NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

NON-LINEAR PSEUDORANDOM BIT GENERATION BY COMBINING BLUM BLUM SHUB AND LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

by

Andrew M. Cammack

June 2020

Thesis Advisor:	Pantelimon Stanica
Second Reader:	Thor Martinsen

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2020	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE NON-LINEAR PSEUDORANDOM BIT GENERATION BY COMBINING BLUM BLUM SHUB AND LINEAR FEEDBACK SHIFT REGISTER SEQUENCES			**5. FUNDING NUMBERS**
6. AUTHOR(S) Andrew M. Cammack			
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)

The purpose of this thesis is to analyze the cryptographic properties of a pseudorandom bit generator that combines Blum Blum Shub and linear feedback shift register sequences using a shrinking generator configuration. We sought to answer the questions: (1) What are the strengths and weaknesses of this type of combiner? (2) What constraints must be placed on the input parameters to ensure good cryptographic properties of the output sequence?

We generated sequences using variations of this combiner. We then evaluated their cryptographic suitability with the National Institute of Standards and Technology (NIST) statistical test suite. We identified lower bounds on the input parameters to increase the probability that the combiner would perform well under the NIST test suite. Our scheme produced consistently excellent results under NIST testing but is computationally too slow for many practical uses as a stream cipher. Future work could focus on methods to increase the speed of the generator without a loss of excellent cryptographic properties.

14. SUBJECT TERMS Blum Blum Shub, BBS, linear feedback shift register, secure communications, pseudorandom bit generators, PRBG, shrinking generator, non-linear sequences, National Institute of Standards and Technology, NIST			**15. NUMBER OF PAGES** 93
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** 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.

NON-LINEAR PSEUDORANDOM BIT GENERATION BY COMBINING BLUM BLUM SHUB AND LINEAR FEEDBACK SHIFT REGISTER SEQUENCES

Andrew M. Cammack
Major, United States Army
BS, U.S. Military Academy, 2009
MS, Missouri University of Science and Technology, 2014

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2020

Approved by: Pantelimon Stanica
Advisor

Thor Martinsen
Second Reader

Wei Kang
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The purpose of this thesis is to analyze the cryptographic properties of a pseudorandom bit generator that combines Blum Blum Shub and linear feedback shift register sequences using a shrinking generator configuration. We sought to answer the questions: (1) What are the strengths and weaknesses of this type of combiner? (2) What constraints must be placed on the input parameters to ensure good cryptographic properties of the output sequence?

We generated sequences using variations of this combiner. We then evaluated their cryptographic suitability with the National Institute of Standards and Technology (NIST) statistical test suite. We identified lower bounds on the input parameters to increase the probability that the combiner would perform well under the NIST test suite. Our scheme produced consistently excellent results under NIST testing but is computationally too slow for many practical uses as a stream cipher. Future work could focus on methods to increase the speed of the generator without a loss of excellent cryptographic properties.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**

1.1 Stream Ciphers 1

1.2 Pseudorandom Bit Generators 2

2 Linear Feedback Shift Registers **5**

2.1 LFSRs as Linear Recurrence Relations 5

2.2 LFSRs as Abstract Algebra Structures 8

3 Blum Blum Shub Generator **17**

3.1 History 17

3.2 Overview 17

3.3 Number Theory 18

3.4 Summary 24

4 Methodology **25**

4.1 Experiment Number One: BBS Driver with Varying Primitive Polynomials . 26

4.2 Experiment Number Two: LFSR Driver with Varying Primitive Polynomials . 27

4.3 Experiment Number Three: BBS Driver with Varying BBS Moduli 29

4.4 Experiment Number Four: BBS Driver with Varying BBS Seed 32

4.5 Experiment Number Five: BBS Driver with Varying Blum Primes 33

4.6 Experiment Number Six: BBS Driver with Varying BBS Seed 34

4.7 Experiment Number Seven: BBS Driver with Degree 18 to 32 Polynomials . 35

4.8 Experiment Number Eight: BBS Driver with Random Parameter Selection within Bounds 36

4.9 Experiment Number Nine: BBS Driver with Degree 26-29 Polynomials and BBS Period $\geq$ 3000 37

5 Conclusion **39**

5.1 Summary of Results 39

5.2 Further Research Ideas 39

Appendix: Python Code **41**

A.1 Main Combiner, with BBS Driver 41

A.2 Random Combiner Parameters Generator 49

A.3 Reversible Combiner 52

A.4 BBS Period vs. M Plot 71

List of References **73**

Initial Distribution List **75**

List of Figures

Figure 2.1 Linear Feedback Shift Register 5

Figure 2.2 Linear Feedback Shift Register Example 6

Figure 4.1 Experiment 1 27

Figure 4.2 Experiment 2 28

Figure 4.3 ln(combiner linear complexity) vs. ln(M) 29

Figure 4.4 ln(combiner linear complexity) vs. ln(combiner period) 30

Figure 4.5 BBS Period vs. M 31

Figure 4.6 Experiment 3 32

Figure 4.7 Experiment 4 33

Figure 4.8 Experiment 5 34

Figure 4.9 Experiment 6 35

Figure 4.10 Experiment 7 36

Figure 4.11 Experiment 8 37

Figure 4.12 Experiment 9 38

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 2.1 Example LFSR States at Time t 6

Table 2.2 Group Table for $\langle \mathbb{Z}_4, + \rangle$ 9

Table 2.3 Group Table for $\langle \mathbb{Z}_3, + \rangle$ 12

Table 2.4 Group Table for $\langle \mathbb{Z}_3, \cdot \rangle$ 12

Table 2.5 Coset Representatives for $\mathbb{F}_2(x)(\mathbf{mod}\ x^4 + x + 1)$ 15

Table 4.1 Example of a Shrinking Generator 25

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AES Advanced Encryption Standard

BBS Blum Blum Shub Pseudorandom Number Generator

CRT Chinese Remainder Theorem

gcd Greatest Common Divisor

lcm Least Common Multiple

NIST National Institute of Standards and Technology

NPS Naval Postgraduate School

PRBG Pseudorandom Bit Generator

PRNG Pseudorandom Number Generator

RC4 Rivest Cipher 4

RSA Rivest, Shamir, and Adleman Cryptosystem

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank my thesis advisor, Dr. Pante Stănică, and second reader, Commander Thor Martinsen, for their inspiration and support to this research. I am also indebted to Captain Michael Troncoso for his invaluable assistance with setting up the NIST test and for his example LFSR code. Lastly, I would like to thank my wife, Josephine, for her love and support throughout the process.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Cybersecurity is a major concern for the modern world. In the information age, vast amounts of critical information are stored in and shared between computers. Governments, businesses, organizations, and individuals rely on the security of their data to function from day-to-day. To secure data, two major types of encryption are used: block and stream ciphers. Block ciphers are used to encrypt completed data files. The widely accepted standard for block ciphers in the Advanced Encryption Standard (AES). Stream ciphers, on the other hand, are used to rapidly encrypt incomplete data as it is generated, such as live streaming video, cellphone, and radio communications. This thesis focuses on a particular configuration of stream cipher which combines features of a Linear Feedback Shift Register (LFSR) and the Blum Blum Shub (BBS) Pseudorandom Number Generator (PRNG).

1.1 Stream Ciphers

Stream ciphers encrypt binary plaintext into ciphertext so that only the intended audience can decrypt the message back into plaintext. For the sake of speed, encryption often occurs one bit at a time rather than with large blocks of bits as in block ciphers. While stream ciphers are inherently less secure than block ciphers, they are essential to secure communications requiring near-instantaneous transmission. A good stream cipher should balance security and speed in accordance with the level of security required by the application.

1.1.1 Current Stream Ciphers

Many variations of stream ciphers have been created. Some of the best known include Rivest Cipher 4 (RC4), Software-Optimized Encryption Algorithm (SEAL), A5/1, A5/2, and the eStream collection of ciphers that include Trivium, Grain v1, and Salsa20/12 [1].

This thesis considers a new scheme for a stream cipher combining the security and linear complexity of BBS with the speed inherent in LFSRs. By using the National Institute of Standards and Technology (NIST) test to evaluate the statistical randomness of the sequence generated, it may be possible to achieve an acceptable level of security with greater speed

using a smaller BBS key (and therefore a faster BBS calculation) than is normally required for BBS-only keystreams. While this thesis does not compare the speed and security of our stream cipher to other existing stream ciphers, it does provide the foundation for future comparisons.

1.1.2 Stream Cipher Encryption

Encryption is the process of converting plaintext to ciphertext, then back to plaintext. For a stream cipher, this typically occurs with a bitwise operation rather than with blocks of bits. First, a secret key is shared between the sender and receiver. For electronic communications, secret key transfer is performed through Public Key Cryptography methods. The key is used to generate a keystring. A binary plaintext message is combined with a keystring using a bit-wise exclusive-or (XOR) operator ($\oplus$) to produce ciphertext.

$$\text{An example of encryption:}\quad \begin{array}{lr} \text{(plaintext)} & 001011100 \\ \text{(keystring)} & \oplus\ \underline{011011000} \\ \text{(ciphertext)} & 010000100 \end{array}$$

The ciphertext is then sent to the receiver who decrypts the message by generating the same keystring (using the secret key) and combining it with the ciphertext via the bit-wise XOR function to decrypt the message back to plaintext.

$$\text{An example of decryption:}\quad \begin{array}{lr} \text{(ciphertext)} & 010000100 \\ \text{(keystring)} & \oplus\ \underline{011011000} \\ \text{(plaintext)} & 001011100 \end{array}$$

1.2 Pseudorandom Bit Generators

The most secure cipher is the one-time pad. This cipher involves a secret key shared between a sender and receiver in which each character in the message is encrypted using a unique, random key. This type of cipher can only be defeated by capturing the secret key. Unfortunately, the one-time pad is too slow for electronic communications and requires sharing key information as large as the message itself. Instead, in modern cryptography, pseudorandom bit generators, relying on a small amount of key data, create a keystream of seemingly random bits that are combined using the XOR operation with plaintext bits to create ciphertext. The key is shared with the desired receiver using Public Key Cryptography

methods. By choosing a good pseudorandom bit generator (PRBG), the sender and receiver can generate an identical keystream with a period of millions of bits from a very small amount of key data.

Note that the terms *key* and *keystream* are used to mean two different things here. The *key* is the information needed to initiate the PRBG. The *keystream* (or *keystring*) is the long sequence of pseudorandom bits that are generated by the PRBG and combined bitwise with the plaintext. A keystream can be of any length, but will have a finite period over which the sequence will repeat.

In order for a PRBG to be considered cryptographically secure, there must not be any statistical test, which can predict the next bit from the previous bits with a greater than 50% likelihood. No stream cipher can truly be *proven* secure because there may be a new statistical test not yet developed that would increase our likelihood of predicting the next bit. However, we can use all known statistical tests to make our PRBG as secure as possible. For this we use the National Institute of Standards and Technology (NIST) suite of statistical tests for PRBGs. This is a collection of the best statistical tests currently available.

The following chapters discuss LFSRs and the BBS PRNG which form the basis for our combiner PRBG.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Linear Feedback Shift Registers

LFSRs are mathematical structures implemented in hardware and software and used commonly as pseudorandom bit generators (PRBGs). An LFSR uses a linear recurrence relation to rapidly generate a seemingly random keystring of bits with a very long period.

LFSRs can be described functionally in terms of a linear recurrence relations and theoretically in terms of abstract algebra structures.

2.1 LFSRs as Linear Recurrence Relations

The linear recurrence relation of the form [2]:

$$s_{t+L} = \sum_{i=1}^{L} c_i s_{t+L-i}, \forall t \geq 0 \tag{2.1}$$

generates a sequence, $\mathbf{s} = (s_t)_{t \geq 0} = s_0 s_1 \ldots$, where t represents the time-step, L is the length (number of cells) of the LFSR, c_i is a *feedback coefficient*, and the subscript i is the cell number from 1 to L. s_t is the output bit from the LFSR at time t.

The recurrence relation is displayed graphically in Figure 2.1.

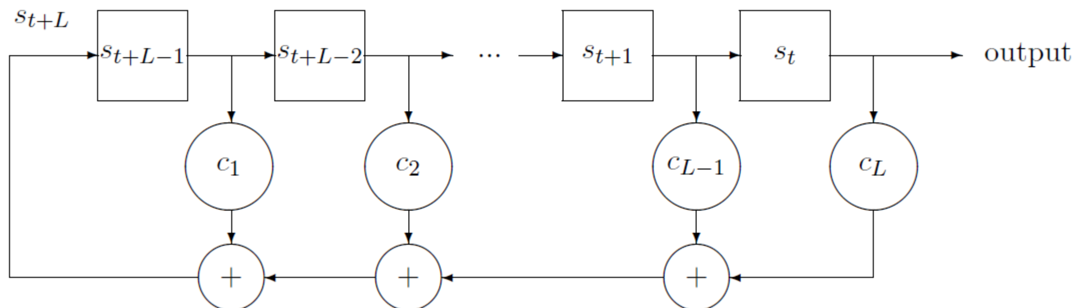

Figure 2.1. Linear Feedback Shift Register. Source: [2].

The LFSR is initialized with each cell containing an element from $\mathbb{F}_q$ (typically $\mathbb{F}_2 = \{0, 1\}$).

The values in each of the LFSR's L cells during a given time-step are called the LFSR's *state*. During each time-step, the cell s_t is outputted and appended to the end of LFSR sequence **s**. Concurrently, the value for s_{t+L} is calculated using the bitwise XOR function from the cells in the register with closed taps ($c_i \neq 0$). This is the same calculation seen in Equation 2.1. s_{t+L} then fills the leftmost cell of the register, and the remaining cell values shift to the right by one cell.

2.1.1 LFSR Example

An LFSR based on the linear recurrence relation $s_{t+4} = s_{t+1} + s_t$ is shown in Figure 2.2:

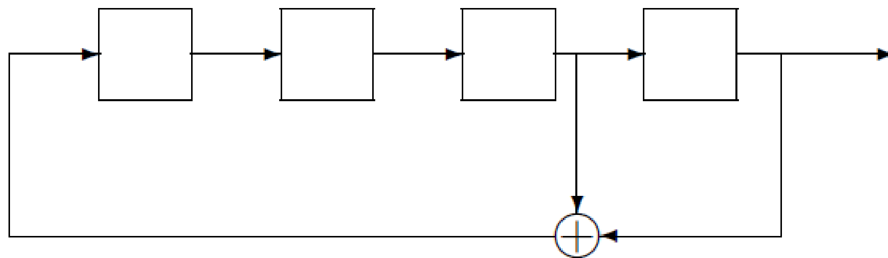

Figure 2.2. Linear Feedback Shift Register Example. Source: [2].

with a starting seed of $s_3 = s_2 = s_1 = 0;\ s_0 = 1$. For this LFSR, the feedback coefficients are: $c_3, c_4 = 1$ and $c_1, c_2 = 0$. The LFSR has a period of $2^4 - 1 = 15$ terms. Table 2.1 displays the LFSR states over a full period. At $t = 15$ the LFSR has completed a full period and returns to its initial state.

