NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A STATISTICAL ANALYSIS OF SOME STANDARD CIPHERS' CRYPTOGRAPHIC PRIMITIVES

by

Devon Zillmer

June 2021

Thesis Advisor:	Pantelimon Stanica
Co-Advisor:	Robert L. Bassett
Second Reader:	Thor Martinsen

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE A STATISTICAL ANALYSIS OF SOME STANDARD CIPHERS' CRYPTOGRAPHIC PRIMITIVES	**5. FUNDING NUMBERS**
6. AUTHOR(S) Devon Zillmer	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Encryption is ubiquitous in the modern environment. While public/private key architecture has provided an amazing and powerful way to encrypt information so that only one intended recipient can decrypt, the computation required for this approach means that this encryption method can quickly grow extremely expensive. With that in mind, there are a variety of open-source stream ciphers that seek to provide relatively inexpensive stream ciphers to securely encrypt information. But these stream ciphers all operate using very different techniques to generate their keystream, as seen in the stark differences in paradigms between ciphers. As such, it is not immediately clear what operations are required to achieve the desired level of encryption. What cryptographic primitives are most common or efficacious in achieving security? Examining the Data Encryption Standard, Advanced Encryption Standard, and the stream cipher winners of the eStream II competition, an underlying trend composed of two operations emerges. Despite observing no clear n-grams defining precise cryptographic primitives, we identify a general structure common to all stream ciphers. Additionally, we identify that substitution boxes or multiplication operations are not necessary for stream ciphers, whereas addition and rotation operations seem to be essential.

14. SUBJECT TERMS cryptography, cipher, algebra, eStream	**15. NUMBER OF PAGES** 97
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

A STATISTICAL ANALYSIS OF SOME STANDARD CIPHERS' CRYPTOGRAPHIC PRIMITIVES

Devon Zillmer
Major, United States Army
BS, U.S. Military Academy, 2010

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

and

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
June 2021

Approved by: Pantelimon Stanica
Advisor

Robert L. Bassett
Co-Advisor

Thor Martinsen
Second Reader

Wei Kang
Chair, Department of Applied Mathematics

W. Matthew Carlyle
Chair, Department of Operations Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Encryption is ubiquitous in the modern environment. While public/private key architecture has provided an amazing and powerful way to encrypt information so that only one intended recipient can decrypt, the computation required for this approach means that this encryption method can quickly grow extremely expensive. With that in mind, there are a variety of open-source stream ciphers that seek to provide relatively inexpensive stream ciphers to securely encrypt information. But these stream ciphers all operate using very different techniques to generate their keystream, as seen in the stark differences in paradigms between ciphers. As such, it is not immediately clear what operations are required to achieve the desired level of encryption. What cryptographic primitives are most common or efficacious in achieving security? Examining the Data Encryption Standard, Advanced Encryption Standard, and the stream cipher winners of the eStream II competition, an underlying trend composed of two operations emerges. Despite observing no clear n-grams defining precise cryptographic primitives, we identify a general structure common to all stream ciphers. Additionally, we identify that substitution boxes or multiplication operations are not necessary for stream ciphers, whereas addition and rotation operations seem to be essential.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AES	Advanced Encryption Standard
ARO	Add Rotate Obscure
ASCII	American Standard Code for Information Interchange
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CTR	Counter (Mode of Block Cipher Operation)
DES	Data Encryption Standard
ECB	Electronic Codebook
ECRYPT	European Network of Excellence in Cryptology
EU	European Union
FSM	Finite State Machine
IBM	International Business Machines (Corporation)
IV	Initial Value
JCR	Joint Capabilities Release
LFSR	Linear Feedback Shift Register
NBS	National Bureau of Standards
NFSR	Nonlinear Feedback Shift Register
NIST	National Institute of Standard and Technology
NSA	National Security Agency

OFB Output Feedback

PRNG Pseudo-Random Number Generator

S-Box Substitution Box

SPN Substitution Permutation Network

UTF Unicode Transformation Format

XOR Exclusive OR logical operator

Acknowledgments

I am greatly indebted to several people and institutions.

Professor Stănică, you provided the spark of an idea and several (insightful) gusts of fresh air that helped motivate and fan the flames from a spark to the resultant bonfire. Thank you for your support, assistance, and patience with a verbose student, as a thesis advisor and instructor.

Professor Bassett, despite my tardiness and poor communication skills, you were willing to work with me and help me fulfill the myriad of requirements for someone in my shoes. Thank you for your assistance, guidance, and feedback through all of this.

Commander Martinsen, amid an already busy schedule you helped to provide your thoughts, feedback, and assistance as a second reader. Thank you, sir.

I would be remiss without mentioning three institutions that formed not only my limited intelligence but also my character. In chronological order these are: the United States Military Academy; the United States Army; and the one, holy, catholic, and apostolic Church. There have been countless individuals within these organizations that have had led, challenged, and shaped me, and I only hope to contribute so meaningfully as I move forward.

To many in the Applied Mathematics and Operations Research departments, including students and instructors, military and civilian, who helped educate and encourage me on my return to academics after a decade of infantry activities, I am most grateful.

Finally, to my beautiful, brilliant, and tenacious bride, Rachel. Who else would have remained with me despite my (apparent) best efforts to chase her away? Through endless months of Ranger school, multiple deployments, and a job (or work ethic) that always demands more time and attention, she remains the highlight of every day and the best part of my life. Your support through this (including such iconic lines as “Oh, I didn’t know you could write!”) has been invaluable. I love you.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

People keep secrets.

Those secrets might be good or bad; this thesis will not engage in the moral dimension of this human phenomenon. In many cases in the modern world, however, people simply wish to be able to share or withhold information: financial information, online transactions, or simply personal matters one does not desire to share with the world. A variety of ways exist to keep information private, especially in the information age. Encryption is a common way to make a message, transaction, or other information private. To use the words from the announcement of the Advanced Encryption Standard (AES) in [1] in 2001: "Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back to its original form, called plaintext."

There are two basic approaches to encryption: symmetric and asymmetric. Symmetric encryption uses the same information for encryption as for decryption, hence its name. Asymmetric encryption, however, uses different techniques to encrypt and decrypt, relying on a fundamentally different set of operations. This paper will primarily focus on symmetric ciphers. The two kinds of symmetric ciphers we discuss here are stream and block ciphers. Block ciphers seek to take a "block" of information and encrypt it, ideally ensuring that no one without access to the proper key can read it. Stream ciphers, on the other hand, seek to produce a "stream" of encryption which is often used to encrypt a "flow" of data to prevent others from being able to interpret the data (without the proper key). This thesis compares and contrasts the foundational mathematical operations and sequences of a collection of symmetric ciphers to search for common effective cryptographic primitives.

In addition to Data Encryption Standard (DES) and AES, the major symmetric ciphers considered in this thesis are the winners of the European Network of Excellence in Cryptology (ECRYPT) II Network of Excellence Competition. ECRYPT is a collection of organizations funded largely by the European Union (EU) through the ECRYPT Coordination and Support Action. The goal of ECRYPT is to support excellence in cryptology in academics, industry, and government agencies. In addition to hosting ongoing opportunities related to

improvement in these areas, ECRYPT also hosted two major conferences to provide open-source advances in the field, usually referred to as ECRYPT I and II. Of the publications from these conferences, there were seven steam ciphers selected as the "best of" ECRYPT II's eSTREAM competition in 2008. To date, none of these ciphers' full algorithms have been successfully attacked. As such, they provide examples of reliably secure ciphers for our consideration of efficacious cryptographic primitives.

1.1 Motivation

Encryption of information is ubiquitous. If an employee has a wireless mouse, headset, or cellular phone that connects to a vehicle via Bluetooth, then they have interacted with a wireless encryption method. If a student conducts online banking, purchases something from a vendor's website, or sends a digitally signed email, then they have first-hand experience with digital authentication and encryption. For those in uniform, a servicemember who has "filled" a radio to facilitate cipher-text transmission, programmed a Joint Capabilities Release (JCR) for operation, or used a classified computer, that servicemember has relied on the information security provided by encryption. All of these examples use the principles of encryption through various implementations to assure data confidentiality. This thesis seeks to examine the cryptographic primitives, or effective building blocks of encryption, in some common ciphers to identify trends and commonalities across some of these kinds of applications.

In large part, we rely on these systems because they are secure. But a balance has to be struck between the strength of the security and the complexity of encryption. On the one hand, such as in the online banking example, a single transfer requires small amounts of information to be exchanged at a time, and so can afford intense computational efforts to ensure security via asymmetric encryption. On the other hand, we consider two constraints that prevent the use of the same high standard. One constraint is seen in the wireless encryption example previously, where we see physical limits to the computation power available for encryption, leading to the development of efficient hardware-oriented ciphers. Another example is when there is a large amount of data that must be encrypted for transmission and quickly decrypted to facilitate a continuous flow of information. Each of these scenarios leverages different methods of symmetric encryption.

This thesis will primarily focus on stream ciphers, which are divided into two different categories to address these last two of the above examples: hardware-oriented and software-oriented ciphers. These cipher paradigms rely on different assumptions, resulting in different approaches to balancing the trade-offs between speed and security. In order to discuss the ciphers in depth, the remainder of this chapter provides an overview of the concepts required to discuss the ciphers. We first build up the mathematics, including integer representation, modular arithmetic, and a brief explanation of algebra from groups to extension fields. After the mathematics overview, we discuss some of the theoretical goals, common inputs, and briefly introduce an unfamiliar reader to the concept of linearity and linear cryptanalysis because of its importance in cryptological system design. We then close the chapter with a quick summary of different cipher types, review the concept of a Substitution Box (S-Box) and a cryptographic hash, and introduce the notation used in subsequent chapters.

We note, before we progress to our background material, that the field of cryptography is necessarily open-ended. Advances in technology (such as quantum computing), new applications of mathematical concepts (as seen in the introduction of public key cryptography), or years of scrutiny (as with DES) can reveal weaknesses in established ciphers or change the way cryptographers think. Perhaps no cipher is truly secure, if only because of advances in computing and brute force attacks. We choose to focus on established, well-studied, and currently secure ciphers to helps us gain insight into secure trends.

1.2 Mathematics of Cryptography

To discuss the mechanics of cryptography, this section provides an overview and introduction to the mathematics required to discuss assorted ciphers. We begin with the integers and their representation, move to modular arithmetic, and then discuss the foundational concepts required from algebra to appreciate the operations involved in symmetric cipher cryptography.

1.2.1 Integer Representation

In everyday usage, people use decimal notation to express values of integers. The word "decimal" is itself an indication that the base unit for this expansion is ten. As an example, the number 114 is used to denote $1 \cdot 10^2 + 1 \cdot 10^1 + 4 \cdot 10^0$. Decimal notation is common in

many everyday circumstances, but in information theory or cryptography, base 2 and base 16 are often used to express values. Values expressed this way are often referred to as binary and hexadecimal, respectively. See Table 1.1 for an example of equivalent expansions in these three major bases.

Decimal	1	2	3	4	5	6	7	8	9
Binary	1	10	11	100	101	110	111	1000	1001
Hexadecimal	1	2	3	4	5	6	7	8	9

Decimal	10	11	12	13	14	15	16
Binary	1010	1011	1100	1101	1110	1111	10000
Hexadecimal	A	B	C	D	E	F	10

Table 1.1. Integer Expansions

A note on notation: sometimes a number in hexadecimal can be written $(10)_{16}$, but in this thesis we will generally write 0x10. There are many reasons to express integers in this fashion, but in this thesis we will often do so for consistency and to express information in a more compact fashion. For a further discussion or efficient algorithms for converting between integer bases, see Chapter 4 in [2] or [3].

1.2.2 Modular Arithmetic

In some circumstances, we are not concerned with all of the details in a given problem. An example: what time will it be in 72 hours? Our answer is, of course: the same time as it is now. In this case, we are not concerned with how many days pass, we are only concerned with the time of day. This is an example of modular arithmetic, which is this section's focus.

If we have two integers p, q, and $q \neq 0$, we commonly say that p divides q if there exists some number r such that $q = pr$. In this case, we say that p and r are factors or divisors of p, and conversely that q is a multiple of p and r. To denote the relationship between p and

q, we use the notation $p \mid q$, read as "p divides q". We note that if this is not true, we say p does not divide q, or $p \nmid q$.

The greatest common divisor of two integers p, q is the largest integer that divides them both, often written $\gcd(p, q)$. If the greatest common divisor is 1, then the two integers are co-prime. The least common multiple of integers p, q is the smallest possible number n such that both $p \mid n$ and $q \mid n$, and is denoted $\text{lcm}(p, q)$.

Modular arithmetic rests squarely on the **division algorithm**, which states that for an integer n, there are a unique combination of integers d, q and r, such that $n = dq + r$. Expressed in words, we would say that an integer n can be represented by combining three unique numbers d, q, r by first taking the product of a divisor (d) and quotient (q), then adding the remainder (r).

In **modular arithmetic**, we are only concerned with the remainder. From the example where we are computing time, we are only concerned with the remainder after all the full days have passed; we are thus operating modulo 24. In mathematical terms, we were saying $72 \mod 24 = 0$. Because we are only concerned with the remainder term in modular arithmetic, we define the concept of **congruence** as: $a \equiv b \mod c$, if there exists an n such that $a = nc + b$. Using the above example, we would say that $0 \equiv 72 \mod 24$.

We note that the idea of dividing an element by another, and only retaining the remainder, is a concept that does not just apply to integers, but to polynomials as well. We will provide additional examples of modular arithmetic in the following section. For more information on the development and applications of modular arithmetic, see Chapter 3 in [4] or Chapter 4 in [2].

1.2.3 Algebra Introduction

To build up to an appreciation of the algebra involved in cryptography, we will begin with an introduction to a subset of common sets, build from groups to rings to fields, discuss the concept of irreducible polynomials, and then outline how to build extension fields. All of these key concepts are discussed in [5]. This is all for the purpose of demonstrating the mathematical underpinnings of the Linear Feedback Shift Register (LFSR), which is a key component in many ciphers.

In our previous time example, there are a few ways we can consider expressing our situation. On the one hand, the number of whole hours that pass can be represented as an element $t \in \mathbb{Z}$, but this is not commonly done. More often, we consider the hour of the day as an element in the set $\{1, 2, \ldots, 12\}$ (or outside the United States represented from 0 to 24), which can be represented with a bijection of the elements from $\mathbb{Z}_{12}$ [1]. In general, a subset of the integers that only includes from 0 to $n-1$ is written $\mathbb{Z}_n$. The remainder of this section will discuss special properties of these kinds of sets when we are using the operation of addition.

Groups

A **group**, denoted $\langle G, * \rangle$, is a set G that is closed under a binary operation, denoted by $*$, which satisfies the following three axioms:

- G_1: Identity. There is an identity element e in the group G which does not change any other element when used with $*$. Formally, we say $\exists e \in G \ni \forall a \in G, a*e = e*a = a$.
- G_2: Invertible. Each element in G has an inverse element, which when input with $*$ returns the identity element from G_1. Formally, $\forall a \in G, \exists a^{-1} \in G \ni a * a^{-1} = e$.
- G_3: Associative. For any three elements in G, we can group the elements together without changing the output. $\forall a, b, c \in G, a * (b * c) = (a * b) * c$.

We note the formal notation $\langle G, * \rangle$ is often referred to simply as Group G in subsequent references to save space, so long as the meaning should be clear. If the axiom G_4 below is included, we say that G is an Abelian group.

- G_4: Commutative. The order of operations between any two elements of the group produces the same output. Formally, $\forall a, b \in G, a * b = b * a$.

The "time of day" example we have been considering is useful: the hours of the day are a group under the operation of addition. Mathematically, we describe this group as $\langle \mathbb{Z}_{12}, + \rangle$. This time example is straightforward, but provides a simple explanation for a behavior that applies equally well to any such $\langle \mathbb{Z}_n, + \rangle$, including when we consider binary or hexadecimal expansions. For an example of how we compute binary addition, or $\langle \mathbb{Z}_2, + \rangle$, see Table 1.2.

[1] To transform the elements of the hours of the day to $\mathbb{Z}_{12}$, we need only add one to each of the elements of $\mathbb{Z}_{12}$ to see the hours of the day.

+	0	1
0	0	1
1	1	0

Table 1.2. Group Table for $\langle \mathbb{Z}_2, + \rangle$

While a table like Table 1.2 can be used to look up values in the group, more often one simply computes the addition modulo n. In bit-wise binary addition as well as in our hours of the day example, we can conduct modular arithmetic or look up the answer from the group table. For more information on Group theory, see Part I in [5].

Rings

A **ring**, denoted $\langle R, +, \cdot \rangle$, is a set R that is closed under two binary operators $+, \cdot$ that satisfy the following three axioms. We note that the traditional meanings for addition and multiplication are associated with these binary operations:

- R_1: $\langle R, + \rangle$ is an abelian group.
- R_2: Multiplication is associative. This is the same meaning as in G_3 above.
- R_3: Distributive Law. This ensures that multiplication "distributes across" addition operations the way one normally sees when operating in $\mathbb{R}$. This includes right- and left-distribution. To be precise, we say that $\forall a, b, c \in R, a \cdot (b + c) = (a \cdot b) + (a \cdot c)$, as well as $\forall a, b, c \in R, (a + b) \cdot c = (a \cdot c) + (b \cdot c)$.
- R_4: Commutative. For the purposes of this thesis, we will only consider commutative rings. This means that $\forall a, b \in R, a \cdot b = b \cdot a$.

These four axioms hold in many of the mathematical circumstances in which we find ourselves. For example, if we take operation over hexadecimal characters, one can easily verify all the above for $\langle \mathbb{Z}_{16}, +, \cdot \rangle$. If we are operating in this ring, and only using one hexadecimal "digit" when we add or multiply two hexadecimal values, we are actually conducting addition and multiplication modulo 16. This can look odd at first, if for example we say 0x9 + 0x9 = 0x2, because (in base 10) $9 + 9 = 18 \mod 16 \equiv 2 \mod 16$. We note that the algebraic concept of a Ring is not enough for our purposes in cryptography, because there is a crucial gap: we cannot invert our "$\cdot$" operation. So we must discuss additional

algebraic structure to complete our mathematical underpinnings.

Fields

In order to define a field, we must first define three additional concepts. First, a **ring with unity** is a ring as defined above that has a multiplicative identity. This is comparable to the additive identity discussed in G_1 above, but under our multiplication operation. Second, each element in that ring that has a multiplicative inverse is called a **unit**. Finally, a **division ring** is one in which every nonzero element of R is a unit. We collect these ideas into axiom R_5:

- R_5: Multiplicative Inverse. $\forall a \in R, a \cdot a^{-1} = a^{-1} \cdot a = 1$, where the multiplicative identity is denoted by 1.

A set R with binary operations addition and multiplication, denoted $\langle R, +, \cdot \rangle$, which meets the conditions R_1 through R_5, is called a **field**. We can also express a field succinctly as a commutative division ring.

We note that many common sets are not fields. As an example, $\langle \mathbb{Z}, +, \cdot \rangle$ is a commutative ring, but not a field, because many elements do not have multiplicative inverses.

The application towards which we are building is the use of elements of a ring or field as coefficients for a polynomial. In general, a polynomial $f(x)$ with coefficients $a \in \langle R, +, \cdot \rangle$, can be expressed as:

$$f(x) = \sum_{i=0}^{\infty} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n + \cdots .$$

As an example, $\mathbb{Z}[x]$ is a ring of polynomials using the variable x and coefficients from $\mathbb{Z}$. We note that this is a ring, not a field, because we cannot form multiplicative inverses of all elements from the integers.

