NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

CHAOTIC COMBINER FOR LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

by

Alexander Gutzler

March 2020

Thesis Advisor:	Thor Martinsen
Second Reader:	Beny Neta

THIS PAGE INTENTIONALLY LEFT BLANK

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

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2020	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
CHAOTIC COMBINER FOR LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

5. FUNDING NUMBERS

6. AUTHOR(S) Alexander Gutzler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

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

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Cryptography is widely used by everybody in day-to-day activities. Many cryptographic algorithms rely on pseudorandom number sequences. One of the quickest methods of pseudorandom number generation is using linear feedback shift registers (LFSR) to generate sequences. LFSR sequences exhibit good statistical properties, but alone are not adequately secure due to their low linear complexity. To enhance the security, separate LFSR sequences can be combined into a single pseudorandom string by using a combiner. In this thesis, creating a combiner function using another pseudorandom sequence derived from the chaotic motion of a double pendulum is investigated. Using the information from this driving function, an iterative process occurs whereby certain LFSR sequence blocks are selected and combined. The resultant sequences are sufficiently random as proven by the 15 tests adopted by the National Institute of Standards and Technology to evaluate the randomness of binary strings.

14. SUBJECT TERMS cryptography, pseudorandom number generation, linear feedback shift register, combiner, hybrid, linear complexity	**15. NUMBER OF PAGES** 73
	16. PRICE CODE

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

NSN 7540-01-280-5500

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

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

CHAOTIC COMBINER FOR LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

Alexander Gutzler
Lieutenant, United States Navy
BSME, Virginia Tech, 2013

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
March 2020

Approved by: Thor Martinsen
Advisor

Beny Neta
Second Reader

Wei Kang
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Cryptography is widely used by everybody in day-to-day activities. Many cryptographic algorithms rely on pseudorandom number sequences. One of the quickest methods of pseudorandom number generation is using linear feedback shift registers (LFSR) to generate sequences. LFSR sequences exhibit good statistical properties, but alone are not adequately secure due to their low linear complexity. To enhance the security, separate LFSR sequences can be combined into a single pseudorandom string by using a combiner. In this thesis, creating a combiner function using another pseudorandom sequence derived from the chaotic motion of a double pendulum is investigated. Using the information from this driving function, an iterative process occurs whereby certain LFSR sequence blocks are selected and combined. The resultant sequences are sufficiently random as proven by the 15 tests adopted by the National Institute of Standards and Technology to evaluate the randomness of binary strings.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ASCII American Standard Code for Information Interchange

BBS Blum-Blum-Shub

LFSR linear feedback shift register

NIST National Institute of Standards and Technology

NPS Naval Postgraduate School

PRNG pseudorandom number generator

STS statistical test suite

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Cryptography is used constantly by billions of people and businesses around the globe. A key component of many of these cryptographic systems are pseudorandom number sequences.

Sequences generated by a linear feedback shift register (LFSR) are extremely fast, easy to generate, and contain many strong statistical randomness properties. However, a significant weakness is their linearity, allowing for algorithms to quickly and easily re-create the LFSR used to create them, given a small number of bits from the sequence. Combiners are used to solve this problem by combining multiple LFSR sequences in a non-linear fashion. Through this process the combiner seeks to destroy the linearity of LFSR sequences and thwart potential attacks while maintaining the statistical randomness properties that are inherent in the original sequences.

This thesis creates such a combiner in Python 3, one that uses a sequence generated by the motion of a double pendulum as its combining function. The combiner is designed in such a way that it concatenates combined subsequences from a selection of the original LFSR sequences. The breaking up of the larger LFSR sequences into subsequences further improves the linear complexity of the output sequences because when combining LFSR sequences in their entirety, there is a maximum linear complexity that can be achieved. This method can result in the output sequence being shorter than, equal to, or longer than the input sequence. Four subsequence block sizes are tested: two, four, eight, and 16. The National Institute of Standards and Technology (NIST)'s test suite is used to evaluate the statistical randomness for the input sequences and the output sequences generated by the combiner. For each trial, 100 sequences are tested using a significance level of 0.01.

The input sequences used to drive the combiner are actually non-random. The output sequences generated by the designed combiner are much better, and it is concluded that the resultant pseudorandom sequences appear to be random. Additionally, the linear complexity of the LFSR sequences that are being combined is completely destroyed.

The combiner created is effective in taking a proven non-random sequence and using it to combine pieces of multiple LFSR sequences to make a statistically random string for use as a pseudorandom sequence.

Acknowledgments

I would like to thank Commander Thor Martinsen for his fantastic guidance throughout the thesis process. At each stage, his expertise empowered me to go even further beyond the routine and into the extraordinary. I would also like to thank my second reader, Beny Neta, for his assistance in revising the thesis through its many drafts. Finally, I would like to thank Lieutenant Ryan Hard for his help in talking through roadblocks and providing as many data sets as I needed.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Technology that is used every day by people across the world has many complexities beneath the surface layer allowing it to do its job. One of those complex pieces, and perhaps among the most important, is the cryptosystem used in transmitting, receiving, and storing information in a secure manner. Company and personal data from password vaults to banking information and private messages all exist in the digital domain. To keep such data secure while maintaining its integrity and disclosing it only to authorized parties, various cryptographic methods are used. Cryptosystems are used 24 hours a day, 7 days a week and are largely transparent to the general public. Important requirements of cryptosystems include security, simplicity, and speed. In the digital age, security of data is essential.

The backbone of any worthwhile cipher is its randomness. To be useful, the cipher must be sufficiently random to prevent a third party from being able to break it. A second major factor is the speed of the cipher. Digital processes occur at a rapid pace and so too must the methods of security. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications is produced by the National Institute of Standards and Technology (NIST) and is the standard for measuring the probability that a sequence of ones and zeros are random. Using linear feedback shift register (LFSR)s as a method of generating sequences is quick and results in sequences with good statistical randomness properties. However, if given an adequate number of the sequence bits, they are easily broken by the Berlekamp-Massey algorithm [1]. A combiner can be used to capitalize on these positive qualities while making the output sequence less susceptible to attack. Such a combiner could be useful in taking a sequence with less than ideal statistical randomness properties as an input and creating a more useful output sequence for use as a pseudorandom sequence.

1.1 Background

Ciphers are far from a modern invention and have been used to convey messages in secret since the days of Julius Caesar—though they have grown far more sophisticated in the spanning centuries. Caesar's merely shifted the alphabet a few characters but, there have

since been hundreds of methods envisioned and implemented. The implications of a strong cryptographic system are clearly visible from the successes of Caesar and the German Enigma machines of World War II. However, the Caesar Cipher is now easily broken, and the Enigma Machine from World War II also succumbed to attacks and exposed German secrets.

This thesis explores a method of combining the simple sequences made from LFSRs into a stronger pseudorandom sequence. The advantages of a sequence generated by a single LFSR are that it is straightforward and easy to construct taking little computational power. However, the trade off is that such sequences are not cryptographically secure. While the resulting sequence has desirable statistical randomness properties, such as an even balance of ones and zeros, no repetition over a single period, and a desirable length and number of repeated bits, it has a very simple linear complexity profile that makes it insecure. By using a combiner, the strengths of these sequences can be used while mitigating the disadvantages. This thesis explores using a combiner based on another, potentially weak, random input sequence.

1.2 Motivation

Data is often obscured through encryption by combining plain text data with a cipher stream to render encrypted data that is then unintelligible if improperly decrypted or not decrypted at all. This part of the system is analogous to a lock on the data keeping the desired information secure. Most cryptosystems are largely open source, which is a major source of their strength. This transparency allows the community to attempt to break the cipher or find major faults that significantly hinder the system's effectiveness. Furthermore, this approach takes into account Kerckhoff's principle: "One should always assume the enemy knows the method being used" [2]. How, and in what order, the operations are carried out is defined and known; the public knows how the lock is constructed. The strength of the system then must come from the key of the system, which is dynamic and secret. Without this key, the lock, that is the cipher, cannot be opened. This changing key must be random to preclude predictability and unauthorized use. While there are methods of true randomness, they are often much slower than the speeds required by digital needs. Additionally, computers are discrete machines from which it is nigh impossible to generate true randomness. Thus, new ways must be invented to approximate random number generation.

CHAPTER 2:
Linear Feedback Shift Registers and Combiners

Chapter 2 explores the benefits and drawbacks of LFSRs, the one-time pad, and combiners. It also delves into the mathematical background of a LFSR and the resulting sequences. The chapter will discuss and use examples to illustrate the binary representation of characters, binary addition, and combiners.

2.1 Binary Representation of Information

Data in the digital domain is stored and transmitted as ones and zeros. Data is encoded using what is known as binary representation. Each one or zero is known as a bit, and to communicate, devices send and receive strings of bits, generally termed sequences. In order to share data, the data must first be converted from alphanumeric characters to a string of bits. For example, the American Standard Code for Information Interchange (ASCII), shown in Figure 2.1, provides for encoding characters into binary and back.

Figure 2.1. ASCII Table. Source: [3].

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

As an example, the name Alex would be represented by the bit string “01000001 01101100

01100101 01111000." Note that spaces were added between each binary representation for the sake of clarity in the example and would not normally be present in the string.

2.2 Binary Computations

With the data in binary, it is now possible to transmit and encrypt it using binary operations. The encryption method used for this thesis makes use of binary addition, also referred to as the "exclusive or" (XOR) operation, represented by "$\oplus$." In the binary environment, only the values zero and one exist. Therefore, $1 \oplus 1 = 0$, all other operations are as expected. Table 2.1 shows the XOR results for each possible combination and an example is shown in Figure 2.2.

Table 2.1. Exclusive or Operations

$\oplus$	0	1
0	0	1
1	1	0

Figure 2.2. Exclusive or Binary Vector Addition Example

$$\begin{array}{rr} & 10110001 \\ \oplus & 10010111 \\ \hline & 00100110 \end{array}$$

More generally, binary addition is conducted modulo 2. This allows us to conduct the addition of more than two digits at a time rather than XORing one pair at a time. This general form is shown in Equation 2.1.

$$x = (n_1 + n_2 + ... + n_k)(mod 2). \tag{2.1}$$

When adding two or more binary strings, each one is treated as a vector with the individual bits as the elements of the vector. The addition then occurs by adding the vectors, element by element, reduced modulo 2. While there are methods to add sequences of different lengths, this thesis will only add together sequences of the same length. Figure 2.3 gives an example of binary addition of three strings each of length four.

Figure 2.3. Addition Modulo 2

$$1010 + 1100 + 1011 =$$
$$x_1 = (1 + 1 + 1)(mod 2) = 1$$
$$x_2 = (0 + 1 + 0)(mod 2) = 1$$
$$x_3 = (1 + 0 + 1)(mod 2) = 0$$
$$x_4 = (0 + 0 + 1)(mod 2) = 1$$
$$1101$$

2.2.1 The One-Time Pad

The one-time pad, when properly used, is a completely secure method of encryption [2]. However, the drawbacks make it impractical to use for most purposes. As its name implies, to remain secure the cipher must be destroyed after use and never used again. A binary example will be used to highlight the major benefit of the XOR operator: its reversibility. The one-time pad works by XORing the unencrypted message (plaintext string) with a random encryption string of equal length, to create the encrypted message (ciphertext).

Then, after receipt, the same random string is XORed with the ciphertext to decrypt it and yield the original plaintext. As long as the random encryption sequence is never used again the one-time pad is secure. An example of this process is shown in Figures 2.4 and 2.5.

Figure 2.4. One-Time Pad Encryption

	A	l	e	x
	01000001	01101100	01100101	01111000
⊕	00111011	00000001	00110110	00010110
	01111010	01101101	01010011	01101110
	z	m	S	n

Figure 2.5. One-Time Pad Decryption

	z	m	S	n
	01111010	01101101	01010011	01101110
⊕	00111011	00000001	00110110	00010110
	01000001	01101100	01100101	01111000
	A	l	e	x

2.3 Linear Feedback Shift Registers

Linear Feedback Shift Registers (though deterministic) create sequences that have many beneficial statistical randomness properties and are used in many modern applications. In this thesis, LFSR sequences will serve as the components that the combiner uses to generate an pseudo-random sequence.