Table 2.1. Example LFSR states at time t. The output sequence $\mathbf{s} = s_0 s_1 ...$ is shown on the s_t row. Adapted from [2].

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s_t	1	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1
s_{t+1}	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1	0
s_{t+2}	0	0	1	0	0	1	1	0	1	0	1	1	1	1	0	0
s_{t+3}	0	1	0	0	1	1	0	1	0	1	1	1	1	0	0	0

2.1.2 Feedback and Characteristic Polynomials

The feedback coefficients for an LFSR can be represented with a *feedback polynomial* [2]:

$$P(X) = 1 - \sum_{i=1}^{L} c_i X^i.$$

Closely related to the feedback polynomial is its reciprocal, the *characteristic polynomial* [2], of the form:

$$P^\star(X) = X^L P(1/X) = X^L - \sum_{i=1}^{L} c_i X^{L-i}.$$

For our example LFSR, the feedback polynomial is $P(X) = 1 + X^3 + X^4$ and its characteristic polynomial is $P^\star(X) = 1 + X + X^4$ [2].

Notice that over one full period of 15 time-steps the LFSR states (columns of Table 2.1) include each of the 15 possible permutations of 4 bits (leaving out the all 0's vector) without repeats. An appropriately selected feedback polynomial will of degree n will generate a maximal-length sequence (m-sequence) which is $2^n - 1$ bits long and cycles through all $2^n - 1$ states exactly once, leaving out the all 0's state. By cycling through all permutations but the all 0's state, LFSRs are approximately balanced over one period with the count of 1-bits = $(period + 1)/2$ and the count of 0-bits = $(period - 1)/2$. The difference in 1's and 0's comes from the fact that the all 0's state does not occur. The only exception is if the all 0's state is the initial LFSR state; in that case, the output sequence will be all 0's. The example LFSR in Figure 2.2 is nearly balanced with an output of eight 1's and seven 0's over a full period of 15 bits. This gives an average Hamming weight of $8/15 = 0.5\overline{3}$, near 50%. As the LFSR period increases, the average Hamming weight approaches 50% over a full period.

This example LFSR generates a sequence with a period of 15 bits from 4 bits of seed information in the initial state. This is a relatively small output period relative to the amount of seed information needed. However, by selecting appropriate feedback polynomials, one can generate extremely long sequences with relatively little seed information. For example,

the feedback polynomial $P(X) = 1 + X^{28} + X^{31}$ (with characteristic polynomial $P^\star(X) = 1 + X^3 + X^{31}$) gives the recurrence relation $s_{t+31} = s_{t+3} + s_t$ and generates a sequence with period $2^{31} - 1 = 2,147,483,647$ bits from only 31 bits of seed information! But how does one select the correct feedback polynomial to generate such long sequences? For this we must consider the abstract algebra behind LFSRs.

2.2 LFSRs as Abstract Algebra Structures

First, we will review the basics of Abstract Algebra to build the foundation for describing how LFSRs work. We will start with basic number theory, then four algebraic structures: groups, rings, fields, and extension fields.

2.2.1 Number Theory

The following basic algebra concepts and definitions are covered in most discrete mathematics or abstract algebra textbooks such as *Discrete Mathematics and Its Applications* by Rosen [3], or *A First Course in Abstract Algebra* by Fraleigh [4].

"If a and b are integers with $a \neq 0$, we say that *a divides b* if there is an integer c such that $b = ac$, or equivalently, if b/a is an integer. When a divides b we say that a is a *factor* or *divisor* of b, and the b is a *multiple* of a. The notation $a|b$ denotes that a divides b. We write $a \nmid b$ when a does not divide b," [3]. The *greatest common divisor* of two integers a and b is the largest integer that divides both a and b, denoted gcd(a,b). If the gcd of two integers is 1, the integers are said to be *co-prime*. "The *least common multiple* of the positive integers a and b is the smallest positive integer that is divisible by both a and b. The least common multiple of a and b is denoted by lcm(a,b)," [3].

Next, we will review the Division Algorithm which is the basis for *modular arithmetic*. "Let a be an integer and d a positive integer. Then there are unique integers q and r, with $0 \leq r < d$, such that $a = dq + r$," [3].

The remainder may be expressed in the form $r = a \textbf{ mod } d$ where r is the *remainder*, a is the *dividend*, and d is the *divisor*. If two integers b and c produce the same remainder when divided by divisor d, we say that b and c are equivalent *modulo d*. This is written $b \equiv c \textbf{ mod } d$.

2.2.2 Groups

Recall [4] that a **group** $\langle G, * \rangle$ is a set G, closed under a binary operation *, satisfying the following axioms:

G_1: For all $a, b, c \in G$, we have $(a * b) * c = a * (b * c)$. (Associativity)

G_2: There is an element $e \in G$ such that for all $x \in G$, $e * x = x * e = x$. (Identity Element e for operation *)

G_3: Corresponding to each $a \in G$, there is an element $a' \in G$ such that $a * a' = a' * a = e$. (Inverse a' of a).

One example of a group is the set of integers modulo 4 = {0,1,2,3} over addition (+), denoted $\langle \mathbb{Z}_4, + \rangle$. Performing the operation (+) on any element in the set returns another element of the set. The addition table for our example $\langle \mathbb{Z}_4, + \rangle$ is shown in Table 2.2.

Table 2.2. Group Table for $\langle \mathbb{Z}_4, + \rangle$.

+	**0**	**1**	**2**	**3**
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Also, a group G is called **abelian** (or commutative) if its binary operation is commutative [4]. An abelian group adds the additional rule:

G_4: For all $a, b \in G$, $a + b = b + a$. (Commutativity)

The group $\langle \mathbb{Z}_4, + \rangle$ shown in Table 2.2 is an example of an abelian group, as seen in its group table's symmetry across the diagonal.

2.2.3 Commutative Rings

The next algebraic structure we will discuss is a commutative ring. A **ring** $\langle R, +, \cdot \rangle$ is a set R together with two binary operations + and $\cdot$, which we call *addition* and *multiplication*, defined on R satisfying the following axioms [4]:

R_1: $\langle R, + \rangle$ is an abelian group.

R_2: Multiplication is associative.

R_3: For all $a, b, c \in R$, the **left distributive law** $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and **right distributive law** $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$ hold.

Commutative Rings are rings with the additional property that multiplication is commutative:

R_4: For all $a, b \in R$, $a \cdot b = b \cdot a$. (Commutative under Multiplication)

Rings with unity are rings that contain the multiplicative identity 1.

Some rings have an additional property that makes them a **division ring**:

R_5: Corresponding to each non-zero element $a \in R$, there is an element $a' \in G$ such that $a \cdot a' = a' \cdot a = 1$. (Multiplicative Inverse a' of a)

An example of a commutative ring with unity (but not a division ring) is the set of integers under addition and multiplication $\langle \mathbb{Z}, +, \cdot \rangle$. The set of integers are closed under addition, subtraction, and multiplication because adding, subtracting, or multiplying two integers together always produces another integer. $\mathbb{Z}$ is not closed under division, however, because dividing an integer by another integer may produce a rational number that is not an element of $\mathbb{Z}$.

Another example of a commutative ring with unity (but not a division ring) is the set of polynomials with integer-valued coefficients under the operations addition and multiplication. This is called the ring of polynomials with integer coefficients, $\langle \mathbb{Z}[x], +, \cdot \rangle$. Polynomials are functions with an indeterminate x of the form $p(x) = \sum_{i=1}^{\infty} a_i * x^i$ where $a_i \in \mathbb{R}$. Addition within $\mathbb{Z}[x]$ is taken in the usual way in which if two polynomials are added, the coefficients with the same power of x are added together to form the new polynomial. Multiplication

within $\mathbb{Z}[x]$ is also defined in the usual way, by multiplying each term of the first polynomial with each term of the second polynomial. When two monomial terms are multiplied, their coefficients are multiplied together, and the exponents associated with x are added. Terms of like powers of x are then added together to form a new polynomial. Because adding, subtracting, or multiplying together two polynomials always results in a polynomial already within the ring, $\mathbb{Z}[x]$ is closed under addition and multiplication and has an additive inverse. However, if a polynomial is divided by a polynomial, the result may be a rational function that cannot be reduced to a polynomial. Because not all elements of $\mathbb{Z}[x]$ have a multiplicative inverse within the set, $\mathbb{Z}[x]$ is not a division ring.

2.2.4 Fields

A commutative division ring with unity is called a **field**. [4] That is to say that a field is any ring with all the properties R_1 to R_5.

Fields can be infinite, such as the real numbers $\langle \mathbb{R}, +, \cdot \rangle$, or finite such as the ring of integers modulo a prime number $\langle \mathbb{F}_p, +, \cdot \rangle$.

The ring of integers modulo a prime number p forms what is known as a Finite Field or Galois Field, denoted $\mathbb{F}_p$ or $GF(p)$. By taking the *quotienting out* the ring of integers modulo a prime number p, we are able to partition the ring into a set with cardinality p, where each integer falls into the partition represented by its remainder modulo p. By "quotienting out", we mean reducing the elements of $\mathbb{Z}$ to their remainder modulo p. Through this process of partitioning, we are able to create a finite field from the ring of integers, albeit with prime number of elements. The advantage, then, is that every element of our new finite field has a multiplicative inverse within the field.

An example of a finite field is the ring of integers $\mathbb{Z}$ modulo the prime number $p = 3$ (also denoted $\mathbb{F}_3$) with the following addition and multiplication tables:

Table 2.3. Group Table for $\langle \mathbb{Z}_3, + \rangle$.

+	**0**	**1**	**2**
0	0	1	2
1	1	2	0
2	2	0	1

Table 2.4. Group Table for $\langle \mathbb{Z}_3, \cdot \rangle$.

$\cdot$	**1**	**2**
1	1	2
2	2	1

The properties that make $\mathbb{F}_3$ a field are seen in the addition and multiplication tables. Notice that both the addition and multiplication tables are symmetric about the diagonal, so $\mathbb{F}_3$ is commutative over addition and multiplication. Note that each column of the addition table contains the 0 element, the additive inverse. Also note that each column of the multiplication table contains the 1 element, the multiplicative inverse, so $\mathbb{F}_3$ also meets the requirements of a division ring with unity.

2.2.5 Field Extensions

Though the ring of polynomials with integer coefficients $\mathbb{Z}[x]$ is not a field because it lacks the division ring property, it can be converted into a field in the same way that the ring of integers was converted into a field $\mathbb{F}_p$ by quotienting it out by a prime number p.

First, we add the constraint that the coefficients of our polynomials must be elements of our Galois Field $\mathbb{F}_p$, for example, a polynomial in $\mathbb{F}_3$ could only have coefficients 0, 1 or 2,

because these are the elements of $\mathbb{F}_3$. At this point, $\mathbb{F}_p[x]$ includes all polynomials of the form $p(x) = \sum_{i=1}^{\infty} a_i * x^i$ where $a_i \in \mathbb{F}_p$, with all, but finitely many coefficients that are zero.

Next, we quotient out the ring of polynomials $\mathbb{F}_p[x]$ by an ideal generated by an irreducible polynomial $q(x)$ of degree n. The irreducible polynomial $q(x)$ is an element of the original ring $\mathbb{F}_p[x]$.

> A polynomial $q(x)$ of degree $n > 0$ is said to be *irreducible* over [a field] $\mathbb{F}$ if and only if there do not exist polynomials $c(x)$ and $d(x)$ of degree greater than 0 over $\mathbb{F}$ such that $c(x) * d(x) = q(x)$, where multiplication is ordinary polynomial multiplication with coefficient operations in $\mathbb{F}$ [5].

By quotienting out $\mathbb{F}_p[x]$ [**mod** $q(x)$] (q is irreducible), we partition the ring $\mathbb{F}_p[x]$ into a smaller set that has the properties of a field. We will call this field $\mathbb{F}_{p^n}[x]$, where n is the degree of the polynomial q. Every polynomial in $\mathbb{F}_p[x]$, when reduced modulo the irreducible polynomial, leaves a remainder polynomial in the field $\mathbb{F}_{p^n}[x]$.

We will not go into details here, but if a root of the irreducible polynomial generates the multiplicative group (of cardinality $p^n - 1$) of the field $\mathbb{F}_{p^n}$ (and hence it has maximal order), the irreducible polynomial is said to be a *primitive polynomial*. If the irreducible polynomial is not primitive, the order of any root of the polynomial is less than $p^n - 1$.

2.2.6 The Algebraic Structure of LFSRs

LFSRs in $\mathbb{F}_2$ are generated by quotienting out the Ring of Polynomials with integer coefficients $\mathbb{F}_2[x]$ with an irreducible polynomial of degree n with coefficients in $\mathbb{F}_2$. If the irreducible polynomial is primitive, then the Ring of Polynomials will be partitioned into $2^n - 1$ cosets. Each coset contains all of the polynomials from the original ring $\mathbb{F}_2[x]$ that reduce to the same polynomial remainder modulo the irreducible polynomial. The coset is *represented* by the polynomial remainder, and all polynomials within a given coset are equivalent **mod** $q(x)$.

We can then use powers of x ($x^0, x^1, ..., x^{2^n-2}$) to generate all of the cosets without repetition, by finding (x^i for $i = 0, 1, ..., 2^n - 2$) modulo the primitive polynomial q(x). For example

if our primitive polynomial is $x^4 + x + 1$ over $\mathbb{F}_2$, our coset representatives are shown in polynomial form and in vector form (adapted from [5]):

$x^0(\textbf{mod}\ x^4 + x + 1) \equiv 1$ <-> 0001

$x^1(\textbf{mod}\ x^4 + x + 1) \equiv x$ <-> 0010

$x^2(\textbf{mod}\ x^4 + x + 1) \equiv x^2$ <-> 0100

$x^3(\textbf{mod}\ x^4 + x + 1) \equiv x^3$ <-> 1000

Computing $x^4(\textbf{mod}\ x^4 + x + 1)$ requires polynomial division:

$$\begin{array}{r|r} & 1 \\ \cline{2-2} x^4 + x + 1) & x^4 \\ & -x^4 - x - 1 \\ \cline{2-2} & -x - 1 \end{array}$$

so $x^4(\textbf{mod}\ x^4 + x + 1) \equiv x + 1$ <-> 0011 (*Note*: In binary, $-1 \equiv 1$, so $-x - 1 \equiv x + 1$)

$$\begin{array}{r|r} & x \\ \cline{2-2} x^4 + x + 1) & x^5 \\ & -x^5 - x^2 - x \\ \cline{2-2} & -x^2 - x \end{array}$$

so $x^5(\textbf{mod}\ x^4 + x + 1) \equiv x^2 + x$ <-> 0110

$\vdots$

$x^{14}(\textbf{mod}\ x^4 + x + 1) \equiv x^3 + 1$ <-> 1001

$x^{15}(\textbf{mod}\ x^4 + x + 1) \equiv 1$ <-> 0001 ($x^{15} \equiv x^0 = 1$)

The cyclic group generated by the characteristic (primitive) polynomial $x^4 + x + 1$ is represented by the vectors:

Table 2.5. Coset Representatives for $\mathbb{F}_2[x] \pmod{x^4 + x + 1}$

Powers of x ($\textbf{mod}\ x^4 + x + 1$)

x^i	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}
x^0	1	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1
x^1	0	1	0	0	1	1	0	1	0	1	1	1	1	0	0	0
x^2	0	0	1	0	0	1	1	0	1	0	1	1	1	1	0	0
x^3	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1	0

The coset vectors correspond to the states from the LFSR example in Table 2.1. Though the coset vectors are not in the same order as the LFSR states, the maximal length sequence (m-sequence) generated by taking the bits from the x^0 row is the same as the example LFSR sequence generated by the linear recurrence relation $x_{m+4} = x_{m+1} + x_m$ in Table 2.1. The LFSR generated the same m-sequence because its characteristic polynomial $P^{\star}(X) = 1 + X + X^4$ is the same as the primitive polynomial used to generate the field shown in Table 2.5. In general, an LFSR in $\mathbb{F}_2$ will produce an isomorphic m-sequence to that produced by coset representatives if both are based on the same characteristic primitive polynomial.