We are especially interested in finite fields. One way to construct a finite field is by starting with the ring of integers and evaluating each element modulo a prime number p. This maps every element of $\mathbb{Z}$ to a coset corresponding to a number between 0 and $p - 1$ (because as discussed in Section 1.2.2, we retain only the remainder from the modulus operation).

This generates the set $\mathbb{Z}_p$, which we incorporate with the binary operations of addition and multiplication as $\langle \mathbb{Z}_p, +, \cdot \rangle$. This is often simplified to $\mathbb{F}_p$, and is also referred to as a Galois Field of order p, or GF(p). If we let $p = 5$, we can see Tables 1.3 and 1.4 to verify all the criteria for fields from above to verify that $\langle \mathbb{Z}_5, +, \cdot \rangle$ is a group.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Table 1.3. Group Table for $\langle \mathbb{Z}_5, + \rangle$

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Table 1.4. Group Table for $\langle \mathbb{Z}_5, \cdot \rangle$

We note from Table 1.4 that each element in $\mathbb{Z}_5$ has an inverse, which addresses a key problem when we were trying to build a ring with the integers. For additional exploration of rings, field, or polynomials, see Parts IV and VI in [5].

Irreducible Polynomials

In order to introduce our final algebraic structure, we must first examine the concept of **irreducible polynomials** over a field F. We recall that, when we are operating in $\mathbb{R}[x]$, a reducible polynomial is one where we can think about "factoring" the polynomial. But in this thesis we consider polynomials whose coefficients are from specific fields, which means we cannot classify a given polynomial, $f(x)$, as irreducible or reducible without understanding the field from which its coefficients are drawn.

As a very well-known example, we consider the polynomial $f(x) = x^2 + 1$. If $f(x) \in \mathbb{R}[x]$, then there are no roots, and so we can assess that $f(x)$ is irreducible. If, however, $f(x) \in \mathbb{C}[x]$, then we know there are two distinct roots, and we can factor as $f(x) = (x + i)(x - i)$. This idea of an irreducible polynomial over a field is required to build extension fields.

Extension Fields

We can use the same technique used to generate the finite field $\mathbb{Z}_p$ to create an extension field. Simply put, an **extension field** is built by "dividing out" by an element from the original set to build a new structure (consisting of cosets). In the case of dividing out by a prime integer above, we used an infinite ring to build a finite field; we can also "divide out" a polynomial ring by an irreducible polynomial to generate a field extension, with which we have additional structure that we can leverage.

As an example, we consider $\mathbb{Z}_2[x]$, the ring of polynomials with binary coefficients. This is an infinite set, but if we divide every element by an irreducible polynomial and retain the remainder we are left with a finite field. We let $p(x) = x^2 + x + 1$, which is irreducible in $\mathbb{Z}_2$ (we can verify by substituting in 0 and 1 to see that there are no roots in $\mathbb{Z}_2$). Below we consider monomials from degree 0 to 5 in $\mathbb{Z}_2[x]$ to see how an infinite set is reduced to a finite collection of cosets, using a reduced vector-style notation to indicate the coefficients of the first 5 powers of x (up to x^5):

- $x^0 \mod (x^2 + x + 1) \equiv 1$
- $x^1 \mod (x^2 + x + 1) \equiv x$
- $x^2 \mod (x^2 + x + 1) \equiv x + 1$
- $x^3 \mod (x^2 + x + 1) \equiv 1$
- $x^4 \mod (x^2 + x + 1) \equiv x$
- $x^5 \mod (x^2 + x + 1) \equiv x + 1$

For a few examples of how to conduct polynomial long division, see Appendix A.1. We note that in the above we have mapped an infinite set into four cosets, and that for polynomials other than those indicated, we can simply add the desired terms and see that we are still left with only the elements of our finite field of size 4. As an example, if we take the sum of two

terms from above, x^3 and x^5, we have

$$x^5 + x^3 \mod p(x) \equiv x + 1 + 1 \mod p(x) = x \mod p(x)$$

because our coefficients are in $\mathbb{Z}_2$.

What we have just done is build a field $\mathbb{F}_4$ by "dividing" the original set by the generator polynomial $p(x)$. This is usually written in the form $\mathbb{F}_4 = \mathbb{Z}_2[x]/\langle p(x)\rangle$. If we consider α to be a root of our polynomial $p(x)$, then our field $\mathbb{F}_4$ now contains this α, despite the original set, $\mathbb{Z}_2$, not including a solution. Tables 1.5 and 1.6 show the addition and multiplication tables for this α. See A.2 for a walkthrough explanation of how these values were calculated.

$+$	0	1	α	$1+\alpha$
0	0	1	α	$1+\alpha$
1	1	0	$1+\alpha$	α
α	α	$1+\alpha$	0	1
$1+\alpha$	$1+\alpha$	α	1	0

Table 1.5. Group Table for $\langle \mathbb{F}_4, + \rangle$

$\cdot$	0	1	α	$1+\alpha$
0	0	0	0	0
1	0	1	α	$1+\alpha$
α	0	α	$1+\alpha$	1
$1+\alpha$	0	$1+\alpha$	1	α

Table 1.6. Group Table for $\langle \mathbb{F}_4, \cdot \rangle$

Field extensions provide the algebraic underpinnings for how LFSRs operate. For additional material treating extension fields, see Part VI in [5].

1.3 Principles of Encryption

There are a few basic principles of encryption that we consider to provide a framework for subsequent comparisons of the effect of different mathematical operations. We begin with

the basic concepts of confusion and diffusion, as proposed by Shannon in [6], then discuss common inputs to a symmetric cipher, and conclude with a brief note on cryptanalysis.

1.3.1 Confusion

In many respects, this is the traditional concept of encryption. Take an input and change it to make it unreadable, or "confuse" the output. Put a slightly different way, confusion takes input and returns a different output. A mono-alphabetic cipher is the simplest example: map every letter to another letter. This might work in very simple circumstances, but is not a robust encryption standard.

A more robust application of the principle of **confusion** entails ensuring the relationship between input and the output is very complicated. There are many ways to achieve this effect. The step from a mono-alphabetic cipher to using a rotation of different substitution alphabets would improve the confusion aspect, because we now require many different "keys" to unlock the cipher. Another way to achieve confusion is to incorporate many different inputs into a function that determines the output character, such as using input from a secret key or Initial Value (IV) (as defined below in Section 1.3.3).

1.3.2 Diffusion

The principle of diffusion is somewhat more complicated. The basic principle is that one change in one place of the input should effect several changes in the output. Put another way: we want any changes in plaintext to "diffuse" through the ciphertext. This is also true of the inverse, meaning if an attacker tries to change some of the ciphertext, the plaintext is affected in several places.

This effect of the application of this principle makes the ciphertext more difficult to decrypt without the proper information, because it means that more than a few characters must be examined at one time in order to conduct cryptanalysis. This can be achieved through many different forms, but a common way this is achieved is by conducting several rounds of operations in encryption, and between each round by rotating the data or swapping the bits in the data. To see a way this is conducted in DES, see Section 2.1.

For a further discussion on confusion or diffusion, see [6] or [4].

1.3.3 Common Inputs

Most modern symmetric ciphers have a few commonalities that we define here to provide a baseline introduction to common inputs across all of the ciphers considered.

Key

This is a critical, unique value of a specified length, that is used by the cipher to encrypt the message. The key must generally be kept secret by the user, and can generally only encrypt a certain amount of information. In many situations, they key is the only part of a cipher that is unknown to a would-be attacker, as the algorithms are often made public.

Initialization Value

The IV is another important component that can be required in a cipher, sometimes called a nonce. It is worth noting that different ciphers may not use these, or even allow variable-length inputs for the IV. In general, an IV can be thought of as either an extension to the key or as the initial internal state of the cipher, and should also be kept secret.

Counters

Some ciphers use a counter, which is simply a number incremented by iteration, to augment the IV or otherwise facilitate the encryption process. This need not necessarily be secret per se, but depending on how it is used is generally part of the IV and is not be shared.

Plaintext

This is the original, easily read message. In most cases in this thesis, we will convert whatever format the original message was into American Standard Code for Information Interchange (ASCII), the backwards-compatible precursor to the increasingly-common Unicode Transformation Format (UTF), to homogenize the message into bits or bytes. We note that efficient transmission of information may also involve lossless, or in some cases lossy, transmission of data; this thesis omits discussion of transmission to facilitate simpler comprehension of the steps unique to cryptography, as compared to compression or transmission. For a detailed discussion and development of codes, see [3]. Even with this simplification, the information may look very different to human observers when in different formats, so we must understand that they are simply different representations of

the same information. We will consider the example message: “Try this!” to demonstrate this conversion.

Table 1.7 provides examples of different expansions of our example message. Each row represents a character in our message, with each column providing a different expansion. We note that the ASCII column values are in decimal format.

Character	ASCII	Binary	Hexadecimal
T	84	01010100	0x54
r	114	01110010	0x72
y	121	01111001	0x79
" "	32	00100000	0x20
t	116	01110100	0x74
h	104	01101000	0x68
i	105	01101001	0x69
s	115	01110011	0x73
!	33	00100001	0x21

Table 1.7. ASCII Expansions

1.3.4 Linearity and Cryptanalysis

While a full discussion of the cryptographic concept of linearity and linear cryptanalysis is beyond the scope of this thesis, these ideas are important concepts that are connected to the ideas of confusion and diffusion. As such, we will provide a very brief introduction to these ideas to provide context for further discussion, using a similar structure as [7].

In mathematics generally, a **linear function** is often defined as a function f, with inputs a, b for which the following is true:

$$f(a + b) = f(a) + f(b).$$

For an algebraic discussion of general linearity, see Section 13 of [5]. We note that in conversational English, the term “linear” is often confused with the more precise term “affine.” We are concerned only with a specific application of linearity, rather than the

general concept.

The term **cryptanalysis** refers to the study or practice of breaking cryptographic systems. The idea of linear cryptanalysis was first proposed in [8] as a technique to attack DES, and later explained more generally in [7]. The very basic idea is that a linear combination of certain input bits can be used to generate the outputs. Thus a highly linear cipher is not considered secure. To prevent this type of attack, cryptographers use nonlinear ciphers.

1.4 Cipher Types

With a basic understanding of the mechanics and inputs of ciphers, someone wishing to encrypt their data must decide how they wish to do so. There are a variety of ways to encrypt, as discussed in Section 1.1, and this thesis will briefly introduce asymmetric ciphers before progressing to symmetric block and stream ciphers. This paper includes a few block encryption schemes as a baseline for comparison, including the AES and the DES, while primarily focusing our analysis on an array of stream ciphers.

1.4.1 Asymmetric Ciphers

A common example of an asymmetric cipher is seen in public-key encryption. The very general analogy is that there are special kinds of locks one can use to secure information with publicly available keys, which can only be opened by someone with a *different* kind of key. We use a simple, standard example to illustrate.

We consider a few people who are sending encrypted emails: Alice, Bob, and Eve. Using asymmetric encryption, if Alice wants to send a private email to Bob, she can use Bob's public key to encrypt so that only Bob's private key can decrypt. If Eve intercepts this email she cannot easily decrypt, because Bob's public key is of no direct use in decrypting the ciphertext. The idea of public and private keys provided a powerful new way to encrypt information when it was proposed, and is still invaluable in many situations today.

However, the basic methods by which asymmetric encryption work rely upon more computationally intensive techniques than symmetric encryption and a different set of assumptions. The underlying mathematics are also distinct from most symmetric techniques, often relying on raising large prime numbers to large powers and reducing via another large prime number

modulus. The solutions of these problems rely on elegant insights resulting in simplification techniques developed by Fermat and other famous mathematicians. For a discussion of these techniques, see [2] or [4].

Asymmetric encryption is ideally suited for some situations. But in other circumstances, such as ones where a separate public and private key are not helpful, hardware constrained environments, or transmitting a large quantity of data, a symmetric cipher may be more efficient. As such, the remainder of this thesis focuses on symmetric techniques and ciphers due to their efficiency and effectiveness in encrypting arbitrarily large volumes of data.

1.4.2 Block Ciphers

In the traditional execution of a block cipher, or Electronic Codebook (ECB) mode, the algorithm divides up the message into blocks of a certain length, then performs a defined sequence of operations on each block, returning a cipher text. This is an invertible process, and decryption involves breaking the ciphertext back into those same blocks and then performing the inverse operations on the data to return the plaintext message. There are other modes of block cipher operation, such as Counter (Mode of Block Cipher Operation) (CTR) or Output Feedback (OFB), that do not operate precisely in this fashion. For simplicity we focus exclusively on the ECB.

Table 1.8 shows the input and output steps using AES to encrypt our sample message. Every row in the table is a single character, and the columns progress from plaintext, to ASCII value for the character (expressed in hexadecimal), the ASCII value output from the cipher in place of that character (also hexadecimal), and the corresponding character for the ciphertext. In this case, our original plaintext of "Try this!" has the resulting ciphertext ",ê(0x08)í{å?(0x91)q" for the same message [2]. See Appendix A.6 for notes on the code used to generate this example.

[2]The non-renderable characters are often instructions used in early computers that are no longer necessary. In this case the codes refer to the commands "backspace" and "undefined." To ensure backwards compatibility, these ASCII and UTF values are retained but seldom used.

Character	ASCII	Cipher ASCII	Cipher Character
T	0x54	0x2c	,
r	0x72	0xea	ê
y	0x79	0x08	0x08 (“backspace”)
" "	0x20	0xed	í
t	0x74	0x7b	{
h	0x68	0xe5	å
i	0x69	0x3f	?
s	0x73	0x91	0x91 (“undefined”)
!	0x21	0x71	q

Table 1.8. Block Cipher Example.

1.4.3 Stream Ciphers

Stream ciphers, on the other hand, function like a Pseudo-Random Number Generator (PRNG) and generate keystream (sometimes compared to “noise”) which is added bit- or byte-wise to the message. This encryption process is very different from block ciphers, where the cipher operates in a complicated fashion on the plaintext to produce ciphertext; the stream cipher creates keystream that is added to the plaintext to encrypt via bit- or byte-wise Exclusive OR logical operator (XOR). This means that decryption consists of simply adding that same keystream stream back to return the original message.

Table 1.9 demonstrates the steps for a stream cipher. Again, each row is a single character in our message. This time, the columns are the binary expansion, the binary keystream output, and the ciphertext binary expansion. We note that for Salsa20, the cipher stream is the bytewise addition of the binary with the keystream. Here, our test message returns “ xÂ(0xad)èÅ(0x1c)£*) ” as our ciphertext.

For an in-depth discussion of Boolean functions and stream ciphers, see chapter 7 in [9].

1.5 Substitution Boxes

The S-Box is a common tool to confuse outputs, and is used in many ciphers. The basic idea is to take and input and return (or substitute) a different output, e.g., a mono-alphabetic cipher, where each letter is exchanged for another letter. We note that in general, the outputs

Character	Binary	Keystream Binary	Cipher Binary	Cipher Character
T	01010100	00100100	01111000	x
r	01110010	01010000	11000010	Â
y	01111001	00110100	10101101	0xad (“soft hyphen”)
" "	00100000	11001000	11101000	è
t	01110100	01010001	11000101	Å
h	01101000	10110011	00011100	0x1c (“file separator”)
i	01101001	00111010	10100011	£
s	01110011	00110110	00101010	*
!	00100001	00001000	00101001	)

Table 1.9. Stream Cipher Example.

and inputs do not have to be of the same size.

Figure 1.1 shows the AES encryption S-Box. In this case, the two-digit input is transformed into a two-digit output. As an example, if input m =0xf2, output c=0x89.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 1.1. AES S-Box. Source: [1]

This is not the only way to configure a S-Box. Another way, as used by DES, takes a 6-bit input and returns a 4-bit output. The first and last bit determine the row accessed, while the inner bits determine the column accessed. Figure 1.2 depicts one of the tables in DES. In

this example, if the input $m = 101010$, the row accessed is 10 in binary, or the third row, and the column accessed is 0101, or the fifth row, to return $c = 6$ or $c = 0110$ in binary.

S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Figure 1.2. DES S-Box S_1. Source: [10]

We note that the inputs and outputs from a given S-Box need not have the same size, but that the process must be invertible to ensure decryption is possible. For more discussion on S-Boxes or the development of the ones in Figures 1.1 and 1.2, see [4].

1.6 Hashes

In some of the documentation for the ciphers discussed in Chapter 2, there are references to hash functions. While direct discussion of hashes is not the express aim of this thesis, their importance in message verification and the use of similar processes merit a brief outline. The following introduction parallels the introduction from [4].

An implicit requirement of encryption is that the ciphertext can be converted back to plaintext, and so any mathematical encryption is said to be invertible because we can "undo" any operations in the encryption process. Hashes work differently; they take a message and turn that message into a string of a fixed length via one-way processes. This is sometimes referred to as a **message digest**, because the plaintext can be arbitrarily long, but the output is set based on the specific hash algorithm.

In general, as outlined in [4], hashes should satisfy three properties:

- H_1: Fast. Calculating the hash of a message m_1, annotated $h(m_1)$, should be relatively fast.
- H_2: One-Way. If we have a hash of a message g, it is not computationally feasible to simply search for some message m such that $h(m) = g$.
- H_3: Collision-Free. It is computationally infeasible to find a message m_2 such that $h(m_1) = h(m_2)$ (can sometimes be relaxed). This means that it is difficult to find a

message whose hash "collides," or has the same hash, as the original message.

These three basic ideas contribute to the utility of hash functions as a way to verify message integrity. If the hash of a message is included at the end of the message, then the receiver simply computes the hash themselves and compares their computed hash with the sent message hash. By H_1, it is cheap for the message receiver to compute the hash, and by H_2 and H_3 the message recipient can be confident that their message was not altered.

For more information on hashes, see [4].

1.7 Operations

As the final section of the introduction, we introduce the broad types of operations used in ciphers, and briefly introduce their notation. The following list of operations provides the categories of operations used in subsequent chapters. We note these operations are distinct from those mentioned in Chapter 4 of [11] (which include bitwise permutations, bitwise Boolean operations, S-Boxes, and modular arithmetic) because we are not examining hardware implementations, but rather searching for patterns in discrete, mathematically distinct operations.

1. *Addition*: In many settings in this thesis, we mean bit- or byte-wise addition, which is addition modulo 2 or 2^8 respectively, but there are other possible moduli. The meaning of the "+" should be clear in the context it is used. This category also includes the XOR operation. In general, the mathematical operation of addition (here usually with a modulus) changes a single input into a different single output, thus directly assisting in confusing the output. Depending on how a collection of operations are structured, different inputs may be combined into one output, which can also assist in diffusing any changes.
2. *Substitution Box*: For an S-Box F with input x, we write $F[x]$, which returns some output y which is often (but not necessarily) of the same size. The use of this operation can directly achieve confusion or reduce linearity (if the relationships between inputs and outputs are constructed properly), but depending on how the S-Box is constructed may also contribute to diffusion.
3. *Multiplication*: As with addition, we generally are referring to multiplication with a

modulus that should be made clear. This is often annotated similar to high-school algebra (two terms immediately adjacent), as an exponent, or with a $\cdot$. This category also includes the logical AND operation. This operation generally contributes to the confusion principle, but can also disrupt linearity by masking or otherwise nonlinearly modifying the outputs (as seen in a Nonlinear Feedback Shift Register (NFSR)).