2.3.1 LFSR Structure

An LFSR can be realized using a feedback loop based on a polynomial or by using a recurrence relation. The polynomial of the form $x^4 + x + 1$ could also be written as the recurrence relation $x_i = x_{i+4} + x_{i+1}$. There are two types of LFSRs: Galois and Fibonacci. Given the same inputs they produce the same output sequence (the only difference being a shift in the sequence's starting point). For this thesis, a Fibonacci LFSR is used for ease of coding. A LFSR must be initialized with a non-trivial (non-zero) seed. Each cycle, the specified operations are carried out, the register is shifted to the next position in the direction of the arrows, and a bit is output. As a feedback loop, it can be modeled by Figures 2.6 and 2.7; the former depicts the LFSR populated with a seed, and the latter illustrates the subsequent clock cycle.

Figure 2.6. Initialized LFSR

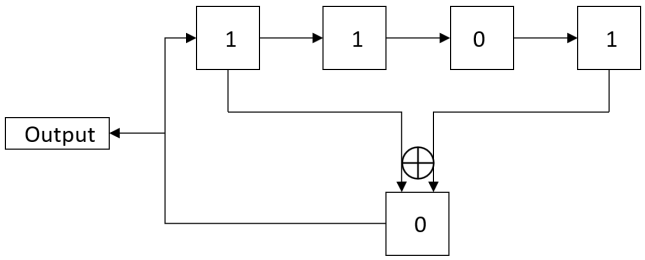
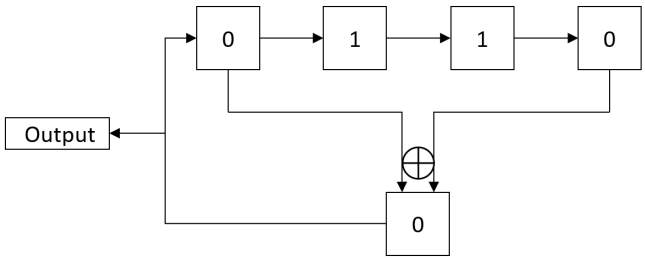


Figure 2.7. Shifted LFSR



Notice that the taps (input sources) are located in the same spot as the exponents/subscripts in the polynomial and recurrence relation forms. When it comes to LFSRs, not all are created equal. Those that are made using a primitive polynomial over GF(2) are better because they will always create a maximal length sequence (m-sequence) with a period of $2^n - 1$, where n in the span of the register, for any seed other than the trivial seed [4]. A primitive polynomial, f, is one in which

$$f \mid x^{p^n-1} - 1,$$

where p is the base field, and n is the order of the polynomial, and

$$f \nmid x^k - 1,$$

for all $k < p^n - 1$. Because of their ability to produce m-sequences, primitive polynomials are used to construct LFSRs in practice and in this thesis. Figure 2.8 shows the difference in using a non-primitive versus a primitive polynomial.

Figure 2.8. Non-Primitive versus Primitive Polynomial Sequences

	Non-Primitive	Primitive
Polynomial	$x^4 + x^2 + x + 1$	$x^4 + x + 1$
Seed	0001	0001
Resulting Sequence	1101000	111101011001000

2.3.2 LFSR Properties

Specific favorable properties that sequences made using LFSRs are balance (having an even distribution of ones and zeros), runs distribution (the number of repeated bit subsequences), and that they do not repeat over the length of one period. Additionally, they are very efficient, requiring a small amount of computing power, and can be quickly generated. Every sequence that is constructed using a primitive polynomial has an almost perfect balance of ones and zeros. For a primitive polynomial of degree n, the LFSR will generate a m-sequence with $2^{n-1} - 1$ zeros and 2^{n-1} ones; there will only be a single more one than zero [5]. A truly random sequence has a 50/50 chance of the next digit being either a one

or a zero. As the degree of a LFSR gets larger the balance in the sequence produced gets closer to this optimal value.

A run in a sequence of ones and zeros is a stretch of repeated bits. In a random binary sequence, the next bit will be either a one or a zero, each with a probability of one half. Therefore, a run of length n occurs with the a probability of $1/2^n$. A m-sequence generated by a polynomial of degree n will always have 2^{n-1} runs in each period [5]. Of these runs half will be of length 1, a quarter will be of length 2, and so on until there is one run of length (n-1) of zeros and one run of length n of ones. It is important that runs are not consistently too short or too long as such an occurrence could lead to predictability in the sequence.

A major cryptological downside of using LFSRs to generate sequences is that they are linear. A sequence made using a primitive polynomial has a linear complexity, L, equal to the degree of that polynomial. For example, the primitive polynomial $x^4 + x + 1$ has a linear complexity of four. Being linear makes the sequences easy to break and re-create using the Berlekamp-Massey algorithm and thus not secure on their own. Given a sequence generated by a LFSR, this algorithm can determine its linear complexity profile in 2L bits [1]. Ideally, a sequence would have a linear complexity of greater than half its length to be immune to the Berlekamp-Massey algorithm, or a linear complexity large enough to render the algorithm computationally infeasible.

Overall, of the 15 tests for randomness, a sequence generated by a LFSR passes 12 of them, failing the Spectral Test, the Binary Matrix Rank Test, and, predictably, the Linear Complexity Test. This is where combiners come in—to mitigate this weakness while maintaining the strengths of LFSR sequences.

2.4 Combiner Principles

A combiner is a mechanism that receives two or more sequences—not necessarily generated by LFSRs—as inputs and outputs a single sequence based on a chosen methodology. The most basic case is using binary addition on two different sequences to create a new string of bits.

Combiners, used effectively, can hide or obscure the input pieces and make the result seem more random. A more sophisticated combiner might take longer to combine sequences, but

typically, the cryptological strength of the output sequence is increased. In this thesis, a fairly straightforward combiner is used to destroy the linearity of the LFSR sequences while attempting to retain their statistically random properties.

As an example, suppose there is a combiner that utilizes three sequences: an input sequence driving the combining process, and two sequences that will be combined, labeled A and B. If the input sequence is a one, the next 4 bits of the two sequences will be combined using binary addition. If it is a zero, they will not. Figure 2.9 illustrates this example. A combiner similar to this will be used for this thesis and explained more in detail in Chapter 3.

Figure 2.9. Example Combiner

Input Sequence: 01 Sequence A: 00111011 Sequence B: 01000001

Output: 1010

Because the first digit of the Input Sequence is 0 and the second is a 1, the first four digits of A and B are not combined but the second group of four is.

The proposed combiner attempts to take an input sequence that does not necessarily have good statistical randomness properties and use it as a driving function to combine LFSR sequences. This method looks to capitalize on the beneficial properties of LFSR sequences to improve a non-random or "weakly" random input sequence and at the same time use the input sequence to decimate the linearity associated with LFSR sequences.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Methodology

Chapter 3 describes the combiner in detail as well as a short description of the Monobit Frequency and Linear Complexity tests. It also briefly explains the code used and the origin of the input sequence. It goes into further detail on how the LFSRs were chosen and how and why the testing parameters were chosen.

3.1 Code

Coding was done in Python 3 (Version 3.7.1) due to familiarity with the language and pre-existing scripts that could be adapted for the purposes of this thesis. The code used to generate LFSR sequences was created by [6] and is used as written. It is a simple code that takes as its input the power representation of the desired polynomial (e.g., $x^4 + x + 1$ as 4,1) and a corresponding seed of appropriate length (e.g., 0001).

The main code is self-written and is the focus of this thesis. It is the combiner that reads the input, fuses the LFSR sequences as directed to by the input sequence, and returns the final output sequence. This provides the polynomials and seeds for generating the LFSR sequences and is written in such a way that it checks for the trivial seed (00...0) and creates a new seed if the trivial seed is obtained. It also provides easy modification of the window size, described in Section 3.3, output bit length desired, and the number of sequences being created and tested. Following the generation of each sequence, the sequence is run through the NIST statistical test suite (STS) and the results are stored in a Python dictionary. This repeats until all sequences are created and tested. Finally, the total number and proportion of sequences that passed the tests are computed and uniformity of the p-values is analyzed per [7]. The results are averaged for ease of plotting and plotted. This code can be found in Appendix B.

The double pendulum input sequences are generated in MATLAB using code written by [8]. The code to generate the Blum-Blum-Shub (BBS) input sequences is self-written. It creates two large Blum primes, an appropriate seed, and uses the least significant bit to generate the sequence. This code can be found in Appendix B.

The code used to execute the NIST STS is written by Gerhardt [9]. Because it was originally written for Python 2 and multiple small errors were discovered, it has been modified to work correctly using Python 3. The NIST documentation found in [7] includes five benchmark sequences and corresponding results. The modified code gives identical results for three out of the five benchmark sequences, two could not be tested due to issues reading the associated files.

The code used to create a visual representation of linear complexity is written by Sachs [10]. It runs the Berlekamp-Massey algorithm on the selected sequence and plots the results as a graph of Linear complexity versus Bit Length.

3.2 Randomness Tests

There are 15 tests that make up the NIST STS. They are listed in Table 3.1. Each test analyzes a property that a random string of bits is expected to have; all have one test with the exceptions of the Serial Test which has two sub-tests, the Random Excursions Test which has eight sub-tests, and the Random Excursions Variant Test which has 18 sub-tests. Every test in the suite is a statistical test using the null hypothesis that the sequence is not random. The results yield the probability that a test is random. A sequence is not necessarily random just because it passes all the tests. For example, sequences generated by a LFSR pass 12 of the 15 tests, but they are definitively not random. On the other hand, if a sequence fails one or even a couple of tests, it does not mean that it is not random. A p-value of 1.0 would indicate that the sequence appears perfectly random, while a value of 0.0 would indicate complete non-randomness. The tests with multiple parts return a p-value for each part. The rationale and mathematical details for each test can be found in [7].

Table 3.1. List of Random Number Generation Tests. Adapted from [7].

Monobit Frequency	Overlapping Template Matching	Linear Complexity
Block Frequency	Non-Overlapping Template Matching	Binary Matrix Rank
Runs	Random Excursions Variant	Cumulative Sums
Spectral	Approximate Entropy	Random Excursions
Serial	Maurer's "Universal Statistic"	Longest Runs

3.2.1 Monobit Frequency Test

This test evaluates the balance of ones and zeros over the entire sequence. A random sequence is expected to produce a zero or one with probability one half. In other words, half of the bits in the sequences are expected to be ones and the other half zeros. If the distribution is skewed towards a certain value, the next bit in the sequence is more likely to be that value, and thus make it less random. This test is the most important to pass. "All subsequent tests depend on the passing of this test" [7].

3.2.2 Linear Complexity Test

The Linear Complexity test works by applying the Berlekamp-Massey algorithm [1] to the sequence in question and determines the shortest LFSR required to re-create it. The longer a LFSR needs to be to generate the sequence, the more random it appears. Because the combiner works using LFSR sequences, this test is of special interest. The Python code written by Sachs [10] will be used to further analyze the linear complexity of the generated pseudo-random sequences.

3.3 Combiner in Detail

The combiner for this thesis combines eight m-sequences, each of which is generated by a primitive polynomial LFSR. Each polynomial is chosen to be primitive to ensure a maximum period for each resulting sequence. Eight m-sequences are used to provide a sufficient number of sequences to draw from. In a static combiner that relies solely on binary addition to combine entire sequences, the linear complexity of the final sequence is at most the sum of the linear complexities of each individual sequence, as shown in Equation 3.1,

$$L_C \leq L_1 + L_2 + ... + L_i, \tag{3.1}$$

where L_C is the combined linear complexity and L_i is the linear complexity for each sequence being combined though binary addition [11]. Equality occurs if and only if the minimal polynomials are coprime. By definition, because each of the selected polynomials is primitive, they are all coprime over GF(2). For example, suppose three sequences generated

by primitive polynomials are being combined with the following linear complexities: $L_1 = 5$, $L_2 = 6$, and $L_3 = 7$ then $L_C \leq 5 + 6 + 7 = 18$. Using eight sequences also mitigates the potential severity of compression of the input sequence. The LFSR sequences are referred to as Sequence 1 through Sequence 8, and are static in their order.