2.2.7 LFSR Summary

By choosing primitive polynomials as the characteristic polynomials for our LFSR, we know that our LFSR using a linear recurrence relation based on the characteristic polynomial will generate an m-sequence of period $2^n - 1$, where n is the degree of the primitive polynomial. We also know that the output of our LFSR will approach an average Hamming weight of 50% as the degree n of our characteristic polynomial is increased. Due to the computational simplicity of the linear recurrence relation, our LFSR can produce an extremely long pseudorandom sequence very quickly. Based on all these properties, LFSRs serve as an excellent building block for cryptographic PRBGs. However, because LFSRs alone are not cryptographically secure they must be combined with non-linear features (such as a BBS PRBG driver) to create a secure keystream.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Blum Blum Shub Generator

3.1 History

The BBS generator is a synthesis of ideas from several important mathematicians from the field of cryptography [6]. The idea of a trapdoor function and its use in public-key cryptography came from W. Diffie and M. Hellman in their landmark paper *New Directions in Cryptography* [7]. Diffie and Hellman's idea was implemented in the RSA (Rivest, Shamir, and Adleman) cryptosystem which relies on a trapdoor function of the form $x^s(\mathbf{mod} n)$ where $n = p * q$ with p, q distinct odd primes and $\gcd(s, \phi(n)) = 1$ [8]. The Euler Totient Function, $\phi(n)$, counts the number of integers from 1 to n that are co-prime to n [9]. Blum Blum Shub modifies the RSA trapdoor function by requiring p and q be congruent to $3(\mathbf{mod}\ 4)$. We will refer to primes of this form as Blum primes.

3.2 Overview

In 1986, Lenore Blum, Manuel Blum, and Michael Shub proposed a pseudorandom number generator of the form [6]:

$$x_i = x_{i-1}^2(\mathbf{mod}\ n),$$

where $n = pq$ is the product of two distinct primes p and q of the form:

$$p, q \equiv 3(\mathbf{mod}\ 4).$$

The generator outputs a sequence of bits $\mathbf{b} = b_0 b_1 \ldots$ where $b_i = parity(x_i)$. Note: The term *parity* can refer to *bit parity* (even or odd number of 1's bits in a string) or *integer parity* (even or odd number). The Blum-Blum-Shub generator uses *integer parity*, with *even* = 0 and *odd* = 1. Taking the integer parity of a number in binary form is the same as taking the least significant bit. The generator is inialized with a seed s that is a positive integer co-prime to n, so $x_0 = s^2(\mathbf{mod}\ n)$. A proof of the cryptographic security of BBS is given

in the original publication by Blum, Blum, and Shub [6], and a more explicit proof can be found in a paper by Pascal Junod [10].

3.3 Number Theory

To explain how the BBS PRBG works, some additional number theory topics are covered.

3.3.1 Quadratic Residues

Let $n \in \mathbb{N}$. Then $a \in \mathbb{Z}_n^*$ is called a *quadratic residue* modulo n if there exists $b \in \mathbb{Z}_n^*$ such that

$$a \equiv b^2 (\textbf{mod } n)$$

where $\mathbb{Z}_n^*$ is the multiplicative group of integers modulo n [10]. The set of quadratic residues modulo n is denoted QR_n. For example, the multiplicative group $\mathbb{Z}_{21}^*$ consists of all integers co-prime to 21: $\{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$. Squaring each of these values modulo n, we are left with $QR_{21} = \{1, 4, 16\}$, quadratic residues modulo 21. The elements of the set $\mathbb{Z}_n^* \setminus QR_n$ are called the *quadratic non-residue* modulo n, denoted by QNR_n. For our example, $QNR_{21} = \{2, 5, 8, 10, 11, 13, 17, 19, 20\}$, the set of quadratic non-residues modulo 21. For n, the product of two primes p and q, as n grows large, it becomes impractical to compute all values of QR_n, so we use a test to determine if a specific number is a quadratic residue modulo n. For this test, we must first define the Legendre symbol and the Jacobi symbol where n is the product of two primes p and q. The full theorems and associated proofs of the equations found in this section may be found in [10].

3.3.2 Legendre Symbol

Let p be an odd prime. For $a \in \mathbb{Z}_p^*$, the Legendre symbol $\left(\frac{a}{p}\right)$ is defined as [10]:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & p|a \\ 1 & a \in QR_p \\ -1 & a \notin QR_p. \end{cases} \tag{3.1}$$

To determine the output of the Legendre symbol, rather than finding all quadratic residues

manually, we can directly calculate the Legendre symbol for a given integer $a \in \mathbb{Z}_p^*$, by Equation (3.2):

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} (\mathbf{mod}\ p). \tag{3.2}$$

One result of Equation (3.2) is Equation 3.3:

$$|QR_p| = |QNR_p| = \frac{p-1}{2}, \tag{3.3}$$

which states that the cardinality of the set of quadratic residues is equal to the cardinality of the set of quadratic non-residues, and that both of these are equal to half of the elements of the multiplicative group $\mathbb{Z}_p^*$.

Another result of Equation (3.2) for integers a and b, with p an odd prime is Equation (3.4) [10].

$$\left(\frac{a}{p}\right) * \left(\frac{b}{p}\right) = a^{\frac{p-1}{2}} * b^{\frac{p-1}{2}} (\mathbf{mod}\ p) = (a * b)^{\frac{p-1}{2}} = \left(\frac{a * b}{p}\right). \tag{3.4}$$

Equation (3.2) shows the multiplicative property of the Legendre symbol which is needed for Equation (3.6).

3.3.3 Jacobi Symbol

The idea of the Legendre symbol is extended to all multiplicative groups with odd moduli by the Jacobi symbol:

Let n be an odd integer with prime factorization:

$$n = \prod_{i=1} p_i^{e_i},$$

where e_i is a positive integer.

Let $a \in \mathbb{Z}_n^*$. Then the Jacobi symbol $\left(\frac{a}{n}\right)$ is defined:

$$\left(\frac{a}{n}\right) = \prod_{i=1} \left(\frac{a}{p_i}\right)^{e_i}. \tag{3.5}$$

Equation (3.5) leads to the multiplicative property of the Jacobi symbol:

$$\left(\frac{a * b}{n}\right) = \left(\frac{a}{n}\right) * \left(\frac{b}{n}\right). \tag{3.6}$$

Having defined the Legendre and Jacobi symbols and some of their basic properties, we review some lemmas that help us find the square roots of a quadratic residue modulo n.

Lemma 1 [10]:

Let p be a prime and α a non-zero element of $\mathbb{Z}_p^*$. Then,

$$-\alpha = \alpha \iff p = 2.$$

This implies that if $p \neq 2$, as is the case for BBS, then $-\alpha \neq \alpha$, no element of the multiplicative group is equal to its additive inverse.

Lemma 2 [10]:

For p an odd prime and for α, β non-zero elements of $\mathbb{Z}_p^*$:

$$\alpha^2 = \beta^2 \iff (\alpha = \beta) \vee (\alpha = -\beta).$$

This follows from Lemma 1, because if $\alpha = \beta$, α cannot also equal $-\beta$, because this only occurs if $p = 2$. Therefore if α and β are unique elements of $\mathbb{Z}_p^*$, then one is in the lower half of the set ($1 \leq \alpha_1 \leq \frac{p-1}{2}$) and one is in the upper half of the set ($\frac{p+1}{2} \leq \alpha_2 \leq p - 1$. For example, if $p = 7$, the square of each of the elements of $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$ is respectively $(1, 4, 2, 2, 4, 1)$. One can see that the squares of the first half of the set are symmetric to the squares of the second half of the set, and conversely that the two unique

square roots of a given quadratic residue are symmetric within the set $\mathbb{Z}_7^*$, or, in other words, $\alpha_1 = -\alpha_2(\textbf{mod}\ p)$.

Lemma 3 [10]:

Let $n = p_1 * p_2 * \cdots * p_k$, where $p_1, \ldots, p_k$ are distinct, odd primes, and $k \geq 2$. An element $\alpha \in \mathbb{Z}_n^*$ is a quadratic residue modulo n if and only if each component of its Chinese Remainder Theorem (CRT) transform with respect to the moduli $p_1, \ldots, p_k$ is a quadratic residue of $\mathbb{Z}_{p_i}^*$, where p_i is the modulus corresponding to that component.

The proof of Lemma 3 is excluded for brevity, but may be found in [10]. The CRT gives a unique solution modulo n for a system of equations modulo the distinct prime factors of n. Lemma 3 uses this fact to show that for every quadratic residue modulo n, there is a unique system of quadratic residues modulo the prime factors of n. Lemmas 1-3 are necessary to prove Theorems 1 and 2.

Theorem 1 [10]:

Let n be an odd integer. If $a \in QR_n$, then the number of distinct square roots of a is exactly 2^k, where k is the number of distinct prime factors of n.

For the case of BBS, n has two distinct prime roots p and q, so by Theorem 1, an element of the quadratic residue of a Blum prime has four distinct square roots.

Theorem 2 [10]:

Let $n = p * q$ be the product of two distinct odd primes. Exactly half the elements of $\mathbb{Z}_n^*$ have Jacobi symbol $(+1)$, and the other half have Jacobi symbol (-1). We denote these two sets respectively by $\mathbb{Z}_n^*(+1)$ and $\mathbb{Z}_n^*(-1)$. None of the elements of $\mathbb{Z}_n^*(-1)$, and exactly half of the elements of $\mathbb{Z}_n^*(+1)$ are quadratic residues.

This tells us that if n is the product of two distinct odd primes, as in the case of BBS, exactly $1/4$ of the elements of $\mathbb{Z}_n^*$ are quadratic residues, and these are the elements whose two CRT factors both have (+1) Legendre symbols (they are both quadratic residues modulo their respective prime). This also tells us that the elements of $\mathbb{Z}_n^*$ with a Jacobi symbol of (+1) are not all quadratic residues modulo n, but that exactly half are. This leads us to the definition of Blum primes and the explanation for the requirement that they be equivalent

to $3(\textbf{mod } 4)$.

3.3.4 Definition of Blum Primes

A prime number p with $p \equiv 3(\textbf{mod } 4$ is called a Blum prime number [10].

The following theorems give the explanation for why this form was chosen.

Theorem 3 [10]:

Let p be an odd prime. Then,

$$-1 \in QNR_p \Leftrightarrow p \; is \; a \; Blum \; prime.$$

By Equation (3.2),

$$\left(\frac{-1}{p}\right) = -1^{\frac{p-1}{2}} (\textbf{mod } p)$$

If $p \equiv 1(\textbf{mod } 4)$, (example: $p = 5, 9, 13, ...$), then $\frac{p-1}{2}$ will always be even, producing a Legendre symbol of (+1), meaning that -1 is an element of the quadratic residue modulo p. With Blum primes, on the other hand, (example: $p = 3, 7, 11, ...$), $\frac{p-1}{2}$ will always be odd, producing a Legendre symbol of (-1). Requiring Blum primes to be equivalent to $3(\textbf{mod } 4)$ ensures that $-1 \in QNR_p$. This fact is used to prove Theorem 4.

Theorem 4 [10]:

Let $n = p * q$ be the product of two Blum primes. Let $a \in QR_n$. Then a has exactly four square roots, exactly one of which is in QR_n itself [10].

Because exactly one of the four roots is a quadratic residue modulo n, we can define $\sqrt{a}$ as the one root that is itself a quadratic residue in QR_n. This allows us to show that the iterative squaring process in BBS is a *permutation*, a bijection from a set onto the same set [10].

Theorem 5:

Let $n = p * q$ be the product of two Blum primes. The function:

$$f : QR_n \to QR_n, \text{ defined by } x \to x^2(\textbf{mod}\, n),$$

is a permutation [10].

If non-Blum primes are used ($p, q \equiv 1(\textbf{mod}\, 4)$, then an element $a \in QR_n$ may have multiple square roots in QR_n. The result is that a permutation is not achieved, and instead the repeated squaring process quickly reduces the pool of quadratic residues to a few of the original elements, regardless of the seed chosen.

In contrast, with Blum primes, each element of QR_n is squared onto a unique element of QR_n. This results in large cycle lengths for a majority of seeds chosen.

For example, with non-Blum primes $p = 5$, $q = 13$, and $n = 65$, then $QR_n = \{1, 4, 9, 14, 16, 29, 36, 49, 51, 56, 61, 64\}$. If the elements of QR_n are then squared modulo n, the result is (1, 16, 16, 1, 61, 61, 61, 61, 1, 16, 16, 1) respectively. While QR_n consisted of 12 elements, the second round of squaring produced a set of only 3 remaining elements (1, 16, and 61) with a cycle length of 2 ($16^2(\textbf{mod}\, 65) = 61$; $61^2(\textbf{mod}\, 65) = 16$).

If Blum primes are used, due to Theorem 4, we see much longer cycle lengths. For example, if Blum primes $p = 7$, $q = 11$ are used, then $n = 77$ and $QR_n = \{1, 4, 9, 15, 16, 23, 25, 36, 37, 53, 58, 60, 64, 67, 71\}$ with 15 elements. Squaring each of these elements modulo n, we get (1, 16, 4, 71, 25, 67, 9, 64, 60, 37, 53, 58, 15, 23, 36) respectively and our set of quadratic residues still has 15 unique elements. Looking at the cycles produced by repeated squaring modulo n, QR_n is broken up into 4 cyclic groups: $(4, 16, 25, 9)$, $(15, 71, 36, 64)$, $(23, 67)$, and $(37, 60, 58, 53)$. While the smallest of these has a cycle length of 2, the majority have a period of 4. The reduced totient function, also called the Carmichael function, $\lambda(x)$, is used to calculate the probable period of a BBS PRBG. The period of a BBS PRBG always divides $\lambda(\lambda(n))$, and is usually equal to $\lambda(\lambda(n))$, depending on the seed [6]. In the example shown here, all of the cycle lengths divide $\lambda(\lambda(77)) = 4$, and most are equal to 4. This example shows the importance of Theorem 5 and the reason why the primes used for p and q must be Blum primes.