4. *Rotation*: This operation is applied in many fashions. If we are looking at blocks of text, to "rotate" them x spaces means to shift each character x spaces in a direction, wrapping around to the other side when necessary. This is commonly annotated with <<< for a left rotation, and >>> for a right rotation. We consider the effects of the "shift" operation to be within this category as well, even as we note that we lose the information from characters shifted out of the string and replace the empty characters with a 0 (unless otherwise specified). The shift is annotated << for left-shift, >> for right-shift. We note two other mathematical perspectives on rotations: first, that if we are considering a polynomial $p(x) \in \mathbb{F}[x]$, this operation is analogous to multiplication by a nonconstant polynomial; second, this operation is a subset of permutation operation (included in the next category) where the ordering of characters is retained. Despite these two observations, we categorize this operation as distinct because of its demonstrated utility in encryption. This operation is the classical operation to achieve the principle of diffusion, but can also contribute to nonlinearity and confusion.
5. *Other Operations*: For comparison of total operations, we also include a few other types of operations, including permutation of values, comparing size of values (e.g., equivalence, larger, or smaller), "slicing" or concatenating strings, and copying a value several times. These operations, while not necessarily mathematically modifying the data, can all contribute to confusion, diffusion, and a reduction of linearity.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Literature Review

This chapter aims to provide an overview and description of the major cryptographic algorithms considered in this thesis. To provide a context for comparison of mathematical computations and operations, we will first consider DES and then AES operating in ECB mode (the traditional mode for introducing these ciphers). While these are block ciphers, and so do not provide a perfect comparison with stream ciphers, they are the first and current (respectively) major ciphers formally adopted by the National Institute of Standard and Technology (NIST) to encrypt data. Following these initial ciphers, we will discuss the winners of the ECRYPT II eSTREAM competition, which culminated in 2008. We again note that these are considered reliable ciphers, meaning they have been closely studied and there have been no successful cryptanalysis against their full implementations.

For those familiar with ciphers, we again note that DES and AES can operate in other modes, such as Cipher Block Chaining (CBC), Cipher Feedback (CFB), OFB, or CTR mode. While these modes, specifically CFB or CTR mode, function more similarly to stream ciphers, we will focus on the basic ECB mode for the sake of simplicity.

For each of the ciphers, we provide an overview of inputs and then three subsections: functions, initialization steps required, and encryption stream generation.

2.1 DES

This block cipher was proposed by International Business Machines (Corporation) (IBM) in response to the National Bureau of Standards (NBS) (predecessor to the NIST) for a national cryptographic standard. After a review and some adjustments by the National Security Agency (NSA), DES was published as the official data encryption standard with a free license for its use, as discussed in [10]. It remained the primary standard despite mounting evidence of weaknesses until 2000, when it was formally replaced by AES in [1]. DES is a block cipher that uses a 64-bit key (which includes eight parity bits) to encrypt 64-bit blocks of plaintext through an initial permutation and 16 rounds of a function to achieve encryption. In addition to the original publication in [10], [4] also provides an

in-depth discussion of DES.

2.1.1 Functions

Only one major function is used in DES, per [10]: the function f used in each round. The inputs to this function are the second 32 bits, or the right half, of the previous intermediate ciphertext, denoted in encryption as R_{i-1}. This function consists of four steps.

1. R is expanded from 32 bits to 48 bits via an expansion permutation, denoted $E(R)$.
2. $E(R) \oplus K_i$ is computed and sliced up into 8 strings of 6 bits, labelled $B_1B_2 \dots B_8$.
3. Each B_i above is input into a separate S-Box to return a 4-bit output, represented as $C_1C_2 \dots C_8$. See Figure 1.2 in the previous section for an example of S_1. We note these are unique to each position i in B_i, but the ith box remains the same through all the rounds.
4. The string $C_1C_2 \dots C_8$ is permuted to produce a final 32-bit string and returned.

2.1.2 Initialization

As described in [10], there is no initialization per se for this algorithm save the creation of the round keys K_i from K. We recall that K was a 64-bit key, of which eight bits were parity bits. The round key K_i is built through three major steps:

1. Parity bits are discarded and the remaining bits are permuted. The resulting string from this permutation is written as C_0D_0, where C_0 and D_0 each have 28 bits.
2. For each round $i \in \{1, 2, \dots, 16\}$, we let $C_i = LS_i(C_{i-1})$ and $D_i = LS_i(D_{i-1})$, where LS_i is a left-shift of on digit for the first, second, ninth, and sixteenth rounds, and two digits for all others.
3. Only 48 bits of the 56-bit string C_iD_i are chosen according to a set list of numbers. Those 48 bits are assigned as K_i.

2.1.3 Encryption

We note we are considering ECB mode of operation for this cipher, as published in [10]. There are three main stages to encrypting each block of 64 bits:

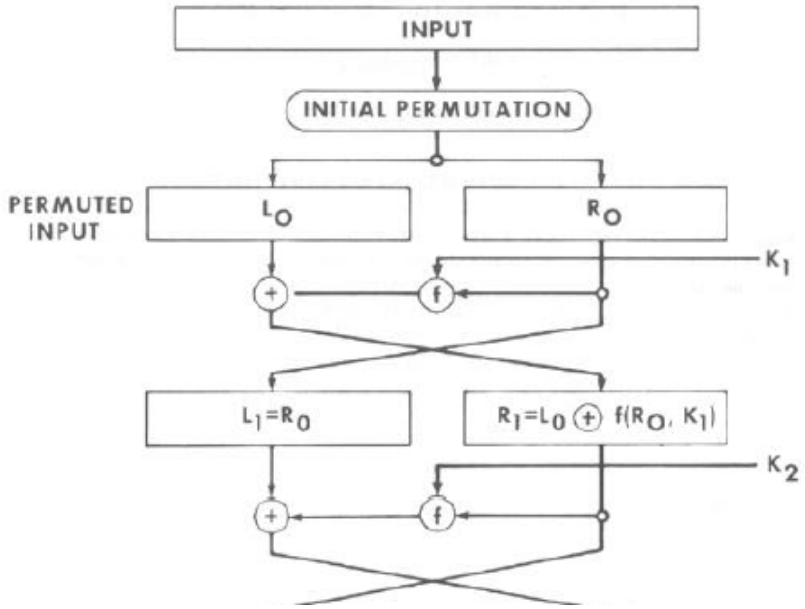

Figure 2.1. DES Feistel System. Source: [10].

- Initial permutation. The plaintext m is permuted by a fixed initial permutation, and then divided into a left- and right-half: $m = L_0R_0$.
- 16 Rounds of Feistel System. See Figure 2.1 for an illustration of the first few rounds. For each round i in $i \in \{1, 2, \ldots, 16\}$, we perform the following:

$$L_i = R_{i-1},$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i).$$

- Final permutation. The final left and right halves are switched and an inverse permutation is performed to return the ciphertext. This is usually depicted $c = IP^{-1}(R_{16}L_{16})$.

2.2 AES

In 1997, the NIST formally issued a call for block cipher submissions to replace DES. After several rounds winnowing the submissions, the Rijndael cipher (pronounced like "Rain Doll," formed by combining the names of the authors J. Daemen and V. Rijman) was selected as the Advanced Encryption Standard. The cipher can take 128-, 192-, or 256-bit keys, and operates on blocks of 128 bits to conduct 10 rounds of operations (each of which

is considered a Substitution Permutation Network (SPN)) to encrypt the plaintext. See the original publication document in [1] or [4] for additional details on the cipher or its history.

2.2.1 Functions

There are two major functions used in the AES encryption process as originally published in [1]: a shift-row function (S) and a mix-column function (M). See Figure 1.1 for the AES S-Box.

1. Shift-Row S. This function takes 16 bytes as inputs (B) and returns 16 bytes as outputs (C). If we divide the input B and output C into four-byte sequences, arranged into a matrix of four rows (with four columns), we return C by performing the following operations on the rows of the matrix:

$$C_1 = B_1 <<< 0$$
$$C_2 = B_2 <<< 1$$
$$C_3 = B_3 <<< 2$$
$$C_4 = B_4 <<< 3.$$

As an example, if $B_2 = (b_1, b_2, b_3, b_4)$, then $C_2 = (b_2, b_3, b_4, b_1)$.

2. Mix-Column M. This function also takes 16 bytes as inputs (C) to return 16 bytes as outputs (D), denoted $D = M(C)$. This function again organizes the bytes into a 4×4 matrix (where the first four bytes are the first row, the second four bytes are the second row, and so forth), and multiplies the matrix by another 4×4 matrix of constants to return a final 4×4 matrix of 16 bytes D (where we again extract the first row as the first four bytes of D and so forth). We note that throughout this operation, we consider each byte an element of GF(2^8).

As an example of the first byte in D, denoted $d_{i,j}$ for the ith row and the jth column of output:

$$d_{1,1} = 2 \cdot c_{1,1} + 3 \cdot c_{2,1} + 1 \cdot c_{3,1} + 1 \cdot c_{4,1}.$$

2.2.2 Initialization

The only initialization required for this cipher is the development of the key schedule from the original key, as prescribed in [1]. We take as an example the 128-bit key, organized into a 4×4 matrix of bytes. The first four columns of this matrix are labeled $W(0), W(1), W(2), W(3)$, with the new columns generated recursively.

For column i where i is a multiple of 4, we first define a special transformation $T(W(i-1))$:

- If $W(i-1)$ = the four-byte sequence a, b, c, d, then we let $e, f, g, h = S[b, c, d, a]$.
- Compute a round constant, where $r(i) = 00000010^{(i-4)/4}$ in GF(2^8).
- We then let $T(W(i-1))$ be the column vector $(e \oplus r(i), f, g, h)^T$.

For column i where i is not a multiple of 4, we set:

$$W(i) = W(i-4) \oplus W(i-1).$$

This generates the key schedule so that round key K_i for the ith round consists of the 4×4 matrix built from the columns $W(4i), W(4i+1), W(4i+2), W(4i+3)$.

2.2.3 Encryption

We note that we are considering the ECB mode of operation from [1]. Encryption for AES consists of one pre-round step and ten rounds of encryption. We first organize the plaintext into a 4×4 matrix of bytes (where the first four bytes of plaintext go in the first row, etc). Add to this the matrix K_0, returning the 4×4 matrix we call A_0.

For rounds $i \in \{1, 2, \ldots, 10\}$, we perform the following steps:

$$\begin{aligned}
B_i &= F[A_{i-1}] \text{ (Byte-wise input into S-Box returned via matrix)} \\
C_i &= S(B_i) \\
D_i &= M(C_i) \text{ (This step omitted for round 10)} \\
E_i &= D_i + K_i.
\end{aligned}$$

The final output is E_{10} as the 128-bit ciphertext.

2.3 HC128

This stream cipher was proposed by Hongjun Wu in [12]. The broad idea is to take a 128-bit key and 128-bit IV to build two 512-register S-Boxes, and use those with the IV to simultaneously update the S-Boxes and create keystream.

2.3.1 Functions

There are six major functions present in this cipher and defined in [12]. We note that in this context, '+' means addition modulo 10^{32}. For h_1, h_2, $P[x_0]$ (or $Q[x_0]$) refers to the 32-bit output of the P (or Q) S-Box, based on the input byte x_0 where x is a 32-bit word such that $x = x_3||x_2||x_1||x_0$,

$$\begin{aligned} f_1(x) &= (x >>> 7) \oplus (x >>> 18) \oplus (x >> 3) \\ f_2(x) &= (x >>> 17) \oplus (x >>> 19) \oplus (x >> 10) \\ g_1(x,y,z) &= ((x >>> 10) \oplus (z >>> 23)) + (y >>> 8) \\ g_2(x,y,z) &= ((x <<< 10) \oplus (z <<< 23)) + (y <<< 8) \\ h_1(x) &= Q[x_0] + Q[256 + x_2] \\ h_2(x) &= P[x_0] + P[256 + x_2]. \end{aligned}$$

2.3.2 Initialization

Per [12], there are three major steps to the initialization.

1. The Key and IV are broken into bytes ($K = K_0||K_1||K_2||K_3$, IV split similarly), then expanded to fill a 1280-byte array W using the key, IV, and a linear combination of f_1, f_2, and previous bytes in W. We also note that in dividing up the key, $K_{i+4} = K_i$. The algorithm is as follows.

$$W_i = \begin{cases} K_i & 0 \le i \le 7 \\ IV_{i-8} & 8 \le i \le 15 \\ f_2(W_{i-2}) + W_{i-7} + f_1(W_{i-15}) + W_{i-16} + i & 16 \le i \le 1279. \end{cases}$$

2. Load the initial values of W into the S-Boxes P, Q as follows:

$$P[i] = W_{i+256} \text{ for } 0 \leq i \leq 511$$
$$P[i] = W_{i+768} \text{ for } 0 \leq i \leq 511.$$

3. Run the cipher 1024 times, feeding the "cipher" output back into the table elements per the below steps. We note in this context that "-" means subtraction modulo 512, and we let each iteration be counted as i mod 512 as we execute 1024 steps. For the first 512 steps:

$$P[i] = (P[i] + g_1(P[i-3], P[i-10], P[i-511])) \oplus h_1(P[i-12]).$$

The second 512 steps are:

$$Q[i] = (Q[i] + g_2(Q[i-3], Q[i-10], Q[i-511])) \oplus h_2(Q[i-12]).$$

2.3.3 Encryption

As outlined in [12], for each iteration of the algorithm we generate a 32-bit output of keystream and update one entry of one S-Box, alternating between the S-Boxes every 512 steps. In the below description, "-" still means subtraction modulo 512, and s_i indicates the output keystream byte of step i.We generate each byte of keystream needed per the below pseudo-code:

$i = 0$
$j = i \mod 512$
if $(i \mod 1024) < 512$ **then**
  $P[j] = P[j] + g_1(P[j-3], P[j-10], P[j-511])$
  $s_i = h_1(P[j-12]) \oplus P[j]$
else
  $Q[j] = Q[j] + g_2(Q[j-3], Q[j-10], Q[j-511])$
  $s_i = h_2(Q[j-12]) \oplus Q[j]$
end if

The keystream output s_i (one byte) is then simply added on top of the message to encrypt.

2.4 Rabbit

The stream cipher Rabbit was proposed by M. Boesgaard, M. Vesterager, and T. Christensen in [13]. The cipher uses a 128-bit key and optional 64-bit IV to build an internal state register of 513 bits, organized into eight internal state variables, eight counters, and a carry bit. The cipher then produces keystream based on the state of the register, concurrently updating the variables, counters, and state bit.

2.4.1 Functions

There are only two functions defined for this cipher in [13], called the "next-state function" and the "counter update," each of which consists of nine equations to update each of the state variables or counters. One system iteration, or step, consists of running each function once. We note that state variables and counters at position j and at step i are represented as $x_{j,i}$ and $c_{j,i}$, respectively, and that in this context '+' means addition modulo 2^{32}.

The nine pieces of the next-state function are as follows:

$$
\begin{aligned}
g_{j,i} &= ((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 >> 32)) \mod 2^{32} \\
x_{0,i+1} &= g_{0,i} + (g_{7,i} <<< 16) + (g_{6,i} <<< 16) \\
x_{1,i+1} &= g_{1,i} + (g_{0,i} <<< 16) + g_{7,i} \\
x_{2,i+1} &= g_{2,i} + (g_{1,i} <<< 16) + (g_{0,i} <<< 16) \\
x_{3,i+1} &= g_{3,i} + (g_{2,i} <<< 16) + g_{1,i} \\
x_{4,i+1} &= g_{4,i} + (g_{3,i} <<< 16) + (g_{2,i} <<< 16) \\
x_{5,i+1} &= g_{5,i} + (g_{4,i} <<< 16) + g_{3,i} \\
x_{6,i+1} &= g_{6,i} + (g_{5,i} <<< 16) + (g_{4,i} <<< 16) \\
x_{7,i+1} &= g_{7,i} + (g_{6,i} <<< 16) + g_{5,i}.
\end{aligned}
$$

The nine equations in the counter update function are below. We note that the a_n terms

below correspond to hexadecimal constants, and the ϕ term is referred to as the "carry bit":

$$\phi_{j,i+1} = \begin{cases} 1 & \text{if } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \wedge j = 0 \\ 1 & \text{if } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \wedge j > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned}
c_{0,i+1} &= c_{0,i} + a_0 + \phi_{7,i} && \mod 2^{32} \\
c_{1,i+1} &= c_{1,i} + a_1 + \phi_{0,i+1} && \mod 2^{32} \\
c_{2,i+1} &= c_{2,i} + a_2 + \phi_{1,i+1} && \mod 2^{32} \\
c_{3,i+1} &= c_{3,i} + a_3 + \phi_{2,i+1} && \mod 2^{32} \\
c_{4,i+1} &= c_{4,i} + a_4 + \phi_{3,i+1} && \mod 2^{32} \\
c_{5,i+1} &= c_{5,i} + a_5 + \phi_{4,i+1} && \mod 2^{32} \\
c_{6,i+1} &= c_{6,i} + a_6 + \phi_{5,i+1} && \mod 2^{32} \\
c_{7,i+1} &= c_{7,i} + a_7 + \phi_{6,i+1} && \mod 2^{32}.
\end{aligned}$$

A visualization of the how state variables and counters feed into one another is illustrated in Figure 2.2 from [13].

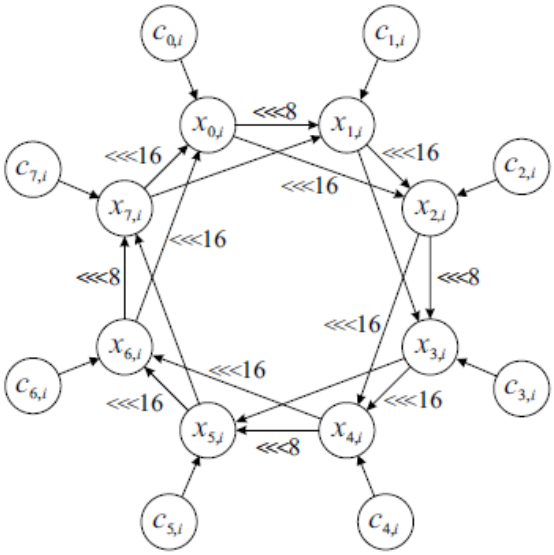

Figure 2.2. Rabbit State Variables and Counter Relationships. Source: [13].

2.4.2 Initialization

Per [13], there are two major phases to this cipher's initialization. Here we use '+' to indicate concatenation of two bit strings.

