University of Memphis
University of Memphis Digital Commons

Electronic Theses and Dissertations

7-21-2023

SAFE-NET: Secure and Fast Encryption using Network of Pseudo-Random Number Generators

Jonathan McCurdy

Follow this and additional works at: https://digitalcommons.memphis.edu/etd

Recommended Citation
McCurdy, Jonathan, "SAFE-NET: Secure and Fast Encryption using Network of Pseudo-Random Number Generators" (2023). *Electronic Theses and Dissertations*. 3017.
https://digitalcommons.memphis.edu/etd/3017

This Dissertation is brought to you for free and open access by University of Memphis Digital Commons. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of University of Memphis Digital Commons. For more information, please contact khggerty@memphis.edu.

SAFE-NET: SECURE AND FAST ENCRYPTION USING NETWORK OF PSEUDO-RANDOM NUMBER GENERATORS

by

Jonathan McCurdy

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Major: Applied Statistics

The University of Memphis

August 2023

Copyright© Jonathan McCurdy

All rights reserved

Acknowledgement

I would like to thank my advisor, Dr. Lih-Yuan Deng, for the guidance and mentorship that he has shown me over the years. It was through his classes that I developed a love for statistics and decided to pursue graduate studies. I would also like to thank Dr. Ching-Chi Yang and Dr. Nirman Kumar, whom I met with weekly, for their patience, insight, and support. Additionally, I would like to thank Dr. Dale Bowman, who has inspired me and many other students through her teaching and passion for statistics.

I would not have been able to complete this without the help and support from my friends and colleagues whom I met along the way. This includes Alexander, who accompanied me on the undergraduate journey and was always happy to partake in our philosophical talks. Andrew, who was always willing to help attack a problem and share his love of cute puppies. Hasan, Talha, and Kevin, who always let me vent over the smallest of things. Ben, Ngan, Tristan, and Clifford who were there for the first year struggles with me in the wild west. I would also like to thank all of the other teachers and colleagues who assisted me on my journey, as there are too many to list personally.

Finally, I would like to thank my family for always being there and encouraging me throughout this process. Mom and Dad for their love, support, and understandingness (now that the dissertation is done I'll need to find a new excuse to get out of yard work!). My siblings Matthew and Caitlin for their constant support and advice. And all of my extended family who motivated me along the way.

Abstract

Jonathan McCurdy, Ph.D. The University of Memphis, August 2023. SAFE-NET: Secure And Fast Encryption using Network of Pseudo-Random Number Generators. Major Professor: Dr. Lih-Yuan Deng.

We propose a general framework to design a general class of random number generators suitable for both computer simulation and computer security applications. It can include newly proposed generators SAFE (Secure And Fast Encryption) and ChaCha, a variant of Salsa, one of the four finalists of the eSTREAM ciphers. Two requirements for ciphers to be considered secure is that they must be unpredictable with a nice distributional property. Proposed SAFE-NET is a network of n nodes with external pseudo-random number generators as inputs nodes, several inner layers of nodes with a sequence of random variates through ARX (Addition, Rotation, XOR) transformations to diffuse the components of the initial state vector. After several rounds of transformations (with complex inner connections) are done, the output layer with n nodes are outputted via additional transformations. By utilizing random number generators with desirable empirical properties, SAFE-NET injects randomness into the keystream generation process and constantly updates the cipher's state with external pseudo-random numbers during each iteration. Through the integration of shuffle tables and advanced output functions, extra layers of security are provided, making it harder for attackers to exploit weaknesses in the cipher. Empirical results demonstrate that SAFE-NET requires fewer operations than ChaCha while still producing a sequence of uniformly distributed random numbers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Stream Ciphers

We send confidential information over the internet every day, whether it is our email communications, banking login, or school dashboard information. However, very rarely do we pause to contemplate the safety of our data and the protection mechanisms for this "secret" information. Secret messages have played an important role throughout history; as they have been used to convey vital information, making encryption and decryption a powerful tool in the hands of those who know how to use them. In ancient times, messages were often encrypted using a simple substitution cipher, where each symbol was replaced with a corresponding symbol. Over the years, more complex methods were developed, with one of the most well-known being the Enigma machine used by the Germans throughout World War II. The ability to send and receive secret messages has played a vital role in international relations. Now, in a world dominated by data, we constantly transmit large amounts of sensitive data across the internet. Given the prevalence of data, we must consider how we can keep this data secure as well as how companies protect our data.

Ciphers play a crucial role in the encryption and decryption process. The role of a cipher is to transform a plaintext message into an unrecognizable form, referred to as the ciphertext. To maintain the security of the transmitted message, only the recipient of the message should then know how to decrypt the ciphertext to obtain the original message using a key [73]. There are two main ways in which this process of encrypting and decrypting a message is completed– namely, using block ciphers and stream ciphers. Block ciphers carry out a transformation on blocks of the plain-text message, while stream ciphers carry out a transformation on individual bits of the plain-text message [63]. Our research focuses on the use of stream ciphers due to their connection to random number generators.

Modern stream ciphers were first used during World War I when Gilbert Vernam [67] proposed a process to combine the plaintext message and a single-use secret key using the "exclusive or" function (XOR) on individual bits. This single-use secret key is often referred to as a one-time pad. One-time pads are secure if the secret key is random, its length is as long as the plaintext message, and it is never reused [61]. While these properties are beneficial in obtaining a safe way to transmit messages, one significant issue that arises is the need for the sender to communicate the secret message, to create an individual key for each message, and distribute it to the receiver. This can potentially become problematic for large messages, as the secret key needs to be random and as long as the original message.

Using a pseudo-random bit generator is one way to resolve the issues associated with using one-time pads. This process requires the input of a smaller key, after which the generator carries out a procedure that produces a stream of bits that appear to be random. One advantage of this method is that the keystream is independent of the plaintext and ciphertext, allowing for it to be generated before encryption/decryption [59]. Furthermore, this method can be used for messages of any length. The general process of the stream cipher can be seen in Figure 1.1. There are various ways in which the keystream can be formed, and our research focuses on investigating and enhancing this specific process. Several different stream ciphers will be discussed in Chapter 2 and Chapter 3.

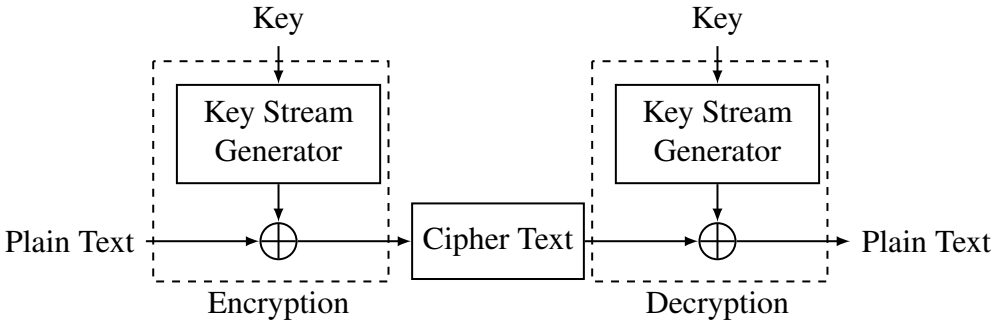

Figure 1.1: Diagram of a Stream Cipher

Once the keystream has been formed, the process of encryption and decryption is a relatively straight forward procedure. The keystream and the plaintext (or ciphertext) are combined using

the XOR transformation, where the XOR operator is often denoted as the symbol $\oplus$. The XOR function is utilized as it is a balanced transformation, meaning that; if our plaintext bit is a 0, then there is a 50% chance the ciphertext bit will be either 1 or 0. Additionally, the XOR operation is invertible allowing the same keystream to be used for both encryption and decryption purposes [54]. Figure 1.2 provides an example of a message being encrypted and decrypted with the XOR operation. In this example, we see that using the same key-stream results in the correct original message after the encryption and decryption process.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

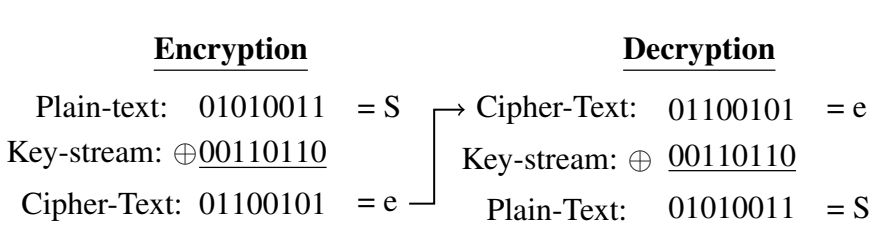

Figure 1.2: Table of XOR Process and Diagram of Encryption & Decryption Process

1.2 Random Number Generators

One of the requirements for a secret key to be secure is that it must be a random sequence of bits. Using a sequence of truly random numbers is not a feasible approach since the sequence, generated in non-deterministic ways, is not replicable. The replicability of the sequence is pivotal to this process so that both the sender and receiver can use the same keystream to encrypt and decrypt the message. With this in mind, Pseudo Random Number Generators (PRNGs) are often used to create the sequence of numbers, as they create a replicable sequence via a deterministic algorithm. This process requires initial starting values, called the "seed", which is then used to run the algorithm that produces a "random" sequence. Changing the starting value will result in a different sequence being outputted, and using the same starting value will result in the same sequence. This consistency is essential for the replicability of the encryption/decryption process.

The key-stream should appear to be a random sequence, and this can be achieved in a several ways. One of the simplest methods to produce a "random" key-stream is with the Linear-

Congruential Generator (LCG). This generator, proposed by Lehmer in the early 1950s [39], produces a sequence of random integers using a recursive process, taking the form:

$$X_i = (BX_{i-1} + A) \mod p, \quad i \geq 0 \tag{1.1}$$

where X_0 is the initial seed and B, A, and p are non-negative integers. The period length will be at most p when $A \neq 0$. Knuth [36] describes several conditions which must be met for the generator of this form to have a maximal period length. When $A = 0$, the maximum period length of $p-1$ is achieved when B and p are relatively prime. These generators are not recommended though, as one can easily compute the next value in the sequence when the modulus is known. Marsaglia [46] showed that consecutive points generated by an LCG will lie on a small number of parallel planes. These phenomena can be seen in Figure 1.3 below:

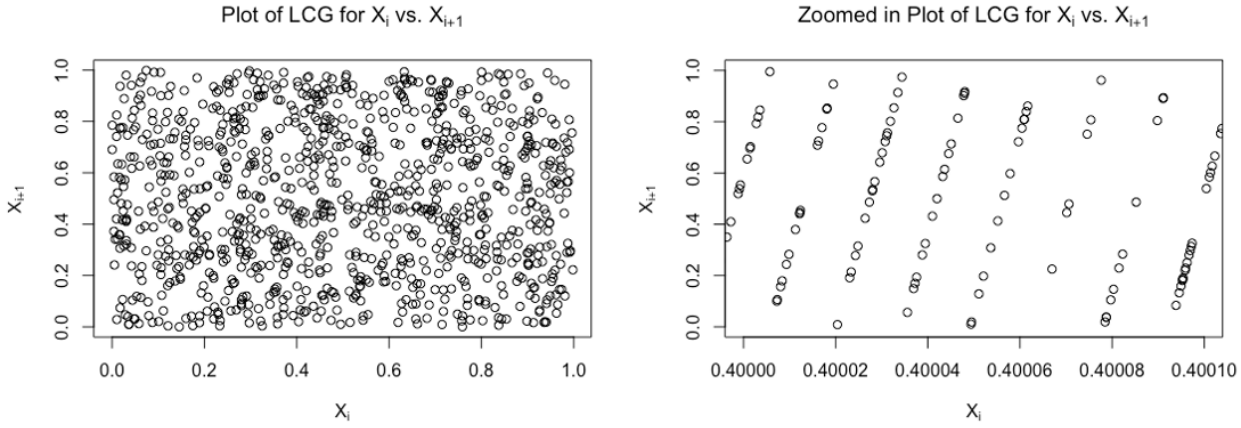

Figure 1.3: Successive points of an LCG lying on parallel planes

Because of this predictable linear pattern, Linear Congruential Generators have been shown to not be suitable for cryptographical purposes [14, 65]. It should be mentioned that there are two different kinds of Random Number Generators. The first is used for computer and simulation purposes when security is not required. These generators are fast, efficient, and have nice empirical properties such as a long period length and good distributional properties. The second are used for security applications. These are often slower generators that aim to mix or transform the values so that one cannot predict what the next value will be.

But, having a non-linear pattern is not enough for the generator to be cryptographically secure. Katz [32] expands on the requirements in that the generator must pass the "next-bit" test, which means that given the past k bits of the sequence, one can only predict the next bit with a success of 50%. Additional requirements and a discussion of common cryptographic attacks will be discussed in future sections.

The Multiple Recursive Generator (MRG) is a natural extension of the linear congruential generator (LCG), but instead of computing the next pseudo random number from the previously computed number, an MRG computes the next value based on a linear combination of the past k random numbers generated. The pseudo random numbers are generated sequentially from the k-th order linear recurrence:

$$X_i = (\alpha_1 X_{i-1} + \cdots + \alpha_k X_{i-k}) \mod p, \quad i \geq k, \tag{1.2}$$

where the multipliers $\alpha_1, \cdots, \alpha_k$ and initial seeds $X_0, \cdots, X_{k-1}$ are integers in $\mathbb{Z}_p$ but are not all zero, and the modulus p, usually a prime number, is a positive integer. It is well known that the maximum period of the MRG in (1.2) is $p^k - 1$, which is achieved if and only if its characteristic polynomial

$$f(x) = x^k - \alpha_1 x^{k-1} - \cdots - \alpha_k \tag{1.3}$$

is a primitive polynomial [36]. While these MRGs have better empirical performances than LCGs, they are still not secure generators due to the fact one can compute the next value by observing the previous values and solving a system of linear recurrence equations [52].

Both MRGs and LCGs have been shown to not be suitable for cryptographic purposes. But, they do have useful attributes in that they are efficient and have nice empirical properties regarding the generation of random numbers, even though they are predictable. Therefore, even though they are not cryptographically secure, they will be utilized in later sections for building and improving cryptographically secure generators.

Another way to generate the keystream that should be mentioned is using a Linear Feedback Shift Register (LFSR), which can be viewed as a Multiple Recursive Generator with modulus

2. These generators generally work by outputting the right-most bit of the current state into the keystream and then updating the current state of the generator by introducing a bit which is a linear combination of the previous state onto the left-most side and shifting the remaining bits to the right. Given n as the length of the generator's state, the period length will be $2^n - 1$ when the feedback polynomial is a primitive polynomial [19]. These LFSRs have played an important role in previous stream ciphers and random number generators. A diagram of the procedure can be seen in Figure 1.4, but it can be designed with any choice of bit combinations and does not necessarily have to be the one drawn.

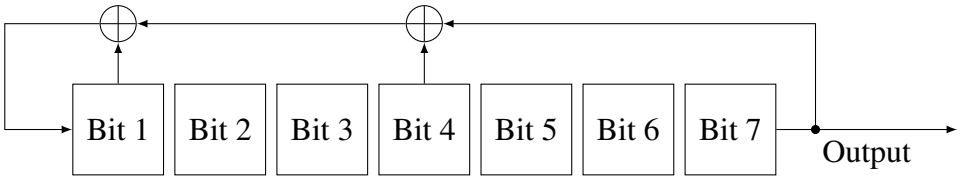

Figure 1.4: Diagram of a General Linear Feedback Shift Register

There are several different cryptographically secure generators available to use when creating a keystream. Some of the more popular ones in use include HC-256, ChaCha20, RC4, and Rabbit. Their general procedures and properties will be discussed in future sections. Most all cryptographically secure generators are susceptible to attackers, and thus precautions should be taken when using them to minimize the risk. We will propose a method to minimize some of the risks and increase the efficiency of the keystream generator.

In this dissertation, a stream cipher called SAFE-NET is proposed to be used for both cryptographic purposes as well as simulation purposes. Utilizing random number generators which have been shown to have nice empirical properties, but are not secure, we can populate a current state matrix and then transform it through mixing and shuffling. A major contribution of this model is the fact that it should require fewer iterations of mixing in order to be secure, resulting in a faster generator. Additionally, utilizing random number generators to populate the matrix increases security as well as provides theoretical support for the outputted keystream to be deemed random.

The main body of the thesis is organized as follows:

Chapter 2 will include an overview of the notation that will be used throughout the dissertation as well as provide a literature review of cryptographically secure ciphers and discuss their security aspects. This chapter will also provide some insight into the different cryptographic attacks that are used to "break" the cipher.

Chapter 3 will give an overview of the popular stream cipher "ChaCha". We will discuss the motivations behind the cipher, as well as how the stream cipher works. Additionally, we will discuss the empirical performance of the cipher, known security attacks on the cipher, and some potential weaknesses of "ChaCha".

Chapter 4 will introduce a general method for Secure And Fast Encryption using Network of Pseudo-Random Number Generators, which we will call SAFE-NET. It will include a discussion on the initialization of the cipher, the procedure carried out, and a few different output functions which could be used.

Chapter 5 covers some special cases of the SAFE-NET method proposed in Chapter 3. In this section, we will look at some extensions of ChaCha and show that they are special forms of the SAFE-NET method.

Chapter 6 will give an overview of the ideal properties of a sequence of random numbers and how we can test for these properties using the TESTU01 library. This chapter will also provide empirical evaluations for all of the methods discussed in the previous chapters.

Chapter 7 will provide a conclusion for the material presented and give insight into potential future work on the topics.

Chapter 2

Overview of Cryptographically Secure Ciphers and Common Attacks

2.1 Notation and Terminology

When it comes to stream ciphers and random number generators in general, some special terminology and notation is often used. A "user-key" is a secret-key used by the sender and the receiver during the encryption and decryption process to generate the keystream. For cryptographic purposes, a "nonce" is a number used once. This is done, often with random numbers, so the keystream generator does not produce the same sequence for different encrypted messages. Multiple messages can be sent under the same "user-key" as long as the "nonce" is changed for each message sent. Producing the same keystream sequence would open one up to replay attacks, thus allowing for the message to be more easily broken. More details will be presented on the nonce in later chapters.