3.4 Summary

The ability to generate long cycle lengths with unpredictable bit outputs using the relatively simple $x_i = x_{i-1}^2 (\mathbf{mod}\ n)$ function makes BBS an excellent PRBG. In practice, the Blum primes must be very large (on the order of 64 bits each) to prevent brute force attacks on the PRBG. Blum primes this large make the PRBG too slow for many applications. This research considers the possibility of combining the security of the BBS PRBG with the speed of the LFSR PRBG to achieve a good mix of speed and security.

CHAPTER 4: Methodology

In order to evaluate the suitability of a BBS/LFSR combiner, we coded a reversible combiner in Python, which was based around the LFSR.py program written by Michael Troncoso [11]. The reversible combiner could produce sequences using an LFSR base stream with a BBS driver or a BBS base stream with an LFSR driver. The base stream generated a pseudorandom string of bits, and the driver took a random sampling of those bits via the following method:

1. When the kth bit of the driver sequence was a 0, the kth bit of the base sequence was skipped.

2. When the kth bit of the driver sequence was a 1, the kth bit of the base sequence was added to the combiner sequence. This type of combiner in known as a *shrinking generator*.

Table 4.1. Example of a shrinking generator. When the driver bit is 0, the corresponding base stream bit is not added to the output sequence. When the driver bit is 1, the base stream bit is added to the output sequence.

(base stream)	001010100
(driver)	011011011
(output)	_01_10_00

We used our combiner to generate strings of 1,000,000 bits in length, which was required for the NIST tests. Our original combiner took around an hour to generate 1,000,000 bits, so a faster implementation was needed to reach the 10,000,000 bits desired for the NIST test.

During initial testing, we found that the combiner using a BBS base stream and LFSR driver had a major weakness. Many BBS sequences are not balanced, so the resulting combiner sequence was not balanced either. Because a truly random sequence should be approximately balanced, this type of combiner performed poorly on every NIST test conducted. Due to

these poor results, we rejected the BBS base stream with LFSR driver as an unsuitable combiner. Instead we focused the remainder of the testing on combiners using an LFSR base stream with a BBS driver. Because our initial program was too slow to generate a large number of bits, we wrote a new combiner that always used a BBS driver and was much faster. With the new combiner, and using a personal computer, we were able to generate balanced pseudorandom sequences of 10,000,000 bits with good linear complexity in under five minutes.

After generating a 10,000,000 bit sequence, we broke it into 10 sets each of 1,000,000 bits in length, and then evaluated it using the NIST tests. The NIST test evaluates the plausibility that a given sequence is truly random through a series of statistical subtests. The null hypothesis for testing is that the sequence is truly random. The alternative hypothesis is that the sequence is not truly random. The test uses a default significance level of $\alpha = .01$. There are 188 subtests and each one is performed for each of the 10 sets of 1,000,000 bits. Each run of 1,000,000 bits receives a p-value based on the likelihood that the null hypothesis is true. The group of 10 sequences also receives an overall p-value which considers the likelihood of the distribution of p-values from the 10 tests, using the χ^2 ("chi-squared") distribution. If the overall subtest p-value is ≥ 0.01, the subtest is considered passing. To judge the overall performance of a given combiner, we considered the number of subtests passed out of the 188 conducted.

We conducted nine experiments using different variations of the combiners.

4.1 Experiment Number One: BBS Driver with Varying Primitive Polynomials

For the first experiment we tested a combiner using an LFSR base stream with a BBS driver. We used primitive polynomials from degree 3 up to deg 21, holding the BBS inputs constant. We noticed that the linear complexity of the combiner increased linearly with the degree of the LFSR primitive polynomial. This tells us that if we are trying to greatly increase the linear complexity, we should look to other factors than the LFSR degree, such as the BBS parameters.

We also observed that the average Hamming weight of the LFSR approaches 50% as the

degree of the polynomial increases. LFSRs do not have a perfectly balanced output because the all 0's state is excluded. Therefore, a full period LFSR output always has one more 1 than 0. Due to this slight imbalance, low degree polynomial LFSR streams are more imbalanced than high degree polynomial LFSR streams. This issue is extended to combiners with an LFSR base stream, because the resulting combiner stream is more unbalanced than a combiner with a higher degree LFSR. Because balancedness is expected in a truly random bitstream, all the sequences we tested with polynomials of degree less than 15 failed the Frequency Test, the first and most basic subtest of the NIST test. The conclusion from this experiment is that the degree of the polynomial for the combiner should be greater than or equal to 15.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Hamming Weight	p	q	M	seed	Period	Hamming Weight	Linear Complexity	Period	Driver	Hamming Weight	Linear Complexity	NIST (**) Passed	NIST(%) Passed
Experiment 1	CB3	3	7	0.57143	59	71	4189	479	84	0.40476	80	84	BBS	0.44118	34	0	0%
	CB4	4	15	0.53333	59	71	4189	479	84	0.40476	80	420	BBS	0.51765	136	0	0%
	CB5	5	31	0.51613	59	71	4189	479	84	0.40476	80	2604	BBS	0.51613	170	0	0%
	CB6	6	63	0.50794	59	71	4189	479	84	0.40476	80	252	BBS	0.47059	64	0	0%
	CB7	7	127	0.50394	59	71	4189	479	84	0.40476	80	10668	BBS	0.50394	238	0	0%
	CB8	8	255	0.50196	59	71	4189	479	84	0.40476	80	7140	BBS	0.50658	272	1	1%
	CB9	9	511	0.50098	59	71	4189	479	84	0.40476	80	6132	BBS	0.49154	306	1	1%
	CB10	10	1023	0.50049	59	71	4189	479	84	0.40476	80	28644	BBS	0.49819	340	3	2%
	CB11	11	2047	0.50024	59	71	4189	479	84	0.40476	80	171948	BBS	0.50024	374	3	2%
	CB12	12	4095	0.50012	59	71	4189	479	84	0.40476	80	16380	BBS	0.50196	408	1	1%
	CB13	13	8191	0.50006	59	71	4189	479	84	0.40476	80	688044	BBS	0.50006	442	126	67%
	CB14	14	16383	0.50003	59	71	4189	479	84	0.40476	80	458724	BBS	0.50049	476	82	44%
	CB15	15	32767	0.50001	59	71	4189	479	84	0.40476	80	393204	BBS	0.49995	510	53	28%
	CB18	18	262143	0.5	59	71	4189	479	84	0.40476	80	1048572	BBS	0.49951	612	73	39%
	CB21	21	2097151	0.49997	59	71	4189	479	84	0.40476	80	25165812	BBS	0.50018	714	160	85%

Figure 4.1. Experiment 1. Polynomial degree increases, while BBS properties are held constant. ** The number of NIST subtests passed is out of a total of 188 subtests.

4.2 Experiment Number Two: LFSR Driver with Varying Primitive Polynomials

Experiment number two used the same polynomials and BBS inputs as experiment number one, but the combiner used BBS as the base sequence and the LFSR as the driver. These experiments all performed very poorly on the NIST test due to lack of balancedness. According to NIST Special Publication 800-22-1a, "All subsequent tests depend on the passing of [the Frequency] test" [12]. Although this test only used one set of BBS inputs,

and therefore the BBS sequence always had the same Hamming weight, most the BBS sequences that we selected had Hamming weights too far from 50% to be acceptable as base sequences. The only way to reliably get a balanced BBS stream would be careful selection of the parameters p, q, and the seed. The problem is that limiting these parameters to just a few options defeats the purpose of having a large pool of secret parameters to choose from, to make a brute force attack infeasible. A *brute force* attack in cryptography is one in which the attacker tries every combination of parameters, in order to reverse engineer the keystream, and decrypt the plaintext message.

Due to the fact that many BBS sequences are too unbalanced to use as a suitable base stream, we did not pursue any more experiments with a BBS base stream and LFSR driver. All subsequent experiments use an LFSR base stream with a BBS driver.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. Comp.	NIST (**)	NIST (%)
Experiment 2	CL3	3	7	0.57143	59	71	4189	479	84	0.40477	80	84	LFSR	0.312505	48	0	0%
	CL4	4	15	0.53333	59	71	4189	479	84	0.40477	80	420	LFSR	0.419655	220	0	0%
	CL5	5	31	0.51613	59	71	4189	479	84	0.40477	80	2604	LFSR	0.404785	1280	1	1%
	CL6	6	63	0.50794	59	71	4189	479	84	0.40477	80	252	LFSR	0.406265	127	0	0%
	CL7	7	127	0.50392	59	71	4189	479	84	0.40477	80	10668	LFSR	0.404725	5120	4	2%
	CL8	8	255	0.50195	59	71	4189	479	84	0.40477	80	7140	LFSR	0.403995	3581	2	1%
	CL9	9	511	0.50098	59	71	4189	479	84	0.40477	80	6132	LFSR	0.40231	3070	2	1%
	CL10	10	1023	0.50049	59	71	4189	479	84	0.40477	80	28644	LFSR	0.406285	14332	1	1%
	CL11	11	2047	0.50023	59	71	4189	479	84	0.40477	80	171948	LFSR	0.40478	81920	35	19%
	CL12	12	4095	0.5001	59	71	4189	479	84	0.40477	80	16380	LFSR	0.40525	8189	6	3%
	CL13	13	8191	0.50001	59	71	4189	479	84	0.40477	80	688044	LFSR	0.40406	100000	36	19%
	CL14	14	16383	0.49997	59	71	4189	479	84	0.40477	80	458724	LFSR	0.404885	99998	38	20%
	CL15	15	32767	0.49979	59	71	4189	479	84	0.40477	80	393204	LFSR	0.405635	99999	37	20%
	CL18	18	262143	0.50011	59	71	4189	479	84	0.40477	80	1048572	LFSR	0.40448	100000	41	22%
	CL21	21	2097151	0.49982	59	71	4189	479	84	0.40477	80	25165812	LFSR	0.404785	100000	42	22%

Figure 4.2. Experiment 2. This is the only experiment with a BBS base sequnce and LFSR driver. Polynomial degree increases, while BBS properties are held constant. These combiners performed poorly due to a lack of balancedness. The lack of balancedness in the combiner output was mainly due to an unbalanced BBS base sequence. ** The number of NIST subtests passed is out of a total of 188 subtests.

4.3 Experiment Number Three: BBS Driver with Varying BBS Moduli

For experiment number three, we kept the LFSR parameters constant, and varied the p and q inputs to the BBS Driver. All of these combiners performed poorly on the NIST test due to the fact that they failed the frequency test. However, these tests helped us to develop some initial ideas about the relationship between the BBS parameters and the resulting combiner characteristics.

Plotting the natural log of the combiner's linear complexity vs. the natural log of M, we see that the combiner's linear complexity tends to increase as M increases, but it is not a strictly linear relationship; there is some variance involved. There is certainly a positive correlation between the two factors. Using this relationship, we can set a lower bound on M by setting minimum values for p and q, in order to increase the probability that our combiner sequence has a high linear complexity, and therefore a better performance on the NIST test.

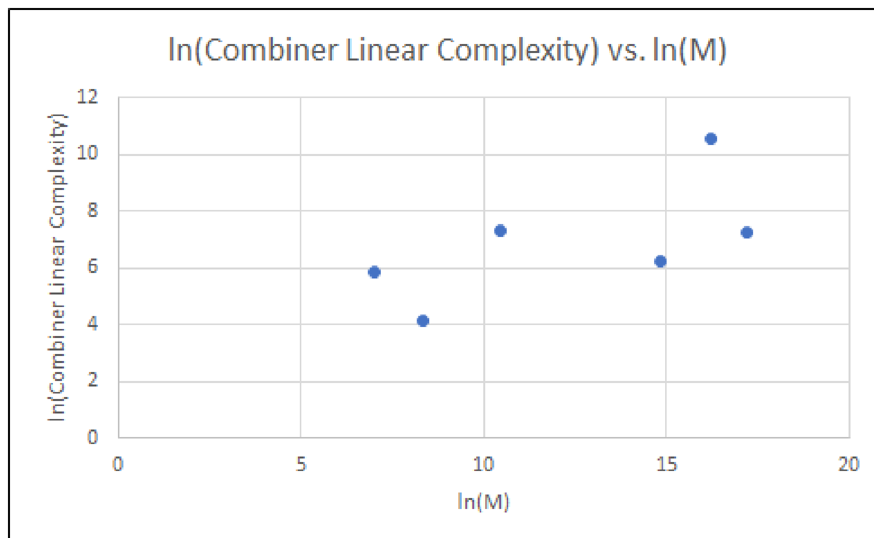

Figure 4.3. ln(combiner linear complexity) vs. ln(M) from experiment number three. The plot shows the effect on a combiner's linear complexity as M increases, on a log-log scale.

With the data from experiment number three, we also plotted the combiner's period vs. the combiner's linear complexity on a log-log scale and saw a positive correlation between these factors.

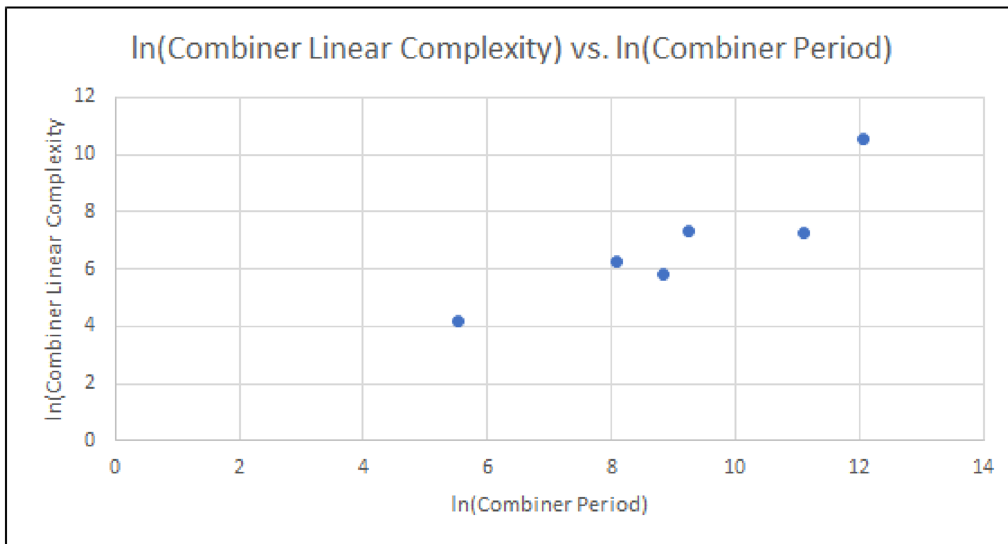

Figure 4.4. ln(combiner linear complexity) vs. ln(combiner period) from experiment number three. The plot shows the effect on combiner linear complexity as combiner period increases, on a log-log scale.

The combiner period is the least common multiple of the BBS period and LFSR period. The LFSR period is fixed for a given primitive polynomial. The BBS period, $p_{BBS} = \lambda(\lambda(M))$ where λ is the Carmichael function. A plot of the BBS period vs. M shows the upper bound of the BBS period is linear and increases with M. However, many values are clustered near the bottom of the plot, even for very large values of M.