1. The key is subdivided into eight subkeys ($K = k_7||k_6||\ldots||k_1||k_0$, where each subkey is sixteen bits), and then used to generate the state (x) and counter (c) variables as follows:

$$x_{j,0} = \begin{cases} k_{j+1 \mod 8} + k_j & \text{for } j \text{ even} \\ k_{j+5 \mod 8} + k_{j+4 \mod 8} & \text{for } j \text{ odd} \end{cases}$$

$$c_{j,0} = \begin{cases} k_{j+4 \mod 8} + k_{j+5 \mod 8} & \text{for } j \text{ even} \\ k_j + k_{j+1 \mod 8} & \text{for } j \text{ odd.} \end{cases}$$

We then iterate the system four times, and modify each of the counter variables as below to prevent key recovery via system inversion:

$$c_{j,4} = c_{j,4} \oplus x_{j+4 \mod 8,4}.$$

2. If an IV is used, each of the counter variables are modified according to the below updates. We use the following notation below to indicate how the IV is sliced: $IV = IV_3||IV_2||IV_1||IV_0$,

$$\begin{aligned} c_{0,4} &= c_{0,4} \oplus (IV_1 + IV_0) \\ c_{1,4} &= c_{1,4} \oplus (IV_3 + IV_1) \\ c_{2,4} &= c_{2,4} \oplus (IV_3 + IV_2) \\ c_{3,4} &= c_{3,4} \oplus (IV_2 + IV_0) \\ c_{4,4} &= c_{4,4} \oplus (IV_1 + IV_0) \\ c_{5,4} &= c_{5,4} \oplus (IV_3 + IV_1) \\ c_{6,4} &= c_{6,4} \oplus (IV_3 + IV_2) \\ c_{7,4} &= c_{7,4} \oplus (IV_2 + IV_0). \end{aligned}$$

We then iterate the system four additional times to complete the initialization. In this case,

that means the "time" variable i begins at 8.

2.4.3 Encryption

As described in [13], keystream is generated by execution of one next-state function and one counter update. This generates a 128-bit sequence of keystream s_i from the state variables $x_{j,i}$, where s_i is divided as $s_i = s_{7,i}||s_{6,i}|| \ldots ||s_{1,i}||s_{0,i}$ and $x_{j,i} = x^1_{j,i}||x^0_{j,i}$,

$$
\begin{aligned}
s_{0,i} &= x^0_{0,i} \oplus x^1_{5,i} & \qquad s_{1,i} &= x^1_{0,i} \oplus x^0_{3,i} \\
s_{2,i} &= x^0_{2,i} \oplus x^1_{7,i} & \qquad s_{3,i} &= x^1_{2,i} \oplus x^0_{5,i} \\
s_{4,i} &= x^0_{4,i} \oplus x^1_{1,i} & \qquad s_{5,i} &= x^1_{4,i} \oplus x^0_{7,i} \\
s_{6,i} &= x^0_{6,i} \oplus x^1_{3,i} & \qquad s_{7,i} &= x^1_{6,i} \oplus x^0_{1,i}.
\end{aligned}
$$

As with most stream ciphers, keystream output s_i (16 bytes at a time) is then simply added on top of the message to encrypt.

2.5 Salsa

The stream cipher Salsa20 was proposed by Daniel Bernstein in [14]. This encryption algorithm is basically a hash function used in counter mode to generate a cipher stream, using a 32- or 16-byte key and 8-byte nonce (or 128/256 bit key and 64-bit nonce). The algorithm for this cipher operates on "words," which are integers between 0 and $2^{32} - 1$, often written in hexadecimal.

2.5.1 Functions

Each function introduced in this section uses previous functions to develop the inner workings of the cipher. We reproduce the seven functions from [14] as follows.

1. Quarterround function. We let y and z be four word sequences, such that $y = (y_0, y_1, y_2, y_3)$ and $z = (z_0, z_1, z_2, z_3)$, and quarterround$(y) = z$, defined as

$$
\begin{aligned}
z_1 &= y_1 \oplus ((y_0 + y_3) <<< 7), \\
z_2 &= y_2 \oplus ((z_1 + y_0) <<< 9),
\end{aligned}
$$

$$z_3 = y_3 \oplus ((z_2 + z_1) <<< 13),$$
$$z_0 = y_0 \oplus ((z_3 + z_2) <<< 18).$$

2. Rowround function. We let y and z be 16 word sequences, such that $y = (y_0, y_1, \ldots, y_{15})$ and $z = (z_0, z_1, \ldots., z_{15})$, and rowround$(y) = z$ defined as

$$(z_0, z_1, z_2, z_3) = \text{quarterround}(y_0, y_1, y_2, y_3),$$
$$(z_5, z_6, z_7, z_4) = \text{quarterround}(y_5, y_6, y_7, y_4),$$
$$(z_{10}, z_{11}, z_8, z_9) = \text{quarterround}(y_{10}, y_{11}, y_8, y_9),$$
$$(z_{15}, z_{12}, z_{13}, z_{14}) = \text{quarterround}(y_{15}, y_{12}, y_{13}, y_{14}).$$

3. Columnround function. This can be conceived as the transpose of the rowround function above. We let x and y be 16 word sequences, such that $x = (x_0, x_1, \ldots, x_15)$ and $y = (y_0, y_1, \ldots., y_15)$, and columnround$(x) = y$ defined as

$$(y_0, y_4, y_8, y_{12}) = \text{quarterround}(x_0, x_4, x_8, x_{12}),$$
$$(y_5, y_9, y_{13}, y_1) = \text{quarterround}(x_5, x_9, x_{13}, x_1),$$
$$(y_{10}, y_{14}, y_2, y_6) = \text{quarterround}(x_{10}, x_{14}, x_2, x_6),$$
$$(y_{15}, y_3, y_7, y_{11}) = \text{quarterround}(x_{15}, x_3, x_7, x_{11}).$$

4. Doubleround function. This is a column round followed by a row round. So if x is a 16 word sequence, doubleround(x) = rowround(columnround(x)).

5. Littleendian function. We let b be a four byte sequence, $b = (b_0, b_1, b_2, b_3)$, and define littleendian$(b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3$. The outputs of the littleendian function are usually written in hexadecimal, hence this function is similar in appearance to concatenating b's elements from right-to-left (or "little end-first") order.

6. Salsa20 hash function. Let x, y, and z be 64-bye sequences, where Salsa20(x) = littleendian(x)+doubleround$^{10}(x) = z$. We note this is a three-stage transition: first x to littleendian, then 10 iterations of doubleround, then the addition and inverse littleendian transformation.

If we split x by byte, annotated $x = (x[0]||x[0]||\ldots||x[63])$, we then let x_i = littleendian$(x[j], x[j+1], x[j+2], x[j+3])$ where $j = i \mod 4$ and $j \in 1, 2, \ldots 16$. This creates a 16-word sequence of x_i from the littleendian of our original x.

We then let y = doubleround$^{10}(x)$.

Finally, to get our $z = (z_0, z_1, \ldots, z_{15})$, we transform littleendian$^{-1}(y_i + x_i)$ for $i \in 1, 2, \ldots, 16$. Input was a 64-byte sequence, and our output is the same.

7. Salsa20 expansion function. This function basically takes a 32-/16-byte sequence and a 16-byte sequence and expands them to a 64-byte sequence by inserting constants and using the above hash function.

If we let σ_1, σ_2, σ_3, and σ_4 be four-byte sequences of constants, with k and n 16-byte sequences, then Salsa20$_k(n)$ = Salsa20$(\sigma_0, k, \sigma_1, n, \sigma_2, k, \sigma_3)$. We note for a 32-byte k, we would split k into 16-byte halves and insert each into one of the two k's input in the hash.

2.5.2 Initialization

This cipher conducts no initialization.

2.5.3 Encryption

Using the same terminology as in [14], we take key k, nonce n, and counter c (where the counter is a unique 8-byte sequence representing 0 to $2^{64} - 1$), and use these to generate 64 bytes of output for each iteration of Salsa20$_k(n, c)$. For each additional 64 bytes of keystream, we increment c and repeat, again simply adding the keystream to our message to encrypt.

2.6 Sosemanuk

The stream cipher Sosemanuk was proposed by a dozen contributors in [15]. Its name is derived from its extensive use of the stream cipher Snow algorithm (direct predecessor to this algorithm) and from use of portions of the block cipher Serpent (one of the original competitors to AES, see [16]), which when combined renders "snow serpent", and this expression in Cree (an Eastern Canadian tribe) translates to Sosemanuk. This cipher works

by taking a 128–256-bit key and 128-bit IV, then using an LFSR, Finite State Machine (FSM), and a special combination of their outputs to generate keystream.

2.6.1 Functions

There are four major components of Sosemanuk, as defined in [15], which we can divide roughly into the two categories that contribute to its name, that is the Serpent cipher and the Snow cipher components. We will provide an overview of the Serpent functions first (as originally proposed in [16]), and then discuss the Snow components (as discussed in [15]).

1. Serpent1. This is the simplest of the four components of Sosemanuk, called Serpent1 because it only uses part of one round of the Serpent block cipher. It simply uses the third substitution box (S_2) from the original Serpent algorithm on groups of four 32-bit words to return four different 32-bit words.

2. Serpent24. This function relies on reducing the original 32 rounds of Serpent to only 24 rounds (hence the name), and is only used for initialization. This function takes as its inputs a 128–256 bit key and returns 12 (twelve) 32-bit words, output at different rounds from Serpent, to populate the initial state of the LFSR and FSM.

Key Schedule

The key is used to generate a key schedule, which is a collection of keys generated from the original key, to be used in subsequent rounds of Serpent24, per the following steps.

First, the key is padded to reach 256 bits by adding a "1" followed by sufficient "0"s on the most-significant-end (right side). The key is then expanded into 33 separate 128-bit subkeys (of which here we only use the first 25), numbered 0 to 32, by writing each subkey as a collection of eight 32-bit words $w_{-8}, \ldots, w_{-1}$. We then expand these to intermediate keys $w_0, \ldots, w_{131}$ with the following recurrence:

$$w_i = (w_{i-8} \oplus w_{i-5} \oplus w_{w-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11,$$

where ϕ is the fraction part of the golden ratio, which can be expressed as the constant 0x9e3779b9.

Rounds keys are now output through the use of S-Boxes operating on collections of four words, where $\{k_0, k_1, k_2, k_3\} = S(w_0, w_1, w_2, w_3)$.

Each key, K_j is returned as: $K_i = \{k_{4i}, k_{4i+1}, k_{41+2}, k_{4i+3}\}$.

Serpent24 Cipher Use

The Sosemanuk IV is then fed into the Serpent block cipher as plaintext, and the cipher is run for 24 rounds using the key schedule as described above. Each round consists of key addition, S-Box, and a set of linear transformations. For each round i, we have the input numbers (B_i), round S-Box (S_i), round key (K_i), and the round output (B_{i+1}), performed as follows:

$$
\begin{aligned}
(X_0, X_1, X_2, X_3) &= S_i(B_i \oplus K_i) \\
X_0 &= X_0 <<< 13 \\
X_2 &= X_2 <<< 3 \\
X_1 &= X_1 \oplus X_0 \oplus X_2 \\
X_3 &= X_3 \oplus X_2 \oplus (X_0 << 3) \\
X_1 &= X_1 <<< 1 \\
X_3 &= X_3 <<< 7 \\
X_0 &= X_0 \oplus X_1 \oplus X_3 \\
X_2 &= X_2 \oplus X_3 \oplus (X_1 << 7) \\
X_0 &= X_0 <<< 5 \\
X_2 &= X_2 <<< 22 \\
B_{i+1} &= X_0, X_1, X_2, X_3.
\end{aligned}
$$

In the final round ($i = 23$), we XOR the 24th key to the last step to produce the final values. The output of each round is fed into the next, but we save the outputs from the 12th, 18th, and 24th round to initialize the Sosemanuk internal state. If we denote Y_i^j as output i from round j, noting that there are four outputs from each round as described above, we assign the values to populate the Sosemanuk initial state, using s to denote LFSR register positions

and R1 or R2 to denote FSM registers:

$$\begin{aligned}(s_7, s_8, s_9, s_{10}) &= (Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12})\\ (s_5, s_6) &= (Y_1^{18}, Y_3^{18})\\ (s_1, s_2, s_3, s_4) &= (Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24})\\ R1_0 &= Y_0^{18}\\ R2_0 &= Y_2^{18}.\end{aligned}$$

3. LFSR. The LFSR for Sosemanuk is the same as used in Snow 2.0, can be seen in Figure 2.3.

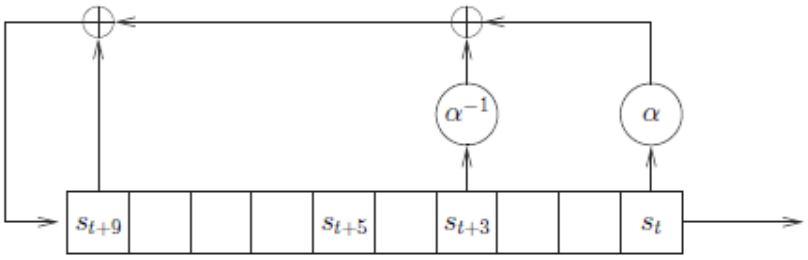

Figure 2.3. The Sosemanuk LFSR. Source: [15].

For a full description of how this LFSR was built, we let $\mathbb{F}_2$ denote the finite field with 2 elements, and β be a root of the following primitive polynomial on $\mathbb{F}_2[X]$:

$$Q(X) = X^8 + X^7 + X^5 + X^3 + 1.$$

We build the field $\mathbb{F}_{2^8}$ by taking the quotient of $\mathbb{F}_2[X]$ via an ideal generated by a polynomial $Q(x)$, $\mathbb{F}_2[X]/\langle Q(X)\rangle$. Noting that $Q(X)$ is primitive, if we represent each element in $\mathbb{F}_{2^8}$ with the basis $(1, \beta, \beta^2, \ldots, \beta^7)$, we know that β is a multiplicative generator of all invertible elements in $\mathbb{F}_{2^8}$. Because we can use β to generate these elements, we can define the following bijection:

$$\phi(\mathbb{F}_{2^8}) \longrightarrow \{0, 1, \ldots, 255\}$$

$$x = \sum_{i=0}^{7} x_i \beta^i \longrightarrow \sum_{i=1}^{7} x_i 2^i.$$

We note in the above, each x_i is either a 0 or a 1, and so addition in $\mathbb{F}_{2^8}$ represents bit-wise XOR between the corresponding integers. Multiplication by β can be executed by a left shift of one bit of the integer representation, then an XOR with a fixed mask if the most significant bit dropped by the shift is 1.

We then define α to be the root of the following primitive polynomial

$$P(X) = X^4 + \beta^{23} X^3 + \beta^{245} X^2 + \beta^{48} X + \beta^{239}$$

on $\mathbb{F}_{2^8}$. The quotient algebraic structure $\mathbb{F}_{2^8}[x]/\langle P(X)\rangle$ defines a new field $\mathbb{F}_{2^{32}}$ that can be represented with its basis $(1, \alpha, \alpha^2, \alpha^4)$. Any element in $\mathbb{F}_{2^32}$ can be identified with a 32-bit integer via the following bijection:

$$\begin{aligned} \phi(\mathbb{F}_{2^{32}}) &\longrightarrow \{0, 1, \ldots, 2^{32} - 1\} \\ y = \sum_{i=0}^{3} y_i \alpha^i &\longrightarrow \sum_{i=1}^{7} \phi(y_i) 2^{8i}. \end{aligned}$$

This clearly defines the α and its inverse, α^{-1} used in the above LFSR. We note that multiplication of some $z \in \mathbb{F}_{2^{32}}$ by α represents a left shift by 8 bits of $\phi(z)$ followed by a 32-bit mask depending on the most significant byte of $\phi(z)$, with multiplication by α^{-1} defined similarly with a right-shift instead of left.

The LFSR is built with the following feedback polynomial:

$$\pi(X) = \alpha X^{10} + \alpha^{-1} X^7 + X + 1 \in \mathbb{F}_{2^{32}}[X].$$

Each element s_i in the LFSR operates over elements of $\mathbb{F}_{2^{32}}$, beginning at time $t = 0$ with the ten 32-bit values of s_i given as outputs from Serpent24. At each new time step, we calculate

$$x_{t+10} = s_{t+9} \oplus \alpha^{-1} s_{t+3} \oplus \alpha s_t, \forall t \geq 1$$

and shift the register. The element shifted out of the s_t position is used for encryption as described in the Encryption Section that follows.

4. FSM. The FSM is a component with 64 bits of memory, corresponding to two 32-bit registers R1 and R2. At each step, the FSM takes input from the LFSR, updates its registers, and produces a 32-bit output denoted f_t. We note that m is a hexadecimal expression for the first ten decimals of π, $\text{lsb}(x)$ is the least significant bit of x, and $\text{mux}(c, x, y)$ is equal to x if $c = 0$, respectively, y if $c = 1$. The outputs at time t as functions of inputs from the previous state are defined as:

$$\begin{aligned} R1_t &= (R2_{t-1} + \text{mux}(\text{lsb}(R1_{t-1}, s_{t+1}, s_{t+1} \oplus s_{t+8})) \mod 2^{32} \\ R2_t &= (m \cdot R1_{t-1} \mod 2^{32}) <<< 7 \\ f_t &= (s_{t+9} + R1_t \mod 2^{32}) \oplus R2_t. \end{aligned}$$

2.6.2 Initialization

To initialize the initial Sosemanuk state the key and IV are used as inputs to the Serpent24 function, and the specific outputs described in the previous section are used to populate the registers.

2.6.3 Encryption

A visual depiction of the process and interactions used to generate keystream are depicted in Figure 2.4.

At each timestep, as prescribed in [15], the following occur:

- FSM is updated as described above, providing input f_t for the keystream.
- LFSR is updated, providing inputs to the FSM and s_t for the keystream.
- Every four steps, four keystream words z are generated as follows:

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = Serpent1(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t).$$

We see here that four 32-bit keystream words are generated at a time, per four clocks of the entire system, by XORing the outptut of Serpent1's S-Box results of the FSM with the

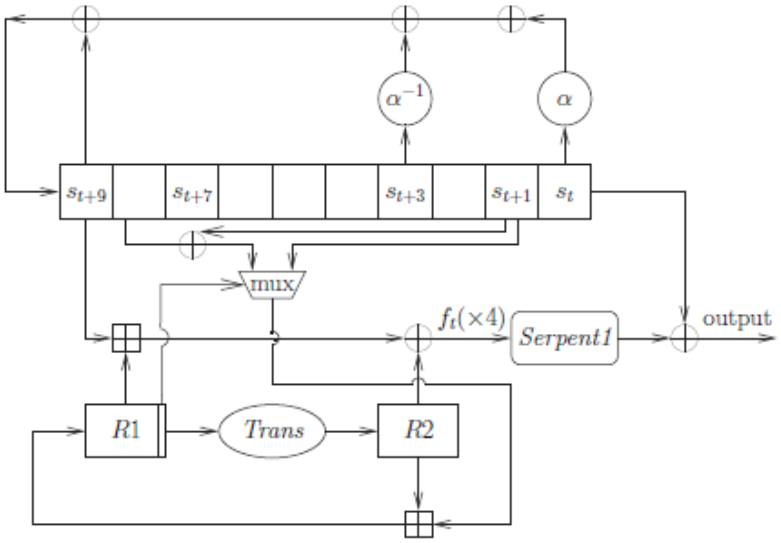

Figure 2.4. Overview of the Sosemanuk Cipher. Source: [15].

LFSR's dropped values. These are then XORed onto the plaintext to generate ciphertext.

2.7 Grain

M. Hell, T. Johansson, and W. Meier proposed the stream cipher Grain in [17]. This is the first of the hardware-based ciphers, designed for lightweight secure implementation in hardware chips. Unlike the software ciphers, the elements of the hardware ciphers are usually represented in bits or elements of GF(2), as opposed to bytes or 32-bit words in hexadecimal format. This cipher takes an 80-bit key and 64-bit IV, and works by combining select outputs from a Nonlinear Feedback Shift Register (NFSR) and an LFSR through a Boolean function to produce keystream.

2.7.1 Functions

There are three functions in the Grain cipher and defined in [17].