When combining sequences in this thesis, the combiner is not combining whole sequences. Instead, a portion, that will be named a window, is taken into consideration and potentially combined. Once a window is considered and used (or not used), it then slides to the next set of bits in the sequence and a new window is considered. This continues until the final output string is constructed. If during this process, the window reaches the end of a sequence it will circle back to the beginning and begin anew. This repeats as necessary until the output sequence is complete. This window size is set by the user and is consistent across each of the eight sequences. Figure 3.1 shows an example with a window size of six.

Figure 3.1. Window Example

Sequences	Window 1	Window 2	Window 3
1110100	111010	011101	001110
111101011001000	111101	011001	000111
0001111100110100100	000111	110011	010010

The first sequence must circle back to the beginning to create the second window and again to create the third. The second sequence must circle back to create the third window.

An input string of ones and zeros is used by the combiner to determine which windows to combine. This input string is divided into blocks of length M, where M is the number of sequences available for combination—eight for this thesis. This process in demonstrated in Figure 3.2. If the input sequence is not divisible by M, then the final block of length less than M is discarded.

The first block of the input sequence is examined to determine the indices that contain a one. For each index that contains a one, the corresponding sequence will be selected, and the current window combined with other selected windows via addition modulo 2. Then the next input block is used, the window moves to the next position, and the process repeats. If the input is all zeros, nothing is combined and the window slides to the next position. Each result is appended to the previous segment to create the overall output sequence. In

Figure 3.2. Input Blocks, M=8

Input Sequence: 10010110101001001010101O...10101110

↓

10010110

10100100

10101010

.

.

.

10101110

the event that the input of all zeros occurs, nothing is appended to the previous segment (as opposed to appending a string of zeros). This is important to ensure excess zeros are not added thereby resulting in an unbalanced sequence. The process repeats until the desired output sequence length is obtained or the input sequence is depleted. An example of the process is shown in Figures 3.3 and 3.4.

Figure 3.3. Indexing Example Part 1

Input Block	Current Window	Selected Windows	Combiner Output
	00011111	00011111	
	11010101	Not Selected	
	10011101	Not Selected	
10010110	01001001	01001001	10111000
	01101010	Not Selected	
	10011001	10011001	
	11110111	11110111	
	01111111	Not Selected	

This method results in a potential shortening or expansion, in regard to length, of the output sequence as compared to the input sequence. The maximum size of the output sequence is bounded by the ratio of window size to input sequence block size,

$$N_o \leq \frac{W}{M}, \tag{3.2}$$

where N_o is the output sequence length, W is the window size, and M is the input block size, determined by the number of LFSRs. Equality occurs if and only if no input block is all zeros.

Figure 3.4. Indexing Example Part 2

Input Block	Current Window	Selected Windows	Combiner Output
	00110100	00110100	
	10011011	Not Selected	
	11010010	11010010	
10100100	10000001	Not Selected	10111000 00110111
	01010011	Not Selected	
	11010001	11010001	
	01011110	Not Selected	
	01111100	Not Selected	

3.3.1 Input Data

The first set of input sequences is generated by the motion of a double pendulum. Analyzing the energy levels of the masses at defined intervals determines if a one or zero is output as the next bit in the sequence. Further discussion on the construction of the double pendulum and the processing used to create the sequences can be found in [8]. A second set of input sequences is generated using a BBS pseudorandom number generator (PRNG) [12].

From both the double pendulum and BBS, sequences of length greater than four million are created and used as inputs to the combiner. The pendulum conditions under which the sequences are generated are listed in Table 3.2. Sequences of length greater than four million are taken to ensure the resulting length from the combiner will reach one million bits, which is the NIST recommended length sequences should be for testing purposes.

Table 3.2. Pendulum Conditions. Adapted from [8].

Variable	Value
θ_1	$\pi/2$
θ_2	Variable
m_1	1 m
m_2	1 m
l_1	1 kg
l_2	1 kg
g	9.8 m/s^2

3.4 Experimental Procedure

Four trial runs are conducted with the window size as the only independent variable. Window size was chosen because of its impact on the output length (see Equation 3.2). Due to the way the combiner is constructed, and the statistical properties of m-sequences, there is high confidence that the generated sequences will pass the battery of statistical tests. Therefore, the window sizes are varied to determine if there is a significant difference in randomness from the small window sizes to the larger. Considerations of manipulating the number of LFSRs and specific polynomials used are left as topics for future research; the insight gained from such changes would be minimal without a prior understanding of the base case. Additionally, limiting the experiment to one independent variable allows for a better understanding of the accompanying results. Window sizes of two, four, eight, and 16 are the chosen test values. Two is the smallest window size desired because a window size of one would require twice as many input bits (8 million) and take at least twice as long to generate the output sequence. The window size of eight is of interest because of the potential for no compression from the input to the output sequence. The largest window size of 16 is over half the size of the shortest LFSR sequence (31), and will require less input bits than the desired number of output bits. The results from each window size will provide insight into the effect on randomness of two key ratios: the ratio of window size to the number of LFSRs, shown in Equation 3.2, and the ratio of window size to minimum sequence length, which is

$$\frac{W}{N_s},$$

where W is the window size and N_s is the smallest LFSR sequence length.

For each trial, 100 sequences of length one million bits are made using the combiner and then tested. The choice of 100 is derived from the selected significance level, α, of 0.01 and also yields an easy interpretation that 1 out of 100 is expected to fail. The length of one million bits is the minimum recommended sequence length for evaluating Linear Complexity, Random Excursions, and Random Excursions Variant Tests. These decisions are based on guidance from the NIST [7].

3.4.1 LFSR Selection and Use

As described in 3.4, the number of LFSRs is chosen to be eight to minimize potential decimation of sequence length from input to output. With eight LFSRs, a block of eight zeros from the input sequence is required to have nothing appended to the output sequence and so reduce its overall length. Having fewer than eight LFSRs would require less and thus be more likely to occur. Greater than eight would likely enhance randomness, but would also be more computationally cumbersome and slower.

The eight LFSRs chosen are listed in Table 3.3 in polynomial form. The eight associated polynomials chosen are all primitive polynomials and coprime to one another. They are also relatively small in size—the smallest has length 31 and the largest has length 32767—to reduce computational time in creating them.

Table 3.3. LFSR Polynomials. Adapted from [13].

$x^5 + x^2 + 1$	$x^9 + x^4 + 1$
$x^6 + x + 1$	$x^{12} + x^6 + x^4 + x + 1$
$x^7 + x + 1$	$x^{13} + x^4 + x^3 + x + 1$
$x^8 + x^4 + x^3 + x^2 + 1$	$x^{15} + x + 1$

Throughout the experiment the polynomials displayed in Table 3.3 are used. Within each trial, for every one of the 100 sequences, a new seed is generated for each LFSR, resulting in a new set of LFSR sequences. Because the structure of the LFSRs is not changing, the sequences generated will be a shifted version of the previous iterations - see Figure 3.5. For a polynomial of degree n, there are $2^n - 1$ unique, non-trivial seeds. Although the sequences are only shifted, it is extremely unlikely that all are shifted by the same amount in the same direction; thus, if given the same input strings, the output strings will be different. Across the trial runs, identical LFSR sequences are used — the first sequence for the trial run of window size two is created using the same LFSR sequences as the first sequences generated for trial runs of window sizes four, eight, and 16. This methodology aids in determining the effects solely due to a changing window size and fulfills the consistency assumption of [7]. The same 100 input sequences from [8] and the same 100 input sequences from the BBS generator are used for all four trials for the same reasons.

Figure 3.5. Variable Seed Effect on LFSR Sequence

Primitive Polynomial: $x^3 + x + 1$		
Seed	001	010
Resulting Sequence	1110100	0111010

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

Chapter 4 presents the NIST STS results for the input sequences and each trial run. These empirical results are evaluated by examining the proportion of sequences that passed, and the distribution of p-values to arrive at a conclusion of their statistical randomness. Lastly, the linear complexity profile is calculated and plotted to show the improvements over a standard combiner of LFSRs.

4.1 NIST Statistical Test Suite

The results of each individual test are averaged and graphed using a logarithmic scale for the y-axis for the input sequence and each trial run. There is also a "Significance Level" line for ease of interpretation—bars above that line indicate the trial likely passed that test. While the input sequences are greater than four million bits in length, only the first one million bits are tested. This is to provide a parallel to the results generated by the combiner and because the time required to run the test suite increases greater than linearly with bit length. Due to almost identical graphs for each window size, when using the double pendulum for the input sequences, only the graphs for the input sequences and the window size equal to two are in this section. Similarly, when using BBS for the input sequences, only when the window size equals 16 is there a noticeable difference, and so only the input results and window size equal to 16 results are shown in this section. All graphs for the results can be found in A.1.

4.1.1 Average P-Values

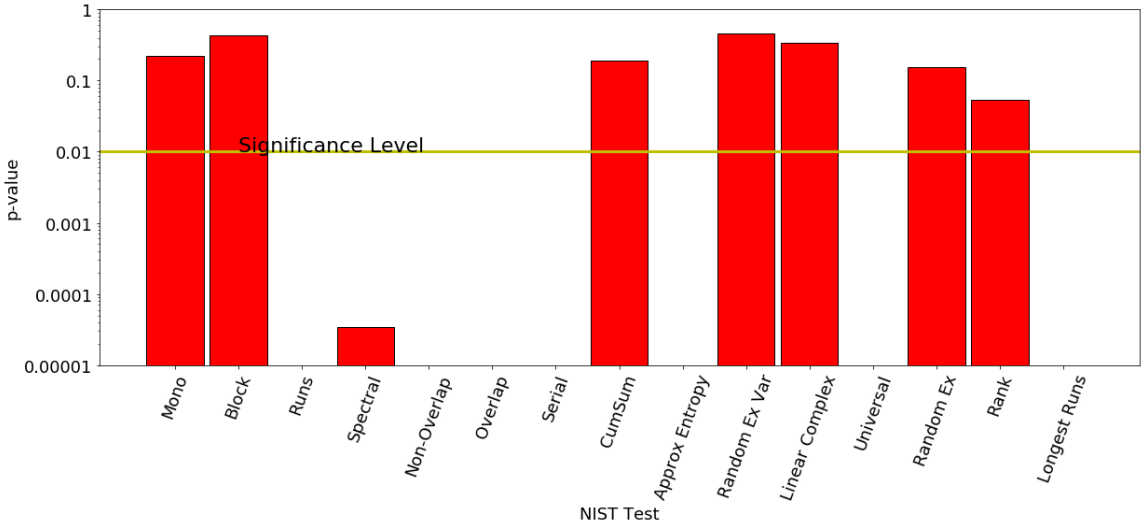
The average of the p-values provides a quick look into the performance of the sequences on each individual NIST test. While it is safe to assume that if the average p-value is less than the chosen significance level that the test fails, the opposite is not true. The p-values provide the data to be evaluated and, following the evaluation, a statement can be made as to the statistical randomness of the sequences generated. To conclude whether the generated sequences are statistically random, the proportion of sequences that pass and the distribution

of p-values must be analyzed to interpret the empirical results [7]; this is done in Sections 4.1.2 and 4.1.3.