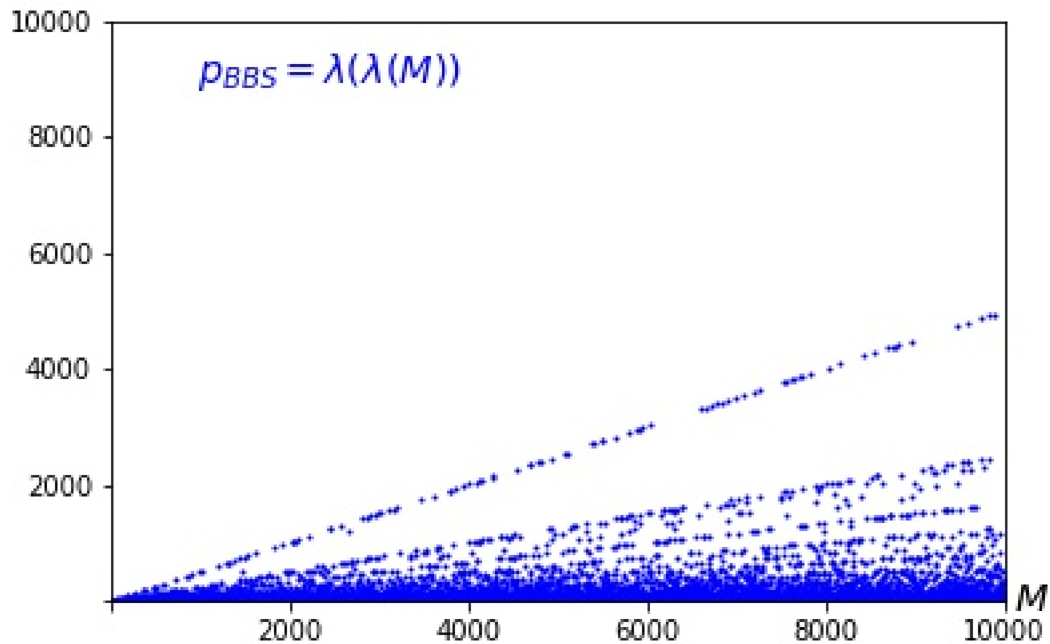

Figure 4.5. BBS Period vs. M. The plot shows that even as M grows very large, though the upper limit on the BBS period is increasing, the actual BBS period may be small. In fact, most of the points are clustered near the lower end of the range. Adapted from [13].

This shows that the combiner algorithm should check the period of BBS (not just the size of M) before generating the sequence to ensure that the BBS period is sufficiently large. If the randomly selected p and q values result in a BBS period that is too small the combiner period and combiner linear complexity will likely be too small to perform well on the NIST test. In the following experiments, we looked for an appropriate lower bound on BBS period and LFSR degree to reliably generate combiner sequences with excellent NIST test results.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. Comp.	NIST (**)	NIST (%)
Experiment 3	CB6_1	6	63	0.50794	23	47	1081	479	110	0.51818	110	6930	BBS	0.50794	342	2	1%
	CB6_2	6	63	0.50794	59	71	4189	479	84	0.40476	80	252	BBS	0.47059	64	0	0%
	CB6_3	6	63	0.50794	167	211	35237	479	492	0.50813	491	10332	BBS	0.50667	1500	1	1%
	CB6_4	6	63	0.50794	1991	1423	2833193	479	156	0.5641	155	3276	BBS	0.49351	528	0	0%
	CB6_5	6	63	0.50794	3259	3343	10894837	479	25020	0.50316	25020	175140	BBS	0.50933	37767	5	3%
	CB6_6	6	63	0.50794	5303	5623	29818769	479	9360	0.50214	935	65520	BBS	0.51672	1410	1	1%

Figure 4.6. Experiment 3. Polynomial degree held constant, while BBS parameters p and q increase. These combiners performed poorly on the NIST test due to a lack of balancedness. The lack of balancedness in the LFSR base sequence caused the overall combiner sequences to be unbalanced. ** The number of NIST subtests passed is out of a total of 188 subtests.

4.4 Experiment Number Four: BBS Driver with Varying BBS Seed

For experiment number four, we kept all combiner inputs constant, except for the BBS seed. The combiner used a degree-15 primitive polynomial and BBS parameters $p = 167$ and $q = 211$. The experiment revealed that, although the BBS Period remained constant for all tests ($p_{BBS} = 492$), the BBS linear complexity varied from 163 to 492. This was reflected the the combiner output in which every sequence had the same period, but the combiner's linear complexity varied from 1200 to 3705. This shows that there is not an exactly linear relationship between BBS period and BBS linear complexity, but that if the BBS period is sufficiently large, the BBS and combiner's linear complexities will likely be sufficiently large as well. Because the BBS and combiner's linear complexities cannot be calculated before the keystream is generated, a good alternative is to place a lower bound on the BBS period (which can be calculated in advance) with sufficient room for the possibility of a drop in linear complexity in the event that a less-than-ideal BBS seed is randomly selected.

The combiners in this experiment all performed very poorly on the NIST test due to the lack of balancedness. A higher degree polynomial was needed to produce a consistently balanced combiner sequence. The linear complexities for these combiners were sufficiently high that the they were not a limiting factor for these combiners' NIST performances.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. Comp.	NIST (**)	NIST (%)
Experiment 4	CB15_1	15	32767	0.50001	167	211	35237	2201	492	0.48577	492	16121364	BBS	0.50002	3585	86	46%
	CB15_2	15	32767	0.50001	167	211	35237	9325	492	0.49797	492	16121364	BBS	0.49997	3675	110	59%
	CB15_3	15	32767	0.50001	167	211	35237	1033	492	0.4939	492	16121364	BBS	0.49997	3645	85	45%
	CB15_4	15	32767	0.50001	167	211	35237	4179	492	0.50203	492	16121364	BBS	0.50001	3705	84	45%
	CB15_5	15	32767	0.50001	167	211	35237	1931	492	0.52439	246	16121364	BBS	0.50007	1935	78	41%
	CB15_6	15	32767	0.50001	167	211	35237	8117	492	0.50203	492	16121364	BBS	0.49996	3705	83	44%
	CB15_7	15	32767	0.50001	167	211	35237	7364	492	0.54268	164	16121364	BBS	0.50006	1335	106	56%
	CB15_8	15	32767	0.50001	167	211	35237	7737	492	0.49797	492	16121364	BBS	0.50012	3675	111	59%
	CB15_9	15	32767	0.50001	167	211	35237	6219	492	0.4878	163	16121364	BBS	0.50002	1200	59	31%
	CB15_10	15	32767	0.50001	167	211	35237	3439	492	0.4878	245	16121364	BBS	0.50004	1800	81	43%

Figure 4.7. Experiment 4. All parameters remained constant, except for BBS seed. These combiners performed poorly on the NIST test due to a lack of balancedness. The lack of balancedness in the LFSR base sequence caused the overall combiner sequences to be unbalanced. ** The number of NIST subtests passed is out of a total of 188 subtests.

4.5 Experiment Number Five: BBS Driver with Varying Blum Primes

With this experiment, we began to see combiner sequences that passed nearly all of the NIST subtests (5 out of 6 sequences passed 182 or more of the 188 NIST subtests). For this experiment we kept the BBS seed and LFSR parameters constant and varied the BBS parameters p and q. The combiner that performed the worst was CB21_2, which had the lowest BBS period of the group ($p_{BBS} = 84$), the lowest BBS linear complexity of 80, and the lowest combiner period and combiner linear complexity (714). CB21_2 passed 155 of 188 NIST subtests. This experiment confirmed the importance of placing a lower bound on the BBS Period. This experiment also confirmed that a combiner sequence with a linear complexity as low as 1197 (CB21_1) could still achieve excellent results on the NIST test (183/188 subtests passed).

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. Comp.	NIST (**)	NIST (%)
Experiment 5	CB21_1	21	2097151	0.49995	23	47	1081	479	110	0.51818	110	230686610	BBS	0.49997	1197	183	97%
	CB21_2	21	2097151	0.49997	59	71	4189	479	84	0.40476	80	25165812	BBS	0.50018	714	155	82%
	CB21_3	21	2097151	0.49995	167	211	35237	479	492	0.50813	491	1031798292	BBS	0.49991	5250	184	98%
	CB21_4	21	2097151	0.49996	1991	1423	2833193	479	156	0.5641	155	327155556	BBS	0.49993	1848	182	97%
	CB21_5	21	2097151	0.49995	3259	3343	10894837	479	25020	0.50316	25020	52470718020	BBS	0.49978	264369	182	97%
	CB21_6	21	2097151	0.49996	5303	5623	29818769	479	9360	0.50214	935	19629333360	BBS	0.49969	9870	185	98%

Figure 4.8. Experiment 5. Polynomial degree held constant, while BBS parameters p and q increase. These combiners performed well on the NIST test. ** The number of NIST subtests passed is out of a total of 188 subtests.

4.6 Experiment Number Six: BBS Driver with Varying BBS Seed

For this experiment, we kept the LFSR parameters constant, with the same degree-21 primitive polynomial from experiment number five. We also kept the BBS parameters p and q constant at $p = 59$ and $q = 71$. We varied the BBS seed parameter using randomly generated seeds. With a constant p and q, the values for M, BBS period, and combiner period remained constant for all combiners. Varying the BBS seed resulted in a range of the BBS Linear Complexities from 26 to 84. BBS linear complexity had a strong correlation to the resulting combiner linear complexity, and combiner linear complexities ranged from 252 to 903.

The performance of these combiners varied from scores of 43/188 to 184/188 on the NIST test. This shows that the BBS seed has a significant impact on resulting BBS linear complexity, which then affects the combiner's linear complexity. This experiment confirmed the need for a lower limit on BBS period and that limit needed to be much higher than 84 to improve the likelihood of excellent performance on the NIST test.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. Comp.	NIST (**)	NIST (%)
Experiment 6	CB21_7	21	2097151	0.49995	59	71	4189	23	84	0.5119	81	25165812	BBS	0.50005	903	183	97%
	CB21_8	21	2097151	0.49997	59	71	4189	131	84	0.38095	78	25165812	BBS	0.49982	672	68	36%
	CB21_9	21	2097151	0.49997	59	71	4189	1619	84	0.53571	28	25165812	BBS	0.50032	315	179	95%
	CB21_10	21	2097151	0.49996	59	71	4189	3323	84	0.42857	26	25165812	BBS	0.49981	252	43	23%
	CB21_11	21	2097151	0.49995	59	71	4189	4159	84	0.5119	81	25165812	BBS	0.50004	903	183	97%
	CB21_12	21	2097151	0.49996	59	71	4189	659	84	0.4881	84	25165812	BBS	0.50016	861	184	98%

Figure 4.9. Experiment 6. All parameters held constant except for BBS seed. These combiners had mixed results on the NIST test. Combiners with lower linear complexity tended to perform worse. A combiner's linear complexity was driven by BBS linear complexity. To get consistently higher BBS linear complexities, a larger BBS period is needed. ** The number of NIST subtests passed is out of a total of 188 subtests.

Another finding from this experiment was a lower range of combiner linear complexity required to pass the linear complexity subtest of the NIST test. $CB21_{10}$ failed the linear complexity subtest with a combiner linear complexity of 252. In contrast, $CB21_9$ passed the linear complexity subtest with a linear complexity of 315. This shows that a combiner linear complexity of at least 315 should be sufficient to pass the linear complexity subtest. By requiring the BBS period to be ≥ 300, experiment number eight had consistently high combiner linear complexities and all combiners passed the NIST linear complexity subtest.

4.7 Experiment Number Seven: BBS Driver with Degree 18 to 32 Polynomials

For experiment number seven, we ran a series of higher degree polynomials (18-32 degree) and for each polynomial we ran two combiners: the first with $p = 23$ and $q = 47$ (with a BBS period of 110) and the second with $p = 59$ and $q = 71$ (with a BBS period of 78). The BBS seed was kept constant at 11 for all tests except the 32-degree polynomials. From this experiment, 6/12 tests scored 182/188 or higher on the NIST test. The others had mixed results, with one combiner failing all tests! During this experiment, we were still trying to find a good lower bound for BBS period, and these tests had BBS periods of 110 or 78, well below the minimum BBS period of 3,000 in our final recommendation.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. Comp.	NIST (**)	NIST (%)
Experiment 7	CB18_1	18	262143	0.50001	23	47	1081	11	110	0.51818	110	28835730	BBS	0.50005	1026	182	97%
	CB18_2	18	262143	0.5	59	71	4189	11	84	0.38095	78	1048572	BBS	0.49964	576	99	53%
	CB19_1	19	524287	0.49995	23	47	1081	11	110	0.51818	110	57671570	BBS	0.49994	542326	39	21%
	CB19_2	19	524287	0.49995	59	71	4189	11	84	0.38095	78	44040108	BBS	0.50274	598612	0	0%
	CB20_1	20	1048575	0.49999	23	47	1081	11	110	0.51818	110	2097150	BBS	0.49988	1140	167	89%
	CB20_2	20	1048575	0.49998	59	71	4189	11	84	0.38095	78	29360100	BBS	0.50002	640	149	79%
	CB22_1	22	4194303	0.49995	23	47	1081	11	110	0.51818	110	461373330	BBS	0.49999	1254	184	98%
	CB22_2	22	4194303	0.49996	59	71	4189	11	84	0.38095	78	117440484	BBS	0.50003	704	141	75%
	CB23_1	23	8388607	0.49996	23	47	1081	11	110	0.51818	110	922746770	BBS	0.4999	1311	184	98%
	CB23_2	23	8388607	0.5	59	71	4189	11	84	0.38095	78	704642988	BBS	0.49982	736	182	97%
	CB32_1	32	4294967295	0.49965	59	71	4189	23	84	0.5119	81	120259084260	BBS	0.49964	1376	187	99%
	CB32_2	32	4294967295	0.49967	5303	5623	29818769	7523	9360	0.4968	4672	2680059592080	BBS	0.4997	74400	188	100%

Figure 4.10. Experiment 7. This experiment included combiners of degrees 18-32 and with various BBS parameters. These combiners had mixed results on the NIST test. Combiners with lower linear complexity or a lack of balancedness tended to perform worse. Combiner linear complexity showed a strong correlation with combiner period. Combiner period is the least common multiple of the LFSR and BBS periods. To get consistently better results, larger LFSR and BBS periods are needed. ** The number of NIST subtests passed is out of a total of 188 subtests.

Of the four best tests, the lowest linear complexity was 736. This shows that a linear complexity above 1000 is not needed for a combiner sequence to perform very well on the NIST test. In contrast, the worst combiner sequence that failed all NIST subtests had a linear complexity of 598,612 and an average Hamming weight of 0.50274. This shows that a good combiner sequence must be very close to balanced (Average Hamming Weight of 0.5000 ± 0.0002) and have a moderately high linear complexity, to pass randomness testing.

The major takeaway from experiment number seven was that we could not guarantee good combiner performance through increased LFSR degree alone, but that we would need to raise the lower limit on BBS period.