Given x and y as two 32-bit integers, we can define some functions that transform their values. To begin with, $x\&y$ will correspond to the bit-wise "logical and". This means the resulting bit is only a 1 if the corresponding bits in both x and y are also 1's. So, given $a = 1001$ and $c = 1100$ then $a\&c = 1000$. Additionally, $x \oplus y$ corresponds to bit-side "exclusive or". This means a resulting bit is only a 1 if only one of the corresponding bits in both x and y are also 1's. Therefore, given $a = 1001$ and $c = 1100$ then $a \oplus c = 0101$. We can also define addition modulo 32 with $x \boxplus y$ corresponding to $(x + y) \bmod 2^{32}$. Additionally, we will let "||" be the concatenation function, allowing for a 32-bit integer to be broken up into smaller pieces.

We can also define $x \gg b$ to correspond to the logical right shift operation of b units with $x \ll b$ corresponding to the logical left shift operation of b units. For example, this means the bits are either shifted the right or left b bits with the right or left b most bits being "dropped off" and b 0's being added on the left or right side respectively. For instance, if $a = 10010101$ and $b = 3$

then $a \gg b = 00010010$. With these functions in place, we can define the transformation $x \lll b$ corresponding to the rotation of b bits. This can be defined as $(x \ll b) \oplus (x \gg (32-b))$. So, if $a = 10010101$ and $c = 3$, then $a \ll c = 10101000$, and $a \gg (8-c) = 00000100$, so $10101000 \oplus 00000100 = 10101100$. Finally, we will say $\mathbf{1}_b$ corresponds to the number $2^b - 1$ which has the binary representation of b 1's,

2.2 Cryptographic attacks

The goal of a stream cipher is to relay sensitive information so that others cannot identify what the information actually is. However, no stream cipher will be completely immune to attackers trying to identify information about the plaintext, user-key, or cipher. There are a number of different cryptographic attacks that have been used over the years, but we will just discuss a few below.

The most basic example of this process would be a brute-force attack, which is also known as an exhaustive search attack. In this attack, every possible key and IV is checked to see if the correct plaintext is generated. This becomes more time consuming and infeasible with larger and larger keys and IVs, as the number of possibilities would grow exponentially. For instance, RC4 accepts a key length of up to 2048-bits, which means a brute-force attack would have a complexity of 2^{2048}, meaning up to 2^{2048} keys would need to be tested in order to find the correct one. A cipher should be considered secure if this method of attack is the least complex computationally, assuming it is sufficiently large [54].

Another possible attack is the related-key attack [11]. This attack requires the attacker to have knowledge about the relationship between the keys and the cipher. If messages are sent under different keys, but portions of the keys remain the same, then it may be possible to identify patterns in the keystream related to the unchanged portions of the key. This might be exploited if the cipher is run with large pieces of the key unchanged.

In addition to attacks on the key, attacks can also be carried out on the keystream. One such

attack is the distinguishing attack [37], which aims to distinguish the keystream from a truly random sequence. If the keystream is distinguishable, then information may be gathered about the plaintext. For instance, there are documented attacks based on the fact that the second output bit for the RC4 cipher is biased towards 0 [45]. This means that the second bit has a higher likelihood of being a 0 when the bit should have an equal probability of being a 0 or 1. These sorts of biases in the keystream could allow attackers to obtain information about the cipher. Because of this reason, it is important to have a keystream generator that acts as a good random number generator.

One of the most powerful and successful attacks is the differential attack, first published by Biham [12] but known by others at the time. The idea of the attack tries to identify and exploit differences in plaintext messages and their ciphertexts. This is done by encrypting a number of messages under the same key with only slight differences to the plaintext and seeing how the ciphertext is changed. No information about the key is known though, so by studying enough pairs of messages that are slightly different from each other, one could figure out the key and algorithm for the cipher.

Finally, the last cryptographic attack which will be mentioned are algebraic attacks. This attack relies on the ability to solve a system of algebraic equations modeled on the cipher's operations with the key and initial states as variables [4]. This process can be very difficult if the cipher is designed well though, as a system of equations may be hard to find. Wong, Carter, and Dawson [70] showed that while it is possible to construct a system of linear equations describing the operations of RC4, solving the system of equations is infeasible with current methods. But, the resistance to algebraic attacks is weakened if one of the operations is compromised. Using nonlinear outputs and feedback functions reduce the susceptibility to algebraic attacks.

2.3 RC4 Stream Cipher

The Rivest Cipher (RC4), developed in 1987 and published in 1994, is a once widely used stream cipher that was appreciated for its speed and simplicity. It is an output feedback algorithm that updates itself when the keystream is outputted. The procedure, outlined in Schneier [60], is relatively straightforward.

Given a key in a 256-byte array $(K_0, K_1, \ldots, K_{255})$, the S-table (substitution-table) is initialized during the Key-Schedule Algorithm (KSA) through a rearrangement of the S table. Then a random byte is outputted for the keystream during the Pseudo Random Generation Algorithm (PRGA) phase, which consists of incrementing the index i, calculating the index j, shuffling the table, and outputting a byte based on the i^{th} and j^{th} index. The PRGA phase is then repeated until the keystream is sufficiently long enough to encrypt/decrypt the plaintext. A general procedure can be seen in the diagram below:

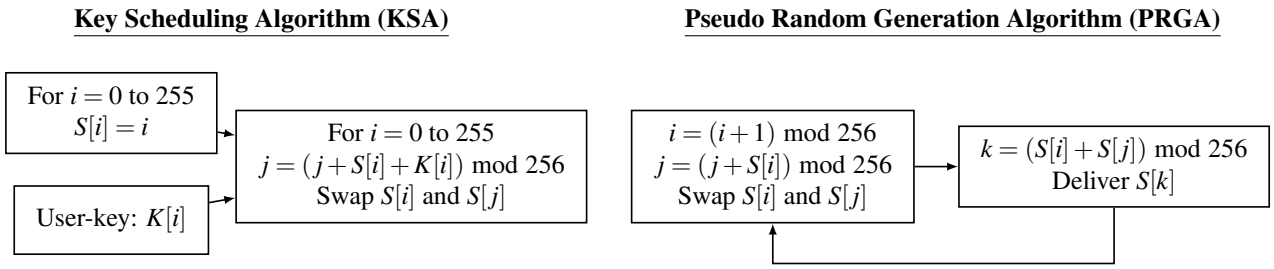

Figure 2.1: Diagram of RC4 Stream Cipher

RC4 outputs a single byte, with the S-table consisting of all the permutations of all 2^8 (256) possible 8-bit words. This set-up could be expanded to both a larger S-table and larger word sizes, but modifications would need to be made due to memory requirements and the large initialization time for the Key Scheduling Algorithm to thoroughly mix the S-table. For instance, a 32-bit iteration of this process would require a table of size 2^{32} and the KSA to perform 2^{32} swaps just to initialize the cipher, which is impractical.

While the algorithm is very fast and was once considered to be secure, weaknesses have been found which make it unsuitable for cryptographic use. For instance, the first portion of the out-

putted keystream has been shown to be biased and thus needs to be discarded before encryption and decryption purposes [32]. Additional attacks on the cipher that have been documented are related to the correlation between the key and the output values, rendering the cipher unusable [27]. An overview of additional attacks are provided in the following articles [31, 69, 72].

Many attempts have been made to remedy these deficiencies. One such method, proposed by Maitra and Paul [44], called RC4+ adds additional layers to the KSA and PRGA to avoid correlation between the user-key and the output byte. They later proposed another extension of RC4, called Quad-RC4 [56], which produces a 32-bit word output using a combination of 4 RC4 states. Additional attempts to expand RC4 to a 32-bit cipher have been made, with Nawaz, Gupta, and Gong [51] proposing RC4(n, m). Their method allows the table size (2^n) and the word size (m-bits) to be changed, with an adjustment being made to the PRGA to include an integer addition variable. This variant is not guaranteed to have nice distributional properties though, as only a few m-bit words are represented in the table and able to be outputted into the keystream.

In all of these extensions, the user-key and thus the current state of the cipher determine the selection of the index values. Additionally, the randomness of the keystream relies on the expectation that the cipher updates itself in a random manner. This can be problematic if the key or starting state is not thoroughly random. Deng et. al [23] proposed eRC4, which utilizes an external random number generator to assist with index selection and table initialization. They also introduced a procedure for selection from multiple mixing output functions to increase the complexity of the cipher and be more resistant to attacks. This incorporates empirically sound random number generators to inject randomness into the cipher.

2.4 HC-256 Stream Cipher

Another prevalent stream cipher is HC-256 [71], which was proposed by Wu in 2004. It accepts a 256-bit key and a 256-bit nonce, where a shorter key and nonce can be repeated to reach the required length. A variant of it, HC-128, was submitted to eStream and selected as a finalist [58].

The cipher relies on two secret tables, which we will call P and Q, which are initialized using a combination of the key and IVs. Given 8 32-bit words for both the key and IV, the initialization is done by populating an array W as:

$$W_i = \begin{cases} K_i, & 0 \leq i \leq 7 \\ IV_{i-8}, & 8 \leq i \leq 15 \\ f_2(W_{i-2}) + W_{i-7} + f_1(W_{i-15}) + W_{i-16} + i, & 16 \leq i \leq 2559 \end{cases}$$

where $f_1(\cdot)$ and $f_2(\cdot)$, which consist of XOR and rotational transformations, are utilized. After W is populated, then the P and Q table are formed as $P[i] = W_{i+512}$ for $0 \leq i \leq 1023$ and $Q[i] = W_{i+1536}$ for $0 \leq i \leq 1023$, respectively.

The incremented index i determines which table will be updated. A series of functions updates the tables based on the index j using XOR and rotational transformations. The outputted value is computed using a function which adds multiple values in the respective table together, with the indices of the selected values based on the value of the word at the ith or jth index. The process of incrementing the index, selecting and updating the table, and outputting the value is repeated until the key stream is the desired length. Further details about the cipher and the function definitions can be found in the referenced literature [71, 25]. A diagram of the process can be seen below.

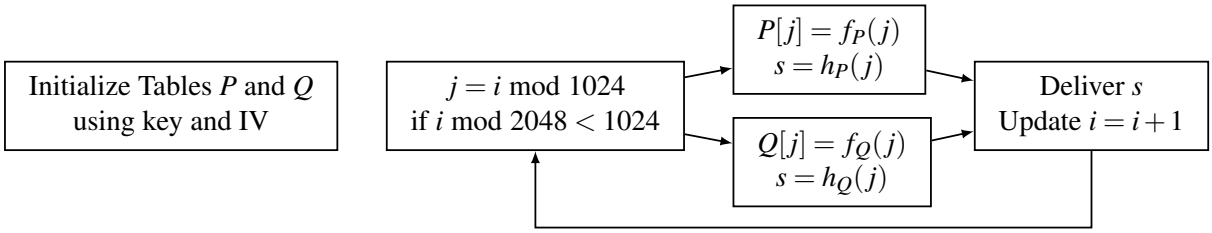

Figure 2.2: Diagram of HC-256 Stream Cipher

It should be noted that a burn-in period is required before generating an output. The author recommends 4096 steps before starting the keystream to allow the tables to be thoroughly mixed. This results in the cipher having a long initialization time as well as very complex functions com-

pared to other stream ciphers. The initialization of the P and Q tables also pose a weakness in that there is no guarantee the values will be uniform or random after transforming the key and IV. Additionally, much like RC4, HC-256 relies on increasing the index sequentially and does not utilize a random number generator to better select the indices. Including a random number generator with good empirical properties to fill in the tables and make index selections could reduce the initialization time and increase the security of the cipher.

Limited analysis has been published about this cipher. Sekar and Preneel demonstrated a distinguishing attack on HC-256 that required $2^{276.8}$ equations. Moreover, differential fault attacks were able to recover the internal state of the cipher with 7968 faults [35], implying if an attacker could induce a random error in the state tables 7968 times, they may gain information about the cipher while observing the outputs. A side-channel attack was also proposed as a feasible method under specific conditions and assumptions about the cipher [55], much like a differential fault attack.

2.5 Rabbit Stream Cipher

Another stream cipher that was featured in eStream due to its security and efficiency was Rabbit [13]. It was inspired by real-valued chaotic maps, meaning small changes cause different maps to exhibit random and unpredictable behaviors. Rabbit takes in a 128-bit key, which helps initialize the state variables $x_{j,i}$ and the counter variables $c_{j,i}$ through a combination and non-linear transformation of the key.

The state of the function is then updated using a non-linear function, dependent on the previous state and counter variable. After that, the counter variables are updated and four 32-bit words are output using an "exclusive or" function between pieces of $x_{j,i}$ words. Additional details about the specification of the functions can be found in the original proposal by the authors [13]. A simplified diagram is provided below, with j typically ranging from 0 to 7:

The design of the stream cipher is fast. A distinguishing attack was recorded by Aumasson

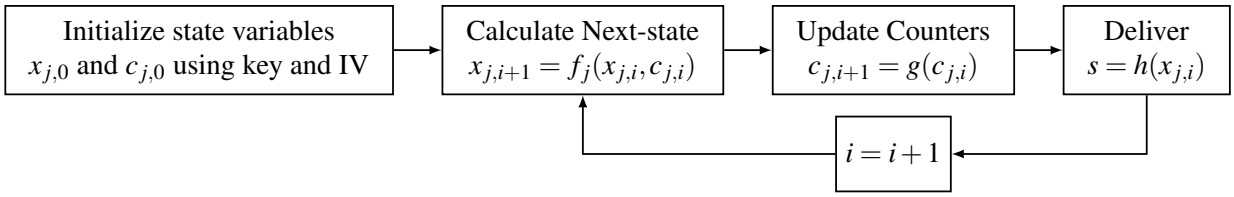

Figure 2.3: Diagram of Rabbit Stream Cipher

[5] with complexity 2^{247}, but Lu [40] was able to reduce the complexity of the attack to 2^{158}. Differential fault analysis was also performed on Rabbit, which revealed that with between 128-256 faults, one can recover the complete internal state in 2^{38} steps [34].

One downside of the cipher is the extreme complexity of its design. Multiple functions are provided for calculating the next state of the cipher, all dependent on the j^{th} index. They all rely on the non-linear transformation, which incorporates the square of the current state and counter with additional transformations taking place. The number of functions and the complexity of the functions could lead to issues on some systems.

2.6 Secure and Fast Encryption (SAFE)

While all of the previous ciphers discussed have nice properties, they do not utilize external generators to infuse randomness throughout the cipher. Deng et. al [22] introduced a class of generators known as Secure and Fast Encryption (SAFE) which uses two baseline generators throughout the generating process.

The key and IV are used to populate the internal states of the two generators, with the two tables populated using either the key or generators. Internal states are then computed by having each random number generator produce a random variate. These values are then used to update the tables and the select values are then used to determine the outputted word. The output function consists of a non-linear mixing function utilizing a combination of the values selected from the table and the values produced by the baseline generators. A general diagram of the process can be seen below:

The generator is very efficient, as minimal operations are carried out in the generating pro-

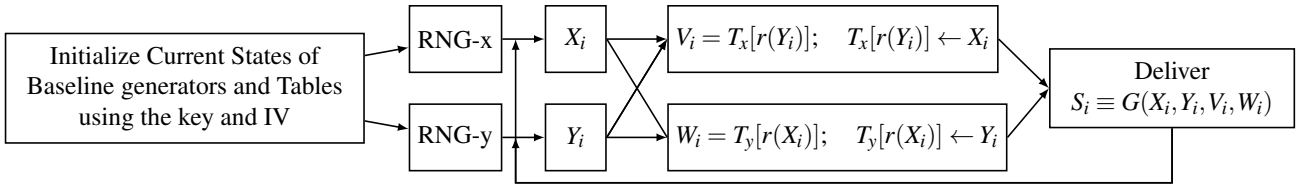

Figure 2.4: Diagram of SAFE Stream Cipher

cess. While there has been no in-depth cryptoanalysis carried out on the generator, Deng et. al [22] argue that it is as secure (if not more) than HC-256 due to the similarity of the ciphers. One thing that contributes to the security of the generator is having multiple layers of defense; such as potentially using large-order MRGs for the generator to make it difficult to identify the next internal value, shuffling the tables, and using a complex mixing function to hide the internal values when outputting the keystream.

An additional stream cipher that was presented to eStream is Salsa20 [8]. More in-depth details on this cipher will be given in future chapters due to its significant impact in motivating our stream cipher SAFE-NET.

Chapter 3

Overview of ChaCha

3.1 Overview of Salsa20 & ChaCha

ChaCha is a 256-bit stream cipher designed by Bernstein [10] for security applications. It came about as a variant of the Salsa20 stream cipher, which was also proposed by Bernstein [8]. Salsa20 is a popular stream cipher that was selected as one of the four finalists for the eSTREAM Project; which sought efficient stream ciphers that could be adopted in a wide variety of applications, including security [58]. The ciphers operate using ARX transformations (Addition, Rotation, XOR) to thoroughly mix their current state, with the only difference between successive starting state matrices occurring in the counter position, which is increased by 1. This leads to questions about the distributional properties of the outputted sequence though, as there is no theoretical support that an increase of 1 in the counter block results in uniform and independent outputs for the keystream. Despite these concerns, ChaCha is widely used due to its light-weight nature and due to no known cryptographic attacks when the number of rounds is large.

Both Salsa20 and ChaCha have the same general structure, with the main differences coming in the implementation of the quarter-round function, as well as the set-up of the initial matrices. These small changes allow for ChaCha to have a better diffusion of bits due to the new quarter-round definition, and due to each block having a chance to influence every other block [10].