Double Pendulum Input Sequence P-Values

Across the board, the input sequences created by the double pendulum perform poorly with respect to the NIST STS. The average p-value for each test is only above the chosen significance level for seven of the 15 tests as shown in Figure 4.1. Of the eight that fail, seven have a p-value of 0.0 indicating a complete statistical lack of randomness. A more detailed analysis of the randomness properties of sequences generated by a double pendulum can be found in [8]. For the purposes of the combiner being tested in this thesis, the failure of the double pendulum sequences is beneficial as it allows for the predicted improvement from input sequence to output sequence to be observed. Of some concern is the pass rate of the Monobit Frequency Test. If the case is that there are too many zeros then the output sequence could have significant reduction in length which is undesirable. While not directly tied to passing the Runs Test, an input sequence with long runs of zeros and ones is detrimental to the combiner. The long runs of zeros would cause severe decimation in the size of the output sequence and the long runs of ones would combine every sequence and have the appearance of a static combiner and reduce the linear complexity.

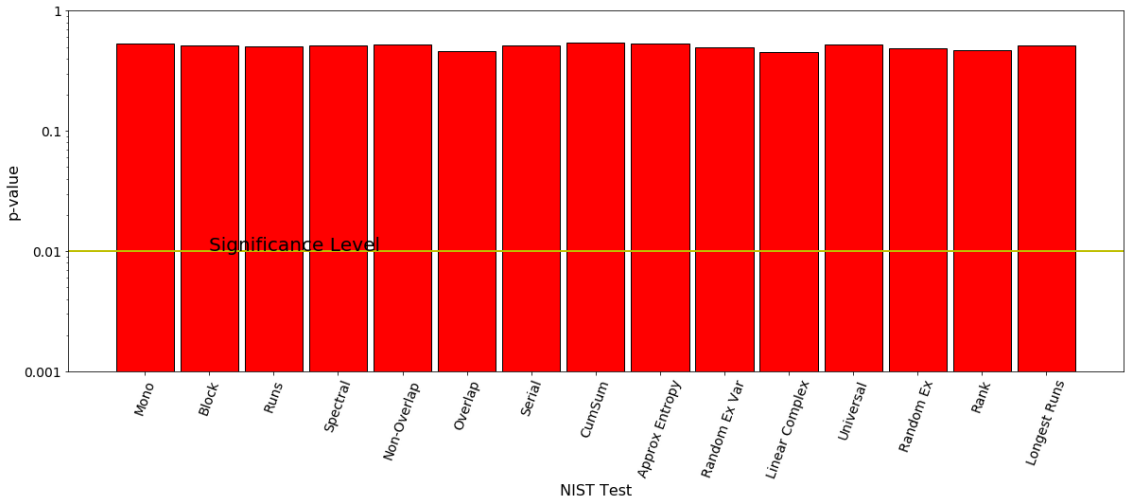
Figure 4.1. NIST STS Results, Double Pendulum



Blum-Blum-Shub Input Sequence P-Values

The input sequences generated using BBS perform well, as expected. The average p-value for all the tests is well above the chosen significance level. Using statistically random input sequences in addition to those generated by the double pendulum allows for comparison of the output sequences based on the two input sequence generation methods.

Figure 4.2. NIST STS Results, Blum-Blum-Shub



Combiner P-Values

For each window size tested, the average of every NIST test is above the α of 0.01. Figure 4.3 shows the graph for the double pendulum input with a window size equal to two, but the information in it is nearly identical to that of window sizes four, eight, and 16. This provides a compelling indication that the sequences generated, for each tested window size and a double pendulum input sequence, can be considered statistically random. These strong results are anticipated due to the statistical properties of the m-sequences and the combiner design. When using BBS for the input sequences, the results are almost identical, with the exception of when the window size is 16 as shown in Figure 4.4. This is likely due to the large window size coupled with the number of trivial input blocks. Over the course all the four million bit input sequences made using BBS there are 196,064 instances of a trivial input block. When using a window size of 16 there are 24,456 trivial input blocks for an average of 244 per input sequence.

Figure 4.3. NIST STS Results, Double Pendulum, Window Size = 2

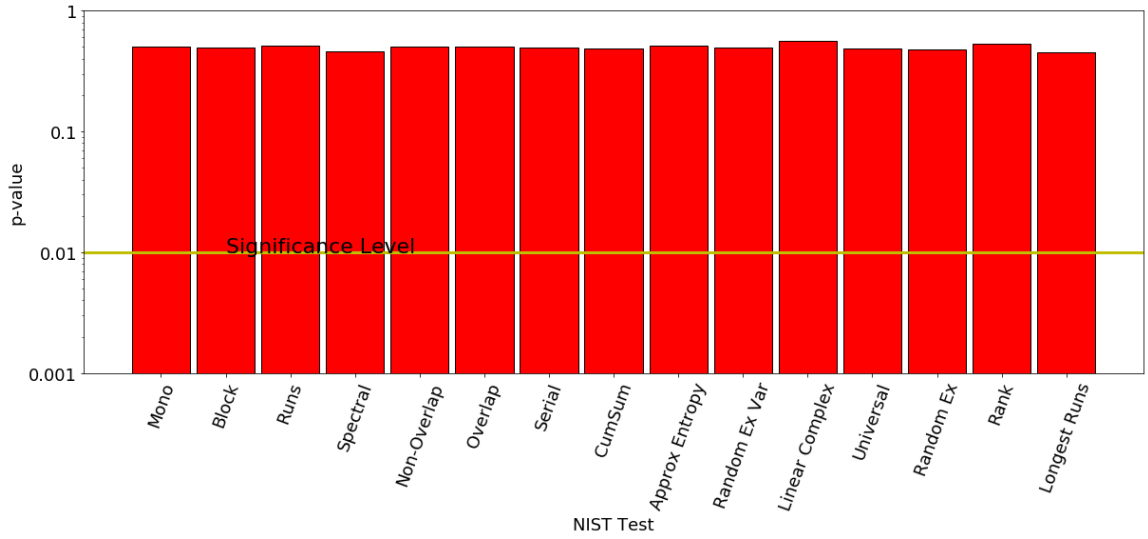


Figure 4.4. NIST STS Results, Blum-Blum-Shub, Window Size = 16

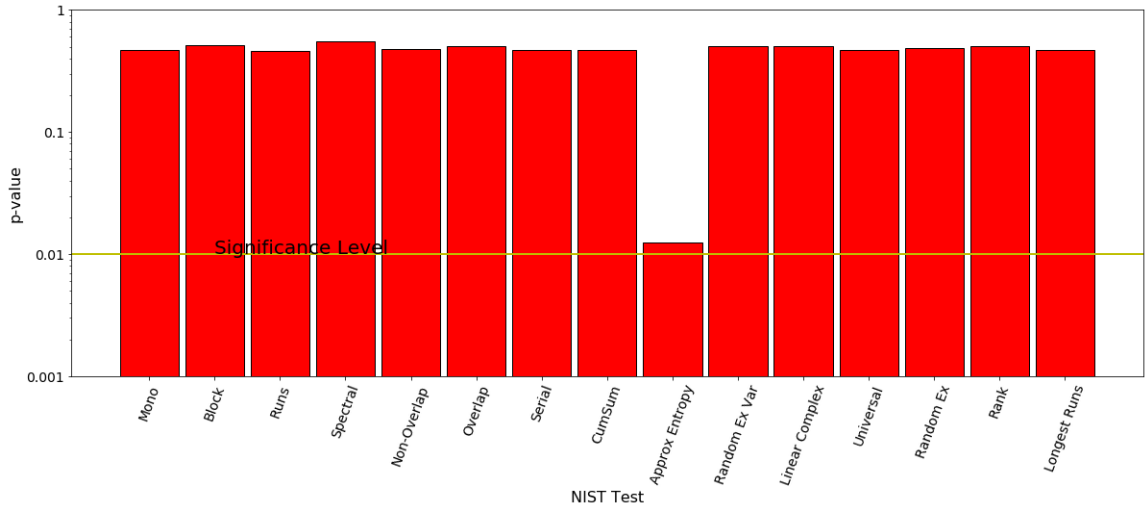


Table 4.1 gives a breakdown of the number of sequences that passed each individual NIST test. Recall that when choosing a significance level of 0.01 and testing 100 sequences, one is expected to fail. In this context, a window size of two performed the best followed by a window size of 16, then eight, and lastly four. The expectation was that a window size

of two would perform the best because it is breaking the LFSR sequences used into the smallest chunks and thus introducing more variability and unpredictability. Using the same logic, it was anticipated a window size of four would perform better than a window size of eight, which would perform better than a window size of 16.

Table 4.1. Number of Sequences that Passed Each Test

Window Size	Pendulum	2	4	8	16
Mono	63	99	98	97	99
Block	51	100	98	100	98
Runs	0	99	100	99	98
Spectral	0	100	98	99	100
Non-Overlap	0	99	98	100	99
Overlap	0	99	98	98	99
Serial	0	99.5	99.5	98	99
CumSum	57	97	98	97	100
Approx Entropy	0	99	98	100	100
Random Ex Var	89.61	98.77	98.27	98.88	98.77
Linear Complex	78	100	99	97	98
Universal	0	99	97	100	98
Random Ex	37.38	98.75	98.38	98.63	98
Rank	21	99	98	99	99
Longest Runs	0	100	99	100	99
Total (out of 1500)	396.99	1487.02	1475.15	1480.51	1482.77

Decimals are due to those sequences that have multiple sub-tests being normalized to 100.

4.1.2 Proportion that Pass

The analysis of the proportion of sequences that pass the tests is vital to ensure that the average p-value is not comprised of a few very high values hiding a significant number of failing tests. A worst case scenario for this would be one sequence having a p-value of 1.0 for a certain test and the remaining 99 having a p-value of 0.0. The average p-value would indicate overall passing, though just barely, at 0.01. This extreme case is highly unlikely, but highlights the potential for bad results to be covered up by good results if

improperly analyzed. The results from the double pendulum input sequences highlight how misleading only looking at the average p-values can be. For the proportions test, the proportion of sequences that passed with a p-value ≥ 0.01 should be above the confidence interval minimum calculated using Equation 4.1

$$\hat{p} \pm \sqrt{\frac{\hat{p}(1-\hat{p})}{m}}, \tag{4.1}$$

where $\hat{p} = 1 - \alpha$ and m is the sample size [7]. Table 4.2 shows "Pass" if the proportion is above the threshold and "Fail" otherwise for each test and window size as well as the input sequences. The tests for the double pendulum sequences predictably fail based on the raw number of sequences that individually passed each test. Based on this, there is sufficient evidence to show that the input sequence is not random. The BBS sequences show much better randomness properties only failing the Block and Random Excursions Variant Tests. Sequences constructed using the double pendulum sequences as an input and with a window size of two, four, and 16 pass all of the proportionality tests fulfilling one the methods of interpretation. Although the sequences with a window size of four fail the proportionality approach for the Approximate Entropy Test, it passes all the others. Using the BBS sequences as an input, has at least one failure for each window size with the exception of when the window size equals four. Further analysis is done on the uniformity of p-values before drawing a conclusion.

Table 4.2. Proportion Test

	Double Pendulum					Blum-Blum-Shub				
Window Size	Input	2	4	8	16	Input	2	4	8	16
Mono	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Block	Fail	Pass	Pass	Pass	Pass	Fail	Pass	Pass	Fail	Pass
Runs	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Spectral	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Non-Overlap	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Overlap	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Serial	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
CumSum	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Approx Entropy	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Fail	Fail
Random Ex Var	Fail	Pass	Fail	Pass	Pass	Fail	Fail	Pass	Fail	Pass
Linear Complex	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Universal	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Random Ex	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Rank	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Longest Runs	Fail	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass

4.1.3 Distribution of p-Values

For the same reasons as the proportions test, the distribution of p-values is analyzed for uniformity to interpret the empirical results. To test the p-values for uniformity they can either be graphed as a histogram and visually inspected or have a χ^2 test applied to them to get a "P-value of the P-values" [7]. It was decided to run the χ^2 test to minimize subjectivity by producing numerical results that either pass or fail. The p-values are defined to be uniformly distributed if the resulting p-value from the χ^2 test is ≥ 0.0001. Table 4.3 displays the results for each test and window size using the double pendulum as the input source. Again, all of the sequences from the double pendulum fail, confirming their non-randomness. For the corresponding output sequences those with window sizes two, four, and eight pass all of the tests for uniform p-value distribution. As shown in Table 4.4,

the only failure is the Random Excursions Variant Test when the window size is 16. When using BBS as the input sequences the only failure is the Approximate Entropy Test when the window size is 16.

