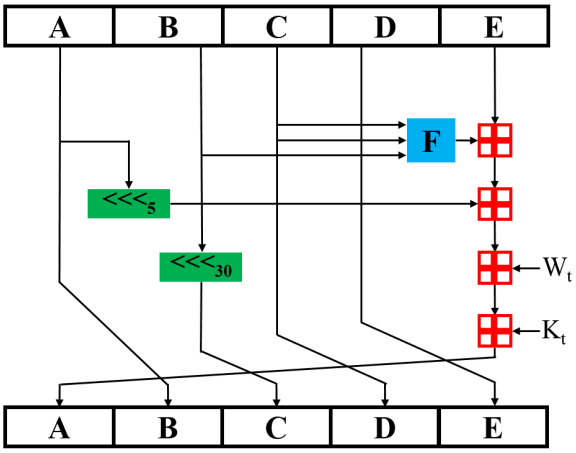


Figure 7.18: The round structure of SHA-1

SHA-1 is an iterative hash function that takes a message of length less than 2^{64} bits, processes 512-bit input message blocks, and produces a 160-bit message digest of the input as an output. As shown in Figure 7.18, SHA-1 follows a similar structure to MD4 and MD5 with slight differences. The algorithm operates on 160-bit state, divided into five 32-bit words that are

denoted as A, B, C, D, and E. The state update transformation of SHA-1 consists of 4 rounds of 20 steps in each. The transformation functions used in each round are very similar to the ones in MD4 and defined as follows;

$$\begin{array}{lcll} F(t;B,C,D) & = & (B \wedge C) \vee ((\neg B) \wedge D) & (0 <= t <= 19) \\ F(t;B,C,D) & = & B \oplus C \oplus D & (20 <= t <= 39) \\ F(t;B,C,D) & = & (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & (40 <= t <= 59) \\ F(t;B,C,D) & = & B \oplus C \oplus D & (60 <= t <= 79) \end{array} \quad (7.12)$$

After the last step of the state update transformation, the initial state variables (A_0, B_0, C_0, D_0, and E_0) and the final state (A_{80}, B_{80}, C_{80}, D_{80}, and E_{80}) are combined using addition in modulo 2^{32}. The result this step is either the final hash value or the initial value to process the next 512-bit message block.

As we did for MD4 and MD5, we added M_i and K_i in the first level. However, adding the state variable E and shifted version of A can also be done in parallel in SHA-1 algorithm. Doing so, we can map one round of SHA-1 onto our processor in 3 cycles. Thus, it takes a total of 240 cycles (80 rounds x 3) to process 512-bit input message block. Again with the help of our toolchain, we were able to extract the parallelism hidden between loops and save 15 cycles in total (yielding 225 cycles) to process one 512-bit input message block.

7.3.4 Secure Hash Algorithm-2 (SHA2)

Like its predecessor, SHA-2 [74] is a cryptographic hash function designed by the U.S National Security Agency in 2001 and published by the NIST as a U.S. Federal Information Processing Standard. The new algorithm was

published as a set of functions (SHA-224, SHA-256, SHA-384, and SHA-512) for varying message digest sizes; 224, 256, 384, and 512 bits. Even though SHA-1 is still a widely used in cryptographic applications, NIST emphasizes that applications that require collision resistance must use the SHA-2 family of hash functions. All standards that include SHA-1 has been either replaced the use of SHA-1 with SHA-2 or supported both at the same time.

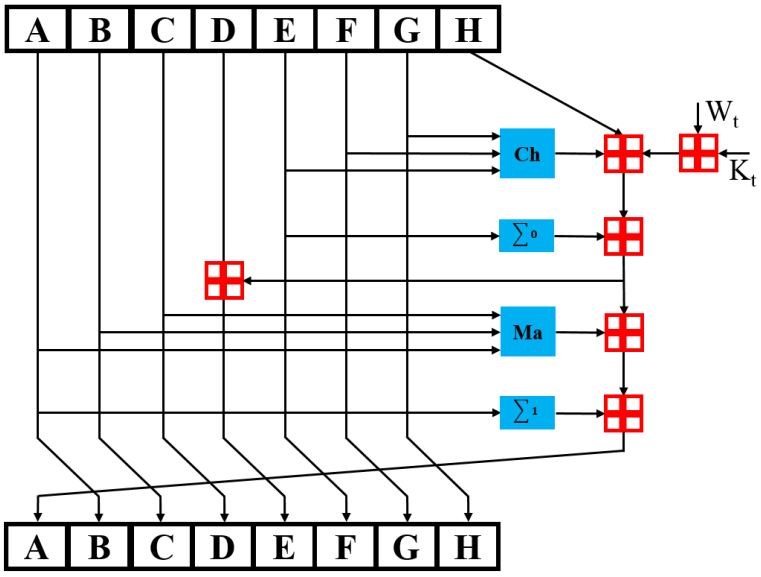


Figure 7.19: The round structure of SHA-2

The SHA-2 hash function takes a message of arbitrary length as input, splits the message into 512-bit message blocks, and transforms message blocks to a message digest of 224, 256, 384, and 512 bits depending on chosen size. SHA-2 algorithm family utilizes eight 32-bit state variables labelled as A, B, ..., H, which are initialised to pre-defined values H_0-H_7 at the start of the hash function. The 32-bit values of the A to H variables are updated in each

round through 64 or 80 rounds, and the new values are used in the next round. The high-level structure of SHA-2 round is shown in Figure 7.19. The round structure and functions used in each round consist of a sequence of addition in modulo 2^{32}, rotation, and logical primitives. The functions used in each round is defined as follows;

$$\begin{array}{rcl} Ch(E,F,G) & = & (E \wedge F) \oplus (: E \wedge G) \\ Ma(E,F,G) & = & (E \wedge F) \oplus (E \wedge G) \oplus (F \wedge G) \\ \Sigma 0(A) & = & (A >>> 2) \oplus (A >>> 13) \oplus (A >>> 22) \\ \Sigma 1(E) & = & (E >>> 6) \oplus (E >>> 11) \oplus (E >>> 25) \end{array} \tag{7.13}$$

At the end of each round, the i^{th} intermediate hash value H^i is computed by adding state variables with corresponding 32-bit word of $(i-1)^{th}$ round's output hash value H^{i-1}, where H^i is defined as $\{H_0^i||H_1^i||...||H_7^i\}$. The output hash value of each 64 or 80 rounds either the final hash value or the initial value to process the next 512-bit message block.

As we did for previous cryptographic hash functions, we choose to process the required operations in parallel by taking the advantage of multiple parallel FUs instead of following traditional operation flow. In the first PE level, we calculate Ch, Ma, add M_i and K_i in one PE and state variables H and D on the other, and process rotation operations for $\Sigma 0$. The second level calculates $Ch + M_i + K_i$, $Ma + H + D$, apply XOR operations to the outputs of rotations, and process rotation operations for $\Sigma 1$. In the third level, $Ch + M_i + K_i$ and $\Sigma 0$ are added and XOR operations to the outputs of rotations are applied. The fourth level generates the new value of E by adding $Ch + M_i + K_i + \Sigma 0$ and $H + D$ while $Ch + M_i + K_i + \Sigma 0 + Ma + H + D$ is

generated on the other PE. Finally, $\Sigma 1$ is added to generate new value of state variable A.

One round of SHA-2 requires five cycles to be mapped onto our processor. Thus, it takes a total of 320 cycles (64 rounds x 5) to process one 512-bit input message block.

Chapter 8

Future Work

In previous chapters, we provided a detailed description of the current architecture of our reconfigurable cryptographic processor. However, there are still potential improvements for functional units as well as overall processor design to achieve a higher degree of flexibility and even higher possible throughput for all.

The lack of dedicated multiplication units is one of the main limitations of Cryptoraptor, which restricts the algorithm coverage of our processor for both existing and future cryptographic algorithms. Therefore, multiplication unit can be a good candidate for next hardware improvement; yielding 8.1% increase in the algorithm coverage for existing algorithms. However, adding a dedicated multiplication unit into existing PE as separate functional unit has a significant impact on processor's cycle time as discussed in the previous chapter. Thus, integrating a multiplication functionality into current processor requires either a pipelined structure or special design for multiplication unit.

Even though the most of the modular arithmetic operations can be implemented using existing functional units, the current design can be extended to support varying and unorthodox modulo bases. Extending existing

hardware for modular arithmetic may increase the algorithm coverage of our processor by 2 percent while having both multiplication unit and modular arithmetic support may increase up to 13.5 percent.