For our purposes, diffusion is defined as the number of bits that are different between two transformed words when the starting words only differ by 1 bit. For instance, if the starting values are $a = 10101010$ and $b = 10101011$, and the transformed values are $a_t = 01101010$ and $b_t = 11101001$, then the diffusion rate would be 3 since a_t and b_t have only 3 bits which are different from each other. Having a better diffusion of bits is important, as with only a difference of 1 between successive matrices, we want the bits in successive matrices to become more different from each other quickly. Bernstein notes that Salsa20 has a diffusion rate of roughly 8 bits per

quarter-round while ChaCha has a diffusion rate of 12.5 bits per quarter-round operation [10].

3.2 General Procedure

Salsa20 and ChaCha both utilize a vector of sixteen 32-bit words, which after undergoing a transformation, outputs a sequence of sixteen 32-bit words called a keystream. The vector of sixteen 32-bit words is composed of eight words from a 256-bit user-key ($\mathbf{k} = (k_0, k_1, \cdots, k_7)$), two words from a 64-bit nonce ($\mathbf{v} = (v_0, v_1)$), four words from a 128-bit constant ($\mathbf{c} = (c_0, c_1, c_2, c_3)$), and two 32-bit words that act as counters ($\mathbf{t} = (t_0, t_1)$). The counter t_0 is increased by 1 every time the 16-word keystream is outputted, and the counter t_1 is increased by 1 only when t_0 reaches 2^{32} and then returns to 0. The sixteen-word vector is often viewed as a 4×4 matrix for visual simplicity. Salsa20 and ChaCha are set up in the following ways, with the initial state matrix of Salsa20 denoted by $\mathbf{X}_{\mathrm{S}}^{(0)}$ and the initial state matrix of ChaCha denoted by $\mathbf{X}_{\mathrm{C}}^{(0)}$:

$$\mathbf{X}_{\mathrm{S}}^{(0)} = \begin{pmatrix} c_0 & k_0 & k_1 & k_2 \\ k_3 & c_1 & v_0 & v_1 \\ t_0 & t_1 & c_2 & k_4 \\ k_5 & k_6 & k_7 & c_3 \end{pmatrix} \qquad \mathbf{X}_{\mathrm{C}}^{(0)} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & v_0 & v_1 \end{pmatrix} \tag{3.1}$$

Both ciphers work on a series of ARX operations, which we will call a quarter-round, in order to diffuse the bits. The quarter-round function is a process that takes 4 words as inputs and then transforms them. In addition to the matrix setup being different between the two ciphers, these quarter-round functions are also slightly different. The Quarter-Round function, $\mathbf{Q}(a, b, c, d)$, is defined in the following ways for both Salsa20 and ChaCha:

Salsa20 Quarter-Round	**ChaCha Quarter-Round**
$b = b \oplus ((a+d) \lll 7)$	$a = a+b; \quad d = d \oplus a; \quad d = d \lll 16$
$d = c \oplus ((b+a) \lll 9)$	$c = c+d; \quad b = b \oplus c; \quad b = b \lll 12$
$d = d \oplus ((c+b) \lll 13)$	$a = a+b; \quad d = d \oplus a; \quad d = d \lll 8$
$a = a \oplus ((d+c) \lll 18)$	$c = c+d; \quad b = b \oplus c; \quad b = b \lll 7$

It should be noted that the Quarter-Round process gets its name as it is run four times in order for 1 Round to be completed. Both Salsa20 and ChaCha partition the 4×4 matrix into four vectors of four words each. Each vector is then inputted into the Quarter-Round function, with four Quarter-Round transformations resulting in 1 Round being completed.

The Quarter-Round transformation is traditionally done for 8, 12, or 20 Rounds depending on how efficient and secure the user wishes the cipher to be. The higher the number of rounds selected, the slower the cipher will be, but it will be a more secure cipher. It should also be noted that there is nothing special about the specific functions/shifts being done on the partitions, as Sobti [64] shows different Quarter-Round functions exist which have a better diffusion of bits. The partitions for both ciphers both are done in a systematic way, with Salsa20 choosing column or row partitions and ChaCha choosing column or diagonal partitions. Given that the current state of the matrix is

$$\mathbf{X} = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} \tag{3.3}$$

the partitions are defined as $P(\mathbf{X}) = (\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$, with $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$, and $\mathbf{x}_3$ being determined depending on if the stream cipher is currently on an odd or even round. A Full Round consists of

four quarter-round transformations of $\mathbf{Q}(\mathbf{x}_0)$, $\mathbf{Q}(\mathbf{x}_1)$, $\mathbf{Q}(\mathbf{x}_2)$, $\mathbf{Q}(\mathbf{x}_3)$. The Quarter-Round partitions are defined differently for even and odd-numbered rounds. Figure 3.1 shows the ways that they are defined.

Salsa20: Odd Rounds

$$\begin{aligned}\mathbf{x}_0 &= (x_0, x_4, x_8, x_{12})\\ \mathbf{x}_1 &= (x_5, x_9, x_{13}, x_1)\\ \mathbf{x}_2 &= (x_{10}, x_{14}, x_2, x_6)\\ \mathbf{x}_3 &= (x_{15}, x_3, x_7, x_{11})\end{aligned}$$

Salsa20: Even Rounds

$$\begin{aligned}\mathbf{x}_0 &= (x_0, x_1, x_2, x_3)\\ \mathbf{x}_1 &= (x_5, x_6, x_7, x_4)\\ \mathbf{x}_2 &= (x_{10}, x_{11}, x_8, x_9)\\ \mathbf{x}_3 &= (x_{15}, x_{12}, x_{13}, x_{14})\end{aligned}$$

ChaCha: Odd Rounds

$$\begin{aligned}\mathbf{x}_0 &= (x_0, x_4, x_8, x_{12})\\ \mathbf{x}_1 &= (x_5, x_9, x_{13}, x_1)\\ \mathbf{x}_2 &= (x_{10}, x_{14}, x_2, x_6)\\ \mathbf{x}_3 &= (x_{15}, x_3, x_7, x_{11})\end{aligned}$$

ChaCha: Even Rounds

$$\begin{aligned}\mathbf{x}_0 &= (x_0, x_5, x_{10}, x_{15})\\ \mathbf{x}_1 &= (x_1, x_6, x_{11}, x_{12})\\ \mathbf{x}_2 &= (x_2, x_7, x_8, x_{13})\\ \mathbf{x}_3 &= (x_3, x_4, x_9, x_{14})\end{aligned}$$

Figure 3.1: Salsa20 and ChaCha Quarter-Round Partitions

After completing R full rounds, the resulting matrix is then added to the initial state matrix and, the sixteen 32-bit words are outputted as the key-stream. Then the initial state matrix is re-set and the counters are increased in their respective ways. The process is then repeated all over again until the stream cipher's key-stream output reaches the desired length. Below is a list of steps and a diagram of the overall procedure of both Salsa20 and ChaCha:

Step 0. Create an initial state matrix $\mathbf{X}^{(0)}$ with running index i of the iteration. Counter (t_0, t_1) can be set as $i = i_0$.

Step 1. At the i^{th} stage, initialize state matrix $\mathbf{X}^{(0)}$ with counter $t_0 = i$, and $t_1 = t_1 + c$

Step 2. Computer R-round transformation as $\mathbf{X}^{(R)} = \mathbf{H}(\mathbf{X}^{(0)}; R)$

Step 3. Output sixteen 32-bit words as $\mathbf{Z} = \mathbf{X}^{(0)} \boxplus \mathbf{X}^{(R)}$ and then the increase the index to $i = (i+1)\&\mathbf{1}_{32}$ while also setting the carry bit to 1 when $i = 0$. We then repeat steps 1-3

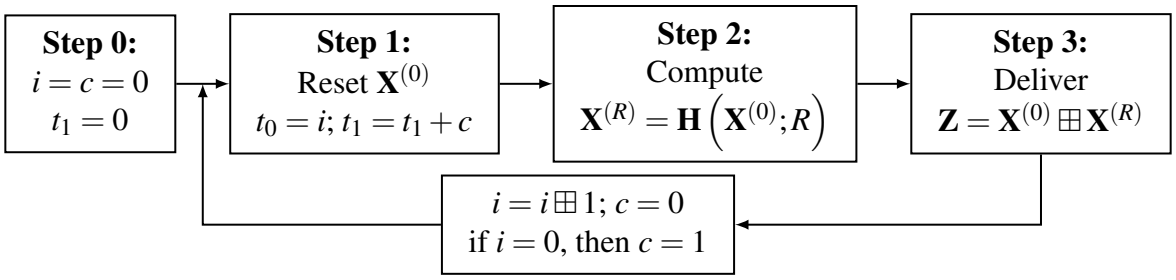

Figure 3.2: General Procedure for ChaCha Stream Cipher

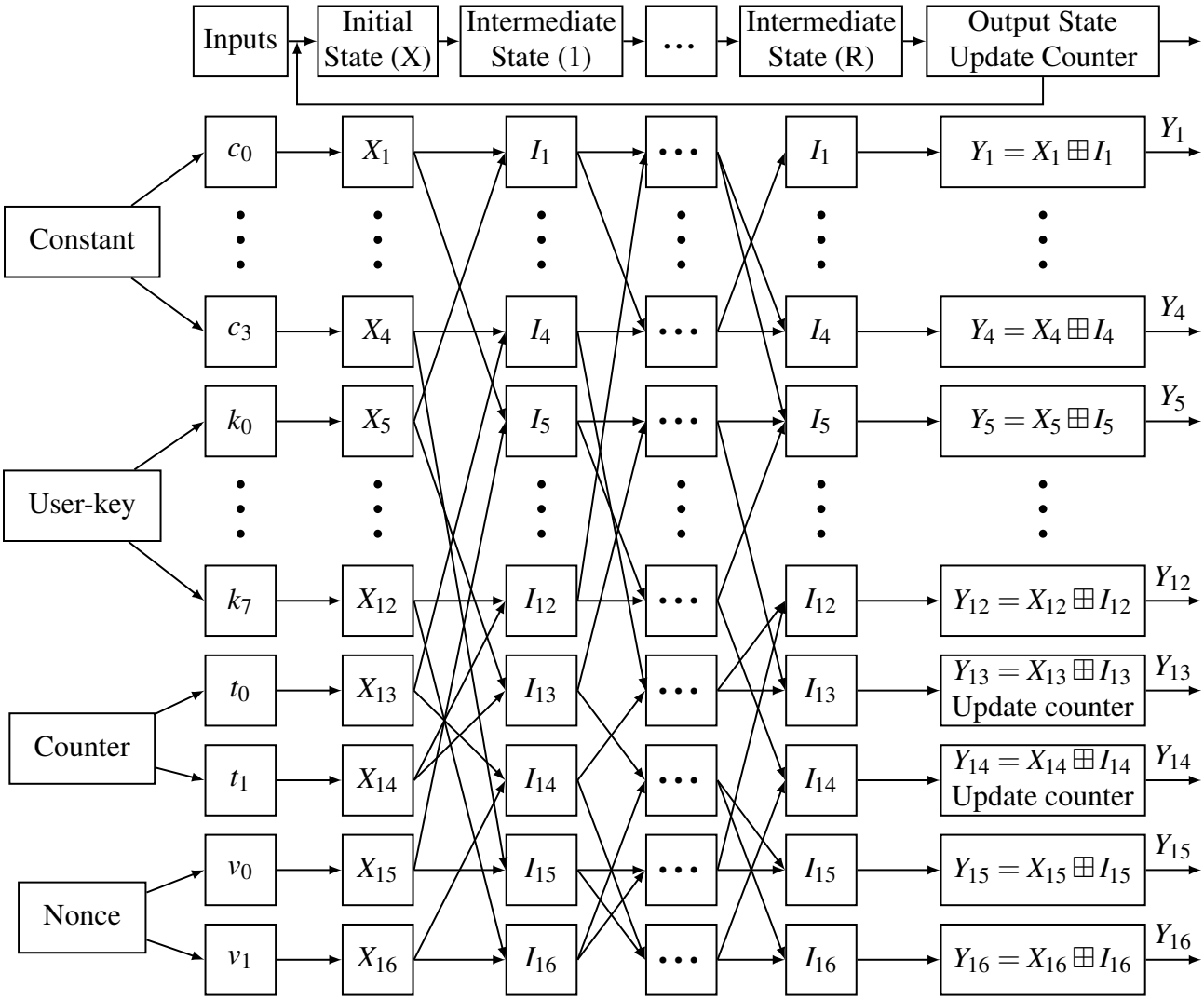

Figure 3.3: Diagram of ChaCha Stream Cipher

3.3 Cryptanalysis

In order for both Salsa20 and ChaCha to avoid potential attacks, one should choose a large enough number of rounds in order to have confidence in the security of the cipher. While a higher num-

ber of rounds, R, will result in the cipher being slower, it will increase the confidence that the cipher will be secure.

One of the first cryptographic attacks to be recorded on Salsa20 was demonstrated by Crowley, who showed that a 2^{165} operation attack using truncated differentials could break parts of the cipher for Salsa20/5, where 5 is the number of rounds [18]. Fisher showed that the key could be recovered in Salsa20/6 with roughly 2^{177} operations and that Salsa20/7 may be open to key-attacks of roughly 2^{217} operations [26]. Additional attacks on Salsa20/8 have been identified [66] to break the starting state matrix. While there is no set number of rounds that will result in Salsa20 being considered completely safe, 12 rounds is normally the traditionally agreed upon number to balance security and efficiency [15].

ChaCha is also susceptible to attacks when the number of rounds is small. Aumasson introduced a method of differential attack called Probabilistic Neutral Bits (PNB) which tries to replace a subset of key bits with fixed bits [6]. This allowed them to identify attacks on ChaCha6 and ChaCha7. Shi expanded on the idea of PNB by introducing a Chaining Distinguishers which breaks ChaCha6 and ChaCha7 in fewer operations [62]. Attacks on Salsa20/9 and ChaCha8 have also been recorded [30]. Additional attacks have been shown on Salsa20 and ChaCha with varying degrees of complexity [16, 24]. There are no proofs that ensure the safety of the cipher after a certain number of rounds, but 12 rounds is traditionally accepted as secure.

One of the properties of Salsa20 and Chacha which opens them up to attacks with differential cryptanalysis is the fact that initial state matrices are virtually identical aside from the counter function being increased by 1. This makes it easier for attackers to “work backward” and identify pieces of the key. Increasing the number of rounds being run will help transform the matrix even more, thus protecting it against this type of attack, but this results in a slower cipher. Additionally, the increased transformations do not guarantee an independent and uniform keystream.

3.4 Weaknesses of the ChaCha cipher

Based on the ChaCha's construction, it appears that there are a few potential weaknesses with the stream cipher:

Poor Distributional Properties: ChaCha only differs by 1 within the counter blocks for successive pages. It is not intuitive and there is no theoretical support that this small change will result in outputted blocks being independent of each other even when the number of rounds is large.

Majority of blocks are unchanged In the ChaCha cipher, the vast majority of blocks are stationary and do not change, both within a single message as well as across different messages (as only the nonce changes between different messages). This stationary property opens the cipher up to attacks, as it is easier to identify the contents of the blocks and thus the user-key and nonce if the blocks are always the same.

Limited Key/IV size The key and nonce are relatively small with ChaCha, with the key being eight 32-bit words and the nonce being two 32-bit words. This results in fewer messages being able to be sent under the same key.

Simple Output Function The output function for ChaCha is 32-bit addition with the initial matrix and the transformed matrix. This might open up the user to attacks as well, as the cipher just uses addition between the unaltered starting block and the transformed block, resulting in it being easier to determine what the starting matrix is as well as what the other blocks are.

Relatively small Period Length The period length of the ChaCha stream cipher is 2^{64} since the counters are two 32-bit words. While this will be long enough for most general purposes, it can be viewed as a negative when other random number generators have much longer periods.

Previous methods have been proposed that attempt to improve upon ChaCha by addressing some of the weaknesses listed above. These will be discussed later in the paper as they are all special cases of SAFE-NET (Secure and Fast Encryption Network of RNGs), a method which we propose.

Chapter 4

SAFE-NET

4.1 Secure and Fast Encryption using Network of RNGs

As has been mentioned previously, ChaCha has a few apparent weaknesses due to the two successive initial state matrices being virtually identical to each other; with the only difference occurring within the counter blocks. This potentially opens one up to several attacks, such as a related-key attack, as a majority of the blocks are stagnant and unchanged. Secondly, there is no theoretical justification for successive pages to be independent of each other when there is just a one-bit difference between them. Finally, since there is a relatively simple output function, attackers are more likely to be able to determine pieces of the starting and intermediate states, potentially allowing them to crack portions of the cipher.

Because of this, a method called Secure And Fast Encryption using Network of Pseudo-Random Number Generators (SAFE-NET) is introduced, with the general idea of populating each block with a baseline random number generator as well as implementing a "shuffle" table and more complex output functions. This method takes advantage of the nice theoretical/empirical properties of the baseline random numbers generator while allowing the quarter-round and output functions to break the linearity of the generators while still maintaining nice empirical properties.

4.2 Design of SAFE-NET

The general method of SAFE-NET uses baseline RNGs to populate the initial state matrices to ensure that the blocks are always populated with new numbers. This property, along with a new initialization procedure to allow for a larger nonce/user-key and a different output function to further hide the starting states, should allow for a more secure generator with fewer rounds. Along with the diagram below, the general method can be stated as:

Step 0. Initialize n RNGs with the user keys and IV using an initialization procedure like the one described below and populate the n shuffle Tables $\mathbf{T}$ using the RNGs already initialized.

Step 1. Generate Internal Values $\mathbf{X}$ using the RNGs.

Step 2. Compute the Initial States $\mathbf{Y}^{(0)}$ as $Y_i = T_i[r(X_{i+1})]$ and then update $T_i[r(X_{i+1})] \leftarrow X_i$

Step 3. Computer R-round transformation as $\mathbf{Y}^{(R)} = \mathbf{H}(\mathbf{Y}^{(0)}; R)$

Step 4. Output n 32-bit words as $\mathbf{Z} = f(\mathbf{Y}^{(0)}, \mathbf{Y}^{(R)})$. Then repeat steps 1-5 as needed.