Table 4.3. p-Value Uniformity: Double Pendulum

Window Size	Pendulum	2	4	8	16
Mono	0.0	0.36692	0.98345	0.10879	0.98790
Block	0.0	0.47499	0.10253	0.63712	0.41902
Runs	0.0	0.59555	0.17187	0.01672	0.49439
Spectral	0.0	0.49439	0.94631	0.12962	0.09094
Non-Overlap	0.0	0.02355	0.41902	0.33454	0.09658
Overlap	0.0	0.57490	0.71975	0.67869	0.69931
Serial	0.0	0.85138	0.05194	0.81654	0.74988
CumSum	0.0	0.21331	0.67869	0.36692	0.24928
Approx Entropy	0.0	0.03757	0.43727	0.63712	0.81654
Random Ex Var	0.0	0.09216	0.01427	0.10987	0.00003
Linear Complex	0.0	0.55442	0.69931	0.75976	0.17187
Universal	0.0	0.16261	0.15376	0.09094	0.97807
Random Ex	0.0	0.19298	0.92100	0.40561	0.01919
Rank	0.0	0.30413	0.94631	0.43727	0.41902
Longest Runs	0.0	0.06688	0.26225	0.27571	0.43727

Since the window sizes of four and 16 made using the double pendulum input sequences have a failure in the evaluation of p-values in one method but not the other, another set of experiments will be conducted on each using different samples generated by the combiner to arrive at a conclusion of their statistical randomness [7]. Due to the number of tests that fail when using the BBS input sequences, and the fact that it is the same test for each method, additional testing will not be done with the combiner using BBS input sequences.

Table 4.4. p-Value Uniformity: Blum-Blum-Shub

Window Size	BBS	2	4	8	16
Mono	0.13728	0.21331	0.59555	0.92408	0.85138
Block	0.23681	0.40120	0.81654	0.21331	0.71975
Runs	0.63712	0.61631	0.81654	0.05194	0.36692
Spectral	0.59555	0.85138	0.75976	0.55442	0.43727
Non-Overlap	0.04567	0.28967	0.02882	0.88317	0.23681
Overlap	0.89776	0.97170	0.26225	0.33454	0.86769
Serial	0.38383	0.11881	0.22482	0.00135	0.23076
CumSum	0.19169	0.79814	0.30413	0.92408	0.31908
Approx Entropy	0.65793	0.38383	0.75976	0.38383	0.0
Random Ex Var	0.18600	0.00336	0.08472	0.29443	0.61053
Linear Complex	0.43727	0.03517	0.71975	0.99944	0.88317
Universal	0.22482	0.89776	0.51412	0.99425	0.95583
Random Ex	0.28613	0.51661	0.52661	0.06482	0.02810
Rank	0.09658	0.36692	0.92408	0.45594	0.73992
Longest Runs	0.77919	0.65793	0.02055	0.49439	0.89776

4.1.4 Additional Numerical Experiments

For these secondary tests, everything is kept the same with the exception of the double pendulum input sequences. Two sets of additional tests are run to reach a conclusion. Tables 4.5 and 4.6 display the results of the additional tests for window sizes four and 16, respectively. For the first additional test, the window size of 16 fails the proportionality test for the Approximate Entropy Test. However, both window sizes pass all of the remaining tests.

Table 4.5. Additional Tests for Window Size 4

	Additional Test 1		Additional Test 2	
Mono	Pass	0.41902	Pass	0.38383
Block	Pass	0.10879	Pass	0.61631
Runs	Pass	0.49439	Pass	0.28967
Spectral	Pass	0.33454	Pass	0.55442
Non-Overlap	Pass	0.26225	Pass	0.16261
Overlap	Pass	0.43727	Pass	0.31908
Serial	Pass	0.21331	Pass	0.66832
CumSum	Pass	0.24928	Pass	0.69931
Approx Entropy	Pass	0.69931	Pass	0.75976
Random Ex Var	Pass	0.28888	Pass	0.25141
Linear Complex	Pass	0.31908	Pass	0.55442
Universal	Pass	0.19169	Pass	0.12233
Random Ex	Pass	0.37110	Pass	0.46066
Rank	Pass	0.67869	Pass	0.05194
Longest Runs	Pass	0.13728	Pass	0.79814

Proportionality Test and p-Value Uniformity

Table 4.6. Additional Tests for Window Size 16

	Additional Test 1		Additional Test 2	
Mono	Pass	0.67869	Pass	0.67869
Block	Pass	0.40120	Pass	0.65793
Runs	Pass	0.00076	Pass	0.35049
Spectral	Pass	0.26225	Pass	0.45594
Non-Overlap	Pass	0.86769	Pass	0.92408
Overlap	Pass	0.59555	Pass	0.99633
Serial	Pass	0.25570	Pass	0.84294
CumSum	Pass	0.79814	Pass	0.63712
Approx Entropy	Pass	0.91141	Pass	0.71975
Random Ex Var	Fail	0.82944	Pass	0.82353
Linear Complex	Pass	0.93572	Pass	0.73992
Universal	Pass	0.35049	Pass	0.23681
Random Ex	Pass	0.26892	Pass	0.18657
Rank	Pass	0.83431	Pass	0.55442
Longest Runs	Pass	0.07572	Pass	0.61631

Proportionality Test and p-Value Uniformity

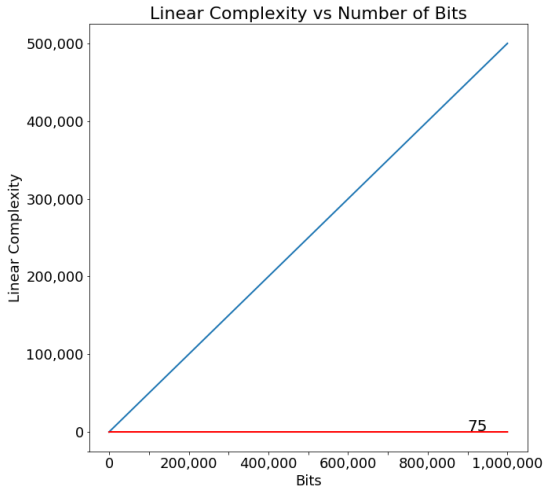
4.2 Linear Complexity

The linear complexity of the combiner's input m-sequences is very small. Of the chosen sequences, the smallest linear complexity is 5 while the largest linear complexity is 15. Based on Equation 3.1 if the sequences were combined using addition the resulting linear complexity would be 75, and very quickly and easily broken by Berlekamp-Massey. The output sequences generated all have a linear complexity that continues to grow as the sequences grows—the Berlekamp-Massey algorithm was unable to find a LFSR that could generate the output sequences. Figure 4.5 shows the linear complexity of the combined LFSR sequences just above the axis in red at 75 and the continually increasing linear

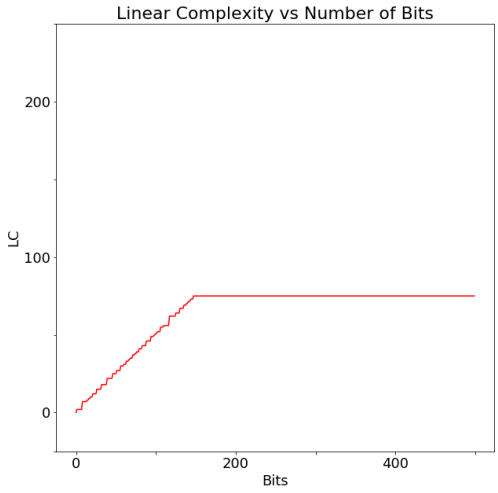
complexity of an output sequence from the combiner in blue. The linear complexity profile shown in blue in Figure 4.5 is representative of the linear complexity profile for each of the output sequences when using both input sequence generation methods.

Figure 4.5. Linear Complexity Profiles of Combined LFSRs

(a) Thesis Combiner and Standard Combiner



(b) Close up of Standard Combiner



4.3 Additional Results

There are some input sequences that result in a complete failure of the combiner. One such sequence comes from the imaging method of extracting ones and zeros from the double pendulum [8]. This input sequence is made up of very long runs of ones followed by very long runs of zeros. The runs of ones result in an inadequate mixing of the LFSR sequences and a low linear complexity. The runs of zeros cause nothing to be combined which severely shortens the output sequence length and would require an impractical input sequence size to generate the desired output sequence length.

An idea that was attempted and did not work was changing the window size as the combiner is running to equal the number of indices selected by the input sequence.

CHAPTER 5:
Conclusion and Future Work

This thesis developed and tested a novel combiner that combines subsequences of multiple LFSR sequences using a separate input sequence. While the input sequences made using the double pendulum performed poorly, with the conclusion that there is significant evidence of non-randomness, the output sequences generated by the combiner demonstrate that they are statistically random. When using the BBS input sequences there is evidence that the output sequence is statistically random using a window size of four, but not for the other window sizes. Additionally, when using the double pendulum input sequences there were no trivial input block occurrences, but when using the BBS input sequences there were an average of 1,961 trivial input blocks per sequence. This likely led to the increased failure rate, but is more indicative of the results when the input is statistically random. The combiner succeeded in the goals of using a statistically non-random input sequence to reconstitute portions of LFSR sequences and destroy their linearity while retaining their statistical randomness properties. Although additional sequences had to be made for testing window sizes four and 16 they ultimately passed and without further testing there is insufficient evidence to conclude that one window size produces a sequence that is more random than another. What is known, is that creating a sequence using a window size of 16 is much quicker and requires a shorter input sequence than the output sequence that is generates.

5.1 Future Work

There is more work to be done using this combiner to fully understand its behavior. For example, how big can the window size be made before the randomness properties begin breaking down? Is there a connection between the window size and the shortest LFSR sequence used and the randomness properties? How many LFSRs should be used to balance speed and randomness? Are certain polynomials or a specific combination of polynomials more efficient? The input sequences used in this thesis worked out well, but are there better ones? Are there worse ones?

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Statistical Test Suite Plots

The plots contained in this Appendix show the average p-value of the results for each test and the chosen significance level.

A.1 Result Plots

These plots give a general idea as to the performance of the specified sequences when run through the NIST STS. It is desired for the p-value to be above the chosen significance level.