Our current toolchain has limited capabilities and does not provide aggressive optimizations. A more powerful automatic mapping structure for cryptographic algorithms using a crypto-specific language or ideally high level language like C/C++ can be developed for robust and high performance mapping of the algorithms. Such a toolchain might also solve the issues related to multiplication and modular arithmetic automatically by mapping these operations to existing hardware on the fly.

Since the FU utilization of the mapped algorithm is low, we might also develop a new synthesis toolchain which takes a set of algorithms targeted to be supported and a set of constraints, design a new architecture using proposed FUs and bundles, and outputs RTL code of new processor. Such methodology will limit the flexibility of the generated processor, but it will achieve higher performance, better area, power, and resource utilization for the targeted algorithms.

Since we initially set the target of our processor as symmetric-key encryption and cryptographic hash functions, public-key cryptography is beyond the scope of this work. Public-key cryptography may require different functional units and/or major changes on datapath due to its different structure and requirements. We left extending our hardware support for public-key cryptography as future work for the ideal and complete cryptographic processor.

Chapter 9

Conclusion

In this thesis, we presented the current architectural structure of our reconfigurable cryptographic processor and the rationales that shaped our processor design. Our first goal was to have a complete, flexible, and high throughput cryptographic processor that can support a wide range of ciphers and cryptographic hash functions.

We provided a comprehensive literature review on cryptographic algorithms and detailed analysis on the specifications and requirements of various crypto-systems. Such a detailed analysis might help both cryptographic algorithm developers and hardware developers while designing new algorithms, standards, and hardware implementations.

After describing our processor design and our rationales, we also provided detailed analysis of our processor in terms of performance, area, power, and the algorithm coverage. We believe providing such detailed study, and evaluation may enable both cryptographic algorithm developers and researchers to explore performance, power, and area trade-offs while designing new algorithms.

We are still exploring design trade-offs for functional units as well as

overall processor design to achieve a higher degree of flexibility and highest possible throughput for all. To achieve that, we plan to eliminate limitations of the current design, further improve its performance, and excessively stress it for flexibility and throughput with more cryptographic algorithms.

We developed a highly reconfigurable cryptographic processor, Cryptoraptor, that supports a wide range of symmetric key encryption algorithms and cryptographic hash functions efficiently and has high potential to support future ones. Our results show that Cryptoraptor with its $1GHz$ clock frequency can compete in term of performance with high-end ASIC cores and FPGA solutions while achieving 25X and 160X higher throughput per area than the best CPU and GPU solutions, respectively. To the best of our knowledge, the proposed cryptographic processor supports the widest set of cryptographic algorithms and the only crypto-specific processor that have the capability of supporting the future algorithms. Through this thesis, we hope to demonstrate the potential of our processor design for high performance cryptographic applications.

Appendices

Appendix A

Detailed Operation Classes Usage

Table A.1: The special functional unit requirements in cryptographic algorithms

Algorithm	Arithmetic Op.	Table Lookup	Logical Op.	Shifter & Rotator	Permutation & Expansion
Block Ciphers					
3WAY			✓	✓	
AES		✓	✓		
Akelarre	✓		✓	✓	
Anubis		✓	✓	✓	
ARIA		✓	✓		
BaseKing			✓	✓	
Blowfish	✓	✓	✓		
Camellia		✓	✓	✓	
CAST-128	✓	✓	✓	✓	
CAST-256	✓	✓	✓	✓	
CIKS-1			✓	✓	✓
Cipherunicorn-A	✓	✓	✓	✓	
Cipherunicorn-E	✓	✓	✓	✓	
CLEFIA			✓		✓
CMEA	✓		✓		
COCONUT98	✓	✓	✓	✓	
Crab	✓	✓	✓	✓	✓
Cryptomeria/C2	✓	✓	✓	✓	✓
CRYPTON		✓	✓		✓
CS-Cipher		✓	✓		✓

DEAL		✓	✓		✓
DES		✓	✓		✓
DESX		✓	✓		✓
DFC	✓	✓	✓		✓
E2		✓	✓		✓
FEAL	✓	✓	✓	✓	
FEALNX		✓	✓		
FEA-M	✓		✓		
FOX		✓	✓		
FROG		✓	✓		✓
GOST	✓	✓	✓	✓	
Grand Cru		✓	✓	✓	
Hasty Pudding cipher		✓	✓	✓	
Hierocrypt-3		✓	✓	✓	✓
Hierocrypt-L1		✓	✓	✓	✓
ICE		✓	✓		✓
IDEA	✓		✓		
Intel Cascade Cipher		✓	✓	✓	
KeeLoq			✓		✓
KHAZAD		✓	✓	✓	
Khufu and Khafre		✓	✓		
KLEIN		✓			✓
KN-Cipher			✓		✓
Ladder-DES		✓	✓		✓
LED		✓	✓	✓	✓
LOKI97	✓	✓	✓		✓
LUCIFER		✓	✓	✓	✓
M6	✓		✓	✓	
M8	✓		✓	✓	✓
MacGuffin		✓	✓	✓	
Madryga			✓	✓	
MAGENTA		✓	✓		
MARS	✓	✓	✓	✓	✓

MBAL		✓	✓		✓
Mercy	✓	✓	✓		
MESH	✓		✓		
Kasumi		✓	✓	✓	✓
MMB			✓		✓
MULTI2	✓		✓	✓	
MultiSwap	✓				
New Data Seal		✓	✓		✓
NewDES		✓	✓		
Nimbus	✓		✓		
Noekeon		✓	✓	✓	
NUSH	✓		✓	✓	
NXT		✓	✓	✓	
PRESENT		✓			✓
PRINCE		✓	✓	✓	✓
Q		✓	✓	✓	✓
RC2	✓		✓	✓	
RC5	✓		✓	✓	✓
RC6	✓		✓	✓	
REDOC III		✓	✓	✓	
SAFER K-128	✓	✓	✓		
SAFER K-64	✓	✓	✓		
SAFER+	✓	✓	✓		
SC2000	✓	✓	✓	✓	
SEED	✓	✓	✓	✓	✓
Serpent		✓	✓	✓	
SHACAL	✓		✓	✓	
SHACAL-2	✓		✓	✓	
Shark		✓	✓	✓	
Skipjack		✓	✓		
SMS4		✓	✓	✓	
Spectr-H64			✓	✓	✓
Square		✓	✓		

SXAL		✓	✓		✓
TEA	✓		✓	✓	
Threefish	✓	✓	✓	✓	✓
Twofish	✓	✓	✓	✓	
UES			✓		✓
Xenon	✓		✓	✓	
Xmx			✓		
XTEA	✓		✓	✓	
XXTEA	✓		✓	✓	
Zodiac		✓	✓		✓
Hash Functions					
BLAKE	✓	✓	✓	✓	
GOST	✓		✓		✓
Groestl	✓	✓	✓	✓	✓
HAS-160			✓	✓	
Haval	✓		✓	✓	
Hamsi			✓	✓	
JH		✓	✓		✓
Keccak			✓	✓	
MD2		✓	✓		✓
MD4	✓		✓	✓	
MD5	✓		✓	✓	
MD6			✓	✓	✓
PANAMA			✓	✓	✓
RadioGatÃžn	✓		✓	✓	
RIPEMD	✓		✓	✓	
RIPEMD-160	✓		✓	✓	
SHA-0	✓		✓	✓	
SHA-1	✓		✓	✓	
SHA-2	✓		✓	✓	
SHAvite3		✓	✓	✓	
SipHash	✓		✓	✓	
Skein	✓		✓	✓	✓

Snefru		✓	✓	✓	
SWIFFT	✓	✓		✓	✓
TIGER	✓		✓	✓	
Whirlpool		✓	✓	✓	
Stream Ciphers					
A5/1	✓		✓	✓	✓
A5/2	✓		✓	✓	✓
Achterbahn			✓	✓	✓
DECIM	✓		✓		✓
FFCSR			✓	✓	✓
FISH	✓		✓	✓	✓
GRAIN		✓	✓	✓	✓
HC256	✓	✓	✓	✓	✓
ISAAC	✓		✓	✓	✓
MICKEY			✓	✓	✓
MUGI		✓	✓	✓	✓
PANAMA	✓		✓	✓	✓
Phelix	✓		✓	✓	
Py	✓	✓	✓	✓	
Rabbit	✓		✓	✓	
RC4	✓	✓			
Salsa20	✓		✓	✓	
Scream	✓	✓	✓	✓	
SEAL	✓	✓	✓	✓	
Sfinks			✓	✓	✓
SNOW	✓	✓	✓	✓	
Trivium	✓		✓	✓	✓
Turing	✓	✓		✓	✓
VEST		✓	✓	✓	✓
WAKE	✓	✓	✓	✓	
Yamb	✓	✓	✓		