For simplicity, we will refer to the initial state matrix $\mathbf{Y}^{(0)}$ as $\mathbf{Y}$ with the i^{th} block being Y_i. Additionally, I_i will refer to the i^{th} block of the intermediate matrix after R rounds of transformation have been carried out, and T_i will refer to the i^{th} Table.

Several different cases that will be examined later on in this paper, mainly focusing on using different numbers of RNGs to populate the blocks as well as methods that can be considered special cases of SAFE-NET. It should also be mentioned that any number of blocks/words could be used to do the mixing, and it is not just limited to the standard 16 words that ChaCha utilizes. We do propose a way to initialize the random number generators using a mixing of the keys and nonces as well as an updated output function which adds an extra layer of security to the cipher.

4.3 Initialization

When it comes to the Salsa and ChaCha stream ciphers, one of the limitations is that it only allows for two 32-bit nonces and eight 32-bit user-keys. This has some drawbacks as it limits the number of messages which can be sent under the same user-key, meaning larger nonces are more desirable. Extensions, such as XSalsa and XChaCha [9, 3], have been proposed which allow for the nonce to be six 32-bit words instead of just two 32-bit words while maintaining all the other characteristics of Salsa and ChaCha. Additionally, the Salsa and ChaCha stream cipher has the key and nonce as unchanging blocks in the initial state matrix. Fischer notes that it is important to thoroughly mix the key so that no portions of it have any "traceable influence on the initial state"

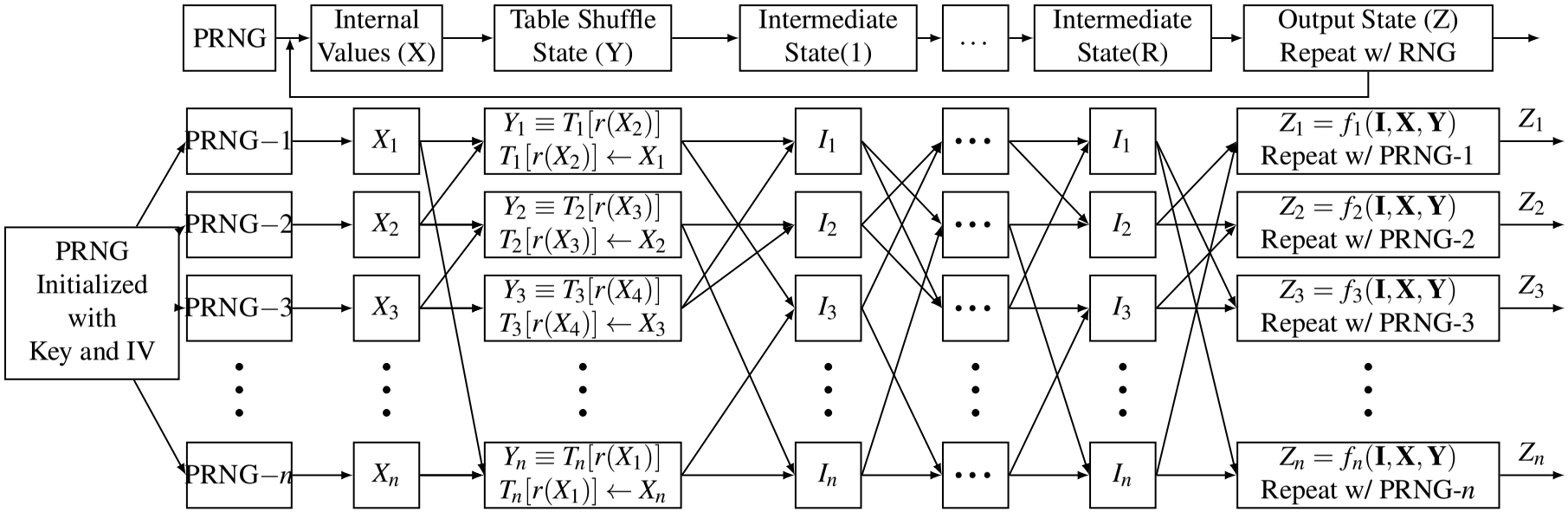

Figure 4.1: General Procedure for SAFE-NET

[26]. Therefore, more should be done to hide the key/IV to add a layer of security, so we propose a new general method for initializing the key/IV outside of the initial state matrix.

With the SAFE-NET family of ciphers, there is no limitation on the size of the nonce or user-key, as the nonces and user-keys will be mixed and then used to populate the initial state of the baseline RNGs. This process could be done in a variety of different ways, but we will only discuss two of them. The first and probably most common way would be to use the key and IV as internal states for a RNG, and then undergo a burn-in period. The length of the burn-in period could be determined based on some value relating to the key and IVs, but it should be long enough to thoroughly mask the key and IV. After this burn-in period is complete, the outputted values will appear independent of the key and IV and thus be suitable to be used as internal states for the baseline RNG. This method requires the key and IV to be large enough to populate all of the internal states for the RNGs, which might be infeasible if large order RNGs are used.

An additional method for generating the internal states would be to use the key and IV with a Matrix Congruential Generator (MCG) as the mixing function. To demonstrate, we take the input of two 32-bit nonces, $\mathbf{v} = (v_0, v_1)$ and eight 32-bit user keys $\mathbf{k} = (k_0, k_1, \ldots, k_7)$, though more or fewer nonces and keys could be used with minimal adjustments to the procedure. To start with, we will define an additional vector $\mathbf{D} = (v_0, v_1, k_0, k_1, k_2, \ldots, k_7)'$. The user-key will also be utilized using some transformations of the key to fill in a 10×10 matrix $\mathbf{B} = (B_{ij})$. We are then able to mix the nonce and user-key by $\mathbf{E} = \mathbf{BD} \bmod 2^{32}$. Therefore, we have:

$$E_j = \sum_{r=1}^{10} \mathbf{B}_{jr}\mathbf{D}_r \bmod 2^{32}$$

which could be changed to the bit-wise "and" function for efficiency instead of multiplication:

$$E_j = \sum_{r=1}^{10} \left(\mathbf{B}_{jr}\&\mathbf{D}_r\right) \bmod 2^{32}$$

After obtaining the E_j values, we can then replace them with $\mathbf{D}$ and repeat the process until we have obtained enough variates to populate all of the internal states of the baseline generator.

As this process will only output ten 32-bit words at a time, it might be necessary to carry out this process multiple times. After obtaining the required amount of internal states, we decide to allow for a small burn-in period to further increase security.

With this process, we have used the nonce and user-key to produce the initial internal states of the RNG that we will use to populate the state matrices in SAFE-NET. It should be noted that we can expand this to cover any size nonce and user-key. This process increases security as the key and nonce are transformed before loading them into the internal states, and the burn-in period will help further hide the user-key and nonce. This makes it infeasible for attackers to determine the user-key and nonce even if they were able to obtain the initial state matrix.

4.4 Shuffle Table

An additional procedure that we can include in the method is using a "Shuffle Table" to break the linearity of the random number generators as well as further hide our internal states. Marsaglia [42] proposed a method, sometimes referred to as Algorithm M [36], which uses 2 generators with one acting as the baseline generator, which we will call X_i, and the other acting as the shuffle generator, which we will call X_{i+1}. A Table T is filled in using the baseline generator X. The value Y_i is then generated using the shuffle generator, with the function $r(x)$ being used to determine the index of the number being outputted within the Table. The outputted number at the index is then re-filled with X_i. An extension of this method was proposed by Bays and Durham [7] which only required one generator instead of two. A diagram of the "shuffle table" is given below:

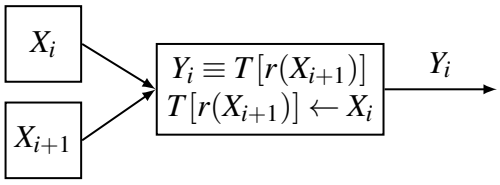

Figure 4.2: Shuffle Table Diagram

4.5 Output Function

After the ChaCha procedure is carried out for R rounds, the resulting matrix $\mathbf{I} = \mathbf{Y}^{(\mathrm{R})}$ is then XOR-ed with the page's starting state matrix. This has a key drawback in that if one knows the constants (which is typical) then they will more easily be able to determine the value of the other words in the matrix $\mathbf{Y}^{(\mathrm{R})}$. This opens one up for attack by allowing the attackers to know and determine words that should be hidden. To get around this drawback, SAFE-NET proposes a method to have the outputs shifted by some function, thus protecting our hidden words even more. Given that j represents the index of the matrix where $1 \leq j \leq 16$, the output function can be written as:

$$Z_j = f_j(\mathbf{I}, \mathbf{Y})$$

These functions theoretically could be any function one wishes it to be, but we propose the following, with Y_j representing the j^{th} block in the initial state matrix and I_j representing the j^{th} block in the R^{th} intermediate state:

$$f_j\left(I_j, I_{j+1}, Y_i\right) = f_{j_1}(I_j) \boxplus f_{j_2}(I_{j+1}) \boxplus f_{j_3}(Y_j)$$

In this case, $f_{j_1}(\cdot)$ could be a shifting function that shifts the bits to the left by a certain number of bits depending on the input word or it could be a pre-set amount. Additionally, a circular design is chosen due to its simplicity and its efficiency. Any combination of the initial matrix and the R^{th} intermediate state could be used though, but it may not be efficient if it utilizes too many blocks/operations. This proposed method gives an extra layer of security as our starting page and ending page are hidden due to the combination of 3 blocks. While this method does require one more addition than ChaCha's output function, it should allow for fewer rounds to be used to be secure, making up for the extra computations. ChaCha's output function is a special case of this general function.

4.6 Advantages of SAFE-NET

Special Cases One thing that should be noted is that ChaCha is a special case of SAFE-NET with 2 "bad" baseline generators being used to update the counter blocks sequentially. Therefore, SAFE-NET will have better diffusion properties between successive pages as all of the blocks are updated with an external RNG. This property will make SAFE-NET at least as secure as ChaCha.

More size flexibility SAFE-NET can be used with any number of terms in the "starting" matrix. There is no reason why 16 is used besides being the standard size for ChaCha. One could use even more with only slight adjustments to the method to have a larger number of blocks.

Larger Nonce/IV size The new initialization procedure allows for a much greater nonce and user-key, which will allow for more messages to be sent under the same user-key.

Increased Period Length The period length has the potential to be much greater than the period length of Chacha, which is 2^{64}. Choosing strategic a strategic modulus for the RNGs could result in a much greater period length. Using the LCGs listed in the table below, the period length will be roughly 0.52×10^{130}.

Extra Layer of Security In addition to the initial state matrices continually changing, the output function gives an extra layer of security, requiring potential attackers to solve a more complicated linear equation before determining the initial state matrix.

Efficiency A disadvantage of SAFE-NET is that it is slightly less efficient than ChaCha when carrying out a quarter-round transformation due to the extra operations in populating the blocks with RNGs. On the other hand, SAFE-NET does require fewer rounds to pass empirical tests, with R as low as 2, meaning the slight inefficiency with the RNGs requires fewer rounds and thus is more efficient than ChaCha.

Chapter 5

Exploring Special Cases of SAFE-NET

There are a few interesting cases of SAFE-NET which should be considered. Previous extensions of ChaCha have been proposed which aim to improve upon the stream cipher while maintaining the structure of ChaCha. Most of these previous extensions are all special cases of the SAFE-NET family of ciphers. Additionally, we will discuss some special cases of SAFE-NET that we have found to have nice empirical properties while still being considered a secure generator.

5.1 ChaCha

First, it should be noted that ChaCha is a special case of SAFE-NET in which all of the blocks are constant random number generators except for the counter blocks. The counter blocks are RNGs with $t_0 = i \bmod 2^{32}$ and $t_1 = \lfloor i/2^{32} \rfloor$ where i is increased every iteration. In this case, all of these generators are poor generators that have bad distributional properties and do not do much to alter the initial state matrix in successive runs. Additionally, ChaCha does not have a mutual-shuffle operation in place to increase the security of the generator. Finally, when it comes to the output function, ChaCha uses a basic modulo addition operation to decide the output words, which is a special/simple case of the SAFE-NET output function. Therefore, since ChaCha is a special case of SAFE-NET, SAFE-NET should be as secure as ChaCha if not more. Empirical evaluations on ChaCha are presented later.

5.2 Using RNGs to populate counter blocks

One possible alternative to improve ChaCha with minimal changes to the structure is to introduce random number generators to fill in the counter blocks on initial state matrices. This method, called eChaCha, can be set up to either update t_1 with a baseline generator or both counters t_0 and t_1 using baseline generators. The way ChaCha is currently structured, the counter blocks do

not provide much of a difference between successive initial state matrices, so by using a baseline generator to fill in those blocks, it should have better diffusion properties than ChaCha and be more secure. A general diagram of the procedure is shown below with only filling in t_1 with a baseline generator, but it could be done with t_0 as well:

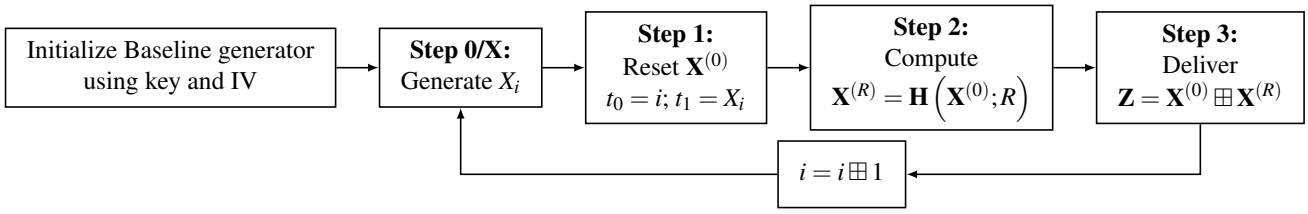

Figure 5.1: General Procedure for eChaCha Stream Cipher

Empirical evaluations shown in Chapter 6 demonstrate that just this small change to the cipher increases the empirical properties of the generator.

5.3 Choice of Baseline Generators

Since we are starting to incorporate random number generators into the ciphers, thought should be given to which generators are used. The most basic example would be the Linear-Congruential Generator (LCG) which has the form

$$X_i = (BX_{i-1} + A) \mod p, \quad i \geq 0. \tag{5.1}$$

These generators would be considered poor generators due to the small period length and because they fail a majority of the empirical tests performed on them. But, even though they are not ideal by themselves, we will later show that incorporating them into SAFE-NET results in excellent empirical performances.

Natural extensions for this type of generator are Multiple-Recursive Generators (MRG) which use the previous k values to compute the next one. The pseudo-random numbers are generated sequentially from the k-th order linear recurrence:

$$X_i = (\alpha_1 X_{i-1} + \cdots + \alpha_k X_{i-k}) \mod p, \quad i \geq k, \tag{5.2}$$

where the coefficients initial seeds are integers such that they are not all zero, and the modulus p is a positive integer. It is well known that the maximum period of the MRG in (5.2) is $p^k - 1$, which is achieved if and only if its characteristic polynomial

$$f(x) = x^k - \alpha_1 x^{k-1} - \cdots - \alpha_k \tag{5.3}$$

is a primitive polynomial [36].

Deng and Xu [21] proposed the class of DX-k generators which take advantage of the nice theoretical properties of k^{th} order MRGs. They are very efficient in that they limit the number of operations needed by having the nonzero coefficients be the same, resulting in only addition and one multiplication. These are fast generators that have been shown to pass all empirical testing [20]. The k-th order linear recurrences are:

$$\text{DX-}k\text{-1} : X_i = BX_{i-1} + X_{i-k} \mod p, \quad i \geq k \tag{5.4}$$

$$\text{DX-}k\text{-2} : X_i = B(X_{i-1} + X_{i-k}) \mod p, \quad i \geq k \tag{5.5}$$

$$\text{DX-}k\text{-3} : X_i = B(X_{i-1} + X_{i-\lceil k/2 \rceil} + X_{i-k}) \mod p, \quad i \geq k \tag{5.6}$$

$$\text{DX-}k\text{-4} : X_i = B(X_{i-1} + X_{i-\lceil k/3 \rceil} + X_{i-\lceil 2k/3 \rceil} + X_{i-k}) \mod p, \quad i \geq k \tag{5.7}$$

These DX generators only achieve a maximal period length when the characteristic polynomial is a primitive polynomial. Knuth [1] described these conditions are:

1. $(-1)^{k-1}\alpha_k$ must be a primitive root $\mod p$.

2. $x^R = (-1)^{k-1}\alpha_k \mod (f(x), p)$, where $R = (p^k - 1)/(p-1)$.

3. For each prime factor q of R, the degree of $x^{R/q} \mod (f(x), p)$ is positive.

Choosing DX generators with different moduli will allow for a longer period length, meaning it will take longer for all of the numbers to repeat at the same time. Choices for different moduli

DX generators are provided in later sections along with their multipliers to achieve larger period lengths.

Additionally, the popular MT19937 [50] could also be used as a baseline generator. But, a downside is its large internal state of 623 values which need to be stored. It should be noted that many different baseline generators could be used, but we will mainly focus on LCGs and DX-k generators due to their speed and simplicity.

5.4 Using 16 RNGs to populate blocks

To expand upon eChaCha a little bit more, instead of just filling in the counter blocks with random variates, we could use baseline generators to fill in all 16 blocks. While SAFE-NET can be carried out using most any baseline random number generator, we evaluate the procedure using 16 different LCGs to populate the blocks due to the simplicity and popularity of the generators. Later we will evaluate the possibility of using 16 higher-order MRGs to populate the blocks, allowing for a much larger period length. Using 16 RNGs helps create a more secure cipher as the blocks are continually changing and thus it is harder for attackers to determine the pattern. Additionally, using 16 different RNGs with different period lengths will result in a cipher with an extremely long period. In the table below, we give an example of 16 LCGs that could be used that will result in a cipher with a very good empirical performance and a long period up to 5.2×10^{129}.