4.8 Experiment Number Eight: BBS Driver with Random Parameter Selection within Bounds

Experiment number eight was the first experiment in which we used randomly selected BBS parameters and LFSR degree rather than only modifying one parameter at a time. For the BBS parameters, we randomly generated combinations of p and q that gave a BBS period

of at least 300. The BBS seed was randomly selected from the range $[2, M-1]$. Note: a seed of 1 produces an all 1's BBS sequence, and a seed of 0 produces an all 0's BBS sequence. The LFSR degree was randomly selected from the range [21,25]. For each degree, we had one representative primitive polynomial. We generated the random parameters using BBS_Primes.py, in Appendix A.2.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. Comp.(*)	NIST (**)	NIST (%)
Experiment 8	CBR1A	25	33554431	0.49995	23	179	4117	3017	440	0.41818	108	14763949640	BBS	0.4999	1150	184	98%
	CBR2	21	2097151	0.49997	83	163	13529	1421	1080	0.49259	537	2264923080	BBS	0.49998	5586	182	97%
	CBR3	21	2097151	0.49995	179	83	14857	1888	440	0.55909	220	922746440	BBS	0.50006	2583	182	97%
	CBR4	22	4194303	0.49998	167	163	27221	2237	2214	0.48961	2212	3095395614	BBS	0.50004	23848	185	98%
	CBR5	23	8388607	0.49995	59	59	3481	1176	812	0.49507	806	6811548884	BBS	0.49998	9246	185	98%
	CBR6	23	8388607	0.49995	107	163	17441	291	1404	0.49217	1404	11777604228	BBS	0.49988	15893	185	98%
	CBR7	23	8388607	0.49994	139	83	11537	2299	440	0.50909	218	3690987080	BBS	0.49994	2576	185	98%
	CBR8	23	8388607	0.49994	163	83	13529	744	1080	0.50926	538	9059695560	BBS	0.49978	6325	185	98%
	CBR9	22	4194303	0.49997	107	107	11449	1720	2756	0.49891	2756	11559499068	BBS	0.49994	30250	184	98%
	CBR10	23	8388607	0.49997	131	227	29737	2128	336	0.53571	84	2818571952	BBS	0.5	1035	184	98%

Figure 4.11. Experiment 8. This experiment included combiners of degrees 21-25 and with randomly selected BBS parameters that gave a BBS period of over 300. These combiners had consistently good results on the NIST test with 10/10 scoring 182/188 or higher. All combiners failed the Approximate Entropy and Serial subtests. ** The number of NIST subtests passed is out of a total of 188 subtests.

The outcome of experiment number eight shows that the lower bounds on LFSR degree (>=21) and BBS period (>=300) show great promise in delivering consistently good results on the NIST test. 10 out of 10 test scored 182/188 or higher.

All combiners failed the Approximate Entropy and two Serial subtests. Both of these subtests evaluate the frequency of the permutations of m-bit runs to determine if there are signs of non-random behavior in the sequence. The minimum polynomial and BBS period needed to be raised to get consistently better results.

4.9 Experiment Number Nine: BBS Driver with Degree 26-29 Polynomials and BBS Period ≥ 3000

In this experiment, our objective was for all combiners to achieve perfect scores on the NIST test (188/188 subtests). We used degree 26-29 polynomials and combinations of p

and q to yield BBS periods of over 3,000. For each of the degrees 26-28, we repeated three combinations of p and q, with random BBS seeds.

Combiner		LFSR Properties			BBS Properties							Combiner Properties				Results	
		Deg.	Period	Ham. Wt.	p	q	M	seed	Period	Ham. Wt.	Lin. Comp.	Period	Driver	Ham. Wt.	Lin. (*) Comp.	NIST (**)	NIST (%)
Experiment 9	CB26_1	26	67108863	0.49997	59	5659	333881	3017	3080	0.5039	1536	206695298040	BBS	0.50006	20176	188	100%
	CB26_2	26	67108863	0.49996	2851	2879	8208029	1421	129420	0.50028	129419	2895076349820	BBS	0.49985	500000	188	100%
	CB26_3	26	67108863	0.49997	7499	7703	57764797	1888	311850	0.50065	311847	6975966308850	BBS	0.50017	500000	188	100%
	CB27_1	27	134217727	0.49983	59	5659	333881	2237	3080	0.4948	1539	59055799880	BBS	0.49985	20574	188	100%
	CB27_2	27	134217727	0.49984	2851	2879	8208029	1176	129420	0.50055	129416	17370458228340	BBS	0.49994	500001	188	100%
	CB27_3	27	134217727	0.49984	7499	7703	57764797	291	311850	0.50061	311845	5979399737850	BBS	0.4999	499999	188	100%
	CB28_1	28	268435455	0.49978	59	5659	333881	2299	3080	0.4948	1539	165356240280	BBS	0.49978	21336	186	99%
	CB28_2	28	268435455	0.49977	2851	2879	8208029	744	129420	0.50028	129419	2316061105740	BBS	0.49994	500000	186	99%
	CB28_3	28	268435455	0.49977	7499	7703	57764797	1720	311850	0.50087	311850	5580773109450	BBS	0.49977	499999	186	99%
	CB29_1	29	536870911	0.50012	2851	2879	8208029	2128	129420	0.49694	14379	69481833301620	BBS	0.50006	207234	186	99%

Figure 4.12. Experiment 9. This experiment included combiners of degrees 26-29 and with BBS parameters that gave a BBS period of over 3,000. These combiners had consistently good results on the NIST test with 10/10 scoring 186/188 or higher, and six achieving a perfect 188/188. The four combiners that scored 186/188 all failed both Serial subtests. ** The number of NIST subtests passed is out of a total of 188 subtests.

While all combiners scored 186 or higher, the 26- and 27-degree combiners scored a perfect 188/188. $CB27_1$ had the lowest combiner period (59,055,799,880) and lowest linear complexity (20,574) of the perfect scoring combiners. The least balanced perfect combiner was $CB26_3$ with an average Hamming weight of 0.50017. These combiner properties give us an idea of the minimum requirements for a perfect combiner. By "perfect combiner", we mean a combiner that passes 188/188 NIST subtests.

CHAPTER 5:
Conclusion

5.1 Summary of Results

A combiner using an LFSR base stream and BBS driver is an effective keystream generator for cryptographic applications if certain constraints are followed. The polynomial for the LFSR should be a primitive polynomial of degree 26 or higher. This ensures balancedness of the keystream, as well as a sufficiently long combiner period. The p and q randomly generated for the BBS sequence must meet the criteria that the resulting BBS period is greater than or equal to 3,000. This increases the probability that the keystream has a long enough period and high enough linear complexity to meet the NIST test requirements. This requirement for a minimum BBS period places a lower bound on acceptable choices for p and q, eliminating only a few possibilities. An upper bound on p and q are only necessary for speed of computation, not cryptographic security. This research does not cover the problem of finding an upper bound for p and q based on desired computation speeds. There are no additional constraints recommended for the choice of seed for BBS other than those normally required for the BBS generator.

The LFSR and BBS inputs (LFSR seed, BBS seed, p, and q) must be randomly generated within the recommended parameters, kept secret, and sent securely from the sender to receiver using a secure Public Key Exchange protocol.

5.2 Further Research Ideas

Early in our research we rejected the idea of a BBS base sequence and an LFSR driver, because the small period BBS sequence we used for experiment number two was unbalanced, resulting in an unbalanced combiner sequence. Unbalanced sequences perform poorly on NIST in general. Near the end of our research we realized that BBS sequences with long periods tend to be closer to balanced. For example, a BBS sequence from experiment number nine, combiner $CB26_2$ with $p = 2851$, $q = 2879$, and a seed of 1421, had an average Hamming weight of 0.50028. This would have made a sufficiently balanced base

sequence to use in a combiner with an LFSR driver. The trouble with BBS base sequences is that they are not consistently balanced even if p and q are large. For example, even with the same p and q as $CB26_2$, combiner $CB28_2$ had a BBS sequence with an average Hamming weight of 0.4948 due to a different BBS seed of 744. This is not close enough to balanced to pass the NIST Frequency subtest. For further research, it would be interesting to see if there was a way to get consistently good (or good enough) results from a combiner with a BBS base sequence.

APPENDIX: Python Code

A.1 Main Combiner, with BBS Driver

Fast Berlekamp Massey Python code by Jason Sachs [14].

List of primitive polynomials by Arash Partow [15].

```
# Andrew Cammack
# Created APR 2020
#
'''Implementation of a shrinking generator type combiner w/
a Fibonacci LFSR base stream and BBS driver.
Generates a combiner sequence .txt file and
a stats.txt file.

Requires a folder called COMBINER in the same
directory as the .py file

An associated file, BBS_primes.py, can be used
to generate random combiner parameters for this
program.
'''

#####IMPORTS#####
import math # for the ceil function
from math import gcd # to check that the BBS Seed is coprime to M
from sympy import * # for the reduced_totient (carmichael) function
import winsound # gives a chime when the program is complete
import sys

#####CONSTANTS#####
file_name = 'COMBINER/CB4_15.txt' # file goes to COMBINER folder
stats_file_name = 'COMBINER/stats.txt' # stats for combiner sequence
```

```
n = 10**7 # desired length of combiner sequence; a minimum of
# 10,000,000 bits is recommended for 10x 1,000,000 bit NIST tests
n_linComp_BBS = 10**6 # desired number of bits to consider for
# estimating linear complexity of BBS; default is 1,000,000 bits
n_linComp = 10**6 # desired number of bits to consider for estimating
# linear complexity of the combiner; default is 1,000,000 bits

poly=[4,1] # characteristic polynomial ex: x^4 + x + 1 = [4,1];
# the brackets should include the powers of x except for x^0
# a list of primitive polynomials can be found at
# https://www.partow.net/programming/polynomials/index.html
p = 5303 # BBS parameters p & q are prime numbers equivalent to 3(mod 4)
q = 5623
seed_BBS = 7523 # the BBS seed must be an integer between [2,M-1]
# and coprime to M = p*q; the program checks that these are co-prime

#####GLOBAL FUNCTIONS#####

#
# below three functions are used from the libgf2 library for fast
# berleykamp-massey calculation
# https://bitbucket.org/jason_s/libgf2/src/default/
#
# Copyright 2013-2017 Jason M Sachs
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
```

```
# limitations under the License.
#

def parity(bitsx):
    '''
    parity of the bits contained in bitsx
    Execution time is logarithmic in bit length.
    This is the preferred general-purpose parity calculation,
    unless there are special circumstances.
    '''

    if bitsx < 0:
        raise ValueError("Parity defined only for nonnegative integers")
    k = 1
    while True:
        y = bitsx >> k
        if y == 0:
            break
        bitsx ^= y
        k <<= 1
    return bitsx & 1

def reverse_bits(x, n=0):
    '''helper function for bm'''

    y = 0
    while x > 0 or n > 0:
        n -= 1
        y <<= 1
        y |= x & 1
        x >>= 1
    return y


def bm(bits, N):
```

```
'''
compute minimal polynomial of LFSR that produces the specified bit
sequence

   bits:    sequence of bits to match
   N:       stop after N bits
   verbose: True to print something

https://en.wikipedia.org/wiki/Berlekamp%E2%80%93Massey_algorithm
'''

b = 1
c = 1
L = 0
m = -1
n = -1
history = 0

for bit in bits:
    n += 1
    if N is not None and n >= N:
        break
    history = (history << 1) | int(bit)
    # history = the first n bits, oldest = most significant
    mask = (1 << (L+1)) - 1
    d = parity(mask & history & c)
    if d == 1:
        t = c
        c ^= b << (n-m)
        if 2*L <= n:
            L = n+1 - L
            m = n
            b = t
    yield L
# c is now in reversed order
```

```
    return (reverse_bits(c,L+1), L)

##### MAIN #####
print("Running script...")
# Generate stream_BBS
M = p*q
# Check that the seed_BBS is coprime to M
if math.gcd(M,seed_BBS) != 1:
    print("Error: seed_BBS must be co-prime to M")
    sys.exit(0)
stream_BBS = str() # initializing the BBS stream
i = 0
n_BBS=0
temp = seed_BBS
while i < n:
    temp = (temp**2) % M
    bit = temp % 2
    stream_BBS += str(bit)
    if bit == 1: i += 1
    n_BBS += 1

#print("stream_BBS is:", stream_BBS)
print("count of 1's in BBS is:",i) # with a BBS driver, to generate a
# k-bit combiner sequence, the number of 1's in BBS must be k-bits
print("length of stream_BBS is:",n_BBS)

# Generate stream_LFSR
deg = poly[0]
period_LFSR = 2**deg - 1
taps = [i % deg for i in poly]
print("taps at", taps)
# Setting up the seed
a=[]
i = 0
while i < deg:
```

```
    a.append(0)
    i += 1
a[0]=1
seed_LFSR = a.copy()
stream_LFSR = str()

j = 0
while (j <= 2**deg - 2) and (j <= n_BBS - 1):
    # Running 1 iteration
    temp = a.copy()
    x = 0
    for tap in taps:
        x = (x + a[tap])%2
    a[deg-1] = x

    i = 0
    while i <= deg-2:
        a[i] = temp[i+1]
        i += 1
    stream_LFSR += str(a[0])
    j+=1

#print("stream_LFSR is:", stream_LFSR)

reps = math.ceil(n_BBS / period_LFSR)
print("reps =", reps)
i = 1
temp = stream_LFSR
while i < reps:
    stream_LFSR += temp
    i += 1
stream_LFSR = stream_LFSR[:n_BBS]
print("Final stream_LFSR length is:", len(stream_LFSR))
# length of LFSR base stream must equal length of BBS driver stream
```

```
# Creating the combiner sequence
stream = str()
j = 0
while j < n_BBS:
    if stream_BBS[j] == "1": stream += stream_LFSR[j]
    j += 1
#print("stream is:", stream)
print("stream length is:", len(stream))

# Saving the stream to a .txt file
with open(file_name, 'w') as fd:
    fd.write(stream)

# Getting the stats
# Avg. Hamming weight of LFSR
count = 0
for bit in stream_LFSR:
    if bit == '1':
        count += 1
hamWt_LFSR = round(count/n_BBS,5)

period_BBS = reduced_totient(reduced_totient(M))
# reduced_totient function is also called the Carmichael function
# Linear Complexity of BBS
print("Linear Complexity of BBS:")
seq = list(map(int, stream_BBS))
tmp = None
count = 0
for item in bm(seq, n_linComp_BBS):
    linComp_BBS = item
    count += 1
    if count % 100000 == 0:
        print(count, ': ', linComp_BBS)

# Avg. Hamming weight of BBS
```

```
count = 0
for bit in stream_BBS:
    if bit == '1':
        count += 1
hamWt_BBS = round(count/n_BBS,5)


period = lcm(period_LFSR,period_BBS) # the Combiner Period
driver = "BBS"
# Avg. Hamming weight of Combiner
count = 0
for bit in stream:
    if bit == '1':
        count += 1
hamWt = round(count/n,5)

# Linear Complexity of Combiner
print("Linear Complexity of the Combiner:")
seq = list(map(int, stream))
tmp = None
count = 0
for item in bm(seq, n_linComp):
    linComp = item
    count += 1
    if count % 100000 == 0:
        print(count, ': ', linComp)

args = [poly,deg,seed_LFSR,period_LFSR,hamWt_LFSR,p,q,M,seed_BBS,
        period_BBS,hamWt_BBS,linComp_BBS,period,driver, hamWt,linComp]
header = ("poly;deg;seed_LFSR;period_LFSR;hamWt_LFSR;p;q;M;seed_BBS;"
          + "period_BBS;hamWt_BBS;linComp_BBS;period;driver;hamWt;linComp")
# Printing the Combiner Stats
stats = str()
for arg in args:
    stats += str(arg) + '; '
```

```
print("Stats are:", stats)

# Saving the stats to a .txt file
with open(stats_file_name, 'w') as fd:
    fd.write(header)
    fd.write("\n")
    fd.write(stats)

print()
print('Complete!')

# Play completion sound
duration = 200  # milliseconds
freqs = [500,400,300,1000,200,200,200]   # Hz
for freq in freqs:
    winsound.Beep(freq, duration)
```