Appendix B

Operation Clusters

Table B.1: Operation clusters and patterns

Operation Classes	**Cryptographic algorithms**
Logical, Shift/Rotate, Table Lookup, Arithmetic	BLAKE, CAST-128, CAST-256, Cipherunicorn-A, Cipherunicorn-E, COCONUT98, Crab, Cryptomeria/C2, FEAL, GOST, Groestl, HC256, MARS, Py, SC2000, Scream, SEAL, SEED, SNOW, Threefish, Twofish, WAKE
Logical, Shift/Rotate, Table Lookup	Anubis, Camellia, GRAIN, Grand Cru, Hasty Pudding cipher, Hierocrypt-3, Hierocrypt-L1, Intel Cascade Cipher, Kasumi, KHAZAD, LED, LUCIFER, MacGuffin, MUGI, Noekeon, NXT, PRINCE, Q, REDOC III, Serpent, Shark, SHAvite3, SMS4, Snefru, VEST, Whirlpool
Logical, Table Lookup, Arithmetic	Blowfish, DFC, LOKI97, Mercy, SAFER K-128, SAFER K-64, SAFER+, Yamb
Logical, Table Lookup	AES, ARIA, CRYPTON, CS-Cipher, DEAL, DES, DESX, E2, FEALNX, FOX, FROG, ICE, JH, Khufu and Khafre, Ladder-DES, MAGENTA, MBAL, MD2, New Data Seal, NewDES, Skipjack, Square, SXAL, Zodiac
Logical, Shift/Rotate	3WAY, A5/1, A5/2, Achterbahn, Akelarre, BaseKing, CIKS-1, FFCSR, FISH, Hamsi, HAS-160, Haval, HC256, ISAAC, Keccak, M6, M8, Madryga, MD4, MD5, MD6, MICKEY, MULTI2, NUSH, PANAMA, PANAMA, Phelix, Rabbit, RadioGatÃžn, RC2, RC5, RC6, RIPEMD, RIPEMD-160, Salsa20, Sfinks, SHA0, SHA1, SHA2, SHACAL, SHACAL-2, SipHash, Skein, Spectr-H64, TEA, TIGER, Trivium, Xenon, XTEA, XXTEA
Logical, Arithmetic	A5/1, A5/2, Akelarre, CMEA, DECIM, FEA-M, FISH, GOST, Haval, HC256, IDEA, ISAAC, M6, M8, MD4, MD5, MESH, MULTI2, Nimbus, NUSH, PANAMA, Phelix, Rabbit, RadioGatÃžn, RC2, RC5, RC6, RIPEMD, RIPEMD-160, Salsa20, SHA0, SHA1, SHA2, SHACAL, SHACAL-2, SipHash, Skein, TEA, TIGER, Trivium, Xenon, XTEA, XXTEA

Logical, Permutation/Expansion	A5/1, A5/2, Achterbahn, CIKS-1, CLEFIA, Crab, Cryptomeria/C2, CRYPTON, CS-Cipher, DEAL, DECIM, DES, DESX, DFC, E2, FFCSR, FISH, FROG, GOST, GRAIN, Groestl, HC256, Hierocrypt-3, Hierocrypt-L1, ICE, ISAAC, JH, Kasumi, KeeLoq, KN-Cipher, Ladder-DES, LED, LOKI97, LUCIFER, M8, MARS, MBAL, MD2, MD6, MICKEY, MMB, MUGI, New Data Seal, PANAMA, PANAMA, PRINCE, Q, RC5, SEED, Sfinks, Skein, Spectr-H64, SXAL, Threefish, Trivium, UES, VEST, Zodiac
Shift/Rotate	SWIFFT, Turing
Table Lookup	KLEIN, PRESENT, RC4, SWIFFT, Turing
Arithmetic	MultiSwap, RC4, SWIFFT, Turing
Permutation/Expansion	KLEIN, PRESENT, SWIFFT, Turing

Appendix C

Operation Bundles

Table C.1: Operation patterns

Operation patterns	**Cryptographic algorithms**
XOR - SBOX	AES, Anubis, ARIA, BLAKE, Camellia, Cipherunicorn-A, Cipherunicorn-E, Cryptomeria/C2, CRYPTON, CS-Cipher, DEAL, DES, DESX, E2, FEAL, FEALNX, FROG, GRAIN, Grand Cru, Hierocrypt-3, Hierocrypt-L1, ICE, Intel Cascade Cipher, Kasumi, KHAZAD, Khufu and Khafre, KLEIN, Ladder-DES, LED, LOKI97, MacGuffin, MAGENTA, MARS, MBAL, MD2, Mercy, MUGI, NewDES, Noekeon, NXT, PRESENT, PRINCE, SAFER K-128, SAFER K-64, SAFER+, SEED, Serpent, Shark, SHAvite3, Skipjack, SMS4, Snefru, Square, SXAL, Turing, Twofish, Whirlpool, Zodiac
SBOX - XOR	AES, Anubis, ARIA, BLAKE, Camellia, CAST-128, CAST-256, Cipherunicorn-A, Cipherunicorn-E, COCONUT98, Crab, Cryptomeria/C2, CRYPTON, CS-Cipher, DFC, E2, FEAL, FEALNX, FOX, FROG, GRAIN, Grand Cru, Hierocrypt-3, Hierocrypt-L1, Intel Cascade Cipher, JH, Kasumi, KHAZAD, Khufu and Khafre, KLEIN, LED, LOKI97, LUCIFER, MacGuffin, MAGENTA, MARS, MBAL, MD2, Mercy, NewDES, Noekeon, NXT, SAFER K-128, SAFER K-64, SAFER+, SEED, Serpent, Shark, SHAvite3, Skipjack, SMS4, Snefru, Square, SXAL, Turing, Twofish, WAKE, Whirlpool, Yamb, Zodiac
XOR - SBOX - XOR	AES, Anubis, ARIA, BLAKE, Camellia, Cipherunicorn-A, Cipherunicorn-E, Cryptomeria/C2, CRYPTON, CS-Cipher, E2, FEAL, FEALNX, FROG, GRAIN, Grand Cru, Hierocrypt-3, Hierocrypt-L1, Intel Cascade Cipher, Kasumi, KHAZAD, Khufu and Khafre, KLEIN, LED, LOKI97, MacGuffin, MAGENTA, MARS, MBAL, MD2, Mercy, NewDES, Noekeon, NXT, SAFER K-128, SAFER K-64, SAFER+, SEED, Serpent, Shark, SHAvite3, Skipjack, SMS4, Snefru, Square, SXAL, Turing, Twofish, Whirlpool, Zodiac