1. LFSR. The LFSR state is 80 bits, and operates on a feedback polynomial operating simply $\mathbb{F}_2$, using the feedback polynomial $f(s)$ outlined below:

$$f(x) = 1 + x^{18} + x^{29} + x^{42} + x^{57} + x^{67} + x^{80}.$$

This polynomial results in an update function for the LFSR as follows:

$$s_{i+8} = s_{i+62} + s_{i+51} + s_{i+38} + s_{i+23} + s_{i+13} + s_i.$$

2. NFSR. The NFSR state is also 80 bits, again operating over $\mathbb{F}_2$, with feedback polynomial $g(x)$ defined below:

$$\begin{aligned} g(x) = 1 &+ x_{18} + x_{20} + x_{28} + x_{35} + x_{43} + x_{47} + x_{52} + x_{59} + x_{66} + x_{71} + x_{80} + \\ &+ x_{17}x_{20} + x_{43}x_{47} + x_{65}x_{71} + x_{20}x_{28}x_{35} + x_{47}x_{52}x_{59} + x_{17}x_{35}x_{52}x_{71} + \\ &+ x_{20}x_{28}x_{43}x_{47} + x_{17}x_{20}x_{59}x_{65} + x_{17}x_{20}x_{28}x_{35}x_{43} + x_{47}x_{52}x_{59}x_{65}x_{71} + \\ &+ x_{28}x_{35}x_{43}x_{47}x_{52}x_{59}. \end{aligned}$$

This polynomial generates an update function for the NFSR as follows:

$$\begin{aligned} b_{i+80} = s_i &+ b_{i+62} + b_{i+60} + b_{i+52} + b_{i+45} + b_{i+37} + b_{i+33} + b_{i+28} + b_{i+21} + \\ &+ b_{i+14} + b_{i+9} + b_i + b_{i+63}b_{i+60} + b_{i+37}b_{i+33} + b_{i+15}b_{i+9} + \\ &+ b_{i+60}b_{i+52}b_{i+45} + b_{i+33}b_{i+28}b_{i+21} + b_{i+63}b_{i+45}b_{i+28}b_{i+9} + \\ &+ b_{i+60}b_{i+52}b_{i+37}b_{i+33} + b_{i+63}b_{i+60}b_{i+21}b_{i+15} + \\ &+ b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} + b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} + \\ &+ b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21}. \end{aligned}$$

3. Boolean Functions. These Boolean functions combine inputs from several locations in the NFSR and LFSR to produce a value of either 1 or 0. We define $x = (x_0, x_1, x_2, x_3, x_4)$, and our Boolean function $h(x)$ as

$$h(x) = x_1 + x_4 + x_0x_3 + x_2x_3 + x_3x_4 + x_0x_1x_2 + x_0x_2x_3 + x_0x_2x_4 + x_1x_2x_4 + x_2x_3x_4.$$

This function is then combined with additional elements from the NFSR, where $A = \{1, 2, 4, 10, 31, 43, 56\}$, to generate keystream bit z_i:

$$z_i = \sum_{k \in A} b_{i+k} + h(s_{i+3}, s_{i+25}, s_{i+46}, s_{i+64}, b_{i+63}).$$

2.7.2 Initialization

As described in [17], the 80 bits of the NFSR are loaded with the 80 bits of the key, $b_i = k_i, 0 \leq i \leq 79$. The first 64 bits of the LFSR are loaded with the IV, with all remaining bits set to 1 to ensure the LFSR cannot be initialized to the all-zero state.

The LFSR is then clocked 160 times, with the output from the Boolean XOR's back into the inputs for the LFSR and NFSR as illustrated in Figure 2.5.

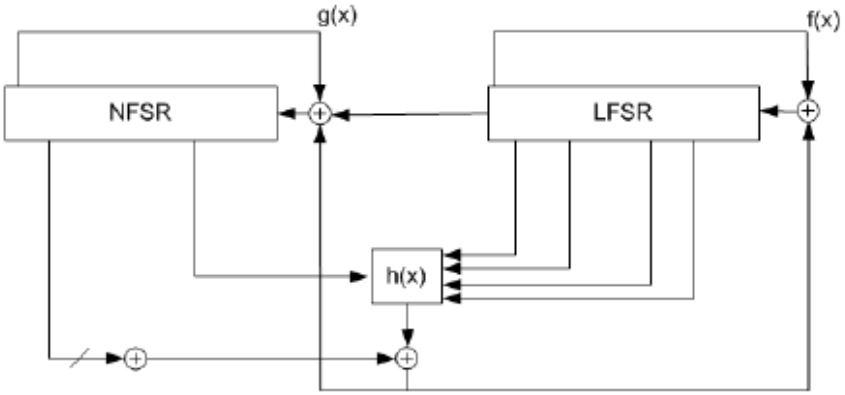

Figure 2.5. Grain Cipher in Initialization Mode. Source: [17].

2.7.3 Encryption

Per [17], once initialized the cipher generates keystream by clocking the system and using the Boolean function as described previously, where each clock of the system provides one bit of keystream. See Figure 2.6 for the system when generating keystream.

2.8 Mickey

The stream cipher, Mickey (which stands for Mutual Irregular Clocking KEYstream generator), was proposed by S. Babbage and M. Dodd in [18]. This cipher takes an 80-bit key and an IV of up to 80 bits, and uses two 100-bit registers, labeled R and S, that behave differently depending on a variable calculated each clocking cycle.

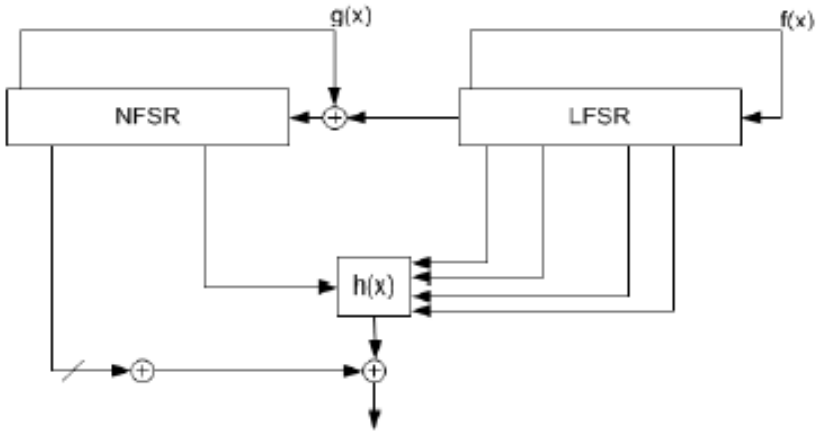

Figure 2.6. The Grain Cipher. Source: [17]

2.8.1 Functions

There are three functions in this cipher, each of which can operate differently in different modes. We paraphrase them from [18] to present in a consistent format here.

1. Linear register clock, operating on register R. This function takes as its inputs register R (where $R = (r_0, r_1, \ldots, r_{99})$), an input bit for R ($IB_r$), and a control bit for R (CB_r), and produces and updated register R' (where $R' = (r'_0, r'_1, \ldots, r'_{99})$). There are also taps placed on half of the positions in the register R, organized in a set called "RTaps". One linear clock consists of the following operations conducted on all $i \in \{0, 1, \ldots, 99\}$:

$$FB_b = r_{99} \oplus IB_r$$

$$r'_i = \begin{cases} 0 & \text{for } i = 0 \\ r_{i-1} \oplus FB_r & \text{if } i \in \text{RTaps} \\ r_{i-1} & \text{else} \end{cases}$$

$$\text{if } CB_r = 1, \text{ then } r'_i = r'_i \text{ (from above)} \oplus r_i.$$

2. Nonlinear register clock, operating on register S. This function takes as its input register S (where $S = (s_0, s_1, \ldots, s_{99})$), an input bit for S ($IB_s$), a control bit for S ($BC_s$), and produces an updated register S' (where $S' = (s'_0, s'_1, \ldots, s'_{99})$). We will use $\hat{s}$ to denote an intermediate value. This function also defines a four-row, 100-column lookup table, where

the ith column is accessed via a function for each row $(C0_i, C1_i, F0_i, F1_i)$,

$$FB_s = s_{99} \oplus IB_s,$$

$$\hat{s_i} = \begin{cases} 0 & \text{for } i = 0 \\ s_{98} & \text{for } i = 99 \\ s_{i-1} \oplus (s_i \oplus CO_i) \cdot (s_{i+1} \oplus C1_i) & \text{else,} \end{cases}$$

$$s'_i = \begin{cases} \hat{s_i} \oplus (F0_i \cdot FB_s) & \text{if } FB_s = 0 \\ \hat{s_i} \oplus (F1_i \cdot FB_s) & \text{if } FB_s = 1. \end{cases}$$

3. System Clock. This function takes as inputs registers R, S, a "mixing" Boolean value (m = true or false), and an input bit (IB_c), and includes executing one iteration of each of the above two functions,

$$CB_r = s_{34} \oplus r_{67},$$

$$CB_s = s_{67} \oplus r_{33},$$

$$IB_r = \begin{cases} IB_c \oplus s_{50} & \text{if } m = \text{true} \\ IB_c & \text{if } m = \text{false} \end{cases},$$

$$IB_s = IB_c,$$

$$\text{Run Linear Register Clock}(R, IB_r, CB_r),$$

$$\text{Run Nonlinear Register Clock}(S, IB_s, CB_s).$$

2.8.2 Initialization

As described in [18], we initialize the values of all registers as 0. We then execute the system clock with the inputs for every bit in the IV, Key, and then additional pre-clock iterations, as follows:

1. For every bit $i \in$IV, we run the System Clock with inputs $(R, S, m = \text{true}, IB_c = IV_i)$.
2. For every bit $j \in$Key, we run the System Clock function, inputting $(R, S, m = \text{true}, IB_c = K_j)$.

3. Run the System clock 100 times, with $(R, S, m = \text{true}, IB_c = 0)$ as inputs.

This completes the initialization of the registers for the Mickey cipher.

2.8.3 Encryption

Per [18], for a desired keystream of length L, we generate keystream z by XOR'ing the inputs from the registers as shown below:

$$z_i = r_0 \oplus s_0,$$
$$\text{System Clock}(R, S, m = \text{false}, IB_c = 0).$$

The keystream is then added bit-wise to the plaintext to encrypt.

2.9 Trivium

This stream cipher was proposed by C. De Canniere and B. Preneel in [19]. This simple cipher uses a key of 80 bits, an IV of 80 bits, and a few simple taps along the 288-bit register to generate keystream.

2.9.1 Functions

This cipher has only one function defined in [19], which is synonymous with one discrete timestep for the system. We note that s_i denotes the current value of register i, and that all operations are defined on GF(2). It is defined as follows:

$$t_1 = s_{66} + s_{93},$$
$$t_2 = s_{162} + s_{177},$$
$$t_3 = s_{243} + s_{299},$$
$$z_i = t_1 + t_2 + t_3,$$
$$t_1 = t_1 + s_{91} \cdot s_{92} + s_{171},$$
$$t_2 = t_2 + s_{175} \cdot s_{176} + s_{264},$$
$$t_3 = t_3 + s_{286} \cdot s_{287} + s_{69}.$$

To progress the rest of the register, we shift the values as follows:

$$\begin{aligned}(s_1, s_2, \ldots, s_{93}) &= (t_3, s_1, \ldots, s_{92}),\\ (s_{94}, s_{95}, \ldots, s_{177}) &= (t_1, s_{94}, \ldots, s_{176}),\\ (s_{178}, s_{179}, \ldots, s_{288}) &= (t_2, s_{178}, \ldots, s_{287}).\end{aligned}$$

2.9.2 Initialization

To initialize the system, per [19], the Key and IV are loaded into the register positions, with the remaining values padded as 0 or 1, per the below assignments:

$$\begin{aligned}(s_1, s_2, \ldots, s_{93}) &= (K_1, \ldots, K_{80}, 0, \ldots, 0),\\ (s_{94}, s_{95}, \ldots, s_{177}) &= (IV_1, \ldots, IV_{80}, 0, \ldots, 0),\\ (s_{178}, s_{179}, \ldots, s_{288}) &= (0, \ldots, 0, 1, 1, 1).\end{aligned}$$

The system is then clocked $1152 = 4 \cdot 88$ times to ensure progression through four full cycles, but with the keystream bits fed back into the system as follows:

$$\begin{aligned}t_1 &= s_{66}s_{91} \cdot s_{92} + s_{93} + s_{171},\\ t_2 &= s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264},\\ t_3 &= s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69},\\ (s_1, s_2, \ldots, s_{93}) &= (t_3, s_1, \ldots, s_{92}),\\ (s_{94}, s_{95}, \ldots, s_{177}) &= (t_1, s_{94}, \ldots, s_{176}),\\ (s_{178}, s_{179}, \ldots, s_{288}) &= (t_2, s_{178}, \ldots, s_{287}).\end{aligned}$$

2.9.3 Encryption

For a visual representation of the system, see Figure 2.7. Once the initialization is complete, to generate each bit of keystream one simply clocks the system, with the keystream bit i output as discussed above,

$$z_i = t_1 + t_2 + t_3.$$

Encryption is achieved by bit-wise addition of the keystream with the plaintext.

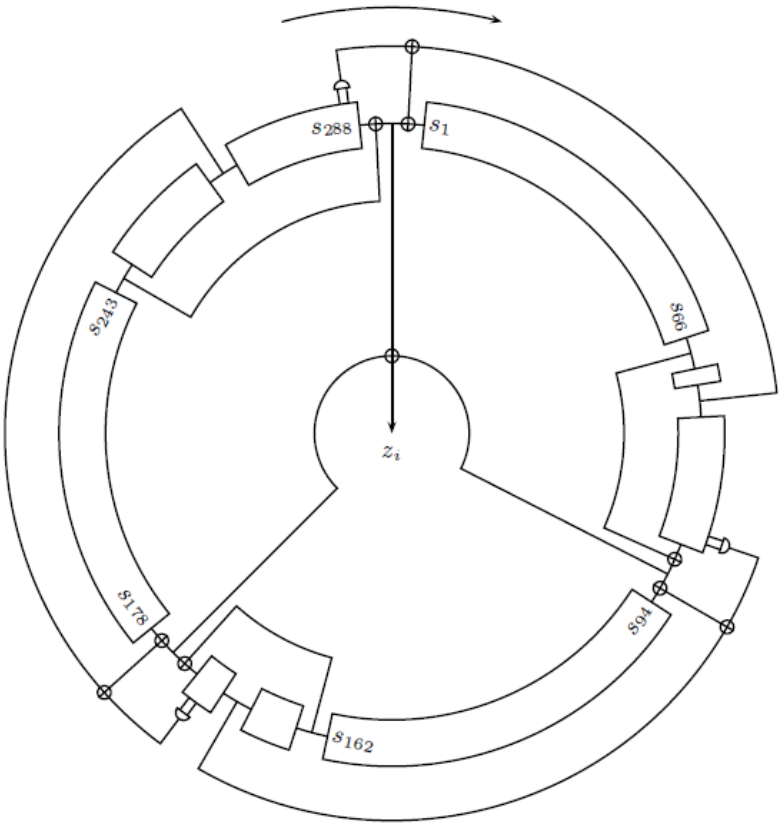

Figure 2.7. The Trivium Cipher. Source: [19].

This completes our brief explanation of each of the ciphers this thesis considers. Now that we have established the process each cipher uses to encrypt or produce keystream, we proceed to Chapter 3 to discuss how we will analyze and compare our ciphers in subsequent chapters.

CHAPTER 3:
Methodology

The aim of this chapter is to identify and outline the processes this thesis will use to analyze and compare the ciphers discussed in Chapter 2. This chapter is separate from the analysis in subsequent chapters to explain the process independent of the ciphers considered. This could facilitate flexibility in considering additional ciphers for future work, if desired. We will discuss a broad overview of the ciphers first, then compare operations counts, and conclude with direct comparisons.

This thesis focuses on mathematical operations and their effects, not a specific program or implementation of a cipher. This is because in an implementation of a specific cipher there may be more efficient techniques that may not execute the same exact sequence of operations in the cipher's algorithm. This may be done for a variety of reasons, but one common reason is to improve efficiency. As an example, very few software implementations of an LFSR will actually "shift" the values from one box to another, but will instead use a designated cache of memory for calculations and only recall as required. This greatly reduces the number of operations the machine has to conduct and is distinct from the way a physical hardware implementation of the LFSR functions. Rather than a line-by-line discussion of implementation code, or of the nuanced discussion of computer memory allocation as compared to hardware function, we focus our analysis on the algorithms directly because we are interested in analyzing the effects of the operations.

3.1 Broad Comparison

The first section of our analysis is a broad comparison of general characteristics between the ciphers discussed in Chapter 2. We will provide an outline and discussion comparing the general architecture, inputs, and outputs to give initial comparisons. We then note some structural similarities or trends between different ciphers, looking at general sequences or trends in types of operations. This provides an initial glimpse into general techniques used in each cipher and gives context for subsequent analysis.

3.2 Operation Counts

The second section of analysis compares mathematical operations counts across the ciphers. We provide a total count for each cipher and then examine the frequency of different operations, with some notes on the code used to generate the data in Appendix A.6. The aim of this section is to consider the frequency and composition of the various ciphers' operations, comparing the ciphers and identifying trends.

To allow for a clear comparison of operations across the ciphers, we make the following general simplification for the operations executed, in line with the types of operations discussed in Section 1.7. The following are the short-hand values used in our analysis:

1. Addition or XOR;
2. Lookup or S-Box;
3. Multiplication or AND;
4. Rotate or shift;
5. Other: permutation, string slicing, or concatenation.

In order to compile a list for each cipher, each algorithm must be evaluated in accordance with a standard order of operations. In this thesis we operated in a manner similar to how many computer programs would interpret the information, working from inside the parentheses out and then left to right. Specifically, we evaluate all function calls (i.e., an S-Box lookup), then parenthetical statements, then proceed with the standard ordering of operations (with rotations given equal ordering with multiplication). We do not consider reduction by a modulus as an operation per se. This is because in most cases we operate in a finite field (e.g., bytewise operations), and thus from an algebraic standpoint we are not conducting an "operation" at all.

As a rudimentary example, we consider the basic "quarterround" function from Salsa20, as discussed in Section 2.5. In each line of this function, we evaluate the addition first, then the rotation, then the XOR. Using our above simplification creates the sequence: 1,4,1. Thus the one iteration of the function performs the sequence (1,4,1) four distinct times, conducting 12 operations total. A function which calls a quarterround (i.e. the rowround) is simply treated as executing those precise operations each time the function is called.

To facilitate comparisons between ciphers, we iterate each algorithm based on the keystream

size. The largest cipher output of 64 bytes per algorithm iteration is from Salsa20, so all other algorithms were repeated to produce this size keystream. As an example, a hardware cipher that generates keystream one bit per cycle will have to clock eight times to generate one byte, then clock $64 \cdot 8 = 512$ additional times to generate the keystream of one iteration of Salsa20. The output of this process is a list of operations for each cipher, which specifies the list of operations executed to generate a common length of keystream. We then use the Python programming language to analyze these lists.

