2016-09

Nonquadratic variation of the Blum-Blum-Shub Pseudorandom Number Generator

Knuth, Thomas

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/50570

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

NONQUADRATIC VARIATION OF THE BLUM-BLUM-SHUB PSEUDORANDOM NUMBER GENERATOR

by

Thomas Knuth

September 2016

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*

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE September 2016	3. REPORT TYPE AND DATES COVERED Master's Thesis 09-28-2015 to 09-23-2016

4. TITLE AND SUBTITLE

NONQUADRATIC VARIATION OF THE BLUM-BLUM-SHUB PSEUDORANDOM NUMBER GENERATOR

5. FUNDING NUMBERS

6. AUTHOR(S)

Thomas Knuth

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

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 document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Cryptography is essential for secure online communications. Many different types of ciphers are implemented in modern-day cryptography, but they all have one common factor. All ciphers require a source of randomness, which makes them unpredictable. One such source of this randomness is a random number generator. This thesis focuses on Pseudorandom Number Generators (PRNG), specifically, a PRNG called Blum-Blum-Shub (BBS). In this thesis, we make two modifications to BBS, and test our modified generators for randomness using the National Institute of Standards and Technology (NIST) tests. The original BBS is a quadratic generator that generates bits based on the output of squaring terms in a sequence. The first modification replaces the quadratic generator with a cubic generator. The second modification generates bits faster by using more bits per iteration. Data collected in this thesis suggests that the cubic modification performs just as well as the original generator. In addition, data from this thesis suggests that taking more bits per iteration can speed up this process while retaining randomness. In addition, we propose a new cryptosystem based upon the modification of the BBS PRNG introduced in this thesis.

14. SUBJECT TERMS

Pseudorandom Number Generator, Blum-Blum-Shub

15. NUMBER OF PAGES 75

16. PRICE CODE

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

THIS PAGE INTENTIONALLY LEFT BLANK

NONQUADRATIC VARIATION OF THE BLUM-BLUM-SHUB PSEUDORANDOM NUMBER GENERATOR

Thomas Knuth
Second Lieutenant, United States Army
B.S., Marquette University, 2015

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
September 2016

Approved by: Pantelimon Stanica
Thesis Advisor

Thor Martinsen
Second Reader

Craig Rasmussen
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Cryptography is essential for secure online communications. Many different types of ciphers are implemented in modern-day cryptography, but they all have one common factor. All ciphers require a source of randomness, which makes them unpredictable. One such source of this randomness is a random number generator. This thesis focuses on Pseudorandom Number Generators (PRNG), specifically, a PRNG called Blum-Blum-Shub (BBS). In this thesis, we make two modifications to BBS, and test our modified generators for randomness using the National Institute of Standards and Technology (NIST) tests. The original BBS is a quadratic generator that generates bits based on the output of squaring terms in a sequence. The first modification replaces the quadratic generator with a cubic generator. The second modification generates bits faster by using more bits per iteration. Data collected in this thesis suggests that the cubic modification performs just as well as the original generator. In addition, data from this thesis suggests that taking more bits per iteration can speed up this process while retaining randomness. In addition, we propose a new cryptosystem based upon the modification of the BBS PRNG introduced in this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

BBS	Blum-Blum-Shub
CRT	Chinese Remainder Theorem
NIST	National Institute of Standards and Technology
PRNG	Pseudorandom Number Generator
RNG	Random Number Generator
XOR	Exclusive Or

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Cryptography has become an essential part of our everyday lives. The rise of online communication and e-commerce has created a constant demand for secure correspondence. Many of our cryptosystems rely upon the use of random numbers. For this reason, cryptographers design pseudorandom number generators (PRNG) as a method of generating random numbers quickly. One such generator is called the Blum-Blum-Shub (BBS) PRNG. The BBS PRNG takes two large Blum-prime numbers, $p, q \equiv 3 \pmod 4$, and multiplies them to create $n = p * q$. A sequence of pseudorandom numbers is created by repeatedly squaring a random seed and reducing it modulo n. BBS can create random bits by taking the parity bit of each term in this sequence.

This thesis proposes two modifications to the original BBS PRNG. First, this thesis generates random numbers by repeated cubing, instead of squaring. Second, this thesis attempts to speed up bit generation by taking more than the parity bit per iteration. Both modifications proposed in this thesis are tested by National Institute of Standards and Technology (NIST) tests for randomness and then compared to the original BBS PRNG.

Data from this thesis suggests that the cubic generator performs just as well as the original. In addition, data gathered in this thesis suggests that bit generation can be significantly increased by taking more than the parity bit, while retaining a sufficient degree of statistical randomness.