A.2 Random Combiner Parameters Generator

List of primitive polynomials by Arash Partow [15].

```
# Andrew Cammack
# Created APR 2020
#
'''Generates random combiner parameters for Fast_BBS_Combiner_v1.py and
saves them to a text file
'''

#####IMPORTS#####
from sympy import * # for the reduced_totient function
from math import gcd
import random
import sys

#### CONSTANTS ####
```

```
file_name = 'random_combiner_parameters.txt'
n = 100 # The first n valid BBS primes p & q; default is 100
min_bbs_period = 3000 # the minimum BBS period to accept; default is 3000
combs = 10 # number of random combiners to generate
deg_min = 26 # minimum primitive polynomial degree; min available is 2
deg_max = 32 # maximum primitive polynomial degree; max available is 32
verbose = False # if True, turns on extra print statements

#### MAIN ####
# Generating the list of BBS Primes p & q
bbs_primes = list()
i = 1 # the index number for the prime numbers
j = 1 # the count of BBS primes
while j <= n:
    x = prime(i)
    if x%4 == 3:
        bbs_primes.append(x)
        j += 1
    i+=1
if verbose:
    print('The first %d BBS primes p, q:' % n)
    print(bbs_primes)
    print()

# Generating the dictionary of p,q combinations
# such that the BBS Period >= min_bbs_period
P = bbs_primes
Q = bbs_primes
dict1 = dict()
count = 0
for p in P:
    for q in Q:
        M = p*q
        BBS_period = reduced_totient(reduced_totient(M))
        if BBS_period >= min_bbs_period:
```

```
            dict1[count] = [p,q,BBS_period]
            count += 1
if verbose:
    print('All combinations of the first %d BBS primes'
          ' that generate a BBS period >= %d:' % (n, min_bbs_period))
    print(dict1)
    print()

# Generating the dictionary of primitive polynomials (one per degree)
# Source: https://www.partow.net/programming/polynomials/index.html
poly_dict = {2:'[2,1]',3:'[3,1]',4:'[4,1]',5:'[5,2]',6:'[6,1]',7:'[7,1]',
             8:'[8,4,3,2]',9:'[9,4]',10:'[10,3]',11:'[11,2]',
             12:'[12,6,4,1]',13:'[13,4,3,1]',14:'[14,8,6,1]',15:'[15,1]',
             16:'[16,12,3,1]',17:'[17,3]',18:'[18,7]',19:'[19,5,2,1]',
             20:'[20,3]',21:'[21,2]',22:'[22,1]',23:'[23,5]',
             24:'[24,7,2,1]',25:'[25,3]',26:'[26,6,2,1]',27:'[27,5,2,1]',
             28:'[28,3]',29:'[29,2]',30:'[30,23,2,1]',31:'[31,3]',
             32:'[32,22,2,1]'}

# Generating a string of combiners with a randomly selected
# polynomial degree, p, q, and BBS seed
print('%d sets of random combiner parameters with primitive'
      ' polynomial degree from %d to %d and BBS period >= %d:\n'
      % (combs,deg_min,deg_max,min_bbs_period))
header = (str(combs) + 'sets of random combiner parameters with'
          + 'primitive' + ' polynomial degree from' + str(deg_min)
          + 'to' + str(deg_max) + 'and BBS Period >='
          + str(min_bbs_period) + ':\n')
print('Copy and paste a line into the CONSTANTS section'
      ' of Fast_BBS_Combiner_v1.py\n')
output = str()
for comb in range(combs):
    deg = random.randint(deg_min,deg_max)
    bbs = dict1[random.randint(0,count-1)]
    seed_BBS = 0
```

```
    p = bbs[0]
    q = bbs[1]
    M = p*q
    while(gcd(M,seed_BBS) != 1):
        seed_BBS = random.randint(2,M-1)
    poly = poly_dict[deg]
    print('poly, p, q, seed_BBS = (%s, %d, %d, %d)' % (poly,p,q,seed_BBS))
    temp_str = ('poly, p, q, seed_BBS = (' + poly + ', ' + str(p) + ', '
                + str(q) + ', ' + str(seed_BBS) + ')\n')
    output += temp_str

# Saving the stats to a .txt file
with open(file_name, 'w') as fd:
    fd.write(output)
```

A.3 Reversible Combiner

Adapted from LFSR.py by Michael Troncoso [11].

List of primitive polynomials by Arash Partow [15].