We again note that in building these lists we are using the algorithms from the ciphers described in Chapter 2, *not* the C or any other programming language implementation. We point this out again because of the significant impacts relevant to this section. Some of the reasons we make this distinction are:

- These counts do not necessarily indicate the processing requirements of a cipher. One example of this apparent discrepancy is the difference between adding two six-digit numbers via software and the hardware implementation of a bitwise AND operation. These are mathematically and conceptually related operations, but do not have the same computational requirements or complexity.
- In order to compare the ciphers, as mentioned previously, we repeat the algorithm for the cipher to generate 64 bytes of keystream. While this allows us to compare the frequency of operations for a comparable amount of encryption, it also means that the hardware ciphers (Grain, Mickey, and Trivium) will have disproportionately large operations counts.
- These counts do not account for the efficiencies gained in a programming language's specific implementation of an algorithm. An example of this would be managing a collection of counters as a vector and incrementing the counters via a single addition operation on that vector. Efficient management of memory and computer-level operations are not the same across different implementations, so this thesis resides at the algorithmic level.
- Differences between a hardware/software implementation are not trivial in computation or efficiency. But if the implementation is faithful, the effects are still retained. The example mentioned previously was the implementation of an LFSR. This thesis considers one LFSR "clock" to be additions (from each tap) with a final shift, which describe the mathematical operations conducted but not precisely how a com-

puter would efficiently manage the memory. Some of the ciphers explicitly discuss computational efficiency in greater detail, such as [14].

3.3 Specific Comparison

In this third component of our analysis, we draw specific comparisons in different parts of the algorithms to discuss structural concrete similarities between ciphers that may not be apparent at the broad level discussed in previously in Section 3.2. We conduct the specific comparisons by analyzing recurring sequences of operations, and use the term "n-gram" to describe a specific sequence of operations of length n.

Using the operations sequences obtained from the process in Section 3.2, we wrote a Python script to count and store these sequences. We analyze the most common n-grams across the ciphers to assess trends, as well as how those trends relate back to the fundamentals of encryption discussed in Section 1.3.

With this framework in mind, we proceed to Chapter 4 to discuss the results of the application of this analysis.

CHAPTER 4:
Results

This chapter provides a comparison of the ciphers discussed in Chapter 2 via the framework discussed in Chapter 3. We progress from a broad comparison of the ciphers to compare operations counts, concluding with comparisons of operations sequences.

4.1 Broad Comparison

Table 4.1 provides an overview of our ciphers. The remainder of this section will discuss the table, then three key structural similarities between the ciphers.

Cipher	Handle	Input	Output
HC128	Changing S-Box	128-bit key, 128-bit IV	1 byte (8 bits)
Rabbit	Internal Variables	128-bit key, 64-bit IV	16 bytes (128 bits)
Salsa20	Based on Salsa Hash	128- or 256-bit key, 64-bit IV	64 bytes (512 bits)
Sosemanuk	LFSR and FSM	128–256-bit key, 128-bit IV	16 bytes (128 bits)
Grain	LFSR and NFSR	80-bit key, 80-bit IV	1 bit
Mickey	Irregular Linear and Nonlinear Registers	80-bit key, optional 80-bit IV	1 bit
Trivium	Three taps on three registers	80-bit key, 80-bit IV	1 bit
DES	Previous Standard (Feistel cipher)	64 (56) bit key	8 bytes (128 bits)
AES	Current Standard, 10 rounds (SPN cipher)	128–256-bit key	16 bytes (256 bits)

Table 4.1. Broad Cipher Comparison.

4.1.1 Broad Notes

The "handle" provided in Table 4.1 is a short description we provide to give a broad contrast in techniques between the ciphers. It is not intended to be rigorous or complete, but rather to highlight differences.

We note that the driving force in the similarities of the key length was likely the ECRYPT competition requirements oriented at a level of security commensurate with a 128-bit key. The differences in initialization values required is also notable, as we see the range of inputs from no IV, to optional, to mandatory IV, depending on how the cipher generates its initial internal state.

The output length discrepancy is highlighted succinctly in Table 4.1, with outputs ranging from 1 to 512 bits. This is due to the nature of these minimal hardware ciphers, but means that the operations counts in the next section will have to multiply the hardware ciphers' algorithm operations list by 512 when comparing the operations sequences and common n-grams.

Due to the very different encryption techniques used in DES and AES, it is also worth highlighting that the way in which the key is used is markedly different between these block ciphers and the stream ciphers in this thesis. Specifically, block ciphers (when functioning via ECB) use information from the key and interact iteratively with the plaintext in order to generate the ciphertext. So when we count operations in the next section for these block ciphers in ECB, we are counting interactions between the key, constants, and the plaintext. In comparison, the stream ciphers only use their key and IV (if required) to create a kind of PRNG that is added on top of the plaintext. Thus, when we count operations with stream ciphers, we are discussing the ways the basic key/IV information is modified in order to generate an apparently random keystream.

4.1.2 Structural Similarities

This section discusses two major structural trends across all stream ciphers considered, and then a similarity between each of the block ciphers and a one additional stream cipher.

Broad Commonality: ARO

Consistent across all of the stream ciphers in this thesis is a very general pattern: two or three additions, alternating with one to three rotations, then obscured in some way prior to producing keystream. We will refer to this as the Add Rotate Obscure (ARO) framework. This framework provides a common cryptographic structure present in every single stream cipher considered. We will consider each element in turn:

- Two or three additions. This is usually in the form of adding in part of a key, incrementing state variables, or combining information in other registers to produce a new value.
- One to three rotations. This is often interspersed with additions to help both confuse and diffuse the effects of the key or counter added into the value, and includes a possible register shift.
- Obscuration. The idea behind this step is that the cipher does not want a cryptanalyst to be able to "see" the entirety of what is being done, and so "mixes up" the output considerably. This could consist of slicing and permuting the data in different ways, using S-Boxes to limit the direct ability to observe the mechanics of keystream generation, or combining the output from multiple systems, so that the cipher is more difficult to analyze.

Table 4.2 provides examples of the ARO framework from each of the stream ciphers. This framework can effectively implement the goals of confusion and diffusion, and can contribute to breaking up linearity if the rotations are incorporated in the right way. In adding values together, the cipher confuses the information that will become the keystream (in addition to diffusing previous rounds' changes), and through rotation it diffuses these changes while further complicating the relationship between input and output. Thus several iterations of adding and rotating result in both effective confusion and thorough diffusion.

While not a cryptographic primitive per se, ARO provides an example of a low-level block or tool that is used through many iterations to achieve encryption. Its repeated use in all of our stream ciphers highlight that it is a valuable tool to facilitate encryption, demonstrating a consistent sequence of operations to that end.

As previously mentioned in Section 3.2, the "quarterround" function in Salsa20 (1,4,1 or add, rotate, XOR) is consistent with this application. This deliberate sequence in the Salsa20

Cipher	Addition	Rotation	Obscuration
HC128	f and g functions	f and g functions	Use of S-Boxes
Rabbit	Next-state function	Next-state function	Variable slice and XOR
Salsa20	Quarterround function	Quarterround function	Salsa20 expansion and hash functions
Sosemanuk	LFSR	LFSR	Serpent1 combination with FSM
Grain	LFSR	LFSR	Boolean function with NFSR bits
Mickey	RTaps in linear registers	Linear register shift	irregular combiner
Trivium	NFSR taps	Register shifts	3 taps and nonlinear structure

Table 4.2. Cipher Structural Comparison.

cipher was explicitly discussed as the building block (Add-Rotate-XOR or ARX) in [14]. In that work, the author discusses how the repeated use of this framework is efficiently achieved in computer memory and execution, rather than as a general cryptographic framework.

Use of Registers

As Table 4.2 points out, LFSRs and NFSRs can fulfill the first two steps in ARO. We can see this visually by examining the diagrams of the hardware ciphers or the conceptual diagram of the Sosemanuk cipher. The use of registers in this manner is somewhat more obscured in the other three ciphers (HC128, Rabbit, and Salsa20), but we will demonstrate how each of the functions listed in Table 4.2 perform a function analogous to a register.

In HC128, the f and g function act like a miniature register: they rotate the input value a few times, then add each of those values together in a finite field. If we refer to Section 2.3, we can see that while the inputs are obscured by the S-Boxes, those inputs are still acting like the "initial value" used to populate a register, which these two functions add together to return a value. In this way, these functions are analogous to a miniature LFSR.

The way the Rabbit cipher relies on its state variable updates is also similar to a miniature

LFSR. As discussed in Section 2.4, the cipher builds a miniature register state with the g function, then builds each state variable based on a different set of taps. Again, we see that reoccurring pattern in operations.

Similarly, Salsa20's rowround function, as discussed in Section 2.5, functions like a very primitive two-stage register. This cipher has additional actions prior to keystream generation, but the basic principle still applies.

Not all of the stream ciphers considered in this thesis have obvious linear or nonlinear feedback registers defined. But we can see how the ARO framework applies a "register-like" action in each that contributes to the cryptographic process.

Block Cipher Similarities

In addition to these broad conceptual similarities between the stream ciphers, we now discuss one structural similarity between each of the block ciphers considered and one of the stream ciphers.

First, we highlight that the FSM component of the Sosemanuk cipher is somewhat similar to the Feistel System used in DES. In each round, one half of the register is passed to the other (Sosemanuk's FSM makes a single addition per round), whereas the other half of the register has several operations performed on it. This is the heart of DES, whereas it is only a component of Sosemanuk, but it is one of the few very clear similar functions between DES and the stream ciphers. See Figure 4.1 for a visual representation of this similarity.

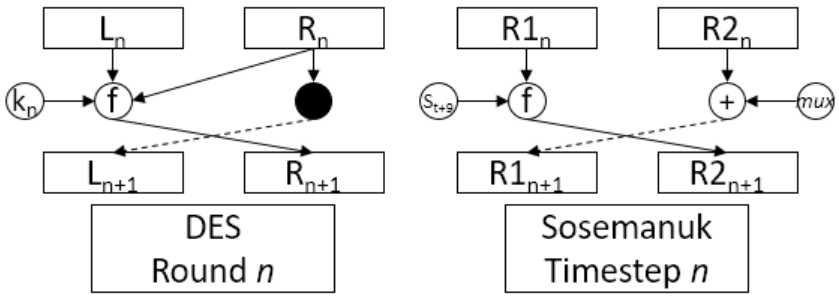

Figure 4.1. DES-Sosemanuk Structural Similarity.

For the purposes of highlighting procedural similarities, we combined some steps from each of the ciphers to build the depiction in Figure 4.1. Specifically, the f symbol shown is not

a single function as defined by the respective algorithms, but is written to more generally indicate that a function is performed with the given inputs. While this is obviously not a perfect similarity or parallel, the structural comparison is worth noting.

Second, we note that the only two ciphers discussed in this thesis to deliberately organize information into a matrix to perform operations on columns or rows are AES and Salsa20. In the case of AES, the plaintext is reshaped into a matrix, and matrix multiplication between the plaintext and a set of constants facilitates rapid diffusion in every round of encryption. This is distinct from Salsa20, which organizes the key and IV intermixed with constants (via its expansion function) in a matrix and then operates on functions of the rows and columns of that matrix, but has a similar effect of ensuring rapid diffusion.

In both cases, a depiction of the ciphers in matrix format can help one visualize the encryption process. This organization is not merely illustrative, but provides an organization to the data that helps describe how the encryption process works. Using a matrix allows elements to be accessed in columns or rows, broadening the utility of the rotation or multiplication operations. As such, the concept of organizing the data is not just conceptual, but is also procedural.

4.2 Operation Counts

In this section we provide the results of the operation count framework laid out in Section 3.2. For notes on the Python code used to generate the lists upon which the subsequent analysis is conducted, see Appendix A.6.

4.2.1 Raw Operation Counts

Figure 4.2 demonstrates the operation counts by cipher to encrypt 64 bytes of information. We note that the x-axis (indicating the counts) is log-scaled. As mentioned in Section 3.2, these counts are not an indicator of the computational resources required, as the very high numbers present in the hardware ciphers are due in part to their bit-wise operation.

Figure 4.2 demonstrates that Sosemanuk algorithm has the lowest count of operations for a comparable keystream generation, followed by Rabbit. If we first consider the two major functions used in Sosemanuk's keystream generation (the LFSR and FSM, as seen

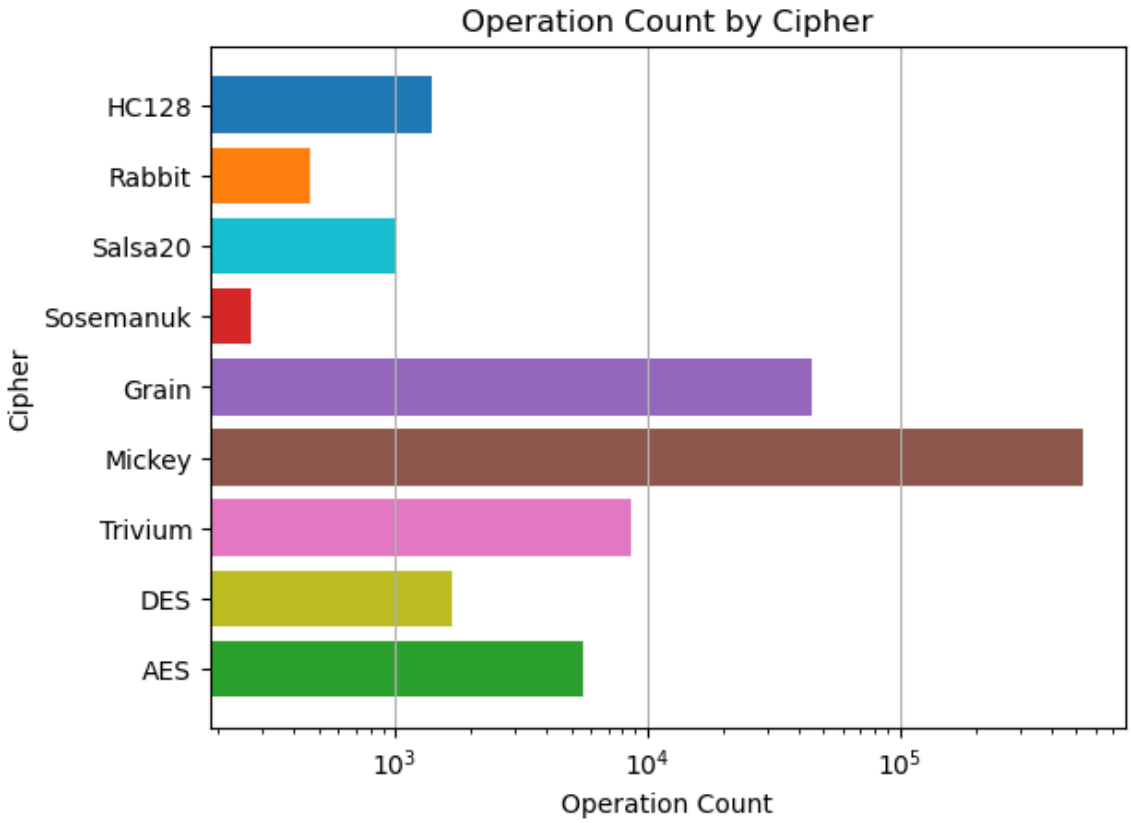

Figure 4.2. Operation Count by Cipher.

in Section 2.6 and Figure 2.4), we see several bytes of output generated at a time. Each of these functions only conduct a few additions, some multiplication, and a rotation, in order to generate keystream, relying on the elegant mathematics of these systems to provide reliable, random-looking keystream.

The Rabbit cipher has the next-fewest operation count, and this seems reasonable as its state variables or counters conduct simple updates (albeit carefully crafted) to generate keystream several bytes at a time. Again we see how a careful system of interconnected parts provides an efficient technique to generate keystream.

One implication of these low-operation-count ciphers is that higher frequencies of operations (or put another way: doing more with the information) may not correspond to more security; careful implementation of structure, as seen in these ciphers, can reduce the total number of

operations. Again, this does not mean that computational complexity bears no correlation with security, simply that effective does not mean complicated.

We note that these counts are for the generation of 64 bytes of keystream, and that they do not indicate the quantity of operations required for initialization of the ciphers. Figure 4.3 demonstrates the way the ciphers compare if we include the counts from initialization.

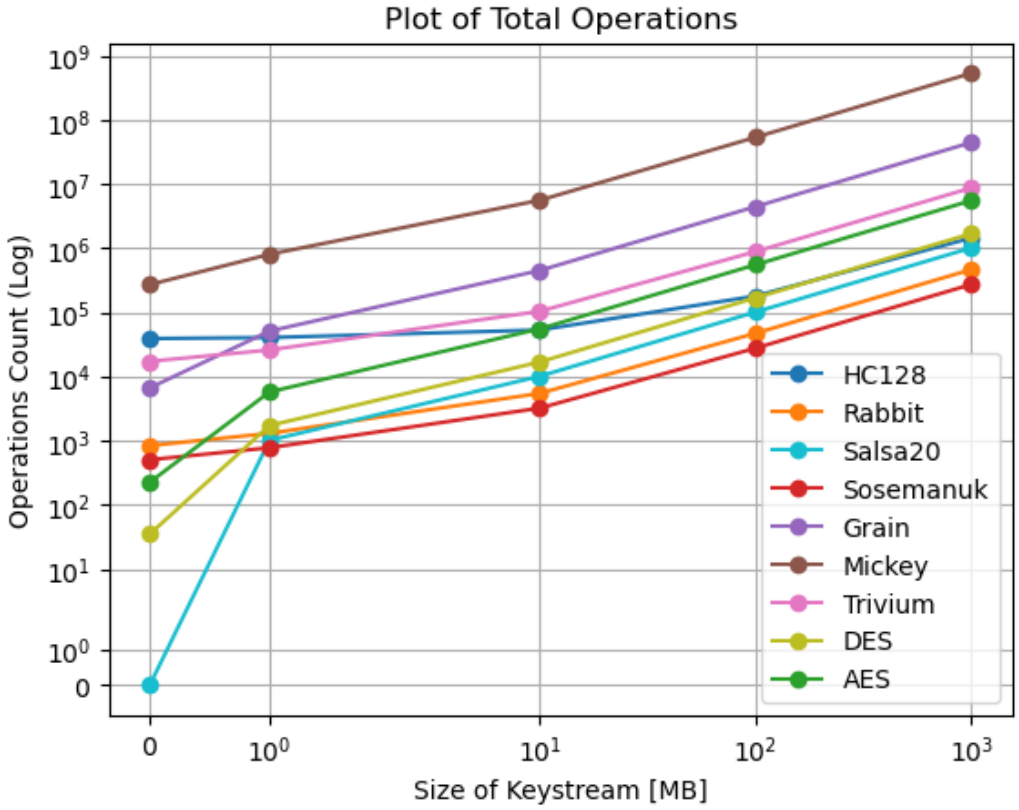

Figure 4.3. Total Operation Count for Keystream Generation.

We note that one cipher, Salsa20, requires no initialization and so has a linear slope in Figure 4.3, whereas in most cases the slope changes at some point due to initialization operations. In general, the ranking of ciphers operations counts remains steady independent of keystream size. The one exception is HC128, which requires a very high count of operations to initialize (second highest of the nine ciphers considered) but at 1GB of keystream is just below the median (third lowest of nine ciphers).

4.2.2 Count by Operation Type

Examination of Figure 4.4 provides more fidelity on how the ciphers use each of the operations in order to encrypt. We again note the log-scaled axis (this time the vertical axis). We make four major observations from this additional information.