Figure A.1. NIST STS Results, Double Pendulum

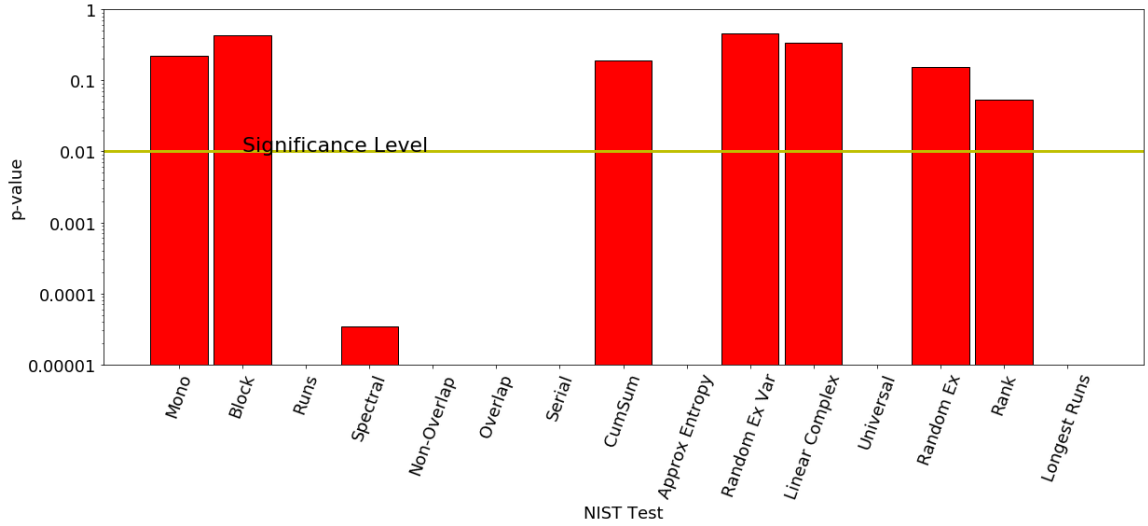


Figure A.2. NIST STS Results, Blum-Blum-Shub

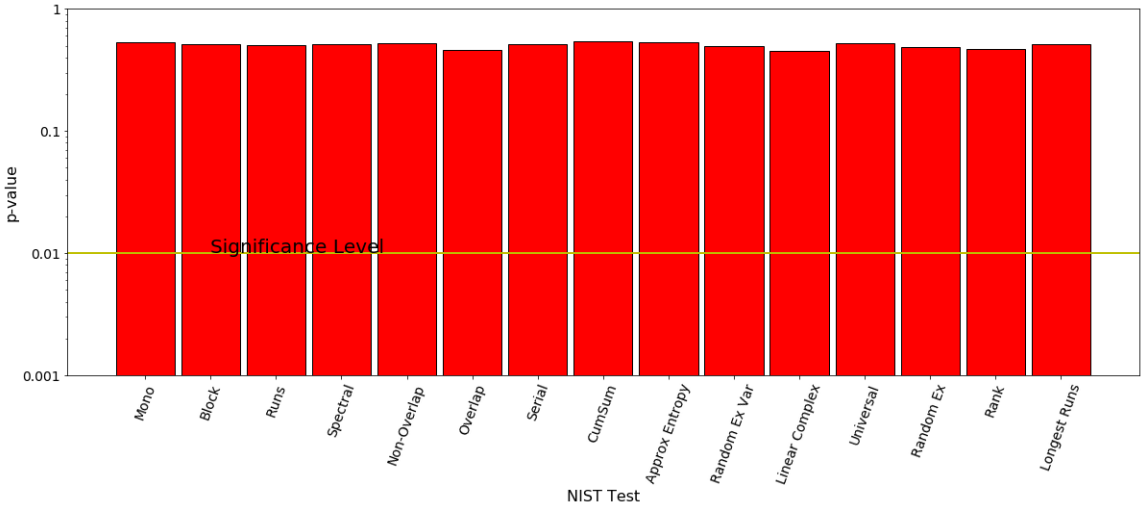


Figure A.3. NIST STS Results, Double Pendulum, Window Size = 2

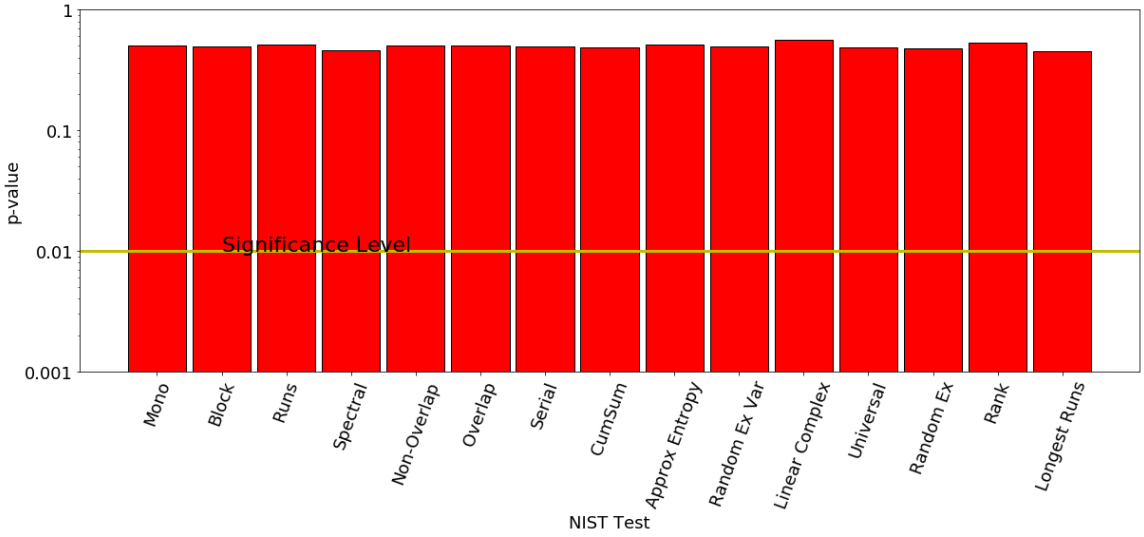


Figure A.4. NIST STS Results, Double Pendulum, Window Size = 4

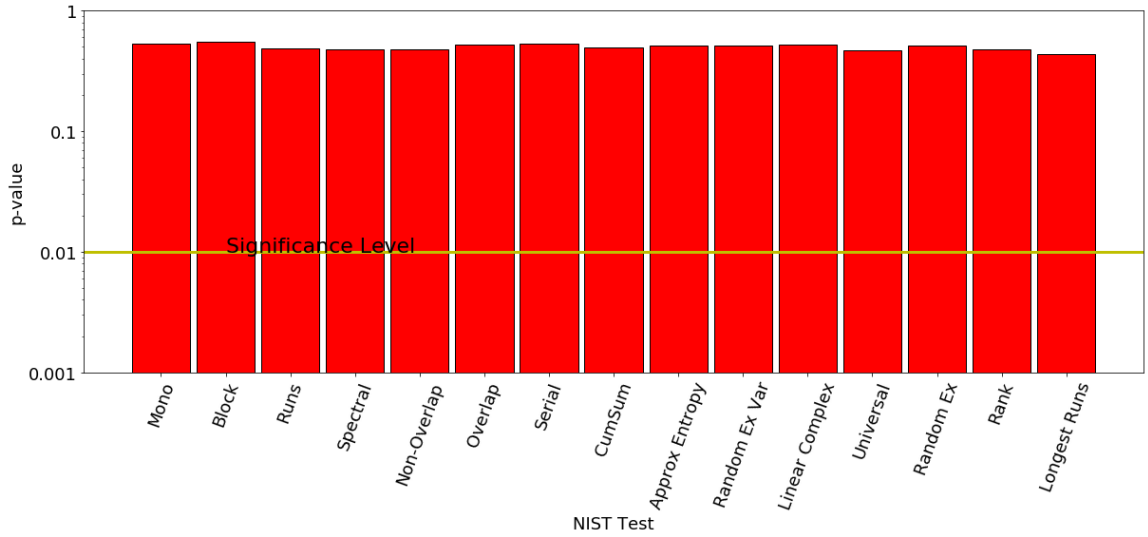


Figure A.5. NIST STS Results, Double Pendulum, Window Size = 8

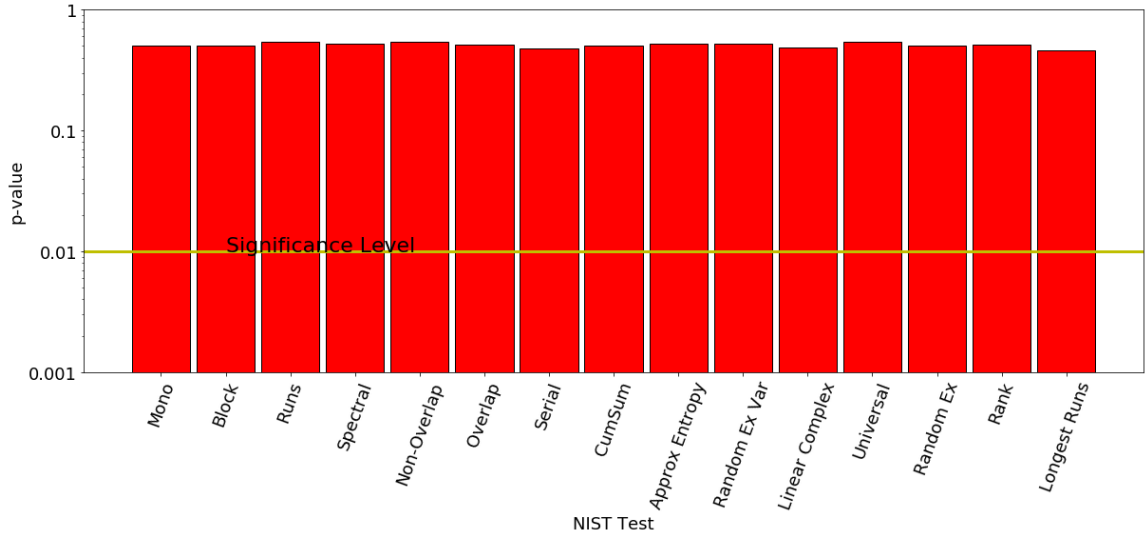


Figure A.6. NIST STS Results, Double Pendulum, Window Size = 16

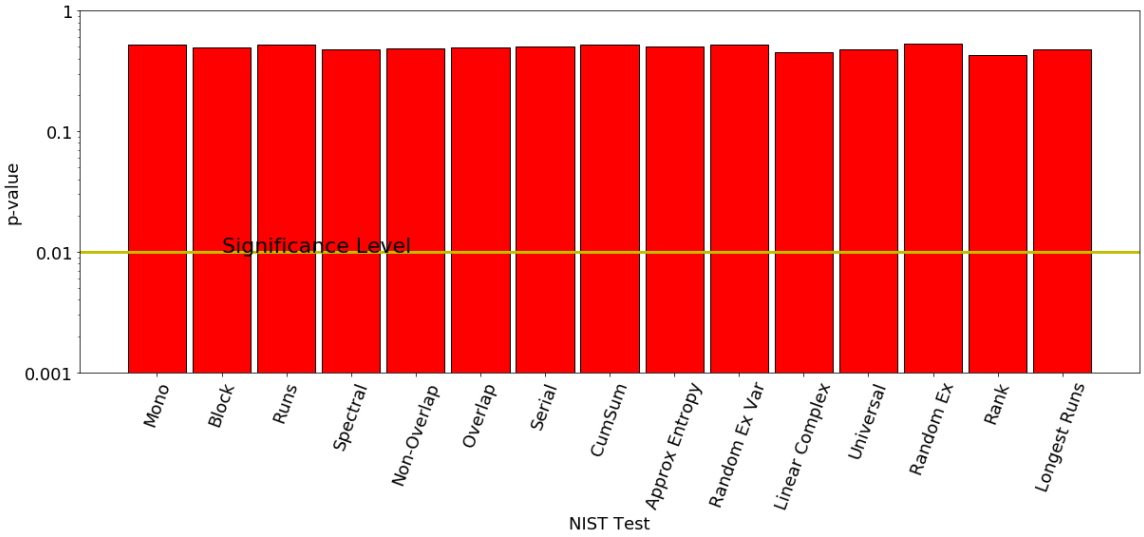