Table 1: Strategic LCGs for maximal period length

Listing of B and m for $\text{LCG}_i = B_i x \bmod m_i$			
$B_1 = 32770$	$m_1 = 2147483647$	$B_2 = 32769$	$m_2 = 2147483629$
$B_3 = 32769$	$m_3 = 2147483587$	$B_4 = 32771$	$m_4 = 2147483579$
$B_5 = 32770$	$m_5 = 2147483563$	$B_6 = 32769$	$m_5 = 2147483549$
$B_7 = 32772$	$m_7 = 2147483543$	$B_8 = 32769$	$m_8 = 2147483497$
$B_9 = 32770$	$m_9 = 2147483489$	$B_{10} = 32770$	$m_{10} = 2147483477$
$B_{11} = 32772$	$m_{11} = 2147483423$	$B_{12} = 32769$	$m_{12} = 2147483399$
$B_{13} = 32772$	$m_{13} = 2147483353$	$B_{14} = 32771$	$m_{14} = 2147483323$
$B_{15} = 32773$	$m_{15} = 2147483269$	$B_{16} = 32769$	$m_{16} = 2147483249$

Additionally, we provide a table of 16 small-order DX-k generators that can be used in the

SAFE-NET procedure. The generators listed are just for testing purposes and any DX generators could be used, as they pass empirical testing even without mixing or transformations. When DX-2-1 generators are used, the period length can be increased to 9.5×10^{244}, which is much larger than when LCGs are used (5.2×10^{129}) or for standard ChaCha (1.8×10^{19}). Additional DX generators are given in the Appendix.

Table 2: Strategic DX-2-1 Generators for maximal period length (9.5×10^{244})

Listing of DX-2-1 Generators with different moduli			
$B_1 = 32758$	$m_1 = 2147483647$	$B_2 = 32753$	$m_2 = 2147483629$
$B_3 = 32760$	$m_3 = 2147483587$	$B_4 = 32773$	$m_4 = 2147483579$
$B_5 = 32756$	$m_5 = 2147483563$	$B_6 = 32763$	$m_5 = 2147483549$
$B_7 = 32753$	$m_7 = 2147483543$	$B_8 = 32769$	$m_8 = 2147483497$
$B_9 = 32750$	$m_9 = 2147483489$	$B_{10} = 32758$	$m_{10} = 2147483477$
$B_{11} = 32753$	$m_{11} = 2147483423$	$B_{12} = 32753$	$m_{12} = 2147483399$
$B_{13} = 32759$	$m_{13} = 2147483353$	$B_{14} = 32755$	$m_{14} = 2147483323$
$B_{15} = 32750$	$m_{15} = 2147483269$	$B_{16} = 32751$	$m_{16} = 2147483249$

Table 3: Strategic DX-3-1 Generators for maximal period length (1.6×10^{260})

Listing of DX-3-1 Generators with different moduli			
$B_1 = 32752$	$m_1 = 2147483647$	$B_2 = 32753$	$m_2 = 2147483629$
$B_3 = 32752$	$m_3 = 2147483587$	$B_4 = 32759$	$m_4 = 2147483579$
$B_5 = 32752$	$m_5 = 2147483563$	$B_6 = 32752$	$m_5 = 2147483549$
$B_7 = 32755$	$m_7 = 2147483543$	$B_8 = 32774$	$m_8 = 2147483497$
$B_9 = 32752$	$m_9 = 2147483489$	$B_{10} = 32758$	$m_{10} = 2147483477$
$B_{11} = 32776$	$m_{11} = 2147483423$	$B_{12} = 32759$	$m_{12} = 2147483399$
$B_{13} = 32757$	$m_{13} = 2147483353$	$B_{14} = 32757$	$m_{14} = 2147483323$
$B_{15} = 32756$	$m_{15} = 2147483269$	$B_{16} = 32750$	$m_{16} = 2147483249$

It should be noted that any number of RNGs could be used to populate the blocks as well as most any kind of random number generator. Empirical testing, which will be discussed in the next chapter, was done with 16 LCGs, 16 MRGs, 1 LCG, and 1 MRG to see how they performed. The fewer the number of RNGs used, the less secure the cipher will be due to multiple blocks being related to one another and the potential to do poorly in empirical testing.

Table 4: Strategic DX-4-1 Generators for maximal period length (1.5×10^{258})

Listing of DX-4-1 Generators with different moduli			
$B_1 = 32756$	$m_1 = 2147483647$	$B_2 = 32759$	$m_2 = 2147483629$
$B_3 = 32753$	$m_3 = 2147483587$	$B_4 = 32809$	$m_4 = 2147483579$
$B_5 = 32755$	$m_5 = 2147483563$	$B_6 = 32810$	$m_5 = 2147483549$
$B_7 = 32751$	$m_7 = 2147483543$	$B_8 = 32773$	$m_8 = 2147483497$
$B_9 = 32755$	$m_9 = 2147483489$	$B_{10} = 32758$	$m_{10} = 2147483477$
$B_{11} = 32758$	$m_{11} = 2147483423$	$B_{12} = 32753$	$m_{12} = 2147483399$
$B_{13} = 32757$	$m_{13} = 2147483353$	$B_{14} = 32755$	$m_{14} = 2147483323$
$B_{15} = 32751$	$m_{15} = 2147483269$	$B_{16} = 32774$	$m_{16} = 2147483249$

Table 5: Strategic DX-5-1 Generators for maximal period length (3.7×10^{276})

Listing of DX-5-1 Generators with different moduli			
$B_1 = 32757$	$m_1 = 2147483647$	$B_2 = 32755$	$m_2 = 2147483629$
$B_3 = 32763$	$m_3 = 2147483587$	$B_4 = 32762$	$m_4 = 2147483579$
$B_5 = 32790$	$m_5 = 2147483563$	$B_6 = 32760$	$m_5 = 2147483549$
$B_7 = 32764$	$m_7 = 2147483543$	$B_8 = 32762$	$m_8 = 2147483497$
$B_9 = 32750$	$m_9 = 2147483489$	$B_{10} = 32757$	$m_{10} = 2147483477$
$B_{11} = 32756$	$m_{11} = 2147483423$	$B_{12} = 32762$	$m_{12} = 2147483399$
$B_{13} = 32779$	$m_{13} = 2147483353$	$B_{14} = 32767$	$m_{14} = 2147483323$
$B_{15} = 32759$	$m_{15} = 2147483269$	$B_{16} = 32756$	$m_{16} = 2147483249$

Table 6: Strategic DX-16-1 Generators for maximal period length (1.02×10^{376})

Listing of DX-16-1 Generators with different moduli			
$B_1 = 32825$	$m_1 = 2147483647$	$B_2 = 32975$	$m_2 = 2147483629$
$B_3 = 32975$	$m_3 = 2147483587$	$B_4 = 32781$	$m_4 = 2147483579$
$B_5 = 32929$	$m_5 = 2147483563$	$B_6 = 32763$	$m_5 = 2147483549$
$B_7 = 32832$	$m_7 = 2147483543$	$B_8 = 32790$	$m_8 = 2147483497$
$B_9 = 32752$	$m_9 = 2147483489$	$B_{10} = 32782$	$m_{10} = 2147483477$
$B_{11} = 32831$	$m_{11} = 2147483423$	$B_{12} = 32757$	$m_{12} = 2147483399$
$B_{13} = 32787$	$m_{13} = 2147483353$	$B_{14} = 32797$	$m_{14} = 2147483323$
$B_{15} = 32825$	$m_{15} = 2147483269$	$B_{16} = 32824$	$m_{16} = 2147483249$

5.5 Additional Extensions

The standard Salsa20 and ChaCha cipher has a 64-bit nonce. Increasing the size of the nonce will not necessarily increase the security of the cipher, but it will allow for more messages to be sent without changing the user-key. A method, which is a case in the SAFE-NET family, proposed to increase the nonce size is to increase the nonce to 96 bits and remove one of the counters (t_1).

Like ChaCha, this can be considered a special case of SAFE-NET with all 15 of the 16 blocks being generated with a constant RNG and one counter updated each time. This does allow for the sender to send more messages under the same key, but it also reduces the number of key-stream blocks that we can output in half to 2^{32}. The argument presented by the authors is that the increased nonce is more beneficial as most 2^{32} blocks will be more than enough for most uses [53]. This smaller period length though could result in the keystream being repeated when encrypting the plaintext, which is a clear weakness [59].

XSalsa and XChaCha

Another method which can be viewed as a special case of SAFE-NET, proposed by [9], called XSalsa20, allows for a 192-bit nonce without sacrificing any of the other components. This method implements a specific "fancy" initialization procedure which allows for a larger nonce and IV size while keeping the ChaCha set-up the same.

The idea of this method is to use a different initial state matrix (to initialize a new state matrix) without the counters and instead composed of 128 bits of the 192 bit nonce as four 32-bit blocks. The Salsa20 transformation is then carried out and sixteen 32-bit words are outputted. Eight words are chosen from the outputted key-stream and are used in place of the user-key. The other unused 64 bits of the nonce are also used in the new initial state matrix.

The nonce for this method consists of six 32-bit words, denoted $(v'_0, v'_1, \ldots, v'_5)$. An initial round of Salsa20, without counters, is then run with the following initial state matrix:

$$\mathbf{Y}_{\mathrm{S}}^{(0)} = \begin{pmatrix} c_0 & k_0 & k_1 & k_2 \\ k_3 & c_1 & v'_0 & v'_1 \\ v'_2 & v'_3 & c_2 & k_4 \\ k_5 & k_6 & k_7 & c_3 \end{pmatrix}$$

After running R rounds of Salsa20 on this initial state matrix, the resulting matrix's, $\mathbf{Y}^{(R)} = \mathbf{H}(\mathbf{Y}^{(0)}; R)$, sixteen 32-bit words are outputted as $\mathbf{K} = (z_0, z_1, \ldots, z_{15})$. It should be noted that

$\mathbf{Y}^{(R)}$ is not added to $\mathbf{Y}^{(0)}$ to get the keystream like in Salsa20 due to security reasons [9]. Of the outputted key-stream, eight 32-bit words are chosen to use in our initial state matrix for Salsa20. We define $(k'_0, k'_1, k'_2, k'_3, k'_4, k'_5, k'_6, k'_7) = (z_0, z_5, z_{10}, z_{15}, z_6, z_7, z_8, z_9)$, with the indices of z chosen for security purposes. The initial state matrix for the regular Salsa20 is then defined as:

$$\mathbf{X}_{\mathrm{S}}^{(0)} = \begin{pmatrix} c_0 & k'_0 & k'_1 & k'_2 \\ k'_3 & c_1 & v'_4 & v'_5 \\ t_0 & t_1 & c_2 & k'_4 \\ k'_5 & k'_6 & k'_7 & c_3 \end{pmatrix}$$

Salsa20 is then carried out like normal without any additional modifications. This process allows for XSalsa20 to have the same components and to maintain the same as shape Salsa20. A similar procedure called XChaCha has been constructed in a way that would have similar properties of allowing a longer nonce while maintaining the shape and other components of ChaCha [3]. Below is a list of steps and a diagram of the overall procedure of both XSalsa and XChaCha:

Step X1. Given 192-bit nonce, $(v'_0, v'_1, v'_2, v'_3, v'_4, v'_5)$, replace counter blocks (t_0, t_1) with (v'_2, v'_3).

Step X2. Initialize state matrix $\mathbf{Y}^{(0)}$ with (v_0, v_1) with (v'_0, v'_1).

Step X3. Computer R-round transformation as $\mathbf{K} = \mathbf{H}(\mathbf{X}^{(0)}; R)$.

Step X4. Given the output $\mathbf{K} = (z_0, z_1, \cdots z_{15}$, choose the new user-key $(k'_0, \cdots k'_7)$ based off of the cipher being used.

Step 0-3 Carry out the standard Salsa/ChaCha stream cipher with the new user-key $(k'_0, \cdots k'_7)$ and the nonce (v'_4, v'_5) instead of (v_0, v_1)

This procedure will take slightly more time, as this method generates an extra Salsa20 block in order to initialize the initial state matrix. Additionally, a similar procedure, XChaCha, could be

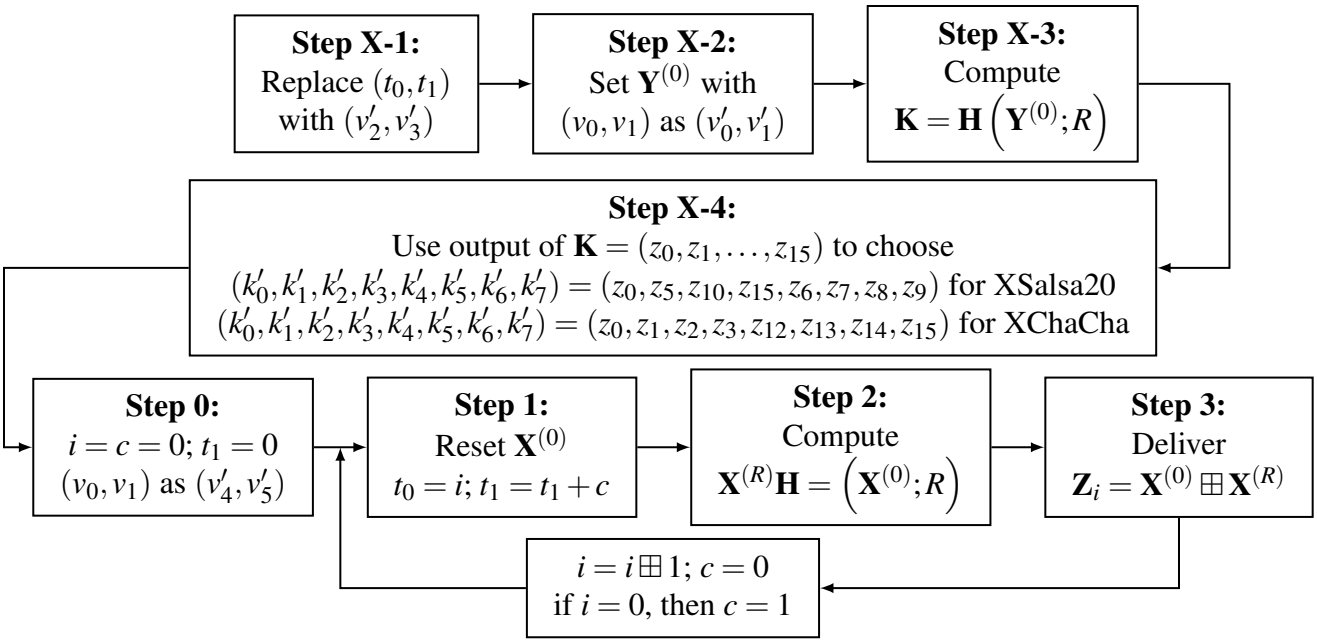

Figure 5.2: General Method for XChaCha

implemented with ChaCha. As for the security of XSalsa20, it has been shown to be as secure, if not more secure, than Salsa20 [53].

Additional extensions have also been proposed to try and remedy some of the perceived weaknesses of the ChaCha cipher by increasing the rate of diffusion. Super ChaCha, put forward by Mahdi et al. [43] is one method that has been proposed as an extension of ChaCha to alter the Quarter-Round function and how the vector is partitioned to form the Quarter-round. The new partition is decided as "Zigzag" and "Alternate" forms instead of "Column" and "Diagonal". Instead of a fixed rotation amount in the Quarter-Round, the rotation amount is variable for each Quarter-Round iteration. These values are decided based on the first 4 bits of 'a', 'b', 'c', and 'd' respectively. so y_0 is the first four bits of 'a', y_1 is the first four bits of 'b', and so on. While this method introduces a process that will help pages diffuse differently, the issue remains of only having 1 block update each successive initial state matrix. Additionally, while the author claims this method passes empirical tests and improves randomness, implementing this method and testing it with BigCrush shows clear failures, even for large R.

"Zigzag": Odd Rounds

$\mathbf{x}_0 = (x_0, x_1, x_4, x_8)$
$\mathbf{x}_1 = (x_5, x_2, x_3, x_6)$
$\mathbf{x}_2 = (x_9, x_{12}, x_{13}, x_{10})$
$\mathbf{x}_3 = (x_7, x_{11}, x_{14}, x_{15})$

"Alternate": Even Rounds

$\mathbf{x}_0 = (x_0, x_4, x_1, x_5)$
$\mathbf{x}_1 = (x_8, x_{12}, x_9, x_{13})$
$\mathbf{x}_2 = (x_2, x_6, x_3, x_7)$
$\mathbf{x}_3 = (x_{10}, x_{14}, x_{11}, x_{15})$

ChaCha Quarter-Round

$a = a + b; \quad d = d \oplus a; \quad d = d \lll 16$
$c = c + d; \quad b = b \oplus c; \quad b = b \lll 12$
$a = a + b; \quad d = d \oplus a; \quad d = d \lll 8$
$c = c + d; \quad b = b \oplus c; \quad b = b \lll 7$

Super ChaCha Quarter-Round

$a = a + b; \quad d = d \oplus a; \quad d = d \lll y_0$
$c = c + d; \quad b = b \oplus c; \quad b = b \lll y_1$
$a = a + b; \quad d = d \oplus a; \quad d = d \lll y_2$
$c = c + d; \quad b = b \oplus c; \quad b = b \lll y_3$

Finally, another extension that falls under SAFE-NET infuses chaos theory in order to have the user-key constantly changing was proposed by [2]. This method creates a 32-bit Chaotic Key using some pre-defined function and updates the initial state matrix by doing XOR on 4 bits of each user-key block using the Chaotic Key (bits 1-4 of block 1, bits 5-7 of block 2, ...). Almazrooie et al. claims this increases the diffusion of bits and requires fewer rounds to be secure. One of the downsides to this method is that this still has the issue of relatively few (up to 33) bits on successive initial state matrices.