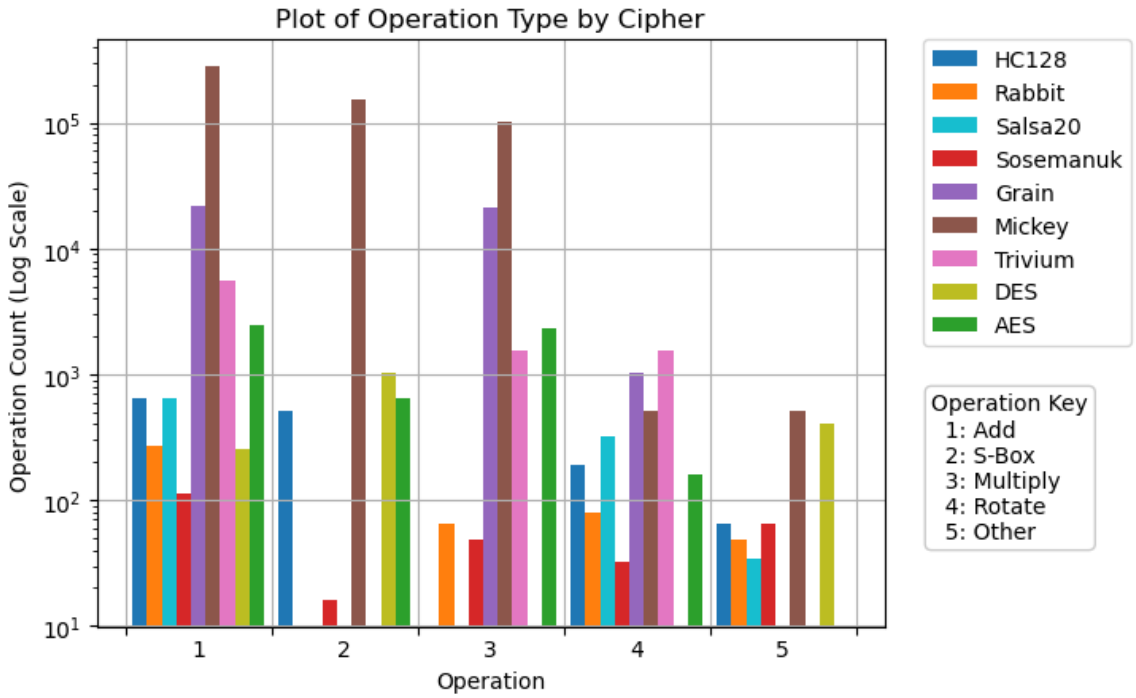

Figure 4.4. Operation Count By Operation and Cipher.

First, as one might expect, the hardware ciphers generally conduct the most operations in every category. This is reasonable: these ciphers operate bitwise, and produce keystream bitwise, so it will take many iterations of the algorithms described to produce a similar size keystream.

A second observation is how several ciphers used no S-Boxes at all. Rabbit, Salsa, Grain, and Trivium all used other cryptographic operations, despite the widespread acceptance and use of the S-Box as a classical tool to stymie linear cryptanalysis. This speaks to the reality that, while S-Boxes can be very valuable, they are not necessary or universally used to achieve effective encryption.

Third, we note that HC128, Salsa, and DES use no multiplication (or AND operation).

The use of multiplication with a modulus can be a computationally challenging barrier to invert, providing a powerful shield from cryptanalysis. In asymmetric cryptography this is the central mathematical operation used to facilitate encryption; but these ciphers achieve their confusion through other means. Thus it is also worth noting that despite its utility, multiplication or the use of NFSRs is also not required or consistently used for secure encryption.

Finally, we note the operations conducted by all ciphers: addition and rotation. This coincides with the general framework observed in Section 4.1.2. All ciphers but AES conduct more additions than any other operation, which makes sense given this operation's utility in key addition, incrementing variables, and combining information. Addition is ubiquitous, and so it is unsurprising to see it as the most common operation.

More interesting is the consistent use of the rotation operation. With the exception of DES, every cipher examined uses frequent rotations. Examining Figure 4.4, we see that for most ciphers, rotation is the second most frequent operation after addition. We note that this holds even though only one rotation is counted regardless of its size (e.g., Trivium has a 93-bit NFSR, or AES rotating four bytes in the row of a matrix of data). This provides a strong indicator on the importance of rotation.

4.3 Specific Comparison

This section discusses specific n-grams of cryptographic primitives and analyzes commonalities between various ciphers. We first present the most common n-grams for each cipher, then compare trends in the most common n-grams.

Table 4.3 provides an overview of the most common n-grams. Each row is a different cipher, with the columns providing the most common 2-, 3-, and 4-gram for that cipher. We note the * indicates the most common n-gram(s) are n copies of a single operation (usually 1), so the next most common non-uniform sequence is shown. In some cases in Table 4.3, we see the depicted n-gram is still a repeated operation; in those situations the most frequent n-gram count is so far beyond the next n-gram that the original sequence is reported. See Appendix A.4 for the listing of the most common n-grams output from our Python script.

We note that, while addition-only n-grams are the most common, sequences containing

Cipher	2-gram	3-gram	4-gram
HC128	1,2*	1,2,2*	1,1,1,2*
Rabbit	1,1	4,1,1*	1,1,1,1
Salsa	1,4*	1,4,1	1,4,1,1
Sosemanuk	1,3*	1,5,5*	1,5,5,5
Grain	3,3*	3,3,3*	3,3,3,3*
Mickey	1,2	3,1,2	1,1,1,1
Trivium	1,3*	1,1,3*	1,1,3,1
DES	2,2	5,1,5*	2,2,2,2
AES	3,3*	3,3,3*	3,3,3,1*

Table 4.3. Most Common *n*-grams.

addition with one or two additions with other operations are also very common. All of the stream ciphers but Grain have addition with another operation (e.g., 1,2 or 4,1,1), providing a partial example of a sequence comparable to the ARO structure discussed in Section 4.1.2. This observation provides additional support of this general structure, but it is not a direct support. Because the sequences are different across *n*-grams, we see that the precise structure across each of the ciphers is different.

4.3.1 2-grams

Figure 4.5 shows a comparison of the five most common *n*-grams from the ciphers, which are all 2-grams. Using the entries in the 2-gram column of Table 4.3, we plot the frequency of each of the 2-grams, indicating the cipher by colour. The quantity of operations to encrypt a comparable amount of data differs greatly (as discussed in Section 3.2 and seen in Figure 4.2), so we continue to use the log scale to compare the counts.

We note that this confirms the high frequency of the addition operation, and demonstrates that the specific *n*-grams used most commonly in different ciphers varies. This indicates that while there are clear examples of the general structure discussed in Section 4.1.2, the specific ciphers' implementations of those structures is not consistent, as expressed in *n*-grams or specific sequences of operations. This is roughly what we would expect, given the distinct characteristics and operations counts of each of the ciphers. We also note that the specific list compiled per order of operations may cause two similar-looking sequences

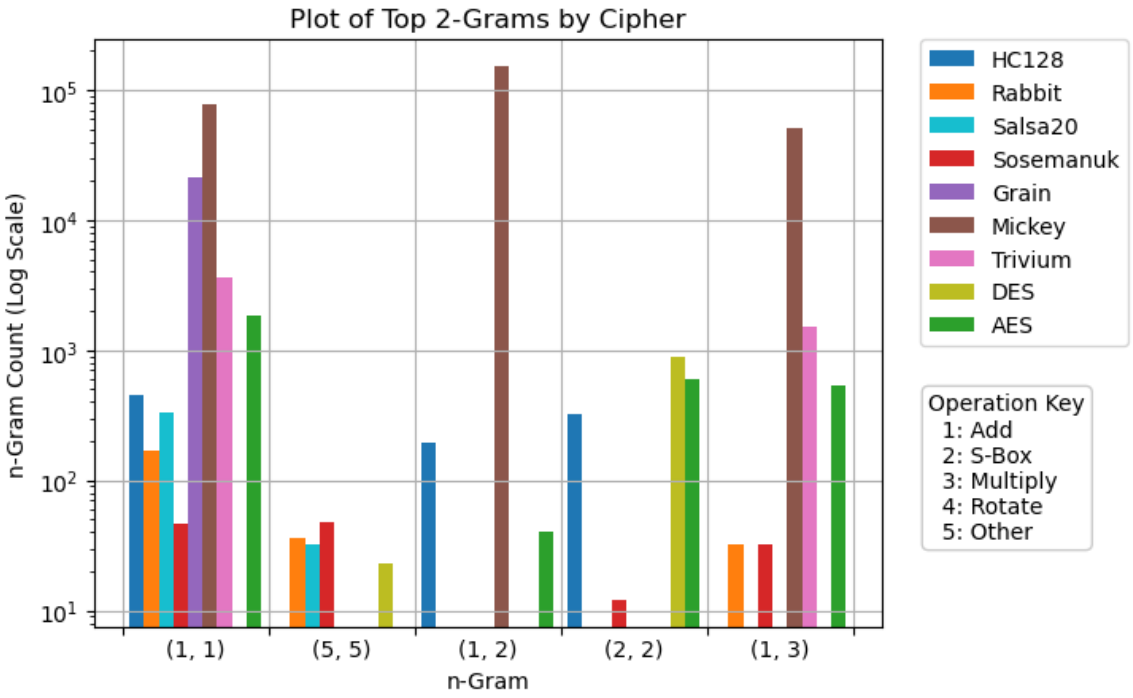

Figure 4.5. 2-gram Comparison.

may not result in the same n-gram.

Further analysis using the most common 3-grams and 4-grams from each of the ciphers from each of the ciphers provides no additional insight. See Appendix A.5 for these plots.

4.3.2 Most Common n-grams

In order to better assess significant operations, we consider which n-grams occur in the most ciphers (ideally all ciphers). To do this, we count which sequences appear in a cipher, then compare the counts to assess sequences that appear most frequently. However, after examining the frequency of n-grams in every cipher, there were none present in every single cipher. Specifically, DES and its use of a Feistel system conducts sequences of operations that did not have common overlap with several of the other ciphers. As a result, we select the five most frequent, non-addition-only n-grams.

Table 4.4 displays the results of the ten most consistently observed n-grams in each of the ciphers. Each row is the n-gram, and the count indicates how many of the ciphers include that

specific sequence of operations. Significantly, we note this table indicates that no sequence of operations is common to all ciphers (as there are nine ciphers considered in this thesis).

***n*-gram**	**Count**
(1, 1)	8
(1, 1, 1)	8
(1, 1, 1, 1)	8
(1, 1, 1, 1, 1)	8
(4, 1)	7
(4, 1, 1)	7
(5, 1)	6
(4, 1, 1, 1)	6
(4, 1, 1, 1, 1)	6
(1, 4)	6

Table 4.4. Most Commonly Observed *n*-grams.

To gain better fidelity, we refine the data from Table A.4 to examine which specific ciphers use these common sequences of operations. We remove the sequences of all one operation (retaining only (1,1)), and then remove highly similar operations (e.g., 4,1,1,1 and 4,1,1,1,1 as they are just extensions of 4,1,1) to consider unique and distinct *n*-grams. Figure 4.6 displays the counts by cipher of the five remaining operations that meet this criteria.

Figure 4.6 displays the counts for the top five most frequently seen *n*-grams in the ciphers considered (excluding addition-only sequences of more than two operations). The counts for each cipher is indicated by the colour of the bar above the specified *n*-gram. Three observations are worth noting from this figure.

First, there is no sequence used by all the ciphers. Additionally, the only specific sequence used by all of the stream ciphers is (1,1). This shows the broad differences between the specific operations conducted by the ciphers.

Second, despite the previous observation, we note that if we consider the 2-grams (1,4) and (4,1), all stream ciphers are included. This provides a concrete example of the alternating addition and rotation operations discussed in the ARO framework. We note that the only stream cipher that does not have the (1,4) operation is the Grain cipher, due to the order

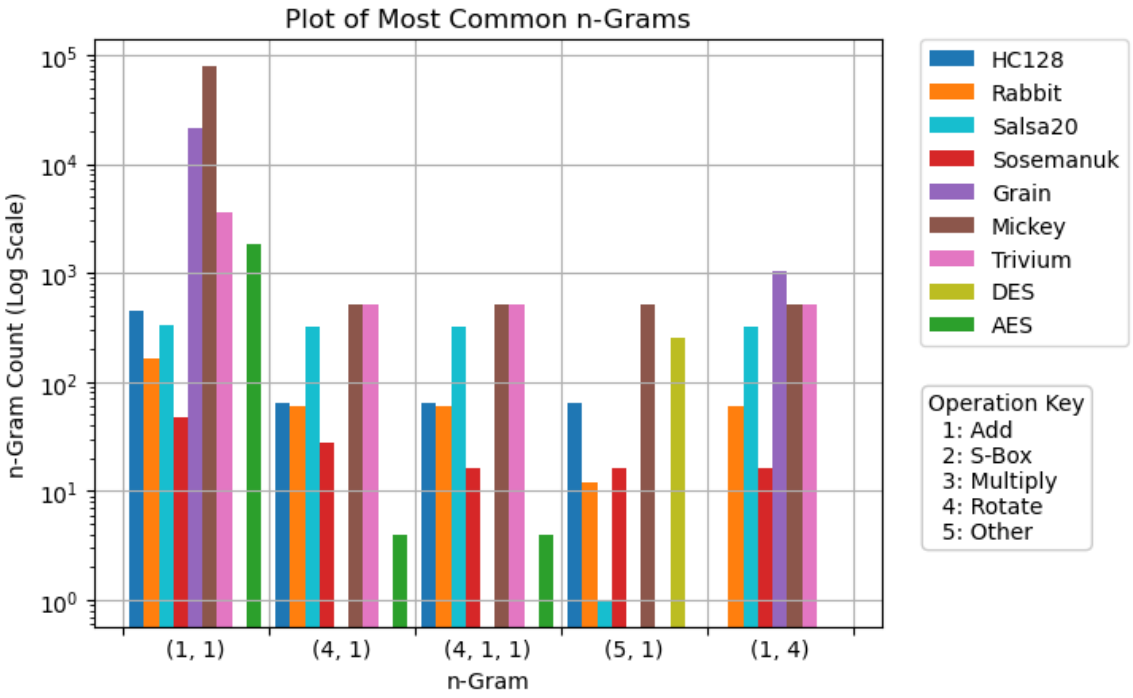

Figure 4.6. Most Commonly used n-gram Comparison.

we evaluated the operations. That said, the cipher does execute a sizeable quantity of (4,1). This discrepancy is due to how we programmed our order of operations.

Finally, we note that besides the (1,1) operation, all of these most common n-grams in Figure 4.6 are confirmation of the significance of the ARO framework.

In this chapter we demonstrated and discussed the outputs of the analysis framework presented in Chapter 3. We conducted a broad comparison, discussed structural similarities across various ciphers, compared operations counts, and investigated how the frequency of specific operations sequences, or n-grams, appeared between different ciphers. In the next chapter, Chapter 5, we summarize the major conclusions of the analysis presented here and present some areas for future research.

CHAPTER 5:
Conclusion

This chapter outlines the major observations and conclusions of the analysis in Chapter 4. In presenting our conclusions, we divide our findings into two general classes: low-level conclusions about the utility of mathematical operations, and higher-level conclusions or major observations about cipher structure. We conclude with additional topics for follow-on research. We note that all conclusions rest on the idea of a concept "correctly implemented," because any operation can be poorly implemented and, consequently, it will fail to achieve the desired result.

5.1 Mathematical Operations

The first four conclusions we highlight are all observations about the utility and necessity of specific mathematical operations. These can also be considered the "low-level" conclusions, because they focus one of the most basic component of the ciphers.

Unsurprisingly, our first conclusion is that addition is the most common operation across all of our ciphers considered. This can be seen in Figure 4.4 and in the annotated entries in Table 4.3 (or in the original output in Tables A.1, A.2, and A.3). This operation is probably the most rudimentary mathematical one, and can directly contribute to confusion and diffusion (via key addition and intermediate value addition, respectively).

Second, as discussed in Section 4.2.2 and seen in Figure 4.4, several secure stream ciphers do not implement S-Boxes in their algorithms. This indicates that S-Boxes are not required for a secure stream cipher, despite their early adoption (in block ciphers) and powerful prospects.

Similarly, while multiplication and modular reduction demonstrate remarkable security in asymmetric encryption, it is not required for an effective symmetric cipher. Examination of Figure 4.4 reveals there are ways to achieve the same cryptographic effects without this operation. Again, we have a useful operation, but not a necessary one.

Our final conclusion on operations, and perhaps much more interesting, is that rotation

appears to be an essential cryptographic operation. Figure 4.4 shows it to be common across our secure ciphers. We again note that rotation does not directly indicate a secure cipher. But as discussed in Sections 1.7 and 4.2.2, its ability to impact all of our cryptographic goals (confusion, diffusion, and nonlinearity) simultaneously hint at the notion that it is a critical operation for a secure cipher.

5.2 Cipher Structure

The second four conclusions are all observations that pertain to the nature and structure of ciphers. These observations can be thought of as "high-level" because they are broader, cipher-level considerations.

Our first structural conclusion is that there is not a clear correlation between quantity or complexity of operations and effective security. Per Figure 4.3, for one megabyte of encryption, we see a frequency spanning four orders of magnitude. We see a stronger lack of trend in the utility of more or less complex initialization processes, spanning five orders of magnitude of counts. This result is the clearest example of how effective encryption is not a product of considerable computation, but rather careful construction.

Second, despite some clear structural trends identified in Section 4.1.2, there were no clear n-grams (aside from sequences of all 1's) that consistently appeared across all ciphers. Table 4.3 and Figure 4.5 shows there were not clear front-runners in specific, low-level sequences. This further supports the notion that how operations are used in the larger perspective on the cipher is more important than which operations or operations sequences are used.

The third structural conclusion is that the implementation of refined mathematical structures can result in fewer operations required. As briefly discussed in Sections 1.2.3 and 1.2.3, careful selection of the LFSR feedback polynomials ensures generation of the entire field extension. Simply restated, this means the LFSR state spans all possible configurations and its outputs are as diverse as possible. If one were to count each timestep of an LFSR or NFSR as a single operation, then the operations counts of several ciphers would drop even further. This conclusion is affirmed in the general acceptance of these kinds of structures in both the hardware and software ciphers considered.

The final conclusion we highlight is the broad conceptual similarity, the ARO, present in

all of the stream ciphers we consider. As shown in Table 4.2, each of our ciphers displays the 2-3 addition, 1-3 rotation, and obscuration paradigm. This common implementation, despite taking very different forms in different ciphers, indicates its utility as a cryptographic primitive.

5.3 Further Topics for Research

As pointed out in Section 1.1, cryptography is a necessarily open-ended field. Based on the results discussed previously, we recommend the following as starting points for additional research.

1. Consider additional ciphers. We recommend ones from more diverse cryptographic backgrounds (e.g., from non-EU or American sources). Examination of additional ciphers may further reinforce or refute the conclusions in Sections 5.1 and 5.2. One examples of a way to extend this would be to consider the block ciphers International Data Encryption Algorithm (IDEA) and GOST (Magma), and include stream ciphers RC4 and A5/1 or A5/2, as these ciphers are fairly well-known and produced from different perspectives.
2. Change the paradigm. Modifying the perspective on the conduct of the order of operations, or the conceptual framework dictating how the operations are grouped, would most certainly return different results. This could also include a permuting of operations conducted at the same or almost the same step. Any of these steps could likely return very different results.
3. Examine modifications. Selective removal of small slices of the ciphers could reveal weaknesses and indicate the importance of certain operations or sequences of operations to security. This could involve extensive work to return results, but would likely indicate the utility of certain operations or n-grams more conclusively.
4. Conduct successful cryptanalysis. Many ciphers are proven insecure only after lengthy study, and perhaps the ones considered in this thesis will be no different. A further analysis and extrapolation of the attacks against simpler versions of these ciphers, including modified ciphers as mentioned in the previous topic, may reveal which operations or sequences are essential to security and which are not helpful.