XOR - Arithmetic	Akelarre, BLAKE, Blowfish, CAST-128, CAST-256, CMEA, Crab, Cryptomeria/C2, GOST, HAS-160, Haval, IDEA, M6, MARS, MD4, MD5, MESH, MULTI2, NUSH, Phelix, SEED, SHA0, SHA1, SHA2, SHACAL, SHACAL-2, SNOW, TEA, TIGER, Twofish, XTEA, XXTEA, Yamb
Arithmetic - XOR	Akelarre, BLAKE, Blowfish, CAST-128, CAST-256, Crab, Cryptomeria/C2, HAS-160, IDEA, MARS, MD4, MD5, Mercy, MESH, MULTI2, NUSH, Phelix, SHA0, SHA1, SHA2, SHACAL, SHACAL-2, SNOW, TEA, Threefish, TIGER, Twofish, XTEA, XXTEA
XOR - Arithmetic - XOR	Akelarre, BLAKE, Blowfish, CAST-128, CAST-256, Crab, Cryptomeria/C2, HAS-160, IDEA, MARS, MD4, MD5, MESH, MULTI2, NUSH, Phelix, SHA0, SHA1, SHA2, SHACAL, SHACAL-2, SNOW, TEA, TIGER, Twofish, XTEA, XXTEA
Logic - SHIFT	3WAY, Akelarre, Anubis, BaseKing, BLAKE, Camellia, CAST-128, CAST-256, Cipherunicorn-A, Cipherunicorn-E, Cryptomeria/C2, E2, FFCSR, GOST, Hamsi, Hasty Pudding cipher, HC256, Kasumi, Keccak, KHAZAD, M8, MacGuffin, MD6, MICKEY, MUGI, MULTI2, Noekeon, NUSH, NXT, Phelix, PRINCE, Q, RC5, RC6, Serpent, SHA2, SHACAL-2, SHAvite3, SipHash, Skein, SMS4, Twofish, Whirlpool, XXTEA
SHIFT - Logic	3WAY, A5/1, A5/2, Achterbahn, Akelarre, Anubis, BaseKing, BLAKE, Camellia, Cipherunicorn-A, Cipherunicorn-E, DECIM, E2, FFCSR, Hamsi, HC256, Kasumi, Keccak, KHAZAD, M6, M8, MARS, MD6, MICKEY, MUGI, MULTI2, Noekeon, NXT, Phelix, PRINCE, RC6, Salsa20, SC2000, SEAL, Serpent, Sfinks, SHA2, SHACAL-2, SHAvite3, SipHash, SMS4, SNOW, Spectr-H64, Threefish, Twofish, WAKE, Whirlpool, XTEA, XXTEA
Logic - SHIFT - Logic	3WAY, Akelarre, Anubis, BaseKing, BLAKE, Camellia, Cipherunicorn-A, Cipherunicorn-E, E2, FFCSR, Hamsi, HC256, Kasumi, Keccak, KHAZAD, M8, MD6, MICKEY, MUGI, MULTI2, Noekeon, NXT, Phelix, PRINCE, RC6, Serpent, SHA2, SHACAL-2, SHAvite3, SipHash, SMS4, Twofish, Whirlpool, XXTEA

Appendix D

Detailed Processing Element Width Usage

The Table D.1 summarizes parallel processing element requirement of each algorithm. Algorithms are clustered based on not only exact requirements but also possible performance gain when more processing element is used.

Table D.1: Operation width (PE way)

PE Width	**Cryptographic algorithms**
1	Blowfish, CAST-128, CAST-256, GOST, ICE, KeeLoq, Madryga, MultiSwap, Xmx
2	Achterbahn, Camellia, Cryptomeria/C2, CS-Cipher, DEAL, DES, DESX, Hasty Pudding cipher, Hierocrypt-L1, KN-Cipher, Ladder-DES, LUCIFER, M6, M8, MacGuffin, MBAL, MD4, MD5, MD6, Mercy, MICKEY, MUGI, MULTI2, Nimbus, Noekeon, NUSH, PRINCE, Py, RC4, RC5, REDOC III, Sfinks, SHA0, SHA1, SHA-CAL, Skipjack, Spectr-H64, SXAL, Threefish, UES, Whirlpool
4	3WAY, A5/1, A5/2, AES, Akelarre, ARIA, BaseKing, BLAKE, CIKS-1, Cipherunicorn-E, CLEFIA, CMEA, COCONUT98, Crab, CRYP-TON, DFC, E2, FEAL, FEALNX, FEA-M, FISH, FOX, FROG, GOST, GRAIN, Grand Cru, Hamsi, HAS-160, HC256, Hierocrypt-3, IDEA, Intel Cascade Cipher, ISAAC, JH, Kasumi, KHAZAD, Khufu and Khafre, KLEIN, LED, LOKI97, MARS, MD2, MESH, MMB, New Data Seal, NewDES, NXT, PRESENT, Q, RadioGatÃžn, RIPEMD, RIPEMD-160, Salsa20, Scream, SEAL, SEED, Serpent, SHA2, SHACAL-2, Shark, SHAvite3, SipHash, Skein, SMS4, Snefru, SNOW, Square, SWIFFT, TEA, TIGER, Trivium, Twofish, VEST, WAKE, Xenon, XTEA, XXTEA, Yamb, Zodiac

8	Cipherunicorn-A, DECIM, FFCSR, Groestl, Haval, MAGENTA, PANAMA, PANAMA, Phelix, Rabbit, RC2, RC6, SAFER K-128, SAFER K-64, SC2000, Turing
16	Anubis, Keccak, SAFER+

Bibliography

[1] Data encryption standard, 1977.

[2] C Adams. Rfc2144: The cast-128 encryption algorithm. *Network Working Group*, 1997.

[3] Carlisle Adams. Cast-256. *AES submission*, 1998.

[4] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturc, G. Worlich, and R. Zohar. Breakthrough AES performance with Intel AES new instructions. *White paper, June*, 2010.

[5] Liakot Ali, Ishak Aris, Fakir Sharif Hossain, and Niranjan Roy. Design of an ultra high speed AES processor for next generation IT security. *Comput. Electr. Eng.*, 37(6):1160–1170, November 2011.

[6] Gonzalo AÌĄlvarez, Dolores De la GuiÌĄa, Fausto Montoya, and Alberto Peinado. Akelarre: a new block cipher algorithm. 1996.

[7] Amphion Semiconductor. *CS5210-40 high performance AES encryption cores*. Amphion Semiconductor.

[8] Ross Anderson and Eli Biham. Tiger: A fast new hash function. In *Fast Software Encryption*, pages 89–97. Springer, 1996.

[9] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, 1998.

[10] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Specification of camellia-a 128-bit block cipher, 2000.

[11] François Arnault and Thierry P Berger. F-fcsr: design of a new class of stream ciphers. In *Fast Software Encryption*, pages 83–97. Springer, 2005.

[12] Telecommunications Technology Association et al. Hash function standard part 2: Hash function algorithm standard (has-160), ttas. Technical report, KO-12.0011, 2008.

[13] Jean-Philippe Aumasson and Daniel J Bernstein. Siphash: a fast short-input prf. In *Progress in Cryptology-INDOCRYPT 2012*, pages 489–508. Springer, 2012.

[14] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C-W Phan. Sha-3 proposal blake. *Submission to NIST*, 2008.

[15] Jean-Philippe Aumasson, Jorge Nakahara Jr, and Pouyan Sepehrdad. Cryptanalysis of the isdb scrambling algorithm (multi2). In *Fast Software Encryption*, pages 296–307. Springer, 2009.

[16] Joppe W. B., Dag A. O., and Deian S. Fast implementations of AES on various platforms. *SPEED-CC - Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers*, 2009.

[17] Steve Babbage and Matthew Dodd. The mickey stream ciphers. In *New Stream Cipher Designs*, pages 191–209. Springer, 2008.

[18] Tomas Balderas-Contreras, Rene Cumplido, and Claudia Feregrino-Uribe. On the design and implementation of a RISC processor extension for the KASUMI encryption algorithm. *Computer Electrical Engineering*, 34(6):531–546, November 2008.

[19] PSLM Barreto and Vincent Rijmen. The anubis block cipher. *Submission to the NESSIE Project*, 2000.

[20] PSLM Barreto and Vincent Rijmen. The khazad legacy-level block cipher. *Primitive submitted to NESSIE*, 97, 2000.

[21] PSLM Barreto and Vincent Rijmen. The whirlpool hashing function. In *First open NESSIE Workshop, Leuven, Belgium*, volume 13, page 14, 2000.

[22] Henry Beker and Fred Piper. *Cipher systems: the protection of communications*. Northwood Books London, 1982.

[23] Come Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Blandine Debraize, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, et al. Decim–a new stream cipher for hardware applications. *ECRYPT Stream Cipher Project Report 2005*, 4, 2005.

[24] Daniel J Bernstein. Salsa20 specification. *eSTREAM Project algorithm description, http://www. ecrypt. eu. org/stream/salsa20pf. html*, 2005.

[25] G.M. Bertoni, L. Breveglieri, F. Roberto, and F. Regazzoni. Speeding up AES By extending a 32 bit processor instruction set. In *International Conference on Application-specific Systems, Architectures and Processors. ASAP'06.*, pages 275–282, 2006.

[26] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Radiogatún, a belt-and-mill hash function. *IACR Cryptology ePrint Archive*, 2006:369, 2006.

[27] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Keccak specifications. 2009.

[28] Eli Biham and Orr Dunkelman. The shavite-3 hash function. *Submission to NIST*, 9, 2008.

[29] Eli Biham and Jennifer Seberry. Py (roo): A fast and secure stream cipher using rolling arrays. *IACR Cryptology ePrint Archive*, 2005:155, 2005.