Chapter 6

Empirical Evaluations using TESTU01

6.1 Overview of TestU01 Library

While random number generators (RNGs) are used for a variety of purposes, including simulation and cryptography, it is important to make sure that one is using a RNG that is empirically sound. This means that the sequence of generated variates closely resembles a uniform distribution and one which would be hard to distinguish from a truly random sequence. The *TESTU01* library [41] allows RNGs to undergo a number of statistical tests for uniformity. The *TESTU01* library differs from other testing packages, such as DIEHARD, the NIST package, and others, in that *TESTU01* is more stringent, contains more statistical tests, and has more flexibility for testing generators compared to the other packages. It also allows one to test both random numbers as well as random bits from a generator. This overview will mainly focus on the former, but that implementation of the latter can easily be done with a similar approach and is in fact strongly related to each other theoretically.

It is important to note that the *TESTU01* library does not contain every imaginable statistical test, as there are an infinite number of possible tests which could be created. The library does contain a variety of different tests which helps give a good idea of the quality of the RNG. These tests are also not meant to prove that a RNG will be "perfect", but passing them gives confidence that the RNG is virtually indistinguishable from a truly uniform sequence. L'Ecuyer notes that "the difference between the good and bad RNGs, in a nutshell, is that the bad ones fail very simple tests whereas the good ones fail only very complicated tests that are hard to figure out or impractical to run" [38].

When testing RNG, *p-values* are computed for the tests. These *p-values* should follow a uniform distribution and thus can be used to measure the uniformity of the generator. If the *p-value* is close to 1 then the generator is too uniform, while values close to 0 indicate the generator is not

uniform enough. The *TESTU01* library considers a generator to fail a test when the *p-values* are extremely close to 0 or 1 (less than 10^{-10} or greater than $1 - 10^{-10}$). If values are near the rejection boundary, then the tests may be replicated with different sequences until it is determined that the test obviously passes or obviously fails. It is noted that a RNG that is expected to pass a test will be labeled as "suspicious" about 2% of the time and undergo additional tests to determine if it should be failed or passed. When generators fail tests decisively, the *p-values* will converge to 0 or 1 quickly.

In testing the RNGs, the *TESTU01* library utilizes what is known as a *two-level* procedure, which essentially means that a test statistic (Y_i) is generated independently N times, and then transformed so that the transformations (U_i) are i.i.d Uniform. This is beneficial in allowing the test to have a larger sample size which increases its power, but it also allows for testing at the local level instead of the global level. The latter benefit could be done through a spacing transformation, which is done by seeing how far apart successive $U_{(i)}$ are from each other. This would help detect clustering of an RNG and help determine if it is a good generator or not. Additionally, it should be noted that most test statistics in the *TESTU01* library will follow a chi-square, a normal, or a Poisson distribution.

In addition to the test mentioned above, the library also carries out tests on a single stream of numbers by measuring global uniformity, measuring clustering, as well as implementing run and gap tests. When testing sub-sequences, a common technique is to divide the unit hyper-cube into partitions and count the number of hits per piece, or how long it takes a partition to reach a certain number of hits. Likewise, once the hyper-cube is partitioned, one can measure the time gaps between visits to the cell. Additional types of tests, such as birthday spacing, are run in order to test for randomness, for more information consult [38] for a more in-depth explanation of the theory behind the tests.

6.2 Background Information for Statistical Tests

Chi-square test

One popular test for testing the goodness-of-fit of a sequence of data is the Chi-square test, first proposed by Pearson [57]. If we have data that falls into a finite number of categories, say k categories, then we might look into using the Chi-square test. Assume we have n independent observations, and that these observations fall into one of k categories. Then, E_i is the theoretical frequency for the observations that will fall into category i, and O_i is the observed frequency for the number of observations that do fall into category i. Then the test statistic will be

$$T = \sum_{x=1}^{k} \frac{(O_i - E_i)^2}{E_i}.$$

We can then consult tables or software to determine the p-value of the test statistic T and then make a conclusion on whether the sample data fit the desired frequency table well.

Kolmogorov-Smirnov test

If our observations fall on a range with infinitely many values, say $U(0,1)$, then we might look into the Kolmogorov-Smirnov (KS) test [1951]. This will allow us to compare the difference between the theoretical distribution $F(x)$ and our empirical distribution $F_n(x)$. We should note that if we have n independent observations ordered from smallest to largest, $X_{(1)}, X_{(2)}, \ldots, X_{(n)}$, then the empirical distribution $F_n(x)$ will be

$$F_n(x) = \frac{\text{number of } X_{(1)}, X_{(2)}, \ldots, X_{(n)} \text{ that are} \leq x}{n}.$$

The test statistic will then be

$$K = \sup_x |F(x) - F_n(x)|.$$

We can then consult tables or software to determine the p-value of the test statistic K and then make a conclusion on whether the sample data fit the desired distribution well.

6.3 Statistical Tests

While there are a great number of statistical tests currently available in the literature, this section aims to discuss a few types of tests which are commonly used, as described in Knuth [36]. For the following section, we will denote $\langle U_n \rangle = U_0, U_1, U_2, \ldots$ as a sequence of numbers between 0 and 1. Additionally we will denote $\langle Y_n \rangle = Y_0, Y_1, Y_2, \ldots$ as a sequence of numbers between 0 and $d-1$. Because of this, we could write $Y_i = \lfloor dU_i \rfloor$. There are a number of different additional ways that we could transform $\langle Y_n \rangle$ into the sequence $\langle U_n \rangle$.

Frequency (Equidistribution) Test

One of the first tests that we might think to come up with is to test if the numbers do in fact follow a uniform distribution. Given the sequence $\langle U_n \rangle$, we could carry out a Kolmogorov-Smirnov test with $F(x) = x$ for $0 \leq x \leq 1$. Additionally, we could also use the Chi-square test if the sequence, $\langle Y_n \rangle$, is integer values or are broken up into a finite number of categories.

Serial Test

In addition to wanting the sequence to be independent and uniform, we also wish to have successive pairs (and t-tuples) to be independent and uniform. The Serial test, proposed by Good [28], is used to see if successive pairs are independent and uniformly distributed. The general idea of this test is to run the frequency test for higher dimensions. Say we want to look at successive pairs, then we will partition the two-dimensional $(0,1)$ square into $k = d^2$ cells of area $1/k$. We will then count the number of times the pair (Y_{2i}, Y_{2i+1}) falls in each category, for $0 \leq j < n$. The chi-square test is then run to compare the expected values for each square vs the observed values. This test can be run for higher dimensions (successive t-tuples) with the same idea of dividing the hyper-cube $(0,1)^t$ into $k = d^t$ cubes.

Gap Test

It may be useful to determine the length of time (or "gap") between a sequence's visit to a certain range. Given $0 \leq \alpha < \beta \leq 1$, we want to find the lengths of the sub-sequence $U_j, U_{j+1}, \ldots U_{j+r}$

such that both U_j and U_{j+r} are in the interval (α, β) but the other values do not fall in the interval. We will run the process n times (to end up with n gaps) and count the number of times each gap length occurs. Since we are dealing with the interval (α, β), we can denote the probability of the next number being in the interval as $p = \alpha - \beta$. We will then compare our observed gap lengths with the expected gap lengths (following a geometric distribution) using a Chi-square test.

Poker Test

Similar to a poker hand, we can also run a test called the Poker test [33] to count the number of times each type of "hand" occurs. In this test, we will take n sets of five consecutive integers $Y_{5j}, Y_{5j+1}, \ldots Y_{5j+4}$, for $0 \leq j < n$ and count the number of distinct values in the set of five. Due to possibly having the range of Y_i be very large, we might need to partition the domain into categories or take $Y_i = \lfloor dU_i \rfloor$ where d is some chosen value. With the 5 consecutive integers, we can either have 5 distinct values, 4 distinct values, ..., or 1 distinct value. After determining the categories for the n sets, we can then do a Chi-square test to compare the observed probabilities with the predicted probabilities. Knuth [36] notes the predicted probabilities of k-tuples for r distinct values is

$$p_r = \frac{d(d-1)\ldots(d-r+1)}{d^k} \left\{ {k \atop r} \right\},$$

where $\left\{ {k \atop r} \right\}$ are Stirling numbers. We can then compare the test statistic to determine the p-value.

Coupon Collector's Test

The Coupon Collector's Test [29] examines how long it will take to "collect" all of the coupons. Given the sequence $Y_j, Y_{j+1}, \ldots$, where $Y_i = \lfloor dU_i \rfloor$ for some d value, we calculate the length of segments needed to "complete" the set of integers from 0 to $d-1$. We repeat this process n times with different sequences and compare the observed lengths to the predicted lengths using a Chi-

square test. Derivations of the predicted probabilities can be found in Knuth [36] as well as Vonn [68].

Permutation Test

Another test that can be run on a sequence of data is the Permutation test. This test divides sequence into n sub-sequences of t elements each, giving us $U_{jt}, U_{jt+1}, \ldots U_{jt+t-1}$ for $0 \leq j < n$. We then classify each sub-sequence according to the relative ordering of the numbers. We can note that we will have $t!$ possible relative orderings. We count the number of times each ordering occurs and run a Chi-square test comparing the observed frequency vs the expected probability of $1/t!$.

Run Test

The Run test focuses on the "runs up" and "runs down" a sequence may exhibit. That is, the test looks at the length of the increasing monotone sections of the sequence. This test is different than previous ones though, as the Chi-square test cannot be carried out on the counts of the runs, as they are not independent of each other. This is because a long run will normally be followed by a few short runs. Due to not having the independence condition met, we will need to find another test statistic. Details about how to carry out this test are presented in [36].

Maximum-of-t Test

For the Maximum-of-t test, we take a sub-sequence of length t and record the maximum value in the sequence a total of n times. So, let $V_j = \max(U_{tj}, U_{tj+1}, U_{tj+t-1})$ for $0 \leq j < n$. We then apply the Kolmogorov-Smirnov (KS) test on the sequence of $\langle V_i \rangle$ with the distribution function $F(x) = x^t$ for $0 \leq x \leq 1$. We choose this distribution function as the sequences are independent, so the probability $P(U_i < x) = x$ and thus the probability $P(U_1, U_2, \ldots U_t) = x^t$ as $xx \cdots x = x^t$.

Collision Test

The Chi-square test should only be used when a nontrivial number of items are expected for each category. When the number of categories is greater than the number of observations then the cri-

teria would not be met. The collision test is rooted in the idea that we have m urns and n balls with $n \ll m$. If we were to randomly throw the balls into the urns then most balls will land in urns that are empty, but if more than one ball lands in the same urn then we have a "collision". We then count the number of collisions that occur. The generator will pass the Collision test if there aren't too few or too many collisions. The probability of c collisions is then

$$\frac{m(m-1)\dots(m-n+c+1)}{m^n} \left\{ \begin{array}{c} n \\ n-c \end{array} \right\},$$

where $\left\{ \begin{array}{c} n \\ n-c \end{array} \right\}$ are Stirling numbers. We can then check the p-value to make sure it is not too low or too high.

Birthday Spacings Test

The Birthday Spacings test proposed by Marsaglia [48] is similar to the Collision test in a few ways. Instead of m urns and n balls, we will think about m days and n birthdays. Given the n birthdays as $Y_1, Y_2, \dots Y_n$, with $0 \leq Y_i < m$, we then order the sequence as $Y_{(1)}, Y_{(2)}, \dots Y_{(n)}$. We then calculate the spacing between the successive observations as $S_i = Y_{(i+1)} - Y_{(i)}$ and then we sort the spacings as $S_{(1)}, S_{(2)}, S_{(n)}$ and count the number of equal spacings called R. We then repeat this process many times, say N times, and carry out a Chi-square test on the observed R's and the expected distribution. The expected distribution should follow a Poisson distribution with a mean equal to $Nn^3/(4m)$. Additional explanation on the distribution can be found in [38].

Serial Correlation Test

Correlation is often used to indicate whether two quantities might be relatively independent of each other. We can calculate the correlation of the observations $(U_0, U_1, \dots U_{n-1})$ and $(U_1, U_2, \dots U_n)$ with the Serial Correlation Test. The test statistic will be

$$C = \frac{n(U_0U_1 + U_1U_2 + \cdots U_{n-2}U_{n-1} + U_{n-1}U_0) - (U_0 + U_1 + \cdots U_{n-1})^2}{n(U_0^2 + U_1^2 + \cdots U_{n-1}^2) - (U_0 + U_1 + \cdots U_{n-1})^2}.$$

Since U_0U_1 and U_1U_2 are not independent of each other, we do not expect the serial correlation to be exactly 0. A "good" value of C should be between $\mu_n \pm 2\sigma_n$ where $\mu_n = \frac{-1}{n-1}$ and $\sigma_n = \frac{n^2}{(n-1)^2(n-1)}$ for $n \geq 2$

Spectral Test

The Spectral test, first formulated by Coveyou [17], was designed to study the lattice structures of Linear Congruential Generators (LCGs). In particular, the Spectral test studies the full period of the LCG to detect global randomness. [36] mentions that this test is the "most powerful test known" because all good generators will end up passing the test and all bad generators will end up failing it. With LCGs, [47] showed that successive k-tuples of an LCG will fall on parallel hyper-planes, with the upper bound being $(k!m)^{1/k}$ hyper-planes. The basic idea of the test is that we want to determine the maximum distance between adjacent hyper-planes.

Given a sequence $\langle U_n \rangle = U_0, U_1, U_2, \ldots$ of period m, we create the set of all m points as:

$$\{(U_n, U_{n+1}, \ldots, U_{n+k-1} |\ 0 \leq n < m\},$$

which are k-tuples in k-dimensional space. When dealing with a full period length LCG with period length $m-1$ then it is necessary to add the point $(0,0,\ldots 0)$ so that there are m points in total. We will rewrite the sequence as

$$\left\{ \frac{1}{m}(x, s(x), s(s(x)), \ldots, s^{[k-1]}(x)) |\ 0 \leq n < m \right\}$$

where $s(x) = (ax + c) \mod m$.

If we were to plot the set of points in k-dimensional space then we will see that we are able to cover all of the points with a number of parallel lines. When we are dealing with a small modulus this is easy to see, when the modulus is large you may be able to alter the scale of the x-axis to view the phenomenon. These figures can be seen in [36] for both 2 and 3-dimensional figures.

We will define $1/v_2$ as the maximum distance between lines taken over all sets of parallel lines that cover the points $\{(x/m, s(x)/m)\}$ in two dimensions. We can also define $1/v_3$ as the

maximum distance between lines taken over all sets of parallel planes that cover the points $\{(x/m, s(x)/m, s(s(x))/m)\}$ in three-dimensions. Likewise, we can extend this idea for $1/v_k$ into k-dimensions. With this, v_2 is the two-dimensional accuracy, v_3 is the three-dimensional accuracy, and so on. We usually only run the Spectral test for small k such that $2 \leq k \leq 6$. Just because we pass the test when k is small does not mean we will always pass when k is larger. Though when k is too large there is no practical significance of the test so we do not test for too large of k.

Through some manipulation, we can show that we wish to determine the value of v_k such that

$$v_k = \min \left\{ \sqrt{x_1^2 + \ldots x_k^2} \;\middle|\; x_a + ax_2 + \cdots + a^{k-1}x_k \equiv 0 \mod m \right\}$$

where $t \geq 2$ and $0 < a < m$ and a is relatively prime to m. We wish to have high values for each v_k where the value is close to the theoretical value of $m^{1/k}$. The value v_k can be tricky to find when working with a large modulus m. Knuth lays out steps to accomplish this task when k is small, as it can be very difficult when $k > 10$.

6.4 Notes on Empirical Testing

The following sections provide the empirical evaluations for a number of different stream ciphers. For each stream cipher, the BigCrush test suite was run, resulting in 106 tests being performed on the generator and 160 p-values being outputted. Running the BigCrush test suite took between 4-8 hours to run, depending on the complexity of the generator and the number of rounds being carried out. For each cipher, BigCrush was run 30 times with random starting seeds. In total, roughly 11,000 hours of computing time on the HPC (High-Performance Computer) at the University of Memphis was used to evaluate the generators. For each of the following tables, the number of "failures" is recorded (meaning the number of tests which result in a p-value smaller than a certain threshold) as well as the proportion of tests that had p-values more extreme than the columns indicate. Since the p-values should follow a uniform distribution, we would expect roughly 0.1% to be less that 0.1%, meaning the proportion should be roughly 0.001. We can do

the same for the other values as well. Ciphers that fail relatively few tests and have roughly the ideal proportion of failed tests are considered empirically sound generators.

6.5 Empirical Testing for ChaCha and Variants

The first stream cipher tested was ChaCha. It is obvious that ChaCha fails spectacularly when only 2 rounds are performed. When 4 and 6 rounds are run it appears that ChaCha passes the empirical tests, and mimics a random uniform distribution. While this demonstrates that ChaCha with 4 or 6 rounds is suitable to be used for simulation/statistical computing, it is already known that ChaCha with 4 or 6 rounds is not secure.

Table 7: Counts/proportions of p-values for Big Crush on R rounds of Chacha

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
2	3880	4039	4057	4084	209	202	186	0
4	0	0	2	9	3	0	0	0
6	0	0	0	6	5	1	0	0
8	0	0	0	3	5	0	0	0
2	0.80833	0.84146	0.84521	0.85083	0.04354	0.04208	0.03875	0.00000
4	0.00000	0.00000	0.00042	0.00188	0.00063	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00125	0.00104	0.00021	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00063	0.00104	0.00000	0.00000	0.00000

A variant of ChaCha called Super ChaCha was also tested. In this version, the quarter-round function does not have fixed rotational constants. The amount of rotation is determined based on the last 4 bits of the inputted word. The author claims that this version is more secure than ChaCha due to this quarter-round function, but it in fact makes it a worse generator. Even for a large amount of rounds it spectacularly fails the empirical tests. Something that became apparent when testing ChaCha is that the slightest change to the cipher can turn it into a bad generator. Because of this, more thorough changes are recommended to make the generator more efficient as well as resilient to small adjustments.