Figure A.7. NIST STS Results, Blum-Blum-Shub, Window Size = 2

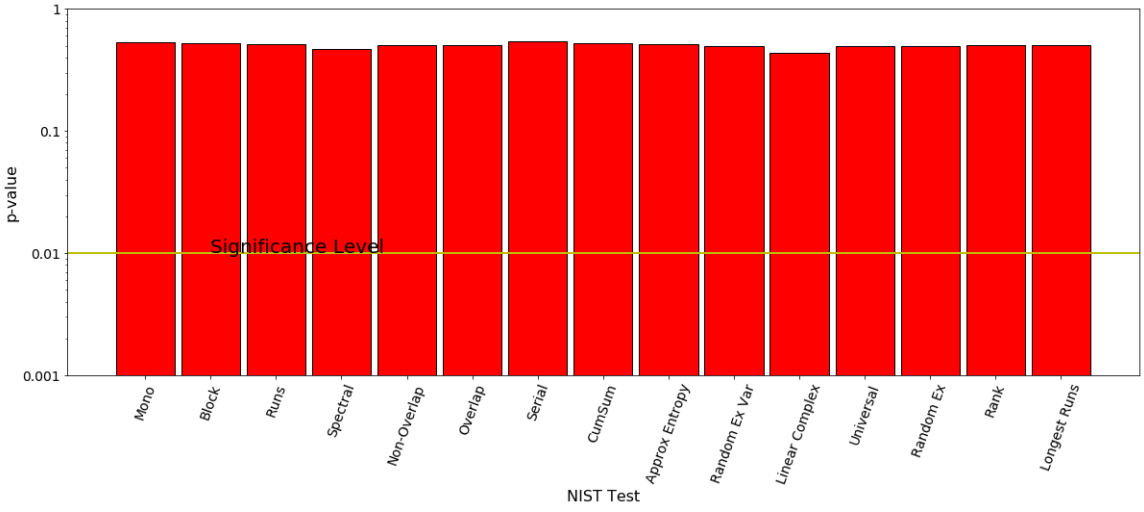


Figure A.8. NIST STS Results, Blum-Blum-Shub, Window Size = 4

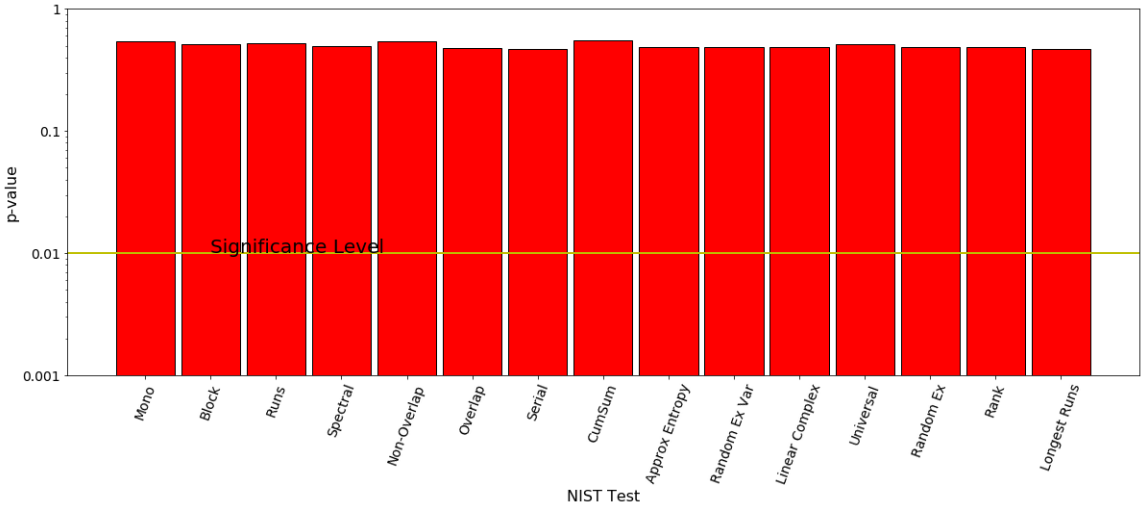


Figure A.9. NIST STS Results, Blum-Blum-Shub, Window Size = 8

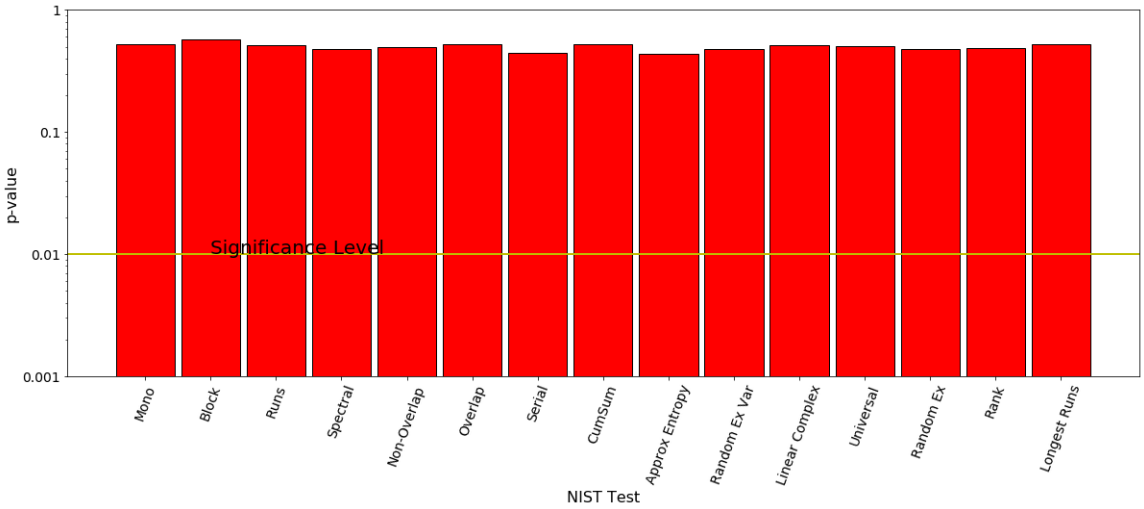
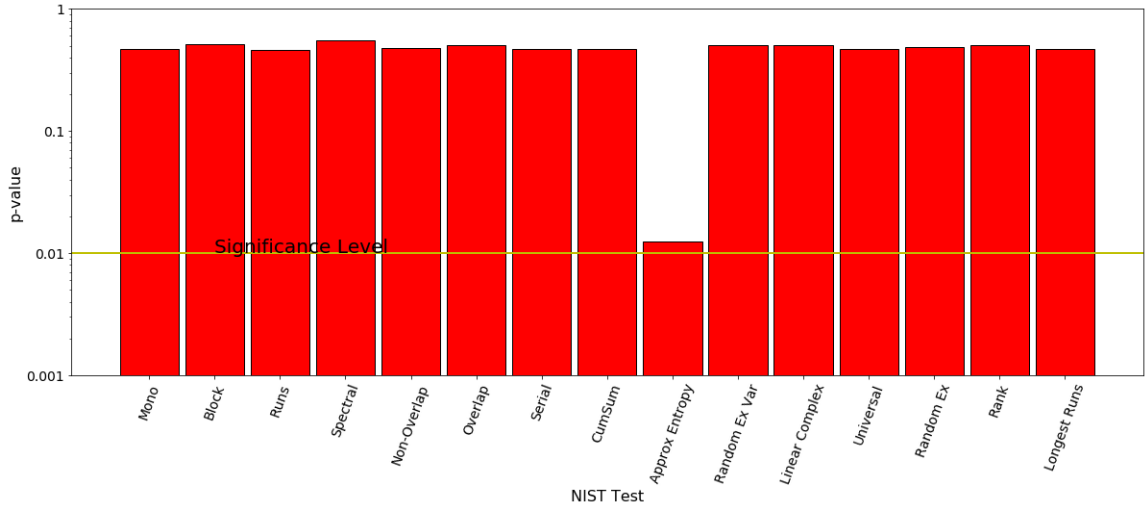


Figure A.10. NIST STS Results, Blum-Blum-Shub, Window Size = 16



APPENDIX B:
Code

The code for the combiner and generation of BBS sequences are duplicated below. Both are written using Python 3 Version 3.7.1. All other code can be found from the provided references.