[30] Eli Biham and Adi Shamir. Differential cryptanalysis of snefru, khafre, redoc-ii, loki and lucifer. In *Advances in CryptologyâĂŤCRYPTOâĂŹ91*, pages 156–171. Springer, 1992.

[31] Alex Biryukov and Johann Grossschadl. Cryptanalysis of the full AES using GPU-like special-purpose hardware. *Fundam. Inf.*, 114(3-4):221–237, 2012.

[32] Matt Blaze and Bruce Schneier. The macguffin block cipher algorithm. In *Fast Software Encryption*, pages 97–110. Springer, 1995.

[33] Uwe Blöcher and Markus Dichtl. Fish: A fast software stream cipher. In *Fast Software Encryption*, pages 41–44. Springer, 1994.

[34] Martin Boesgaard, Mette Vesterager, Thomas Pedersen, Jesper Christiansen, and Ove Scavenius. Rabbit: A new high-performance stream cipher. In *Fast Software Encryption*, pages 307–329. Springer, 2003.

[35] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 450–466. Springer, 2007.

[36] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince-a low-latency block cipher for pervasive computing applications. In *Advances in Cryptology–ASIACRYPT 2012*, pages 208–225. Springer, 2012.

[37] Julia Borghoff, Lars R Knudsen, Gregor Leander, and Krystian Matusiewicz. Cryptanalysis of c2. In *Advances in Cryptology-CRYPTO 2009*, pages 250–266. Springer, 2009.

[38] An Braeken, Joseph Lano, Nele Mentens, Bart Preneel, and Ingrid Verbauwhede. Sfinks: A synchronous stream cipher for restricted hardware environments. In *SKEW-Symmetric Key Encryption Workshop*, 2005.

[39] Ernie F Brickell and Gary L Graunke. Method and apparatus for increasing the speed of cryptographic processing, April 5 2012. US Patent App. 13/440,624.

[40] Jeffrey D. Brown. *The IBM Power Edge of Network^{TM} Processor*. IBM Corporation.

[41] Lawrie Brown and Josef Pieprzyk. Introducing the new loki97 block cipher. In *First AES Candidate Conference*, pages 20–22, 1998.

[42] Frederick J Bruwer, Willem Smit, and Gideon J Kuhn. Microchips and remote control devices comprising same, May 14 1996. US Patent 5,517,187.

[43] Rainer Buchty, Nevin Heintze, and Dino Oliva. Cryptonite - a programmable crypto processor architecture for high-bandwidth applications. In *Organic and Pervasive Computing - ARCS 2004*, volume 2981, pages 184–198, 2004.

[44] Jerome Burke, John McDonald, and Todd Austin. Architectural support for fast symmetric-key cryptography. In *Proceedings of the 9th International Conference on Architectural support for programming languages and operating systems*, ASPLOS IX, pages 178–189, New York, NY, USA, 2000. ACM.

[45] Jerome Burke, John McDonald, and Todd Austin. Architectural support for fast symmetric-key cryptography. In *ASPLOS'00*, pages 178–189, 2000.

[46] Carolynn Burwick, Don Coppersmith, Edward DâĂŹAvignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, S Matyas Jr, Luke OâĂŹConnor, Mohammad Peyravian, David Safford, et al. The mars encryption algorithm, 1999.

[47] Jed Kao-Tung Chang, Chen Liu, Shaoshan Liu, and Jean-Luc Gaudiot. Workload characterization of cryptography algorithms for hardware acceleration. In *ICPE '11*, pages 381–390, 2011.

[48] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis. Cost-efficient SHA hardware accelerators. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(8):999–1008, 2008.

[49] Dong Chen, Guochu Shou, Yihong Hu, and Zhigang Guo. Efficient architecture and implementations of AES. In *3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 6, pages V6–295–V6–298, 2010.

[50] Paul Crowley. Mercy: A fast large block cipher for disk sector encryption. In *Fast Software Encryption*, pages 49–63. Springer, 2001.

[51] Joan Daemen. Cipher and hash function design strategies based on linear and differential cryptanalysis. *Doctoral Dissertation, KU Leuven*, 1995.

[52] Joan Daemen and Craig Clapp. Fast hashing and stream encryption with panama. In *Fast Software Encryption*, pages 60–74. Springer, 1998.

[53] Joan Daemen and Craig Clapp. Fast hashing and stream encryption with panama. In *Fast Software Encryption*, pages 60–74. Springer, 1998.

[54] Joan Daemen, Rene Govaerts, and Joos Vandewalle. Block ciphers based on modular arithmetic. In *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography, Rome, Italy*, page 418, 1993.

[55] Joan Daemen, René Govaerts, and Joos Vandewalle. A new approach to block cipher design. In *Fast Software Encryption*, pages 18–32. Springer, 1994.

[56] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The block cipher square. In *Fast Software Encryption*, pages 149–165. Springer, 1997.

[57] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: Noekeon. In *First Open NESSIE Workshop*, 2000.

[58] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[59] Christophe De Canniere. Trivium: A stream cipher construction inspired by block cipher design principles. In *Information Security*, pages 171–186. Springer, 2006.

[60] D. Denning, J. Irvine, and M. Devlin. A high throughput FPGA Camellia implementation. In *Research in Microelectronics and Electronics*, volume 1, pages 137–140 vol.1, 2005.

[61] Whitfield Diffie and George Ledin. Sms4 encryption algorithm for wireless networks. *IACR Cryptology ePrint Archive*, page 329, 2008.

[62] Hans Dobbertin. Ripemd with two-round compress function is not collision-free. *Journal of Cryptology*, 10(1):51–69, 1997.

[63] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160: A strengthened version of ripemd. In *Fast Software Encryption*, pages 71–82. Springer, 1996.

[64] Donald Eastlake and Paul Jones. Us secure hash algorithm 1 (sha1), 2001.

[65] Patrik Ekdahl and Thomas Johansson. Snow-a new stream cipher. In *Proceedings of First Open NESSIE Workshop, KU-Leuven*, 2000.

[66] Adam J. Elbirt. Reconfigurable computing for symmetric-key algorithms, 2002.

[67] A.J. Elbirt. Fast and efficient implementation of AES via instruction set extensions. In *21st International Conference on Advanced Information Networking and Applications Workshops, AINAW'07.*, volume 1, pages 396–403, 2007.

[68] A.J. Elbirt and C. Paar. An instruction-level distributed processor for symmetric-key cryptography. *Parallel and Distributed Systems, IEEE Transactions on*, 16(5):468–480, 2005.

[69] EnSilica. *eSi-8110 product brief*. EnSilica.

[70] Chih-Peng Fan and Jun-Kui Hwang. Implementations of high throughput sequential and fully pipelined AES processors on FPGA. In *International Symposium on Intelligent Signal Processing and Communication Systems. ISPACS'07*, pages 353–356, 2007.

[71] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228:15–23, 1973.

[72] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family (2008). *Submitted to SHA-3 Competition*.

[73] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hask function family. 2009.

[74] NIST FIPS. 180-2: Secure hash standard (shs). Technical report, Technical report, National Institute of Standards and Technology (NIST), 2001.

[75] A. Murat Fiskiran and Ruby B. Lee. On-chip lookup tables for fast symmetric-key encryption. In *ASAP'05*, ASAP'05, pages 356–363, 2005.

[76] A.M. Fiskiran and R.B. Lee. On-chip lookup tables for fast symmetric-key encryption. In *16th IEEE International Conference on Application-Specific Systems, Architecture Processors, ASAP'05*, pages 356–363, 2005.

[77] D. Fronte, A. Perez, and E. Payrat. Celator: A multi-algorithm cryptographic co-processor. In *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, pages 438–443, 2008.

[78] M.D. Galanis, P. Kitsos, G. Kostopoulos, N. Sklavos, O. Koufopavlou, and C.E. Goutis. Comparison of the hardware architectures and FPGA implementations of stream ciphers. In *Proceedings of the 11th IEEE International Conference on Electronics, Circuits and Systems, ICECS'04*, pages 571–574, 2004.

[79] Berndt M Gammel, Rainer Göttfert, and Oliver Kniffler. The achterbahn stream cipher. *Submission to eSTREAM*, 2005.

[80] Praveen Gauravaram, Lars R Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S Thomsen. Grøstl–a sha-3 candidate. *Submission to NIST*, 2008.

[81] Dianelous Georgoudis, Damian Leroux, and Billy Simon Chaves. The âĂIJfrogâĂİ encryption algorithm. *NIST AES Proposal*, 1998.