Table 8: Counts/proportions of p-values for Big Crush on R rounds of Super-ChaCha

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
4	1908	2267	2350	2442	189	180	173	0
6	175	253	266	286	29	18	15	0
8	169	240	258	281	22	16	13	0
20	131	173	190	209	13	10	9	0
4	0.3975	0.47229	0.48958	0.50875	0.03938	0.0375	0.03604	0.00000
6	0.03646	0.05271	0.05542	0.05958	0.00604	0.00375	0.00313	0.00000
8	0.03521	0.05	0.05375	0.05854	0.00458	0.00333	0.00271	0.00000
20	0.03899	0.05149	0.05655	0.0622	0.00387	0.00298	0.00268	0.00000

6.6 Empirical Testing of eChaCha

One way to incorporate external random number generators into the cipher is to have it populate the counter blocks. This was done by having a low-order DX generator populate the counter block t_1 since it rarely changes. This resulted in a generator that passed most tests with the number of rounds as low as 2. Similar results were also displayed when both counter blocks were updated with external DX-generators. One could even use an LCG to populate the counters, but the period length will be reduced.

Table 9: Counts/proportions of p-values for Big Crush on R rounds of eChacha (1 counter)

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
2	0	0	1	5	6	1	0	0
4	0	0	1	6	3	0	0	0
6	0	0	0	8	3	0	0	0
8	0	0	1	5	6	2	0	0
2	0.00000	0.00000	0.00021	0.00104	0.00125	0.00021	0.00000	0.00000
4	0.00000	0.00000	0.00021	0.00125	0.00063	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00167	0.00063	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00021	0.00104	0.00125	0.00042	0.00000	0.00000

This shows that small adjustments which routinely change the bits after each iteration using random number generators result in the stream cipher passing tests in as few as 2 Rounds. There-

Table 10: Counts/proportions of p-values for Big Crush on R rounds of eChacha (2 counters)

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
2	0	0	3	8	1	0	0	0
4	0	0	2	9	2	0	0	0
6	0	0	0	7	6	0	0	0
8	0	0	0	2	2	0	0	0
2	0.00000	0.00000	0.00063	0.00167	0.00021	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00042	0.00188	0.00042	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00146	0.00125	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00042	0.00042	0.00000	0.00000	0.00000

fore, by changing all the blocks after each run, we would anticipate the cipher to have even better empirical performances.

6.7 Empirical Testing for SAFE-NET

Similar to ChaCha, one could use SAFE-NET which utilizes an external Random Number Generator which has been shown to have nice statistical and empirical properties of uniformity and independence to populate all of the blocks. With this setup, ARX operations in the SAFE-NET method help break up the linearity of the Random Number Generator and passes all tests with just 2 Rounds when the baseline generator is chosen properly. A downside to this method is that initially, it is not as efficient as ChaCha, as an external generator populates every block for each initial state matrix. But fewer rounds would most likely be required for security which could lead to it being more efficient overall. Additional combinations of this method could be explored in order to improve the security of the cipher.

One could use a number of different baseline generators to populate the blocks. For instance, 16 different LCGs with differing moduli allow for simple generators with minimal memory. Therefore, if the moduli are all relatively prime with one another, then the period length has the potential to be very large. In addition to the extremely long period length, using multiple LCGs also protects against potential attacks, as figuring out the pattern of one starting block will not

make figuring out the other baseline generators any easier, as each individual block as they are all moving targets that are independent of each other.

SAFE-NET with 16 LCGs seemed to do very well in all tests, which indicates that increasing the number of LCGs being used to populate the blocks increases empirical performance. Additionally, one would be able to argue that it also increases the security of the generator as well as it uses 16 different generators. This constant changing of blocks makes it difficult to determine the pattern for each of the 16 blocks, as they are virtually unrelated in successive iterations.

Table 11: Counts/proportions of p-values for Big Crush on R rounds of SAFE-NET (16 LCGs)

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
2	0	0	0	3	5	1	0	0
4	0	0	0	8	5	1	0	0
6	0	0	2	4	6	1	0	0
8	0	0	1	6	2	0	0	0
2	0.00000	0.00000	0.00000	0.00063	0.00104	0.00021	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00167	0.00104	0.00021	0.00000	0.00000
6	0.00000	0.00000	0.00042	0.00083	0.00125	0.00021	0.00000	0.00000
8	0.00000	0.00000	0.00021	0.00125	0.00042	0.00000	0.00000	0.00000

The same can be said when 16 low-order MRGs are used. They do very well even when the number of rounds is as low as 2. This is expected though as the class of DX generators used pass empirical testing without any transformation. The SAFE-NET procedure does help break the linearity of the generators though which improves security while maintaining nice distributional properties.

Another possibility that we have discussed is filling in all the blocks using a singular LCG. This method is not recommended unless mutual-shuffle tables are used due to the fact that the period length will usually be less than 2^{32} without one. Choosing a RNG with a small period length could result in the SAFE-NET stream cipher having bad empirical performance due to the numbers repeating sooner. But, using 1 LCG to populate the shuffle the tables and populate the blocks does remarkably well. It is known that an LCG by itself fails the empirical tests miserably, but

Table 12: Counts/proportions of p-values for Big Crush on R rounds of SAFE-NET (16 MRGs)

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
2	0	0	0	8	9	2	0	0
4	0	0	1	7	3	1	0	0
6	0	0	0	7	9	0	0	0
8	0	0	0	5	4	1	0	0
2	0.00000	0.00000	0.00000	0.00167	0.00188	0.00042	0.00000	0.00000
4	0.00000	0.00000	0.00021	0.00146	0.00063	0.00021	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00146	0.00188	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00104	0.00083	0.00021	0.00000	0.00000

using one LCG to fill and shuffle the tables does decently when no quarter-rounds are even performed. Increasing the number of rounds results in a cipher that passes most tests and is empirically sound.

Table 13: Counts/proportions of p-values for Big Crush on R rounds of SAFE-NET (1 LCG)

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
0	266	287	305	337	175	153	141	0
2	0	0	0	6	6	0	0	0
4	0	0	0	3	2	0	0	0
6	0	0	0	2	0	0	0	0
8	0	0	0	7	2	1	0	0
0	0.05542	0.05979	0.06354	0.07021	0.03646	0.03188	0.02938	0.00000
2	0.00000	0.00000	0.00000	0.00125	0.00125	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00063	0.00042	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00042	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00146	0.00042	0.00021	0.00000	0.00000

Another aspect that can be investigated is determining the effects of the table size on the empirical properties of the cipher. For all previous analyses, a table size of 16 was in the shuffle-table. For the results below, a table of size 2 was used. With it, we can see much of the same results. When 1 LCG or 1 small order DX generator is used, it struggles to pass all tests with 0 rounds, but once the cipher starts transforming the values using the quarter-round function it ap-

Table 14: Counts/proportions of p-values for Big Crush on R rounds of SAFE-NET (1 MRG)

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
0	0	1	2	11	3	1	1	0
2	0	0	0	5	5	2	0	0
4	0	0	0	4	2	1	1	0
6	0	0	0	10	4	0	0	0
8	0	0	2	7	2	1	0	0
0	0.00000	0.00021	0.00042	0.00229	0.00063	0.00021	0.00021	0.00000
2	0.00000	0.00000	0.00000	0.00104	0.00104	0.00042	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00083	0.00042	0.00021	0.00021	0.00000
6	0.00000	0.00000	0.00000	0.00208	0.00083	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00042	0.00146	0.00042	0.00021	0.00000	0.00000

pears to do quite well. The same can be said regarding using 16 LCGs with a shuffle table of size 2.

Table 15: Counts/proportions of p-values for Big Crush on R rounds of SAFE-NET (1 LCG) with Table size 2

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
0	1839	1992	2024	2115	191	161	144	0
2	0	0	0	3	5	0	0	0
4	0	0	0	8	2	0	0	0
6	0	0	1	8	6	1	0	0
8	0	0	0	9	5	1	0	0
0	0.38313	0.41500	0.42167	0.44063	0.03979	0.03354	0.03000	0.00000
2	0.00000	0.00000	0.00000	0.00063	0.00104	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00167	0.00042	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00021	0.00167	0.00125	0.00021	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00188	0.00104	0.00021	0.00000	0.00000

After empirical testing, it can be seen that the SAFE-NET method does extremely well and passed empirical testing for small number of rounds. This can be attributed to routinely updating each block so that the values are greatly different in successive state matrices. Additionally, the implementation of the shuffle table and mutual-shuffling technique further aids in the empiri-

Table 16: Counts/proportions of p-values for Big Crush on R rounds of SAFE-NET (1 MRG) with Table size 2

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
0	60	67	77	95	5	0	0	0
2	0	0	0	7	6	0	0	0
4	0	0	0	6	2	0	0	0
6	0	0	1	5	1	0	0	0
8	0	0	0	8	3	0	0	0
0	0.0125	0.01396	0.01604	0.01979	0.00104	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00146	0.00125	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00125	0.00042	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00021	0.00104	0.00021	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00167	0.00063	0.00000	0.00000	0.00000

Table 17: Counts/proportions of p-values for Big Crush on R rounds of SAFE-NET (16 LCGs) with Table size 2

R	$< 10^{-15}$	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	$> 1-10^{-3}$	$> 1-10^{-4}$	$> 1-10^{-5}$	$> 1-10^{-15}$
0	60	60	60	65	25	13	6	0
2	0	0	0	6	11	1	0	0
4	0	0	0	5	5	0	0	0
6	0	0	0	5	1	0	0	0
8	0	0	2	4	4	1	0	0
0	0.0125	0.0125	0.0125	0.01354	0.00521	0.00271	0.00125	0.00000
2	0.00000	0.00000	0.00000	0.00125	0.00229	0.00021	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00104	0.00104	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00104	0.00021	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00042	0.00083	0.00083	0.00021	0.00000	0.00000

cal performance of the generators. While it should be noted that passing empirical tests does not make the generator cryptographically secure, it is an important aspect. Additional cryptanalysis is needed in order to make the claim that it is secure for a certain number of rounds. But, given its similarity to ChaCha, we can claim that the SAFE-NET procedure is as secure as ChaCha.

Chapter 7

Conclusion and Future Work

In this dissertation, a stream cipher was proposed that could be used for both simulation and security purposes. Secure And Fast Encryption using Network of Pseudo-Random Number Generators (SAFE-NET) is a cipher that utilizes baseline generators, shuffle-tables, as well as complex output functions to produce a secure generator. The attributes address some of the weaknesses shown in other popular stream ciphers such as ChaCha. Importantly, any "bad" random number generator can be used as a baseline generator to inject randomness into the cipher and give good empirical results.

The proposed cipher exhibits excellent empirical results, passing the TESTU01 battery of tests with minimal number of rounds required to transform the initial state matrix. While the procedure does require more operation and initialization each round, it should require fewer rounds to be considered secure. This is because our method incorporates uniform numbers throughout the generator instead of sequentially increasing a counter block.

Future studies can be done in this field to improve the efficiency of the generator as well as to increase security. Much of what we did was motivated by ChaCha, and thus we kept a lot of the general framework the same. Investigating different approaches and mixing functions could potentially increase the appeal of the cipher. In-depth cryptoanalysis can be performed on the cipher to determine how secure the cipher is and to find potential weaknesses.

Bibliography

[1] J. Alanen and D. E. Knuth. Tables of finite fields. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 305–328, 1964.

[2] M. Almazrooie, A. Samsudin, and M. M. Mahinderjit Singh. Improving the diffusion of the stream cipher salsa20 by employing a chaotic logistic map. *Journal of Information Processing Systems*, 11:310 324, 01 2015. doi: 10.3745/JIPS.02.0024.

[3] S. Arciszewski. Xchacha: extended-nonce chacha and aead_xchacha20_poly1305. Internet-Draft draft-irtf-cfrg-xchacha-02, IETF Secretariat, January 2020. URL `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-xchacha-02.txt`. `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-xchacha-02.txt`.

[4] F. Armknecht. Improving fast algebraic attacks. In *Fast Software Encryption: 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004. Revised Papers 11*, pages 65–82. Springer, 2004.

[5] J.-P. Aumasson. On a bias of rabbit. In *State of the Art of Stream Ciphers Workshop (SASC 2007), eSTREAM, ECRYPT Stream Cipher Project, Report*, volume 29. Citeseer, 2007.

[6] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger. New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. In K. Nyberg, editor, *Fast Software Encryption*, volume 5086, pages 470–488. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-71039-4_30.

[7] C. Bays and S. D. Durham. Improving a poor random number generator. *ACM Transactions on Mathematical Software (TOMS)*, 2(1):59–64, Mar. 1976. ISSN 0098-3500.

[8] D. J. Bernstein. The salsa20 family of stream ciphers. *New stream cipher designs: the eSTREAM finalists*, pages 84–97, 2008.

[9] D. J. Bernstein. Extending the salsa20 nonce. In *Workshop record of Symmetric Key Encryption Workshop*, volume 2011, 2011.

[10] D. J. Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer, 2008.

[11] E. Biham. New types of cryptanalytic attacks using related keys. *Journal of Cryptology*, 7: 229–246, 1994.

[12] E. Biham and A. Shamir. *Differential cryptanalysis of the data encryption standard*. Springer Science & Business Media, 2012.

[13] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, and O. Scavenius. Rabbit: A new high-performance stream cipher. In *Fast Software Encryption: 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003. Revised Papers 10*, pages 307–329. Springer, 2003.

[14] J. Boyar. Inferring sequences produced by pseudo-random number generators. *Journal of the Association for Computing Machinery*, 36:129–141, 1989.

[15] A. R. Choudhuri and S. Maitra. Differential cryptanalysis of salsa and chacha – an evaluation with a hybrid model. Cryptology ePrint Archive, Paper 2016/377, 2016. URL `https://eprint.iacr.org/2016/377`. `https://eprint.iacr.org/2016/377`.

[16] A. R. Choudhuri and S. Maitra. Significantly improved multi-bit differentials for reduced round salsa and chacha. *IACR Transactions on Symmetric Cryptology*, pages 261–287, 2016.

[17] R. Coveyou and R. D. MacPherson. Fourier analysis of uniform random number generators. *Journal of the ACM (JACM)*, 14(1):100–119, 1967.

[18] P. Crowley. Truncated differential cryptanalysis of five rounds of salsa20. *Cryptology ePrint Archive*, 2005.

[19] P. Deepthi, D. S. John, and P. Sathidevi. Design and analysis of a highly secure stream cipher based on linear feedback shift register. *Computers & Electrical Engineering*, 35(2): 235–243, 2009.

[20] L.-Y. Deng. Efficient and portable multiple recursive generators of large order. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(1):1–13, 2005.

[21] L.-Y. Deng and H. Xu. A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(4):299–309, 2003.

[22] L.-Y. Deng, J.-J. H. Shiau, H. H.-S. Lu, and D. Bowman. Secure and fast encryption (safe) with classical random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 44(4), 2018.

[23] L.-Y. Deng, D. Bowman, C.-C. Yang, and H. H.-S. Lu. Extending rc4 to construct secure random number generators. In *2021 Annual Modeling and Simulation Conference (ANNSIM)*, pages 1–12. IEEE, 2021.

[24] S. Dey and S. Sarkar. Improved analysis for reduced round salsa and chacha. *Discrete Applied Mathematics*, 227:58–69, 2017.

[25] X. Fei, S. Zhang, and W. Sun. New analysis on security of stream cipher hc-256. In *2013 International Conference on Computational and Information Sciences*, pages 1766–1769. IEEE, 2013.

[26] S. Fischer, W. Meier, C. Berbain, J.-F. Biasse, and M. J. Robshaw. Non-randomness in estream candidates salsa20 and tsc-4. In *Progress in Cryptology-INDOCRYPT 2006: 7th International Conference on Cryptology in India, Kolkata, India, December 11-13, 2006. Proceedings 7*, pages 2–16. Springer, 2006.

[27] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In *Selected Areas in Cryptography: 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16–17, 2001 Revised Papers 8*, pages 1–24. Springer, 2001.

[28] I. Good. The serial test for sampling numbers and other tests for randomness. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 49, pages 276–284. Cambridge University Press, 1953.

[29] R. E. Greenwood. Coupon collector's test for random digits. *Mathematical Tables and Other Aids to Computation*, pages 1–5, 1955.

[30] T. Ishiguro, S. Kiyomoto, and Y. Miyake. Latin dances revisited: new analytic results of salsa20 and chacha. In *Information and Communications Security: 13th International Conference, ICICS 2011, Beijing, China, November 23-26, 2011. Proceedings 13*, pages 255–266. Springer, 2011.

[31] P. Jindal and B. Singh. Rc4 encryption-a literature survey. *Procedia Computer Science*, 46: 697–705, 2015.

[32] J. Katz and Y. Lindell. Introduction to modern cryptography: principles and protocols. *Chapman and Hall*, 2008.

[33] M. G. Kendall and B. Babington-Smith. Second paper on random sampling numbers. *Supplement to the Journal of the Royal Statistical Society*, 6(1):51–61, 1939.

[34] A. Kircanski and A. M. Youssef. Differential fault analysis of rabbit. In *Selected Areas in Cryptography: 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers 16*, pages 197–214. Springer, 2009.

[35] A. Kircanski and A. M. Youssef. Differential fault analysis of hc-128. In *Progress in Cryptology–AFRICACRYPT 2010: Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings 3*, pages 261–278. Springer, 2010.

[36] D. E. Knuth. *The art of computer programming, vol 2: seminumerical algorithms*. Addison-Wesley, 3 edition, 1998.

[37] S. Künzli and W. Meier. Distinguishing attack on mag. *ECRYPT Stream Cipher Project Report*, 1:2005, 2005.

[38] P. L'Ecuyer and R. Simard. Testu01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):22, 2007.