B.1 Combiner Code

```
#Alex Gutzler
# Import required modules
import sys
import os
import numpy as np
import random
import time
import LFSR1
import NIST_Suite_Updated as nist
import matplotlib.pyplot as plt
import scipy.special as spc
np.set_printoptions(threshold=sys.maxsize)

tests = {'testlist': [nist.monobitfrequencytest,
    nist.blockfrequencytest,
    nist.runstest,
    nist.spectraltest,
    nist.nonoverlappingtemplatematchingtest,
    nist.overlappingtemplatematchingtest,
    nist.serialtest,
    nist.cumultativesumstest,
    nist.aproximateentropytest,
    nist.randomexcursionsvarianttest,
    nist.linearcomplexitytest,
    nist.maurersuniversalstatistictest,
    nist.randomexcursionstest,
    nist.binarymatrixranktest,
    nist.longestrunones10000]}
```

```
29
30 rlist = ['Mono', 'Block', 'Runs', 'Spectral', 'Non-Overlap', 'Overlap',
31          'Serial', 'CumSum', 'Approx Entropy', 'Random Ex Var',
32          'Linear Complex', 'Universal', 'Random Ex', 'Rank', 'Longest
       Runs']
33
34 results = {str(item): np.array([]) for item in rlist}
35
36 random.seed(0)
37 ## Initialize variables
38 ## These are the degrees that make up each primitive polynomial.
39 p1 = (5,2)
40 p2 = (6,1)
41 p3 = (7,1)
42 p4 = (8,4,3,2)
43 p5 = (12,6,4,1)
44 p6 = (9,4)
45 p7 = (13,4,3,1)
46 p8 = (15,1)
47 # This is the order of each primitive polynomial
48 primdeg = (p1[0],p2[0],p3[0],p4[0],p5[0],p6[0],p7[0],p8[0])
49 primpoly = (p1,p2,p3,p4,p5,p6,p7,p8)
50
51 #len(namelist) = 1
52 bitlen = 1000000 # This is the desired length of the sequence
53
54 # Set the current working directory (cwd):
55 cwd = os.getcwd()
56 # Set the read directory
57 rd = os.path.join(cwd, 'finaldata\\E_THETA2_2MIL\\')
58 #rd = os.path.join(cwd, 'Input Data\\')
59 namelist = os.listdir(rd)
60 namelist = namelist[0:len(namelist)]
61 w1 = 3 # Start of window
62 w2 = 19 # End of window
63 # The possible symbols that need to be removed from the strings
64 remove_from_strings = [ ".0"," ", "[", "]", ",", "\\n","'","\n", "."]
65 readryanstrings = ['1', '0',"'", "[", "]"]
66 for item in namelist:
67     t0 = time.time()
```

```
    inputdata = open(rd+item,'r')
    x = inputdata.readlines()
    inputdata.close()
    x = str(x)
    data = x
    for s in remove_from_strings[1:8]:
        data = data.replace(s, " ")
    for s in readryanstrings:
        if s == '1':
            data = data.replace(s, "1 ")
        else:
            data = data.replace(s, "0 ")
    data = np.fromstring(data,sep=' ')
    #This creats an empty dictionary for the seeds for each primitive
poly.
    seeds = {'seed'+str(l): [] for l in range(1,len(primdeg)+1)}
    t = 0
    #Generate a seed for each LFSR to be used and convert format into
string.
    # Steps through each primitive polynomial
    for i in list(primdeg):
        # Increments counter to the next entry in dictionary 'seeds'.
        t += 1
        #This prevents the seed from being the 0 seed.
        while sum(seeds['seed'+str(t)])==0:
            # Resets the seed to null in case the 0 seed is generated.
            seeds['seed'+str(t)] = []
            # Creates the seed one bit at a time of the appropriate
length for each polynomial.
            for j in range(0,i):
                seeds['seed'+str(t)].append(random.randint(0,1))

    # Creates the empty dictionary to convert the seeds into a string (
the NIST Test Suite code reads in strings)
    seedstr = {'seed'+str(i)+'str': str(seeds['seed'+str(i)]) for i in
               range(1,len(primdeg)+1)}

    for i in range(1,len(primdeg)+1):
        # Removes enwanted symbols from strings and replaces them with
nothing leaving a continuous string
```

```
        for s in remove_from_strings:
            seedstr['seed'+str(i)+'str'] = seedstr['seed'+str(i)+
                    'str'].replace(s, "")
    # Generates the seq from each LFSR and seed generated and puts it in
     a dictionary.
    lfsrseq = {'lfsr'+str(i): np.array(LFSR1.lfsr(seedstr['seed'+str(i)+
                                        'str'],(primpoly[i-1])))
    for i in range(1,len(primdeg)+1)}
    # Create an empty dictionary that will be popultated with select
    strings later
    strings = {'string'+str(i): np.array([]) for i in
               range(1,len(primdeg)+1)}

    # Initialize to enter loop
    y = np.zeros(bitlen)
    counter = 0
    inputcount = 0
    zerocount = 0
    totzero = 0
    # This will populate the strings dictionary with a piece of an LFSR
    sequence if the
    # chaotically generated seed has a 1 corresponding to the LFSR
    sequence entry.
    while counter<bitlen and inputcount !=len(data): # Run until desired
     bit length is achieved
        z = [] # Zero out the XOR operator each time
        # Zero out retrieved strings each time
        strings = {'string'+str(i): np.array([]) for i in
                   range(1,len(primdeg)+1)}
        chaosseed = data[inputcount:inputcount+len(primdeg)]
        cs = np.nonzero(chaosseed)[0] # The indices of the 'chaotic'
    seed that are 1.

        # Picks a windowed piece of the LFSR sequence corresponding with
     each 1 in the input seed.
        # Then appends the chosen sequence on the appropriate string in
    the strings dictionary.
        for i in range(0,len(cs)):
            uplim = (counter+w2)%len(lfsrseq['lfsr'+str(cs[i]+1)])
            lolim = (counter+w1)%len(lfsrseq['lfsr'+str(cs[i]+1)])
```

```
            end = len(lfsrseq['lfsr'+str(cs[i]+1)])
            if uplim<lolim:
                strings['string'+str(i+1)] = np.append(strings['string'+
                        str(i+1)],
                       lfsrseq['lfsr'+str(cs[i]+1)][lolim:end])
                strings['string'+str(i+1)] = np.append(strings['string'+
                        str(i+1)],
                       lfsrseq['lfsr'+str(cs[i]+1)][0:uplim])
            else:
                strings['string'+str(i+1)] = np.append(strings['string'+
                        str(i+1)],
                       lfsrseq['lfsr'+str(cs[i]+1)][lolim:uplim])
        if sum(chaosseed) != 0:
            # XORs the pieces of strings from the strings dictionary
            z = sum(strings['string'+str(i)] for i in
                           range(1,len(cs)+1))%2

            # Put selected pieces into final string
            if len(z) != 0 and bitlen-counter>=w2-w1:
                y[counter:counter+w2-w1] = z
            if bitlen-counter<w2-w1:
                y[counter:bitlen] = z[0:bitlen-counter]
            counter = counter + w2 - w1
        else:
            zerocount += 1
        inputcount += len(primdeg)
        totzero += zerocount

    y = y[0:int((w2-w1)*len(data)/len(primdeg)-zerocount*(w2-w1))]
    print(item)
    print(len(y))

    # Convert the sequence to a string that can be read by the NIST Test
    Suite.
    g = np.array2string(y)
    for s in remove_from_strings:
        g = g.replace(s, "")
    for i in range(0,len(rlist)):
        results[rlist[i]] = np.append(results[rlist[i]],
               tests['testlist'][i](g))
```

```
#print(totzero)
avgresults = {str(item): np.array([]) for item in rlist}
numpass = {str(item): np.array([]) for item in rlist}
for i in range(0,len(rlist)):
    if i == 6:
        avgresults[rlist[i]] = np.append(avgresults[rlist[i]],
                  np.mean(results[rlist[i]].reshape(-1, 2), axis=0))
        numpass[rlist[i]] = np.append(numpass[rlist[i]],
                np.sum(results[rlist[i]].reshape(-1, 2)>.01,axis=0))
    elif i == 9:
        avgresults[rlist[i]] = np.append(avgresults[rlist[i]],
                  np.mean(results[rlist[i]].reshape(-1, 18), axis=0))
        numpass[rlist[i]] = np.append(numpass[rlist[i]],
                np.sum(results[rlist[i]].reshape(-1, 18)>.01,axis=0))
    elif i == 12:
        avgresults[rlist[i]] = np.append(avgresults[rlist[i]],
                  np.mean(results[rlist[i]].reshape(-1, 8), axis=0))
        numpass[rlist[i]] = np.append(numpass[rlist[i]],
                np.sum(results[rlist[i]].reshape(-1, 8)>.01,axis=0))
    else:
        avgresults[rlist[i]] = np.append(avgresults[rlist[i]],
                  np.mean(results[rlist[i]]))
        numpass[rlist[i]] = np.append(numpass[rlist[i]],
                sum(results[rlist[i]]>.01))

temp = {str(item): np.array([]) for item in rlist}
temppass = {str(item): np.array([]) for item in rlist}
for i in range(0,len(rlist)):
    if i == 6:
        temp[rlist[i]] = np.append(temp[rlist[i]],
              np.mean(results[rlist[i]]))
        temppass[rlist[i]] = np.append(temppass[rlist[i]],
              np.sum(results[rlist[i]]>.01)/2)
    elif i == 9:
        temp[rlist[i]] = np.append(temp[rlist[i]],
              np.mean(results[rlist[i]]))
        temppass[rlist[i]] = np.append(temppass[rlist[i]],
              np.sum(results[rlist[i]]>.01)/18)
    elif i == 12:
        temp[rlist[i]] = np.append(temp[rlist[i]],
```

```
                np.mean(results[rlist[i]]))
        temppass[rlist[i]] = np.append(temppass[rlist[i]],
                np.sum(results[rlist[i]]>.01)/8)
    else:
        temp[rlist[i]] = np.append(temp[rlist[i]],
                np.mean(results[rlist[i]]))
        temppass[rlist[i]] = np.append(temppass[rlist[i]],
                np.sum(results[rlist[i]]>.01))

for i in range(0,len(rlist)):
    n,bins,patches = plt.hist(results[rlist[i]])
    if i == 6:
        chisq = sum((n[i]-len(namelist)*2/10)**2/(len(namelist)*2/10)
        for i in range(10))
    elif i == 9:
        chisq = sum((n[i]-len(namelist)*18/10)**2/(len(namelist)*18/10)
        for i in range(10))
    elif i == 12:
        chisq = sum((n[i]-len(namelist)*8/10)**2/(len(namelist)*8/10)
        for i in range(10))
    else:
        chisq = sum((n[i]-len(namelist)/10)**2/(len(namelist)/10)
        for i in range(10))
    pvalt = spc.gammaincc(9/2,chisq/2)
    pvalt = np.round(pvalt,5)
    print(pvalt)

phat = .99
for i in range(0,len(rlist)):
    if i == 6:

        ciu = phat + 3*np.sqrt(phat*(1-phat)/(len(namelist)*2))
        cil = phat - 3*np.sqrt(phat*(1-phat)/(len(namelist)*2))
    elif i == 9:
        ciu = phat + 3*np.sqrt(phat*(1-phat)/(len(namelist)*18))
        cil = phat - 3*np.sqrt(phat*(1-phat)/(len(namelist)*18))
    elif i == 12:
        ciu = phat + 3*np.sqrt(phat*(1-phat)/(len(namelist)*8))
        cil = phat - 3*np.sqrt(phat*(1-phat)/(len(namelist)*8))
    else:
```

```
        ciu = phat + 3*np.sqrt(phat*(1-phat)/len(namelist))
        cil = phat - 3*np.sqrt(phat*(1-phat)/len(namelist))
    count = temppass[rlist[i]]/len(namelist)
    prop = (cil<count<ciu)
    print(prop)

d = {k:float(v) for k,v in temp.items()}
plt.figure(figsize=(20,7))
plt.bar(*zip(*d.items()),.9,color = 'r', edgecolor = 'k', log = True)
plt.yticks(ticks = (.001,.01,.1,1), labels = ('0.001', '0.01', '0.1',
           '1'), fontsize = 14)
plt.xticks(rotation=70, fontsize = 14)
plt.xlabel('NIST Test', fontsize = 16)
plt.ylabel('p-value', fontsize = 16)
plt.axhline(y=.01, color = 'y', linewidth = 2)
plt.text(1,.01,'Significance Level', fontsize = 20, color = 'k')

passed = {k:float(v) for k,v in temppass.items()}
plt.figure(figsize=(20,7))
plt.bar(*zip(*passed.items()),.9,color = 'r', edgecolor = 'k')
plt.xticks(rotation=70, fontsize = 14)
plt.xlabel('NIST Test', fontsize = 16)
plt.ylabel('Number of Sequences that passed', fontsize = 16)
```

B.2 Blum-Blum-Shub Code

```
#Alex Gutzler
# Import required modules
import random
import numpy as np
import sys
import time
import sympy
np.set_printoptions(threshold=sys.maxsize)

random.seed(0)
num = 100 # Number of sequences to generate
for j in range(num):
    p = 0 # Initialize to enter loop
    q = 0
```

```
    while sympy.isprime(p) == False or 3 != p%4: # Ensure p is Blum Prime
        p = random.getrandbits(64) | 1
    while sympy.isprime(q) == False or 3 != q%4: # Ensure q is Blum Prime
        q = random.getrandbits(64) | 1
        m = p*q
    t0 = time.time()
    seed = random.randint(0,m)
    # Make seed that cannot be factored by wrt p or q
    while seed/p == np.floor(seed/p) or seed/q == np.floor(seed/q) or seed == 0 or seed == 1:
        seed = random.randint(0,m)
    xold = seed
    n = 4200000 # Length of sequences
    g = np.zeros(n)
    # Take leasat significant bit
    for i in range(n):
        xnew = (xold**2)%m
        g[i] = xnew%2
        xold = xnew
    print("--- %.5s seconds ---" % (time.time() - t0))
    g = ",".join(map(str, g))
    remove_from_strings = [ ".0"," ", "[", "]", ",", "\\n","'","\n", "." ] # The possible symbols that need to be removed from the strings
    for s in remove_from_strings[0:8]:
        g = g.replace(s, "")
    okay = open("Input Data//BBS_data2/input"+str(j+1)+".txt","w+")
    okay.write(g)
    okay.close()
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] J. Massey, "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.

[2] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory*, 2nd ed. Prentice-Hall, 2007.

[3] J. Hamlin. ASCII Code: Character to Binary. Accessed Jan. 27, 2020. [Online]. Available: http://www.mrhamlin.ca/home/programming-11/binary

[4] T. Helleseth, "Linear and nonlinear sequences and applications to stream ciphers," in *Recent Trends in Cryptography*, I. Luengo, Ed. Providence, RI: American Mathematical Society, 2009, pp. 21–45.

[5] S. W. Golomb, "Shift-register sequences and spread-spectrum communications," in *Proceedings of IEEE 3rd International Symposium on Spread Spectrum Techniques and Applications (ISSSTA'94)*, July 1994, pp. 14–15 vol.1.

[6] MattiaG (https://stackoverflow.com/users/359298/mattiag), "Linear feedback shift register?" Stack Overflow. Accessed Jan. 27, 2020. [Online]. Available: https://stackoverflow.com/questions/3735217/linear-feedback-shift-register

[7] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh *et al.*, "NIST special publication 800-22 revision 1a: A statistical test suite for random and pseudorandom number generators for cryptographic applications," NIST, U.S. Department of Commerce, USA, Tech. Rep., 2010.

[8] R. Hard, "Double pendulum chaotic model for pseudorandom number generation," M.S. Thesis, Dept. Applied Math., Naval Postgraduate School, Monterey, CA, U.S., 2020.

[9] I. Gerhardt, "Random number testing." Accessed Jan. 27, 2020. [Online]. Available: https://gerhardt.ch/random.php

[10] J. M. Sachs, "Galois field GF2 arithmetic for Python." Accessed Jan. 27, 2020. [Online]. Available: https://bitbucket.org/jason_s/libgf2/src/default/src/libgf2/util.py

[11] A. Canteaut, "Combination generator," in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Boston, MA: Springer US, 2011, pp. 222–224. Available: https://doi.org/10.1007/978-1-4419-5906-5_338.

[12] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on Computing*, vol. 15, no. 2, pp. 364–383, 1986.

[13] E. Watson, "Primitive polynomials (mod 2)," *Math. Comp*, vol. 16, no. 79, pp. 368–369, 1962.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California