[82] Henri Gilbert, Marc Girault, Philippe Hoogvorst, Fabrice Noilhan, Thomas Pornin, Guillaume Poupard, Jacques Stern, and Serge Vaudenay. Decorrelated fast cipher: an aes candidate. In *Extended Abstract.) In Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, 1998.

[83] Jovan Dj. Golic. Cryptanalysis of alleged a5 stream cipher. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*,

EUROCRYPT'97, pages 239–255, Berlin, Heidelberg, 1997. Springer-Verlag.

[84] Robert Golla and Paul Jordan. T4: a highly threaded server-on-a-chip with native support for heterogeneous computing, August, 2011. Slides of a talk given at Hot Chips: A Symposium on High Performance Chips.

[85] Zheng Gong, Svetla Nikova, and Yee Wei Law. Klein: a new family of lightweight block ciphers. In *RFID. Security and Privacy*, pages 1–18. Springer, 2012.

[86] T. Good and M. Benaissa. Pipelined AES on FPGA with support for feedback modes (in a multi-channel environment). *Information Security, IET*, 1(1):1–10, 2007.

[87] Tim Good and Mohammed Benaissa. AES on FPGA from the fastest to the smallest. In JosyulaR. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES'05*, volume 3659, pages 427–440, 2005.

[88] Nick D Goots, Alexander A Moldovyan, and Nick A Moldovyan. Fast encryption algorithm spectr-h64. In *Information Assurance in Computer Networks*, pages 275–286. Springer, 2001.

[89] Philipp Grabher, Johann Grossschadl, and Dan Page. Light-weight instruction set extensions for bit-sliced cryptography. In *Proceedings of the 10th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '08, pages 331–345, 2008.

[90] M. Grand, L. Bossuet, G. Gogniat, B. Le Gal, J.-P. Delahaye, and D. Dallet. A reconfigurable multi-core cryptoprocessor for multi-channel communication systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 204–211, 2011.

[91] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The led block cipher. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 326–341. Springer, 2011.

[92] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 12–23. ACM, 2011.

[93] Wang Haixin, Bai Guoqiang, and Chen Hongyi. Zodiac: System architecture implementation for a high-performance network security processor. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 91–96, 2008.

[94] Shai Halevi, Don Coppersmith, and Charanjit Jutla. Scream: A software-efficient stream cipher. In *Fast Software Encryption*, pages 195–209. Springer, 2002.

[95] Helena Handschuh, H Helena, and David Naccache. Shacal (-submission to nessie-). 2000.

[96] Helena Handschuh and Serge Vaudenay. A universal encryption standard. In *Selected Areas in Cryptography*, pages 1–12. Springer, 2000.

[97] Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing*, 2(1):86–93, 2007.

[98] Martin E Hellman, Bailey W Diffie, and Ralph C Merkle. Cryptographic apparatus and method, April 29 1980. US Patent 4,200,770.

[99] A. Hodjat and I. Verbauwhede. Speed-area trade-off for 10 to 100 Gbits/s throughput AES processor. In *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 2147–2150 Vol.2, 2003.

[100] A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. FCCM'04*, pages 308–309, 2004.

[101] A. Hodjat and I. Verbauwhede. Area-throughput trade-offs for fully pipelined 30 to 70 gbits/s aes processors. *Computers, IEEE Transactions on*, 55(4):366–372, April 2006.

[102] F.S. Hossain, M.L. Ali, and M.A. Al Abedin Syed. A very low power and high throughput AES processor. In *14th International Conference on Computer and Information Technology (ICCIT)*, pages 339–343, 2011.

[103] IBM Corporation. *IBM 4765 PCIe Cryptographic Coprocessor*. IBM.

[104] Intel Coorporation. Pin - A Dynamic Binary Instrumentation Tool, 2012.

[105] IP cores, Inc. AES-GCM MACsec (IEEE 802.1AE) and FC-SP Cores GCM1/GCM2/GCM3, 2013. http://www.ipcores.com/macsec_802.1ae_gcm_aes_ip_core.htm [Online; accessed 22-October-2013].

[106] Keisuke Iwai, Naoki Nishikawa, and Takakazu Kurokawa. Acceleration of AES encryption on CUDA GPU. *International JOURNAL of Networking and Computing*, 2(1), 2012.

[107] N.C. Iyer, P.V. Anandmohan, D.V. Poornaiah, and V. D. Kulkarni. High throughput, low cost, fully pipelined architecture for AES crypto chip. In *Annual IEEE India Conference*, pages 1–6, 2006.

[108] MJ Jacobson Jr and Klaus Huber. The magenta block cipher algorithm. *NIST AES Proposal*, 1998.

[109] Kimmo U. Järvinen, Matti T. Tommiska, and Jorma O. Skyttä. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proceedings of ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays*, FPGA '03, pages 207–215, 2003.

[110] Christipher Jenkins, Michael Schulte, and John Glossner. Instruction set extensions for the advanced encryption standard on a multithreaded software defined radio platform. *International Journal on High Performance System Architecture*, 2(3/4):203–214, August 2010.

[111] Robert J Jenkins Jr. Isaac. In *Fast Software Encryption*, pages 41–49. Springer, 1996.

[112] J Jonsson and B Kaliski. Rfc 3447: Public-key cryptography standards (pkcs)# 1: Rsa cryptography specifications version 2.1. *Request for Comments (RFC)*, 3447, 2003.

[113] P Junod and S Vaudenay. âĂŹidea-nxt specifications, version 1.2,âĂŹ. Technical report, EPFL Technical Report IC/2004/75, 15-04-2005, 2005.

[114] Pascal Junod and Serge Vaudenay. Fox: a new family of block ciphers. In *Selected Areas in Cryptography*, pages 114–129. Springer, 2005.

[115] Burton S Kaliski Jr and Matthew JB Robshaw. Fast block cipher proposal. In *Fast Software Encryption*, pages 33–40. Springer, 1994.

[116] John Kelsey, Bruce Schneier, and David Wagner. Mod n cryptanalysis, with applications against rc5p and m6. In *Fast Software Encryption*, pages 139–155. Springer, 1999.

[117] Joe Kilian and Phillip Rogaway. How to protect des against exhaustive key search. In *Advances in CryptologyâĂŤCRYPTOâĂŹ96*, pages 252–267. Springer, 1996.

[118] Lars Knudsen. Deal-a 128-bit block cipher. *complexity*, 258:2, 1998.

[119] Lars R Knudsen, Vincent Rijmen, Ronald L Rivest, and Matthew JB Robshaw. On the design and security of rc2. In *Fast Software Encryption*, pages 206–221. Springer, 1998.

[120] Kunio Kobayashi and Kazumaro Aoki. On linear cryptanalysis of mbal ciphers. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 82(10):1–8, 1999.

[121] Özgül Küçük. The hash function hamsi. *Submission to NIST (updated)*, 33:167, 2009.

[122] Matthew Kwan. The design of the ice encryption algorithm. In *Fast Software Encryption*, pages 69–82. Springer, 1997.

[123] Daesung Kwon, Jaesung Kim, Sangwoo Park, Soo Hak Sung, Yaekwon Sohn, Jung Hwan Song, Yongjin Yeom, E-Joong Yoon, Sangjin Lee, Jaewon Lee, et al. New block cipher: Aria. In *Information Security and Cryptology-ICISC 2003*, pages 432–445. Springer, 2004.

[124] Xuejia Lai. *On the design and security of block ciphers*. ETH SERIES in Information Processing. Hartung Gorre Verlag, v.1 edition, 1992.

[125] Xuejia Lai and James L Massey. A proposal for a new block encryption standard. In *Advances in CryptologyâĂŤEUROCRYPTâĂŹ90*, pages 389–404. Springer, 1991.

[126] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek. MARC: a many-core approach to reconfigurable computing. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 7–12. IEEE, 2010.

[127] C Lee, K Jun, M Jung, S Park, and J Kim. Zodiac version 1.0 (revised) architecture and specification. In *Standardization Workshop on Information Security Technology, Korean Contribution on MP18033, ISO/IEC JTC1/SC27 N*, volume 2563, page 2000, 2000.

[128] Jaeil Lee, Jongwook Park, Sungjae Lee, and Jeeyeon Kim. The seed encryption algorithm. *SEED*, 2005.

[129] R.B. Lee, Z. Shi, and X. Yang. Efficient permutation instructions for fast software cryptography. *Micro, IEEE*, 21(6):56–69, 2001.