[39] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery*, pages 141–146. Harvard Univ. Press Cambridge, MA, 1951.

[40] Y. Lu, H. Wang, and S. Ling. Cryptanalysis of rabbit. In *ISC*, volume 8, pages 204–214. Springer, 2008.

[41] P. L'Ecuyer and R. Simard. *TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators*. University of Montreal, 2013.

[42] M. D. MacLaren and G. Marsaglia. Uniform random number generators. *Journal of the ACM*, 12:83–89, 1965.

[43] M. S. Mahdi, N. F. Hassan, and G. H. Abdul-Majeed. An improved chacha algorithm for securing data on iot devices. *SN Applied Sciences*, 3, 2021.

[44] S. Maitra and G. Paul. Analysis of RC4 and proposal of additional layers for better security margin. *INDOCRYPT 2008, Lecture Notes in Computer Science*, 5365:27–39, 2008.

[45] I. Mantin and A. Shamir. A practical attack on broadcast rc4. In *Fast Software Encryption 2001, Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[46] G. Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 61(1):25, 1968.

[47] G. Marsaglia. The s of linear congruential sequences. In *Applications of number theory to numerical analysis*, pages 249–285. Elsevier, 1972.

[48] G. Marsaglia. A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface. Elsevier Science Publishers, North-Holland, Amsterdam*, pages 3–10, 1985.

[49] F. J. Massey. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951. ISSN 01621459. URL `http://www.jstor.org/stable/2280095`.

[50] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[51] Y. Nawaz, K. C. Gupta, and G. Gong. A 32-bit rc4-like keystream generator. *Cryptology ePrint Archive*, 2005.

[52] H. Niederreiter. A pseudorandom vector generator based on finite field arithmetic. *Math. Japonica*, 31(5):759–774, 1986.

[53] Y. Nir and A. Langley. Chacha20 and poly1305 for ietf protocols. *RFC*, 7539:1–45, 2015.

[54] C. Paar and J. Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[55] G. Paul and S. Raizada. Impact of extending side channel attack on cipher variants: a case study with the hc series of stream ciphers. In *Security, Privacy, and Applied Cryptography Engineering: Second International Conference, SPACE 2012, Chennai, India, November 3-4, 2012. Proceedings*, pages 32–44. Springer, 2012.

[56] G. Paul, S. Maitra, and A. Chattopadhyay. Quad-rc4: merging four rc4 states towards a 32-bit stream cipher. *Cryptology ePrint Archive*, 2013.

[57] K. Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[58] M. Robshaw and O. Billet. *New Stream Cipher Designs – The eSTREAM Finalists*, volume LNCS 4986. Springer-Verlag, 2008.

[59] M. J. Robshaw. Stream ciphers. *RSA Labratories*, 25, 1995.

[60] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, 1995. ISBN 9780471128458.

[61] C. E. Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[62] Z. Shi, B. Zhang, D. Feng, and W. Wu. Improved key recovery attacks on reduced-round salsa20 and chacha. In *Information Security and Cryptology–ICISC 2012: 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers 15*, pages 337–351. Springer, 2013.

[63] G. J. Simmons. *Contemporary cryptology: The science of information integrity*. IEEE press, 1994.

[64] R. Sobti and G. Ganesan. Analysis of Quarter Rounds of Salsa and Chacha Core and Proposal of an Alternative Design to Maximize Diffusion. *Indian Journal of Science and Technology*, 9(3), 1 2016. ISSN 0974-5645, 0974-6846. doi: 10.17485/ijst/2016/v9i3/80087.

[65] J. Stern. Secret linear congruential generators are not cryptographically secure. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 421–426. IEEE, 1987.

[66] Y. Tsunoo, T. Saito, H. Kubo, T. Suzaki, and H. Nakashima. Differential cryptanalysis of salsa20/8. In *Workshop Record of SASC*, volume 28, 2007.

[67] G. S. Vernam. Cipher printing telegraph systems: For secret wire and radio telegraphic communications. *Journal of the AIEE*, 45(2):109–115, 1926.

[68] H. Von Schelling. Coupon collecting for unequal probabilities. *The American Mathematical Monthly*, 61(5):306–311, 1954.

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

[70] K. Wong, G. Carter, and E. Dawson. An analysis of the rc4 family of stream ciphers against algebraic attacks. *Information Security 2010*, pages 73–80, 2010.

[71] H. Wu. A new stream cipher HC-256. In B. Roy and W. Meier, editors, *Proceedings of FSE 2004, Lecture Notes in Computer Science*, volume 3017, pages 226–244. Springer, 2004.

[72] H. Wu. Cryptanalysis of 32-bit RC4-like stream cipher. *IACR Eprint Server,* `eprint.iacr.org`, 2005.

[73] K. Zeng, C.-H. Yang, D.-Y. Wei, and T. Rao. Pseudorandom bit generators in stream-cipher cryptography. *Computer*, 24(2):8–17, 1991.

Appendix A

Appendix: Small Order DX Generators

Table 18: Strategic DX-2-2 Generators for maximal period length

Listing of DX-2-2 Generators with different moduli			
$B_1 = 32750$	$m_1 = 2147483647$	$B_2 = 32753$	$m_2 = 2147483629$
$B_3 = 32761$	$m_3 = 2147483587$	$B_4 = 32761$	$m_4 = 2147483579$
$B_5 = 32756$	$m_5 = 2147483563$	$B_6 = 32755$	$m_5 = 2147483549$
$B_7 = 32758$	$m_7 = 2147483543$	$B_8 = 32753$	$m_8 = 2147483497$
$B_9 = 32752$	$m_9 = 2147483489$	$B_{10} = 32757$	$m_{10} = 2147483477$
$B_{11} = 32752$	$m_{11} = 2147483423$	$B_{12} = 32752$	$m_{12} = 2147483399$
$B_{13} = 32757$	$m_{13} = 2147483353$	$B_{14} = 32759$	$m_{14} = 2147483323$
$B_{15} = 32754$	$m_{15} = 2147483269$	$B_{16} = 32751$	$m_{16} = 2147483249$

Table 19: Strategic DX-3-2 Generators for maximal period length

Listing of DX-3-2 Generators with different moduli			
$B_1 = 32752$	$m_1 = 2147483647$	$B_2 = 32753$	$m_2 = 2147483629$
$B_3 = 32755$	$m_3 = 2147483587$	$B_4 = 32764$	$m_4 = 2147483579$
$B_5 = 32752$	$m_5 = 2147483563$	$B_6 = 32750$	$m_5 = 2147483549$
$B_7 = 32750$	$m_7 = 2147483543$	$B_8 = 32753$	$m_8 = 2147483497$
$B_9 = 32750$	$m_9 = 2147483489$	$B_{10} = 32763$	$m_{10} = 2147483477$
$B_{11} = 32760$	$m_{11} = 2147483423$	$B_{12} = 32759$	$m_{12} = 2147483399$
$B_{13} = 32757$	$m_{13} = 2147483353$	$B_{14} = 32753$	$m_{14} = 2147483323$
$B_{15} = 32754$	$m_{15} = 2147483269$	$B_{16} = 32756$	$m_{16} = 2147483249$

Table 20: Strategic DX-4-2 Generators for maximal period length

Listing of DX-4-2 Generators with different moduli			
$B_1 = 32750$	$m_1 = 2147483647$	$B_2 = 32753$	$m_2 = 2147483629$
$B_3 = 32753$	$m_3 = 2147483587$	$B_4 = 32753$	$m_4 = 2147483579$
$B_5 = 32783$	$m_5 = 2147483563$	$B_6 = 32774$	$m_5 = 2147483549$
$B_7 = 32768$	$m_7 = 2147483543$	$B_8 = 32779$	$m_8 = 2147483497$
$B_9 = 32755$	$m_9 = 2147483489$	$B_{10} = 32766$	$m_{10} = 2147483477$
$B_{11} = 32753$	$m_{11} = 2147483423$	$B_{12} = 32755$	$m_{12} = 2147483399$
$B_{13} = 32773$	$m_{13} = 2147483353$	$B_{14} = 32761$	$m_{14} = 2147483323$
$B_{15} = 32759$	$m_{15} = 2147483269$	$B_{16} = 32769$	$m_{16} = 2147483249$

Table 21: Strategic DX-5-2 Generators for maximal period length

Listing of DX-5-2 Generators with different moduli			
$B_1 = 32752$	$m_1 = 2147483647$	$B_2 = 32766$	$m_2 = 2147483629$
$B_3 = 32767$	$m_3 = 2147483587$	$B_4 = 32754$	$m_4 = 2147483579$
$B_5 = 32752$	$m_5 = 2147483563$	$B_6 = 32752$	$m_5 = 2147483549$
$B_7 = 32756$	$m_7 = 2147483543$	$B_8 = 32774$	$m_8 = 2147483497$
$B_9 = 32753$	$m_9 = 2147483489$	$B_{10} = 32753$	$m_{10} = 2147483477$
$B_{11} = 32750$	$m_{11} = 2147483423$	$B_{12} = 32751$	$m_{12} = 2147483399$
$B_{13} = 32757$	$m_{13} = 2147483353$	$B_{14} = 32774$	$m_{14} = 2147483323$
$B_{15} = 32759$	$m_{15} = 2147483269$	$B_{16} = 32786$	$m_{16} = 2147483249$

Table 22: Strategic DX-16-2 Generators for maximal period length

Listing of DX-16-2 Generators with different moduli			
$B_1 = 32760$	$m_1 = 2147483647$	$B_2 = 32832$	$m_2 = 2147483629$
$B_3 = 32760$	$m_3 = 2147483587$	$B_4 = 32813$	$m_4 = 2147483579$
$B_5 = 32818$	$m_5 = 2147483563$	$B_6 = 32754$	$m_5 = 2147483549$
$B_7 = 32800$	$m_7 = 2147483543$	$B_8 = 32836$	$m_8 = 2147483497$
$B_9 = 32787$	$m_9 = 2147483489$	$B_{10} = 33021$	$m_{10} = 2147483477$
$B_{11} = 32870$	$m_{11} = 2147483423$	$B_{12} = 32868$	$m_{12} = 2147483399$
$B_{13} = 32891$	$m_{13} = 2147483353$	$B_{14} = 32895$	$m_{14} = 2147483323$
$B_{15} = 32964$	$m_{15} = 2147483269$	$B_{16} = 32759$	$m_{16} = 2147483249$

Table 23: Strategic DX-3-3 Generators for maximal period length

Listing of DX-3-3 Generators with different moduli			
$B_1 = 32754$	$m_1 = 2147483647$	$B_2 = 32755$	$m_2 = 2147483629$
$B_3 = 32752$	$m_3 = 2147483587$	$B_4 = 32752$	$m_4 = 2147483579$
$B_5 = 32765$	$m_5 = 2147483563$	$B_6 = 32750$	$m_5 = 2147483549$
$B_7 = 32755$	$m_7 = 2147483543$	$B_8 = 32785$	$m_8 = 2147483497$
$B_9 = 32754$	$m_9 = 2147483489$	$B_{10} = 32753$	$m_{10} = 2147483477$
$B_{11} = 32751$	$m_{11} = 2147483423$	$B_{12} = 32756$	$m_{12} = 2147483399$
$B_{13} = 32791$	$m_{13} = 2147483353$	$B_{14} = 32757$	$m_{14} = 2147483323$
$B_{15} = 32751$	$m_{15} = 2147483269$	$B_{16} = 32751$	$m_{16} = 2147483249$

Table 24: Strategic DX-4-3 Generators for maximal period length

Listing of DX-4-3 Generators with different moduli			
$B_1 = 32760$	$m_1 = 2147483647$	$B_2 = 32753$	$m_2 = 2147483629$
$B_3 = 32761$	$m_3 = 2147483587$	$B_4 = 32756$	$m_4 = 2147483579$
$B_5 = 32798$	$m_5 = 2147483563$	$B_6 = 32752$	$m_5 = 2147483549$
$B_7 = 32763$	$m_7 = 2147483543$	$B_8 = 32753$	$m_8 = 2147483497$
$B_9 = 32772$	$m_9 = 2147483489$	$B_{10} = 32757$	$m_{10} = 2147483477$
$B_{11} = 32785$	$m_{11} = 2147483423$	$B_{12} = 32753$	$m_{12} = 2147483399$
$B_{13} = 32757$	$m_{13} = 2147483353$	$B_{14} = 32761$	$m_{14} = 2147483323$
$B_{15} = 32756$	$m_{15} = 2147483269$	$B_{16} = 32760$	$m_{16} = 2147483249$

Table 25: Strategic DX-5-3 Generators for maximal period length

Listing of DX-5-3 Generators with different moduli			
$B_1 = 32752$	$m_1 = 2147483647$	$B_2 = 32753$	$m_2 = 2147483629$
$B_3 = 32752$	$m_3 = 2147483587$	$B_4 = 32777$	$m_4 = 2147483579$
$B_5 = 32770$	$m_5 = 2147483563$	$B_6 = 32754$	$m_5 = 2147483549$
$B_7 = 32765$	$m_7 = 2147483543$	$B_8 = 32762$	$m_8 = 2147483497$
$B_9 = 32765$	$m_9 = 2147483489$	$B_{10} = 32767$	$m_{10} = 2147483477$
$B_{11} = 32756$	$m_{11} = 2147483423$	$B_{12} = 32769$	$m_{12} = 2147483399$
$B_{13} = 32772$	$m_{13} = 2147483353$	$B_{14} = 32757$	$m_{14} = 2147483323$
$B_{15} = 32750$	$m_{15} = 2147483269$	$B_{16} = 32751$	$m_{16} = 2147483249$

Table 26: Strategic DX-16-3 Generators for maximal period length

Listing of DX-16-3 Generators with different moduli			
$B_1 = 32827$	$m_1 = 2147483647$	$B_2 = 32832$	$m_2 = 2147483629$
$B_3 = 32851$	$m_3 = 2147483587$	$B_4 = 32756$	$m_4 = 2147483579$
$B_5 = 32797$	$m_5 = 2147483563$	$B_6 = 32855$	$m_5 = 2147483549$
$B_7 = 32833$	$m_7 = 2147483543$	$B_8 = 32772$	$m_8 = 2147483497$
$B_9 = 32753$	$m_9 = 2147483489$	$B_{10} = 32822$	$m_{10} = 2147483477$
$B_{11} = 32877$	$m_{11} = 2147483423$	$B_{12} = 32883$	$m_{12} = 2147483399$
$B_{13} = 32933$	$m_{13} = 2147483353$	$B_{14} = 33157$	$m_{14} = 2147483323$
$B_{15} = 32816$	$m_{15} = 2147483269$	$B_{16} = 32827$	$m_{16} = 2147483249$

Table 27: Strategic DX-4-4 Generators for maximal period length

Listing of DX-4-4 Generators with different moduli			
$B_1 = 32772$	$m_1 = 2147483647$	$B_2 = 32811$	$m_2 = 2147483629$
$B_3 = 32771$	$m_3 = 2147483587$	$B_4 = 32756$	$m_4 = 2147483579$
$B_5 = 32802$	$m_5 = 2147483563$	$B_6 = 32813$	$m_5 = 2147483549$
$B_7 = 32763$	$m_7 = 2147483543$	$B_8 = 32753$	$m_8 = 2147483497$
$B_9 = 32750$	$m_9 = 2147483489$	$B_{10} = 32780$	$m_{10} = 2147483477$
$B_{11} = 32752$	$m_{11} = 2147483423$	$B_{12} = 32760$	$m_{12} = 2147483399$
$B_{13} = 32772$	$m_{13} = 2147483353$	$B_{14} = 32755$	$m_{14} = 2147483323$
$B_{15} = 32756$	$m_{15} = 2147483269$	$B_{16} = 32774$	$m_{16} = 2147483249$

Table 28: Strategic DX-5-4 Generators for maximal period length

Listing of DX-5-4 Generators with different moduli			
$B_1 = 32770$	$m_1 = 2147483647$	$B_2 = 32767$	$m_2 = 2147483629$
$B_3 = 32766$	$m_3 = 2147483587$	$B_4 = 32752$	$m_4 = 2147483579$
$B_5 = 32752$	$m_5 = 2147483563$	$B_6 = 32771$	$m_5 = 2147483549$
$B_7 = 32786$	$m_7 = 2147483543$	$B_8 = 32753$	$m_8 = 2147483497$
$B_9 = 32770$	$m_9 = 2147483489$	$B_{10} = 32763$	$m_{10} = 2147483477$
$B_{11} = 32760$	$m_{11} = 2147483423$	$B_{12} = 32756$	$m_{12} = 2147483399$
$B_{13} = 32764$	$m_{13} = 2147483353$	$B_{14} = 32767$	$m_{14} = 2147483323$
$B_{15} = 32809$	$m_{15} = 2147483269$	$B_{16} = 32760$	$m_{16} = 2147483249$

Table 29: Strategic DX-16-4 Generators for maximal period length

Listing of DX-16-4 Generators with different moduli			
$B_1 = 32773$	$m_1 = 2147483647$	$B_2 = 32829$	$m_2 = 2147483629$
$B_3 = 32826$	$m_3 = 2147483587$	$B_4 = 32884$	$m_4 = 2147483579$
$B_5 = 32841$	$m_5 = 2147483563$	$B_6 = 32855$	$m_5 = 2147483549$
$B_7 = 32760$	$m_7 = 2147483543$	$B_8 = 32753$	$m_8 = 2147483497$
$B_9 = 32791$	$m_9 = 2147483489$	$B_{10} = 32787$	$m_{10} = 2147483477$
$B_{11} = 32902$	$m_{11} = 2147483423$	$B_{12} = 32764$	$m_{12} = 2147483399$
$B_{13} = 32817$	$m_{13} = 2147483353$	$B_{14} = 32942$	$m_{14} = 2147483323$
$B_{15} = 32754$	$m_{15} = 2147483269$	$B_{16} = 32812$	$m_{16} = 2147483249$