Fast Berlekamp Massey Python code by Jason Sachs [14].

```
# Andrew Cammack
#
# Adapted from LFSR.py by Michael Troncoso.
# Primitive polynomials taken from Arash Partow at
# https://www.partow.net/programming/polynomials/index.html.
#
# Fibonacci LFSR implementation for dynamic creation and execution of LFSR
# also includes local instances of Berlekamp-Massey Algorithm
# Created: 13MAR2020

#####IMPORTS#####

#import numpy as np
#import sys
```

```
#import matplotlib.pyplot as plt
import math
from math import gcd # for the lcm function
from sympy import * # for the reduced_totient (carmichael) function

#####CONSTANTS#####

COMBINER_FILE_NAME = 'COMBINER/COMBINER' # prefix for output files
STATS_FILE_NAME = 'COMBINER/STATS' # prefix for output files
#LCG_FILE_NAME = 'LCG/LCG' # prefix for output files
#SEED = 1
verbose = True
N = 10**5 # The combiner sequence length; If N > 100000, STATS should be
         # set to False to reduce computation time
STATS = True  # If STATS is true, stats are on.  If STATS is False,
               #only the combiner sequence is produced.

#####CLASSES#####

class COMBINER:

    def __init__(self, poly, seed_LFSR, n, p, q, seed_BBS, d):
        '''init a Combiner with specified characteristic poly and
        start state
        '''

        self.__poly = poly # exponents in charateristic polynomial
        self.__bits = self.__poly[0] # number of bits in register
        self.__taps = [self.__bits - i for i in self.__poly] # placement
                                                              # of taps
        self.__seed_LFSR = seed_LFSR # init state of LFSR must be non-zero
        self.__n = n  # desired length of output from the combiner
        self.__p, self.__q, self.__seed_BBS = p, q, seed_BBS # BBS inputs
        self.__M = self.__p * self.__q
        self.__d = d # driver for the combiner.
```

```
                # d=0 for LFSR driver, d=1 for BBS driver
        self.__driver = str()
        if self.__d == 0:
            self.__driver = 'LFSR'
        elif self.__d == 1:
            self.__driver = 'BBS'
        else:
            print("Error: d must be 0 or 1.   d=0 for LFSR driver."
                  "  d=1 for BBS driver")
        # to get just BBS sequence, set LFSR as driver, and poly = [1]
        # to get just LFSR sequence, set BBS as driver, and seed_BBS = 1.
        self.__state = self.__seed_LFSR # current state of LFSR
        self.__mask = 2 ** (self.__bits) - 1 # bit mask
        self.__stream_LFSR = str() # LFSR bit stream
        self.__stream_BBS = str() # BBS bit stream
        self.__stream = str() # Combiner bit stream
        if STATS == True:
            self.__period_BBS = reduced_totient(reduced_totient(self.__M))
            # aka carmichael(carmichael(M))
            self.__period = lcm(self.__mask,self.__period_BBS)
        else:
            self.__period_BBS = -1
            self.__period = -1
        self.__Hamming_weight = 0  # Hamming weight of combiner sequence
        self.__bm = None
        self.__stats = str()
        self.__seed_BBS_temp = self.__seed_BBS

    def next(self):
        '''run one iteration of combiner, update state info'''

        bit = self.__state

        for j in self.__taps[1:]:
            bit ^= self.__state >> j # fibonacci lfsr...
```

```
                                    # compute bit from xor gates
        bit &= self.__mask
        self.__state = (self.__state >> 1) | (bit << (self.__bits - 1))
            # update lfsr
        self.__state &= self.__mask
        output_LFSR = self.__state & 1 # grab low order bit
        self.__stream_LFSR += str(output_LFSR) # update bit stream

        # Generating one iteration (one bit) of the BBS sequence

        self.__seed_BBS_temp = (self.__seed_BBS_temp**2) %self.__M
        output_BBS = self.__seed_BBS_temp%2
        self.__stream_BBS += str(output_BBS) # update bit stream

        # Generating one iteration of the combiner sequence
        if self.__d == 0: # d=0 means LFSR is the driver
            if output_LFSR == 1:
                self.__stream += str(output_BBS)
                output = output_BBS
            else:
                output = None
        else: # BBS is the driver, d=1
            if output_BBS == 1:
                self.__stream += str(output_LFSR)
                output = output_LFSR

            else:
                output = None

        return output

    def run(self):
        '''run Combiner until it produces a sequence of length n;
        outputs bit stream in list
        '''
```

```
if verbose:
    print("poly is",self.__poly)
    print("bits =", self.__bits)
    print("taps are",self.__taps)
    print("seed_LFSR = ", self.__seed_LFSR)
    print("sequence length, n =", self.__n)
    print("p =", self.__p)
    print("q =", self.__q)
    print("M =", self.__M)
    print("seed_BBS =", self.__seed_BBS)
    print("d =", self.__d)
    print("state is", self.__state)
    print("LFSR period is", self.__mask)
    print("BBS period is", self.__period_BBS)
    print("Combiner period is", self.__period)

output = list()
i = 1
while i <= self.__n:
    bit = self.next()
    if bit == 1:
        self.__Hamming_weight += 1
    if bit == None:
        i -= 1
    output.append(bit)
    i += 1
print("Avg. Hamming Weight is", self.__Hamming_weight/self.__n)


### Running bm on the sequence
if STATS == True:
    count = 0
    for item in bm(self.__stream, self.__n):
        self.__bm = item
        count += 1
```

```
            if count % 10000 == 0:
                print(count, ': ', self.__bm)
        if verbose: print("Linear Complexity =", self.__bm)
    else:
        self.__bm = -1
    self.__stats = (str(self.__poly) + ';' + str(self.__bits) + ';'
                    + str(self.__seed_LFSR) + ';' + str(self.__mask)
                    + ';' + str(self.__p) + ';' + str(self.__q)
                    +  ';' + str(self.__M) + ';'
                    + str(self.__seed_BBS) + ';'
                    + str(self.__period_BBS) + ';'
                    + str(self.__period) + ';' + str(self.__driver)
                    + ';' + str(self.__Hamming_weight/self.__n)
                    + ';' + str(self.__bm))
    '''
    data = [bit for bits in self.__]
    plt.plot(data)
    plt.xlabel('Bits')
    plt.ylabel('LC')
    plt.show()
    '''
    ### End of bm section
    return output   # output consists only of the combiner sequence

def getLFSRStream(self):
    '''return currently saved LFSR stream'''

    return self.__stream_LFSR

def getBBSStream(self):
    '''return currently saved BBS stream'''

    return self.__stream_BBS

def getStream(self):
```

```
        '''return currently saved Combiner stream'''

        return self.__stream

    def getStats(self):
        ''' returns the Combiner stats in a csv format'''

        return self.__stats

    def getPoly(self):
        '''return charateristic polynomial'''

        return self.__poly

#####GLOBAL FUNCTIONS#####

def runCombiner(*args):
    '''runs the Combiner for a given length and writes their strings to
    named files
    '''

    count = 0
    for arg in args:

        if type(arg) != type(COMBINER([1], 1,1,1,1,1,1)):
            print('all arguments must be a valid Combiner')
            print("Correct Argument format: COMBINER([poly],seed_LFSR,n,"
                  "p,q,seed_BBS,d)")
            return

        print('Running Combiner:', count)
        file_name = COMBINER_FILE_NAME + '_' + str(count) + '.txt'
        file_name_stats = STATS_FILE_NAME + '.txt'
        #file_name_stats = COMBINER_FILE_NAME + '_' + str(count) + 'STATS'
        arg.run()
```

```
        with open(file_name, 'w') as fd:
            fd.write(arg.getStream())

        with open(file_name_stats, 'a') as fd:
            fd.write(arg.getStats())
            fd.write('\n')

        count += 1
        print('Complete!')
        print()
    print('All Complete!')

def createCOMBINERDict(*args):
    '''create dict of COMBINERS'''

    count = 0
    output_dict = dict()
    for arg in args:
        if type(arg) != type(COMBINER([1], 1,1,1,1,1,1)):
            print('all arguments must be a COMBINER')
            return

        output_dict[count] = arg
        count += 1

    return output_dict

def lcm(a, b):
    return abs(a*b) // math.gcd(a, b)

#
# below three functions are used from the libgf2 library for fast
# berleykamp-massey calculation
# https://bitbucket.org/jason_s/libgf2/src/default/
```

```
#
# Copyright 2013-2017 Jason M Sachs
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

def parity(bitsx):
    '''
    parity of the bits contained in bitsx
    Execution time is logarithmic in bit length.
    This is the preferred general-purpose parity calculation,
    unless there are special circumstances.
    '''

    if bitsx < 0:
        raise ValueError("Parity defined only for nonnegative integers")
    k = 1
    while True:
        y = bitsx >> k
        if y == 0:
            break
        bitsx ^= y
        k <<= 1
    return bitsx & 1
```

```
def reverse_bits(x, n=0):
    '''helper function for bm'''

    y = 0
    while x > 0 or n > 0:
        n -= 1
        y <<= 1
        y |= x & 1
        x >>= 1
    return y


def bm(bits, N):
    '''
    compute minimal polynomial of LFSR that produces the specified bit
    sequence

       bits:    sequence of bits to match
       N:       stop after N bits
       verbose: True to print something

    https://en.wikipedia.org/wiki/Berlekamp%E2%80%93Massey_algorithm
    '''

    b = 1
    c = 1
    L = 0
    m = -1
    n = -1
    history = 0

    for bit in bits:
        n += 1
        if N is not None and n >= N:
            break
```

```
        history = (history << 1) | int(bit)
        # history = the first n bits, oldest = most significant
        mask = (1 << (L+1)) - 1
        d = parity(mask & history & c)
        if d == 1:
            t = c
            c ^= b << (n-m)
            if 2*L <= n:
                L = n+1 - L
                m = n
                b = t
        yield L
    # c is now in reversed order
    return (reverse_bits(c,L+1), L)

#####MAIN#####
'''
n = number of bits to output, p & q are primes equivalent to 3(mod 4);

For p & q pick large PRIMES that end in 03,07,11,19,23,31,43,47,59,67,
71,79, or 83 because these will always come out to 3(mod4).

seed_BBS must be a random integer greater than or equal to 2.  If the
seed is 1, the BBS sequence is 1111...
'''

if __name__ == '__main__':

    # code to init 5 Combiners and produce files containing their
    # respective sequences

    C0 = COMBINER(poly=[4,3],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=1,d=1)
    # p(x)=x^4 + x^3 + 1; BBS driver is 1111...
    #C1 = COMBINER([4,3],1,100000,59,71,479,1) # p(x)= x^4 + x^3 + 1;
        # BBS driver is turned on, with small M and seed
```

```
#C2 = COMBINER([4,3],1,100000,59,71,479,0) # p(x)= x^4 + x^3 + 1;
    # LFSR driver is turned on.  BBS with small M and seed.
#C3 = COMBINER([1],1,100000,59,71,1,0) # p(x)= x + 1; LFSR driver
    # is 11111....  BBS is 1111...  Sequence 11111...
runCombiner(C0)

# BBS Drivers - seeing how lin. complexity increases with deg. of
# primitive polyn.
'''
CB3 = COMBINER(poly=[3,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=1)
CB4 = COMBINER(poly=[4,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=1)
CB5 = COMBINER(poly=[5,2],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=1)
CB6 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=1)
CB7 = COMBINER(poly=[7,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=1)
CB8 = COMBINER(poly=[8,4,3,2],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,
               d=1)
CB9 = COMBINER(poly=[9,4],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=1)
CB10 = COMBINER(poly=[10,3],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,
                d=1)
CB11 = COMBINER(poly=[11,2],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,
                d=1)
CB12 = COMBINER(poly=[12,6,4,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,
                d=1)
CB13 = COMBINER(poly=[13,4,3,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=1)
CB14 = COMBINER(poly=[14,8,6,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=1)
CB15 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=1)
CB18 = COMBINER(poly=[18,7],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=1)
CB21 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=1)
#runCombiner(CB8)
```

```
runCombiner(CB8,CB9,CB10,CB11,CB12,CB13,CB14,CB15,CB18,CB21)
'''

'''
# LFSR Drivers - seeing how lin. complexity increases with deg of
# primitive polyn.
#CL3 = COMBINER(poly=[3,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL4 = COMBINER(poly=[4,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL5 = COMBINER(poly=[5,2],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL6 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL7 = COMBINER(poly=[7,1],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL8 = COMBINER(poly=[8,4,3,2],seed_LFSR=1,n=N,p=59,q=71,
               seed_BBS=479,d=0)
CL9 = COMBINER(poly=[9,4],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL10 = COMBINER(poly=[10,3],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL11 = COMBINER(poly=[11,2],seed_LFSR=1,n=N,p=59,q=71,seed_BBS=479,d=0)
CL12 = COMBINER(poly=[12,6,4,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=0)
CL13 = COMBINER(poly=[13,4,3,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=0)
CL14 = COMBINER(poly=[14,8,6,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=0)
CL15 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=0)
CL18 = COMBINER(poly=[18,7],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=0)
CL21 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                seed_BBS=479,d=0)
#runCombiner(CL3)
runCombiner(CL4,CL5,CL6,CL7,CL8,CL9,CL10,CL11,CL12,CL13,CL14,
            CL15,CL18,CL21)
'''
'''
# Seeing how linear complexity increases with M
CB6_1 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=23,q=47,
```

```
                 seed_BBS=479,d=1)
CB6_2 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=59,q=71,
                 seed_BBS=479,d=1)
CB6_3 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=167,q=211,
                 seed_BBS=479,d=1)
CB6_4 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=1991,q=1423,
                 seed_BBS=479,d=1)
CB6_5 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=3259,q=3343,
                 seed_BBS=479,d=1)
CB6_6 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=5303,q=5623,
                 seed_BBS=479,d=1)
CB6_7 = COMBINER(poly=[6,1],seed_LFSR=1,n=N,p=7499,q=7687,
                 seed_BBS=479,d=1)
#runCombiner(CB6_2)
runCombiner(CB6_1, CB6_2, CB6_3, CB6_4, CB6_5, CB6_6, CB6_7)
'''

'''
# Test 4: Stability Testing: BBS Driver, same LFSR, vary seed_BBS
CB15_1 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=2201,d=1)
CB15_2 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=9325,d=1)
CB15_3 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=1033,d=1)
CB15_4 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=4179,d=1)
CB15_5 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=1931,d=1)
CB15_6 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=8117,d=1)
CB15_7 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=7364,d=1)
CB15_8 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=7737,d=1)
```

```
CB15_9 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=6219,d=1)
CB15_10 = COMBINER(poly=[15,1],seed_LFSR=1,n=N,p=167,q=211,
                   seed_BBS=3439,d=1)
#runCombiner(CB15_1)
runCombiner(CB15_2, CB15_3, CB15_4, CB15_5, CB15_6, CB15_7, CB15_8,
            CB15_9, CB15_10)
'''

'''
# Test 5: Seeing how linear complexity increases with M
CB21_1 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=23,q=47,
                  seed_BBS=479,d=1)
CB21_2 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=479,d=1)
CB21_3 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=167,q=211,
                  seed_BBS=479,d=1)
CB21_4 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=1991,q=1423,
                  seed_BBS=479,d=1)
CB21_5 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=3259,q=3343,
                  seed_BBS=479,d=1)
CB21_6 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=5303,q=5623,
                  seed_BBS=479,d=1)
#runCombiner(CB6_2)
runCombiner(CB21_1, CB21_2, CB21_3, CB21_4, CB21_5, CB21_6)
'''

'''
# Test 6: Stability Testing: BBS Driver, same degree-21 LFSR,
# vary seed_BBS
CB21_7 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=23,d=1)
CB21_8 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=131,d=1)
CB21_9 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
```

```
                  seed_BBS=1619,d=1)
CB21_10 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                   seed_BBS=3323,d=1)
CB21_11 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                   seed_BBS=4159,d=1)
CB21_12 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=59,q=71,
                   seed_BBS=659,d=1)
runCombiner(CB21_7, CB21_8, CB21_9, CB21_10, CB21_11, CB21_12)
'''

'''
Test 7: Performance of large degree characteristic polynomials.
CB32_1 = COMBINER(poly=[32,22,2,1],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=23,d=1)
CB32_2 = COMBINER(poly=[32,22,2,1],seed_LFSR=1,n=N,p=5303,q=5623,
                  seed_BBS=7523,d=1)
CB18_1 = COMBINER(poly=[18,7],seed_LFSR=1,n=N,p=23,q=47,
                  seed_BBS=11,d=1)
CB18_2 = COMBINER(poly=[18,7],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=11,d=1)
CB19_1 = COMBINER(poly=[19,5,2],seed_LFSR=1,n=N,p=23,q=47,
                  seed_BBS=11,d=1)
CB19_2 = COMBINER(poly=[19,5,2],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=11,d=1)
CB20_1 = COMBINER(poly=[20,3],seed_LFSR=1,n=N,p=23,q=47,
                  seed_BBS=11,d=1)
CB20_2 = COMBINER(poly=[20,3],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=11,d=1)
CB22_1 = COMBINER(poly=[22,1],seed_LFSR=1,n=N,p=23,q=47,
                  seed_BBS=11,d=1)
CB22_2 = COMBINER(poly=[22,1],seed_LFSR=1,n=N,p=59,q=71,
                  seed_BBS=11,d=1)
CB23_1 = COMBINER(poly=[23,5],seed_LFSR=1,n=N,p=23,q=47,
                  seed_BBS=11,d=1)
CB23_2 = COMBINER(poly=[23,5],seed_LFSR=1,n=N,p=59,q=71,
```

```
                  seed_BBS=11,d=1)
runCombiner(CB18_1, CB18_2, CB19_1, CB19_2, CB20_1, CB20_2, CB22_1,
            CB22_2, CB23_1, CB23_2)
'''
'''
# Checking the balancedness of the deg-19 polyn LFSR ONLY (BBS set to
# all 1's)
CB19_3 = COMBINER(poly=[19,5,2],seed_LFSR=1,n=N,p=23,q=47,seed_BBS=1,
                  d=1)
runCombiner(CB19_3)
'''

'''
# Test 8: Checking random Combiners with the following constraints:
# 1. BBS Driver
# 2. Primitive poly of deg 21 to 25
# 3. p & q chosen randomly from a set of the first 50 valid choices
# for p & q s.t. the BBS period is >= 300.
# 4. seed_BBS was chosen randomly between 2 to 3000.
# Note: Inputs were generated using BBS Primes.ipynb
# Note: The R in the combiner name identifies these as randomly
# generated (within constraints)
CBR1 = COMBINER(poly=[25,3],seed_LFSR=1,n=N,p=23,q=179,
                seed_BBS=1955,d=1)
CBR2 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=83,q=163,
                seed_BBS=1421,d=1)
CBR3 = COMBINER(poly=[21,2],seed_LFSR=1,n=N,p=179,q=83,
                seed_BBS=1888,d=1)
CBR4 = COMBINER(poly=[22,1],seed_LFSR=1,n=N,p=167,q=163,
                seed_BBS=2237,d=1)
CBR5 = COMBINER(poly=[23,5],seed_LFSR=1,n=N,p=59,q=59,
                seed_BBS=1176,d=1)
CBR6 = COMBINER(poly=[23,5],seed_LFSR=1,n=N,p=107,q=163,
                seed_BBS=291,d=1)
CBR7 = COMBINER(poly=[23,5],seed_LFSR=1,n=N,p=139,q=83,
```

```
                seed_BBS=2299,d=1)
CBR8 = COMBINER(poly=[23,5],seed_LFSR=1,n=N,p=163,q=83,
                seed_BBS=744,d=1)
CBR9 = COMBINER(poly=[22,1],seed_LFSR=1,n=N,p=107,q=107,
                seed_BBS=1720,d=1)
CBR10 = COMBINER(poly=[23,5],seed_LFSR=1,n=N,p=131,q=227,
                 seed_BBS=2128,d=1)
runCombiner(CBR1, CBR2, CBR3, CBR4, CBR5, CBR6, CBR7, CBR8,
            CBR9, CBR10)
'''

# Seeds were generated using the Python Random module:
'''
# generate random integer values
from random import seed
from random import randint
# seed random number generator
seed(1)
# generate some integers
for _ in range(10):
    value = randint(0, 10000)
    print(value)
'''
'''
CB6_8 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                 seed_BBS=2201,d=1)
CB6_9 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                 seed_BBS=9325,d=1)
CB6_10 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                  seed_BBS=1033,d=1)
CB6_11 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                  seed_BBS=4179,d=1)
CB6_12 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                  seed_BBS=1931,d=1)
CB6_13 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
```

```
                  seed_BBS=8117,d=1)
CB6_14 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                  seed_BBS=7364,d=1)
CB6_15 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                  seed_BBS=7737,d=1)
CB6_16 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                  seed_BBS=6219,d=1)
CB6_17 = COMBINER(poly=[6,1],seed_LFSR=1,n=200000,p=167,q=211,
                  seed_BBS=3439,d=1)
runCombiner(CB6_8,CB6_9,CB6_10,CB6_11,CB6_12,CB6_13,CB6_13,
            CB6_14,CB6_15,CB6_16,CB6_17)
'''

'''
# use this section of code to run bm
with open('data/data.period', 'r') as fd:
    seq = fd.read()
seq = list(map(int, seq))
with open('data/data.bm_period', 'w') as fd:
    tmp = None
    count = 0
    for item in bm(seq, 100000):
        tmp = item
        count += 1
        fd.write(str(tmp) + ',')
        if count % 10000 == 0:
            print(count, ': ', tmp)
    print(tmp)


with open('data/data.bm_period', 'r') as fd:
    data = fd.read()
data = [int(bit) for bit in data.split(',')[:-1]]
plt.plot(data)
plt.xlabel('Bits')
```

```
    plt.ylabel('LC')
    plt.show()
    '''
```

A.4 BBS Period vs. M Plot

This code was adapted from existing code by Wikimedia contributor 'Proz' with minor changes [13].

```
# BBS Period vs. M
'''
This code was adapted from Carmichael Function code written by user Proz.
https://commons.wikimedia.org/wiki/File:CarmichaelLambda.svg
Posted: 05FEB2018
Accessed: 16APR2020
'''
from sympy import *
import matplotlib.pyplot as plt

nmin, nmax = 1, 10000
l = range(nmin,nmax)
#plt.plot(l, [totient(n) for n in l], 'g.', alpha=0.2, markersize=4)
plt.plot(l, [reduced_totient(reduced_totient(n)) for n in l], 'b.',
         alpha=1, markersize=2)
plt.axis([nmin,nmax,nmin,nmax])
plt.xlabel('$M$', fontsize='x-large')
ax = plt.gca()
ax.xaxis.set_label_coords(1.03,0.03)
xticks = ax.xaxis.get_major_ticks()
xticks[0].label1.set_visible(False)
yticks = ax.yaxis.get_major_ticks()
yticks[0].label1.set_visible(False)
plt.figtext(0.2,0.8,'$p_{BBS}=\lambda(\lambda(M))$', fontsize='x-large',
            color='b', alpha=1)
#plt.figtext(0.2,0.74,'$\\varphi(n)$', fontsize='x-large', color='g',
```

```
            # alpha=0.6)
plt.savefig('BBS_period.jpg')
```

List of References

[1] C. Paar and J. Pelzl, *Understanding Cryptography*. Heidelberg, DEU: Springer, 2010.

[2] A. Canteaut, “Linear feedback shift register,” in *Encyclopedia of Cryptography and Security*, H. van Tilborg, Ed. Springer, 2005.

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

[4] J. B. Fraleigh, *A First Course in Abstract Algebra*, 7th ed. Noida, IND: Pearson Education India, 2014.

[5] H. Fredricksen and G. Krahn, “Determining the generator of a linear recursive sequence,” Naval Postgraduate School, Monterey, CA, USA, Tech. Rep., 1994.

[6] 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.

[7] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[8] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[9] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory*. Pearson Education India, 2006.

[10] P. Junod, “Cryptographic secure pseudo-random bits generation: The Blum Blum Shub generator,” University of Miami, August 1999.

[11] M. Troncoso, private communication, Aug. 2019.

[12] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” National Institute for Standards and Technology, Gaithersburg, MD, Tech. Rep. Special Publication 800-22, Revision 1a, Apr. 2010.

[13] U. Proz. (2018). Carmichael function. Wikimedia Commons. [Online]. Available: https://commons.wikimedia.org/wiki/File:CarmichaelLambda.svg. Accessed Apr. 16, 2020.

[14] J. M. Sachs. Fast Berlekamp Massey Python Code. Bitbucket libgf2. [Online]. Available: https://bitbucket.org/jason_s/libgf2/src/default/. Accessed Apr. 25, 2020.

[15] A. Partow. Primitive Polynomial List. Personal Website. [Online]. Available: https://www.partow.net/programming/polynomials/index.html. Accessed Apr. 25, 2020.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California