[130] Ruby B. Lee and Yu-Yuan Chen. Processor accelerator for AES. In *Proceedings of the IEEE 8th Symposium on Application Specific Processors (SASP)*, SASP '10, pages 16–21. IEEE Computer Society, 2010.

[131] Chae Hoon Lim. Crypton: A new 128-bit block cipher. *NIsT AEs Proposal*, 1998.

[132] Bin Liu and Bevan M. Baas. Parallel AES encryption engines for many-core processor arrays. *IEEE Transactions on Computers*, 62(3):536–547, 2013.

[133] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifft: A modest proposal for fft hashing. In *Fast Software Encryption*, pages 54–72. Springer, 2008.

[134] Alexis Warner Machado. The nimbus cipher: A proposal for nessie. *NESSIE Proposal, September*, 2000.

[135] WE Madryga. A high performance encryption algorithm. In *Proceedings of the 2nd IFIP international conference on Computer security: a global challenge*, pages 557–569. North-Holland Publishing Co., 1984.

[136] James L Massey. Safer k-64: A byte-oriented block-ciphering algorithm. In *Fast Software Encryption*, pages 1–17. Springer, 1994.

[137] James L Massey. Safer k-64: One year later. In *Fast Software Encryption*, pages 212–241. Springer, 1995.

[138] James L Massey, Gurgen H Khachatrian, and Melsik K Kuregian. Nomination of safer+ as candidate algorithm for the advanced encryption standard (aes). *NIST AES Proposal*, 1998.

[139] Sanu Mathew, Farhana Sheikh, Michael E. Kounavis, Shay Gueron, Amit Agarwal, Steven Hsu, Himanshu Kaul, Mark Anders, and Ram Krishnamurthy. 53 gbps native gf(2 ^{4}) ^{2} composite-field aes-encrypt/decrypt accelerator for content-protection in 45nm high-performance microprocessors. *J. Solid-State Circuits*, 46(4):767–776, 2011.

[140] Mitsuru Matsui. New block encryption algorithm misty. In *Fast Software Encryption*, pages 54–68. Springer, 1997.

[141] Mitsuru Matsui. New block encryption algorithm MISTY. *Fast Software Encryption*, 1267:54–68, 1997.

[142] L McBride. Q: A proposal for nessie v2.00. In *First NESSIE Workshop, Leuven, Belgium*, 2000.

[143] M McLoone. Hardware performance analysis of the shacal-2 encryption algorithm. In *Circuits, Devices and Systems, IEE Proceedings-*, volume 152, pages 478–484. IET, 2005.

[144] M. McLoone and J.V. McCanny. High-performance FPGA implementation of DES using a novel method for implementing the key schedule. *IEEE Proceedings of Circuits, Devices and Systems*, 150(5):373–8–, 2003.

[145] Mercora Technologies. *AES ultra fast ip core for Xilinx FPGAs* . Mercora Technologies.

[146] Ralph C Merkle. A fast software one-way hash function. *Journal of Cryptology*, 3(1):43–58, 1990.

[147] Ralph Charles Merkle. Secrecy, authentication, and public key systems. 1979.

[148] Markus Michels, David Naccache, and Holger Petersen. Gost 34.10âĂŤa brief overview of russia's dsa. *Computers & Security*, 15(8):725–732, 1996.

[149] Shoji Miyaguchi. The feal cipher family. In *Advances in Cryptology-CRYPT0âĂŹ90*, pages 628–638. Springer, 1991.

[150] Alexander A Moldovyan and Nick A Moldovyan. A cipher based on data-dependent permutations. *Journal of Cryptology*, 15(1):61–72, 2002.

[151] S. Morioka and A. Satoh. A 10 Gbps full-AES crypto design with a twisted-BDD S-Box architecture. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 98–103, 2002.

[152] David MâĂŹRaÏhi, David Naccache, Jacques Stern, and Serge Vaudenay. Xmx: A firmware-oriented block cipher based on modular multiplications. In *Fast Software Encryption*, pages 166–171. Springer, 1997.

[153] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *MICRO'07*. 3-14.

[154] Ghulam Murtaza, Azhar Ali Khan, Syed Wasi Alam, and Aqeel Farooqi. Fortification of aes with dynamic mix-column transformation. *IACR Cryptology ePrint Archive*, 2011:184, 2011.

[155] Jorge Nakahara Jr, Vincent Rijmen, Bart Preneel, and Joos Vandewalle. The mesh block ciphers. In *Information Security Applications*, pages 458–473. Springer, 2004.

[156] National Institute of Standards and Technology (NIST). Advanced encryption standard (AES). *Federal Information Processing Standards (FIPS) Publication*, 197, November 2001.

[157] Roger M Needham and David J Wheeler. Tea extensions. computer laboratory, cambridge university, england (1997).

[158] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. High-performance symmetric block ciphers on multicore CPU and GPUs. *IJNC*, 2(2):251–268, 2012.

[159] NIST. Skipjack and kea algorithm speciiňĄcation,. *NIST Technical Report*, 1998.

[160] Secure Hash Standard NIST and NIST FIPS PUB. 180. *Secure hash standard, National Institute of Standards and Technology, US department of Commerce, DRAFT*, 1993.

[161] W.M. Nunan Zola and L.C.E. De Bona. Parallel speculative encryption of multiple AES contexts on GPUs. In *Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[162] Kaisa Nyberg and Lars Ramkilde Knudsen. Provable security against a differential attack. *Journal of Cryptology*, 8(1):27–37, 1995.

[163] Register of Cryptographic Algorithms. Iso/iec9979-0012 register entry, 1995.

[164] Register of Cryptographic Algorithms. Iso/iec9979-0020 register entry, 1995.

[165] K Ohkuma, H Muratani, F Sano, M Motoyama, and S Kawamura. ; security and performance evaluations for the block ciphers hierocrypt-3 and hierocrypt-l1. *IEIC Technical Report (Institute of Electronics, Information and Communication Engineers)*, 100(324):71–100, 2000.

[166] Kenji Ohkuma, Hirofumi Muratani, Fumihiko Sano, and Shinichi Kawamura. The block cipher hierocrypt. *"Lecture Notes in Computer Science"*, 2012:72, 2001.

[167] Sean O'Neil, Benjamin Gittins, and Howard A Landman. Vest hardware-dedicated stream ciphers. *IACR Cryptology ePrint Archive*, 2005:413, 2005.

[168] Slobodan Petrovic and Amparo Fuster-Sabater. Cryptanalysis of the a5/2 algorithm. *IACR Cryptology ePrint Archive*, 2000:52, 2000.

[169] Josef Pieprzyk and Leonid Tombak. Soviet encryption algorithm. *preprint*, pages 94–10, 1993.

[170] T. Pionteck, T. Staake, T. Stiefmeier, L.D. Kabulepa, and M. Glesner. Design of a reconfigurable AES encryption/decryption engine for mobile terminals. In *Proceedings of*

the 2004 International Symposium on Circuits and Systems, ISCAS'04., volume 2, pages 545–556. IEEE, 2004.

[171] Shanxin Qu, Guochu Shou, Yihong Hu, Zhigang Guo, and Zongjue Qian. High throughput, pipelined implementation of AES on FPGA. In *International Symposium on Information Engineering and Electronic Commerce. IEEC'09*, pages 542–545, 2009.

[172] James A Reeds III. Cryptosystem for cellular telephony, October 27 1992. US Patent 5,159,634.

[173] Vincent Rijmen, Joan Daemen, Bart Preneel, Antoon Bosselaers, and Erik De Win. The cipher shark. In *Fast Software Encryption*, pages 99–111. Springer, 1996.

[174] Terry Ritter. Ladder-des: A proposed candidate to replace des, appeared in the usenet newsgroup sci. crypt, 1994.

[175] Ronald Rivest. Rfc 1321: The md5 message-digest algorithm, april 1992. *Status: INFORMATIONAl*.

[176] Ronald Rivest. The md4 message-digest algorithm, rfc 1320, 1992.

[177] Ronald Rivest. Rfc 1319: The md2 message digest algorithm, 1992.

[178] Ronald L Rivest. The rc5 encryption algorithm. In *Fast Software Encryption*, pages 86–96. Springer, 1995.