This chapter presented the major conclusions of this thesis. We discussed the efficacy

of different mathematical operations and contrasted the essential and the apparently non-essential, then discussed how structure contributes to successful encryption, and offered our ideas for additional work to build on our findings.

As discussed in Section 1.1, there are many ways and reasons that people choose to encrypt information. Our hope is that these findings and future projects contribute to a larger discussion of what operations and sequences are most important to successful encryption. We hope our findings contribute to ensuring information security and helping to guarantee privacy in an increasingly information- and technology-driven society.

APPENDIX: Supporting Work and Additional Data

A.1 Polynomial Long Division

The principle of polynomial long division is demonstrated in many places, including many standard calculus textbooks and [5]. We remember that when calculating the below examples from $\mathbb{F}_4 = \mathbb{Z}_2[x]/\langle p(x)\rangle$ (where $p(x) = x^2+x+1$) we are only concerned with the remainder term at the bottom. To demonstrate the technique used to solve the polynomial division from Section 1.2.3, we demonstrate the first two problems where polynomial division is required.

A.1.1 Example (operations over $\mathbb{Z}_2$): x^2 divided by $p(x)$

Using traditional polynomial division, we have:

$$\begin{array}{r|l} & \quad\;\; 1 \\ \hline x^2+x+1 & \;\; x^2 \\ & -x^2-x-1 \\ \hline & \quad -x-1 \end{array}$$

the remainder term above, because the coefficients are from $\mathbb{Z}_2$, simplifies to $x + 1$.

A.1.2 Example (operations over $\mathbb{Z}_2$): x^3 divided by $p(x)$

Again traditional polynomial division returns the following:

$$\begin{array}{r|l} & \quad\;\; x-1 \\ \hline x^2+x+1 & \;\; x^3 \\ & -x^3-x^2-x \\ \hline & \quad\;\; -x^2-x \\ & \quad\;\; x^2+x+1 \\ \hline & \quad\;\; 1 \end{array}$$

the remainder term here is simply 1 as we saw in Section 1.2.3.

A.2 Group Table Work

We remember that we are discussing elements from $\mathbb{F}_4 = \mathbb{Z}_2[x]/\langle p(x)\rangle$ in terms of a root α of the irreducible polynomial $p(x) = x^2 + x + 1$. In order to reproduce the terms in Table 1.5, we must also keep in mind that the coefficients must be in $\mathbb{Z}_2$.

In Table 1.6, we must multiply the terms out and see how they simplify.

- For the 0 and 1 columns and rows, the answers are apparent.

- For $\alpha \cdot \alpha$, we can take advantage of how we defined α and $p(x)$ and our knowledge about $\mathbb{Z}_2$:

$$\begin{aligned} \alpha^2 + \alpha + 1 &= 0 \\ \alpha^2 &= -\alpha - 1 \\ \alpha^2 &= \alpha + 1. \end{aligned}$$

- For the $(1+\alpha)\cdot(1+\alpha)$ result, we use the above simplification to assist in the second step:

$$\begin{aligned} (1+\alpha)\cdot(1+\alpha) &= 1 + \alpha + \alpha + \alpha^2 \\ &= 1 + \alpha^2 \\ &= 1 + \alpha + 1 \\ &= \alpha. \end{aligned}$$

A.3 Algorithm Operation Lists

This section includes the lists of a single iteration of the encryption component of each of the ciphers discussed in Chapter 2. We note that these lists do not indicate how many iterations of the algorithm must be run in order to generate a comparable length of keystream or encryption. See Appendix A.6 for notes on the Python script that helped generate the data. .

1. DES: 5, 5, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 5,

1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5, 1, 5, 2, 2, 2, 2, 2, 2, 2, 2, 5, 1, 5.

2. AES: Too long to display; see Python code notes in Appendix A.6.
3. HC128: 1, 1, 1, 2, 2, 2, 2, 4, 4, 4, 1, 1, 1, 1, 2, 2, 1, 2, 2, 5, 1, 1.
4. Rabbit: 1, 1, 3, 3, 1, 4, 1, 1, 3, 3, 1, 4, 1, 1, 3, 3, 1, 4, 1, 1, 3, 3, 1, 4, 1, 1, 3, 3, 1, 4, 1, 1, 3, 3, 1, 4, 1, 1, 3, 3, 1, 4, 1, 1, 3, 3, 1, 4, 4, 4, 1, 1, 4, 1, 1, 4, 4, 1, 1, 4, 1, 1, 4, 4, 1, 1, 4, 1, 1, 4, 4, 1, 1, 4, 1, 1, 1, 1, 5, 5, 1, 1, 5, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 5, 5, 5, 5, 5, 5, 5, 1, 1, 1, 1, 1, 1, 1, 1.
5. Salsa20: Too long to display; see note in Appendix A.6.
6. Sosemanuk: 1, 5, 5, 5, 5, 1, 3, 4, 1, 1, 3, 3, 1, 1, 4, 1, 5, 5, 5, 5, 1, 3, 4, 1, 1, 3, 3, 1, 1, 4, 1, 5, 5, 5, 5, 1, 3, 4, 1, 1, 3, 3, 1, 1, 4, 1, 5, 5, 5, 5, 1, 3, 4, 1, 1, 3, 3, 1, 1, 4, 2, 2, 2, 2, 1, 1, 1, 1.
7. Grain: 1, 1, 1, 1, 1, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1.
8. Mickey: Too long to display; see note in Appendix A.6.
9. Trivium: 1, 1, 1, 1, 1, 3, 1, 1, 3, 1, 1, 3, 1, 1, 4, 4, 4.

A.4 Most Common n-grams

This section includes the returns when examining the highest 5 counts and their n-grams for a sequence of at least m operations long. In other words, we only see the 5 most common m-or-higher n-grams to return these values. We note the parentheses are because we are examining tuples (the Python data type).

We note there are some sequences of operations that appear on multiple tables; this is because in some ciphers there were sequences of operations that occurred very frequently even when considering only a minimum of $m = 2$. We also note that as we increase m, we see a decrease in the count of the most common n-grams. This follows our intuition.

Some entries in Table A.3 had to be abbreviated to properly display all of the information.

Cipher	1	2	3	4	5
HC128	(1, 1)	(2, 2)	(1, 1, 1)	(1, 2)	(1, 2, 2)
Rabbit	(1, 1)	(1, 1, 1)	(1, 1, 1, 1)	(1, 1, 1, 1, 1)	(1, 1, 1, 1, 1, 1)
Salsa20	(1, 1)	(1, 4)	(4, 1)	(1, 4, 1)	(4, 1, 1)
Sosemanuk	(5, 5)	(1, 1)	(1, 3)	(5, 5, 5)	(4, 1)
Grain	(1, 1)	(3, 3)	(1, 1, 1)	(3, 3, 3)	(1, 1, 1, 1)
Mickey	(1, 2)	(3, 1)	(3, 1, 2)	(2, 1)	(1, 2, 1)
Trivium	(1, 1)	(1, 3)	(3, 1)	(1, 1, 1)	(1, 1, 3)
DES	(2, 2)	(2, 2, 2)	(2, 2, 2, 2)	(2, 2, 2, 2, 2)	(2, 2, 2, 2, 2, 2)
AES	(1, 1)	(3, 3)	(1, 1, 1)	(3, 3, 3)	(1, 1, 1, 1)

Table A.1. Most Common Results: $m = 2$

Cipher	1	2	3	4	5
HC128	(1, 1, 1)	(1, 2, 2)	(1, 1, 1, 1)	(1, 1, 2)	(2, 2, 2)
Rabbit	(1, 1, 1)	(1, 1, 1, 1)	(1, 1, 1, 1, 1)	(1, 1, 1, 1, 1, 1)	(4, 1, 1)
Salsa20	(1, 4, 1)	(4, 1, 1)	(1, 4, 1, 1)	(1, 1, 4)	(4, 1, 1, 4)
Sosemanuk	(5, 5, 5)	(1, 5, 5)	(5, 5, 1)	(5, 1, 3)	(1, 3, 4)
Grain	(1, 1, 1)	(3, 3, 3)	(1, 1, 1, 1)	(3, 3, 3, 3)	(1, 1, 1, 1, 1)
Mickey	(3, 1, 2)	(1, 2, 1)	(1, 1, 1)	(1, 1, 1, 1)	(1, 1, 1, 1, 1)
Trivium	(1, 1, 1)	(1, 1, 3)	(1, 3, 1)	(3, 1, 1)	(1, 1, 3, 1)
DES	(2, 2, 2)	(2, 2, 2, 2)	(2, 2, 2, 2, 2)	(2, 2, 2, 2, 2, 2)	(5, 1, 5)
AES	(1, 1, 1)	(3, 3, 3)	(1, 1, 1, 1)	(1, 1, 1, 1, 1)	(1, 1, 1, 1, 1, 1)

Table A.2. Most Common Results: $m = 3$

We note that these longer sequences of the same operation are the reason for the very high frequency of shorter sequences of that same operation. As an example of this, in the sequence "$(1, 1, 1, 1)$" we have the n-gram "$(1, 1)$" three times.

A.5 Most Common 3- and 4-gram Plots

In this section we present the plots of the most common 3- and 4-grams from Table 4.4. We again note that the reason this is presented here is because it does not provide substantial new insight beyond what is already discussed in Section 4.3.2.

Cipher	1	2	3	4	5
HC128	(1, 1, 1, 1)	(1, 1, 1, 2)	(1, 1, 2, 2)	(1, 1, 1, 2, 2)	(1, 1, 1, 1, 2)
Rabbit	(1, 1, 1, 1)	(1, 1, 1, 1, 1)	(1, 1, 1, 1, 1, 1)	(seven 1's)	(eight 1's)
Salsa20	(1, 4, 1, 1)	(4, 1, 1, 4)	(1, 1, 4, 1)	(1, 4, 1, 1, 4)	(4, 1, 1, 4, 1)
Sosemanuk	(1, 5, 5, 5)	(5, 5, 5, 5)	(5, 5, 5, 1)	(5, 5, 1, 3)	(5, 1, 3, 4)
Grain	(1, 1, 1, 1)	(3, 3, 3, 3)	(1, 1, 1, 1, 1)	(3, 3, 3, 3, 3)	(1, 1, 1, 1, 1, 1)
Mickey	(1, 1, 1, 1)	(1, 1, 1, 1, 1)	(1, 1, 1, 1, 1, 1)	(seven 1's)	(eight 1's)
Trivium	(1, 1, 3, 1)	(1, 3, 1, 1)	(1, 1, 3, 1, 1)	(1, 1, 1, 1)	(3, 1, 1, 3)
DES	(2, 2, 2, 2)	(2, 2, 2, 2, 2)	(2, 2, 2, 2, 2, 2)	(seven 2's)	(5, 1, 5, 2)
AES	(1, 1, 1, 1)	(1, 1, 1, 1, 1)	(1, 1, 1, 1, 1, 1)	(3, 3, 3, 3)	(3, 3, 3, 1)

Table A.3. Most Common Results: $m = 4$

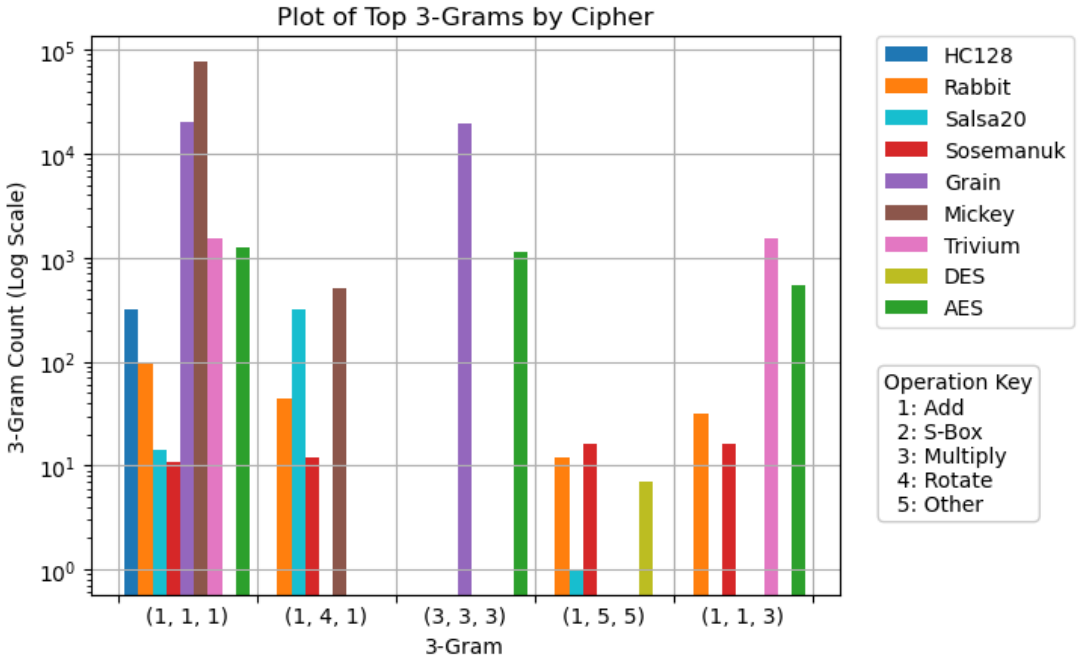

Figure A.1. 3-gram Comparison.

Figure A.1 illustrates the frequency of the most common 3-grams in each cipher. With the exception of the (1,1,1) 3-gram, there are not universally-used sequences to highlight.

Figure A.2 illustrates the frequency of the most common 4-grams in each cipher. Again,

With the exception of the (1,1,1,1) 4-gram, there are no major trends or specific sequences worth highlighting.

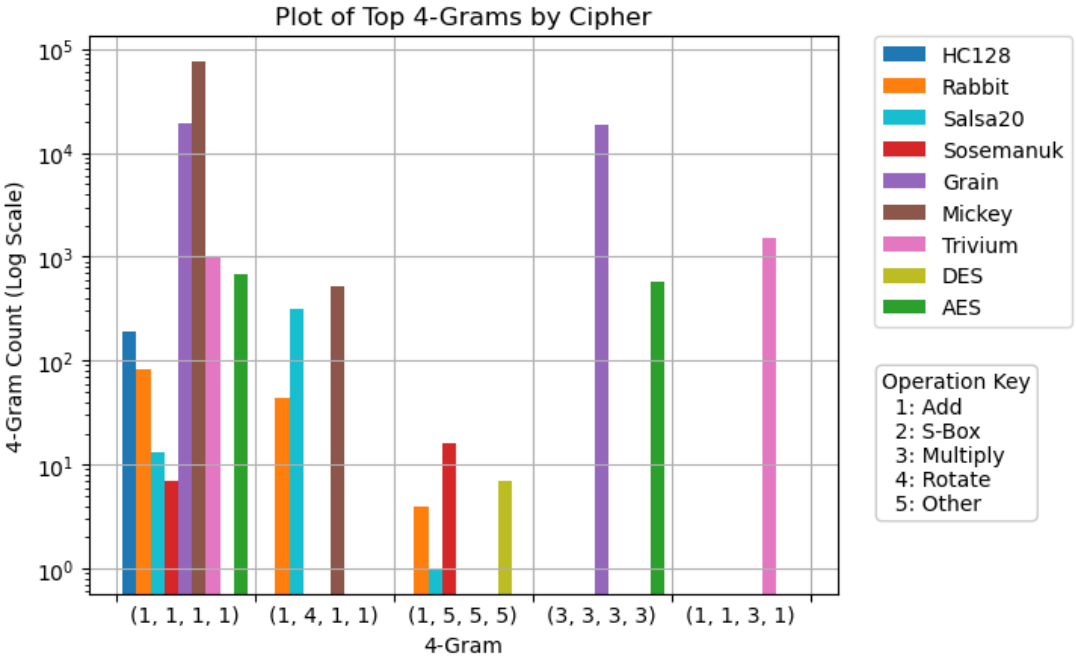

Figure A.2. 4-gram Comparison.

A.6 Python Code

Source code is available upon request.

For each of the examples in Chapter 1, including Tables 1.7, 1.8, and 1.9, we used the same key "0123456789012345" and a suitable nonce as required. We note that these examples require the use of the Cryptodome package.

To generate the tables in Chapter 4, some of the code will have to be modified to examine different aspects of the counts.

List of References

[1] *Announcing the Advanced Encryption Standard (AES)*, NIST FIPS-197, 2001. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/197/final

[2] K. H. Rosen, *Discrete Mathematics and its Applications*, 7th ed. New York, NY: McGraw-Hill, 2012.

[3] S. Roman, *Introduction to Coding and Information Theory*. New York, NY: Springer, 1992.

[4] W. Trappe and L. Washington, *Introduction to Cryptography with Coding Theory*, 2nd ed. New York, NY, USA: Pearson, 2005.

[5] J. B. Fraleigh, *A First Course in Abstract Algebra*, 7th ed. Addison Wesley, 2003.

[6] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[7] H. M. Heys, "A tutorial on linear and differential cryptanalysis," *Cryptologia*, vol. 26, no. 3, pp. 189–221, 2002.

[8] M. Matsui, "Linear cryptanalysis method for DES cipher," in *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, 1993, pp. 386–397.

[9] T. W. Cusick and P. Stănică, *Cryptographic Boolean Functions and Applications*. Academic Press, 2017.

[10] NIST-FIPS, "Data Encryption Standard (DES)," *Federal Information Processing Standards Publication*, pp. 46–3, 1999.

[11] J. Daemen, "Cipher and hash function design strategies based on linear and differential cryptanalysis," Ph.D. dissertation, Computer Security and Industrial Cryptography (COSIC), KU Leuven, Leuven, Belgium, 1995.

[12] H. Wu, "The stream cipher HC-128," in *ECRYPT eSTREAM II Proceedings*, 2008. [Online]. Available: https://www.ecrypt.eu.org/stream

[13] M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner, "The stream cipher Rabbit," in *ECRYPT eSTREAM II Proceedings*, 2008. [Online]. Available: https://www.ecrypt.eu.org/stream

[14] D. J. Bernstein, "Salsa20 specification," in *ECRYPT eSTREAM II Proceedings*, 2008. [Online]. Available: https://www.ecrypt.eu.org/stream

[15] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier *et al.*, "Sosemanuk, a fast software-oriented stream cipher," in *ECRYPT eSTREAM II Proceedings*, 2008. [Online]. Available: https://www.ecrypt.eu.org/stream

[16] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A proposal for the Advanced Encryption Standard," *NIST AES Proposal*, vol. 174, pp. 1–23, 1998.

[17] M. Hell, T. Johansson, and W. Meier, "Grain: A stream cipher for constrained environments," in *ECRYPT eSTREAM II Proceedings*, 2008. [Online]. Available: https://www.ecrypt.eu.org/stream

[18] S. Babbage and M. Dodd, "The stream cipher MICKEY 2.0," in *ECRYPT eSTREAM II Proceedings*, 2008. [Online]. Available: https://www.ecrypt.eu.org/stream

[19] C. De Canniere and B. Preneel, "Trivium," in *ECRYPT eSTREAM II Proceedings*, 2008. [Online]. Available: https://www.ecrypt.eu.org/stream

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California