[179] Ronald L Rivest, Benjamin Agre, Daniel V Bailey, Christopher Crutchfield, Yevgeniy Dodis, Kermin Elliott Fleming, Asif Khan, Jayant Krishnamurthy, Yuncheng Lin, Leo

Reyzin, et al. The md6 hash function–a proposal to nist for sha-3. *Submission to NIST*, 2:3, 2008.

[180] Ronald L Rivest, MJB Robshaw, Ray Sidney, and Yiqun Lisa Yin. The rc6 block cipher. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[181] M. R M Rizk and M. Morsy. Optimized area and optimized speed hardware implementations of AES on FPGA. In *2nd International Design and Test Workshop. IDT'07.*, pages 207–217, 2007.

[182] MJB Robshaw. A cryptographic review of cipherunicorn-a. 2001.

[183] MJB Robshaw. A cryptographic review of cipherunicorn-e. 2001.

[184] Phillip Rogaway and Don Coppersmith. A software-optimized encryption algorithm. In *Fast Software Encryption*, pages 56–63. Springer, 1994.

[185] Gregory G Rose and Philip Hawkes. Turing: A fast stream cipher. In *Fast Software Encryption*, pages 290–306. Springer, 2003.

[186] Leelavathi G. Sagar D. Design and implementation of extended version of AES algorithm with DSP units. In *International JOURNAL of Engineering and Advanced Technology (IJEAT)*, volume 2-6, pages 360–364, 2013.

[187] P. Saravanan, N. Renuka Devi, G. Swathi, and Dr. P. Kalpana. A high-throughput ASIC implementation of configurable advanced encryption standard (AES) processor. *IJCA Special Issue on Network Security and Cryptography*, NSC(3):1–6, December 2011. Published by Foundation of Computer Science, New York, USA.

[188] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption*, pages 191–204. Springer, 1994.

[189] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. *NIST AES Proposal*, 15, 1998.

[190] Rich Schroeppel. Hasty pudding cipher specification. *NIST AES Proposal*, 1998.

[191] Robert Scott. Wide-open encryption design offers flexible implementations. *Cryptologia*, 9(1):75–91, 1985.

[192] Beale Screamer. MicrosoftâĂŹs digital rights management scheme–technical details (october 2001).

[193] Z. Shi and R.B. Lee. Bit permutation instructions for accelerating software cryptography. In *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 138–148, 2000.

[194] Akihiro Shimizu and Shoji Miyaguchi. Fast data encipherment algorithm feal. In *Advances in CryptologyâĂŤEUROCRYPTâĂŹ87*, pages 267–278. Springer, 1988.

[195] Takeshi Shimoyama, Hitoshi Yanami, Kazuhiro Yokoyama, Masahiko Takenaka, Kouichi Itoh, Jun Yajima, Naoya Torii, and Hidema Tanaka. The block cipher sc2000. In *Fast Software Encryption*, pages 312–327. Springer, 2002.

[196] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher clefia. In *Fast software encryption*, pages 181–195. Springer, 2007.

[197] Ken Shirriff. Differential cryptanalysis of redoc iii.

[198] M.I. Soliman and G.Y. Abozaid. Performance evaluation of a high throughput crypto coprocessor using VHDL. In *International Conference on Computer Engineering and Systems (ICCES)*, pages 231–237, 2010.

[199] Mostafa I. Soliman and Ghada Y. Abozaid. FastCrypto: parallel AES pipeline extension for general-purpose processors. *Neural, Parallel Sci. Comput.*, 18(1):47–58, March 2010.

[200] Arthur Sorkin. Lucifer, a cryptographic algorithm. *Cryptologia*, 8(1):22–42, 1984.

[201] Jacques Stern and Serge Vaudenay. Cs-cipher. In *Fast Software Encryption*, pages 189–204. Springer, 1998.

[202] J.E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, Julie Oh, and R. Jenkal. Freepdk: An open-source variation-aware design kit. In *IEEE International Conference on Microelectronic Systems Education, MSE '07*, pages 173–174, 2007.

[203] Sun Microsystems. x86 Assembly Language Reference Manual, 2012.

[204] E. J. Swankoski, R.R. Brooks, V. Narayanan, M. Kandemir, and M.J. Irwin. A parallel architecture for secure FPGA symmetric encryption. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*, pages 132–, 2004.

[205] Eric Swankoski and Vijaykrishnan Narayanan. Dynamic high-performance multi-mode architectures for AES encryption. In *Intenational Conference on Military and Aerospace Programmable Logic Devices*, pages 1–9, 2005.

[206] R. Reed Taylor and Seth Copen Goldstein. A high-performance flexible architecture for cryptography. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems 1999 (CHES99)*, pages 231–245, August 1999.

[207] Nippon Telegraph. Telephone corporation,âĂIJspecification of e2âĂŤa 128-bit block cipher,âĂİ 1999.

[208] Dimitris Theodoropoulos, Alexandros Siskos, and Dionisis Pnevmatikatos. Ccproc: A custom vliw cryptography co-processor for symmetric-key ciphers. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 318–323. Springer, 2009.

[209] Toshio Tokita and Mitsuru Matsui. Linear cryptanalysis of block cipher xenon. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 86(1):13–18, 2003.

[210] Serge Vaudenay. Provable security for block ciphers by decorrelation. In *STACS 98*, pages 249–275. Springer, 1998.

[211] VIA Technologies, Inc. VIA C7 processor, 2013. http://www.via.com.tw/en/products/processors/c7/ [Online; accessed 22-October-2013].

[212] Yi Wang and Yajun Ha. FPGA-based 40.9-Gbits/s masked AES with area optimization for storage area network. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(1):36–40, 2013.

[213] Rick Wash. Lecture notes on stream ciphers and rc4. *Reserve University*, pages 1–19, 2001.

[214] Dai Watanabe, Soichi Furuya, Hirotaka Yoshida, Kazuo Takaragi, and Bart Preneel. A new keystream generator mugi. In *Fast Software Encryption*, pages 179–194. Springer, 2002.

[215] David J Wheeler. A bulk data encryption algorithm. In *Fast Software Encryption*, pages 127–134. Springer, 1994.

[216] David J Wheeler and Roger M Needham. Tea, a tiny encryption algorithm. In *Fast Software Encryption*, pages 363–366. Springer, 1995.

[217] Doug Whiting, Bruce Schneier, Stefan Lucks, and Frédéric Muller. Fast encryption and authentication in a single cryptographic primitive. *ECRYPT Stream Cipher Project Report*, 27(200):5, 2005.

[218] Hongjun Wu. A new stream cipher hc-256. In *Fast Software Encryption*, pages 226–244. Springer, 2004.

[219] Hongjun Wu. The hash function jh. *Submission to NIST (round 3)*, 2011.

[220] Hongjun Wu and Bart Preneel. Distinguishing attack on stream cipher yamb. *eSTREAM The ECRYPT Stream Cipher Project*, (2005/043), 2005.

[221] L. Wu, C. Weaver, and T. Austin. CryptoManiac: a fast flexible architecture for secure communication. In *Proceedings of 28th Annual International Symposium on Computer Architecture*, pages 110–119, 2001.

[222] Wenling Wu and Dengguo Feng. Linear cryptanalysis of nush block cipher. *Science in China Series F: Information Sciences*, 45(1):59–67, 2002.

[223] P. Yalla and J. Kaps. Compact FPGA implementation of Camellia. In *International Conference on Field Programmable Logic and Applications, FPL'09.*, pages 658–661, 2009.

[224] Ming Yan, Ziyu Yang, Lei Liu, and Sikun Li. Prodfa: Accelerating domain applications with a coarse-grained runtime reconfigurable architecture. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 834–839. IEEE, 2012.

[225] Elias Yarrkov. Cryptanalysis of xxtea. *IACR Cryptology ePrint Archive*, 2010:254, 2010.

[226] Xun Yi, Chik How Tan, Chee Kheong Slew, and M Rahman Syed. Fast encryption for multimedia. *Consumer Electronics, IEEE Transactions on*, 47(1):101–107, 2001.

[227] Seong-Moo Yoo, Deen Kotturi, W. David Pan, and John Blizzard. An AES crypto chip using a high-speed parallel pipelined architecture. *Microprocessors and Microsystems*, 29(7):317–326, 2005.

[228] Xinmiao Zhang and K.K. Parhi. High-speed VLSI architectures for the AES algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(9):957–967, 2004.

[229] Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry. Haval-a one-way hashing algorithm with variable length of output. In *Advances in CryptologyâĂŤAUSCRYPT'92*, pages 81–104. Springer, 1993.