Additionally this thesis proposes a new public key cryptosystem based upon the cubic generator.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I am very grateful that I was able to study Mathematics at Naval Postgraduate School for my first assignment in the U.S. Army. The Department of Applied Mathematics is filled with extraordinary professors who are leaders in their field and truly care about their students. I cannot thank my advisor, Pante Stanica, enough for his continued patience and guidance during this process. He is a brilliant mathematician and excellent to work with. I would also like to thank CDR Thor Martinsen for his mentorship as my second reader. He took many hours out of his day to make extremely detail-oriented revisions, and this thesis is much better off for it. Finally, I would like to thank Bijesh Shrestha, my colleague who worked on a similar project for his thesis. It was helpful to have a colleague to help work through difficult problems.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Cryptography is a field spread between mathematics and computer science focused on secrecy and security of communications. Cryptographers attempt to fill the demand for secrecy in a very difficult and dynamic environment. An important ingredient in the encryption process is a pseudorandom number generator (PRNG). This thesis will focus on a particular PRNG, the Blum-Blum-Shub (BBS) generator. This thesis will substitute the quadratic generator used in BBS with a similar cubic generator. National Institute for Standards and Technology (NIST) tests for randomness will be used to evaluate the cubic generator. Additionally, this thesis seeks to improve the efficiency of the cubic generator by increasing the number of bits taken per iteration and again testing these sequences for randomness. Previous research has shown that given a modulus of size n, we can take up to $\log(\log(n))$ bits per iteration and guarantee statistical randomness [1]. This thesis will attempt to take the cubic generator beyond that bound. Data gathered in this thesis research suggests that the cubic generator performs just as well as the quadratic generator in NIST testing. It is worth noting, as the data suggests, that even when taking more than $\log(\log(n))$ bits, the sequences still pass NIST tests over 90 percent of the time.

1.1 Motivation

BBS is probabilistically secure, making it an excellent control group for an experiment [2]. Using BBS as a control, we investigate modifications and see how well they perform. This type of approach has been done previously on RSA. Hansen, Larsen, and Olsen [3] of the University of Copenhagen compare decryption times in variants of RSA on mobile phones. These variants include (Chinese Remainder Theorem) CRT-RSA, multi-prime RSA, multi-power RSA, rebalanced RSA, and R-prime RSA. All of these variations deviate from RSA with a purpose of improving the original, or simply having an alternative. This thesis will attempt to do the same with cubic BBS in hopes of improvement.

A great deal of research has been done on the quadratic generator. In particular, research has been done on how quadratic residues behave. This is critical to understanding BBS. However, thus far little research has been carried out on cubic residues. While quadratic residues follow some patterns, the cubic residues are more difficult to fit into a discernible pattern, and this seems to be a source of randomness. Thus, a pseudorandom number generator based purely on cubic residues shows promise.

Although it is considered to be cryptographically secure, BBS is relatively slow compared to other stream ciphers. While substituting the quadratic generator with a cubic generator could slow down bit generation further, we may be able to improve speed by taking more bits per iteration. By testing the bounds of $\log(\log(n))$ bits per iteration, this thesis will propose a faster alternative, which retains randomness. In addition, we propose a new cryptosystem based upon our modification of the generator.

1.2 Background

Before we delve into the inner workings of the BBS PRNG, it is important to understand how pseudorandom number generators are relevant to cryptography. This section briefly highlights cryptographic goals and history to emphasize the application of this study. It then defines a standard for perfect secrecy as an ideal to strive for. Finally, it hones in on the time and resource issues of using random number generators (RNG), which leads to the need for PRNGs.

1.2.1 Cryptographic Goals

Cryptography seeks to establish three goals:

1. *Confidentiality*: The method of encryption must be strong enough that, given an encrypted message, an adversary has no understanding of the underlying meaning.
2. *Integrity*: The original message must reach its destination unaltered, and if it has been altered, the recipient will know. In addition, integrity means that the authenticity of the sender is proven to the recipient.
3. *Availability*: The message can be transmitted in a timely fashion.

1.2.2 Cryptographic Methods

One of the world's most renowned cryptographers, Claude Shannon, laid out three methods to send confidential messages:

1. *Steganography*: The message is hidden in plain view, but obscured.
2. *Privacy Systems*: Special equipment is required to encrypt or decrypt messages.
3. *True Secrecy*: The system of transmission is known, as well as the method of encryption, but the message cannot be seen without a key [4].

Although cryptography has become a staple of modern communications, it is much older than the computer or the Internet. History is full of examples where messages needed to be sent secretly. Yet, methods that were used in the past are no longer secure today. For example, when the ancient Greeks wanted to send a private message, they wrote it on the bald head of a slave. Once the slave's hair grew back he would journey to the recipient. Once the slave arrived, a shave of the head revealed the message [5]. This method is an example of steganography. The message is written in plaintext, but the system of transmission is hidden. While this system is quite slow and infeasible today, it shows that secrecy has been necessary for hundreds of years.

Modern cryptography deals with large-scale communication over open channels such as the Internet. Thus, Shannon's third principle of True Secrecy is the only method of encryption used today.

One of the first examples of a True Secrecy cipher was used in Rome eons ago. Rather than rely on the slave to reach his destination safely, Julius Caesar used a new cipher, currently known as the Caesar cipher. This simple cipher is one of the first demonstrations of basic mathematics being implemented in cryptography. The cipher shifts the letters of the alphabet over three places. To implement this quickly, let the English letters be represented by the numbers 0 through 25. Let M represent a plaintext letter in a message. Let C represent the ciphertext letter corresponding to M. Then each letter undergoes the mathematical operation:

$$C \equiv M + 3 \pmod{26} \tag{1.1}$$

Thus, the message "ready the troops" translates to "UHDGB WKH WURRSV."

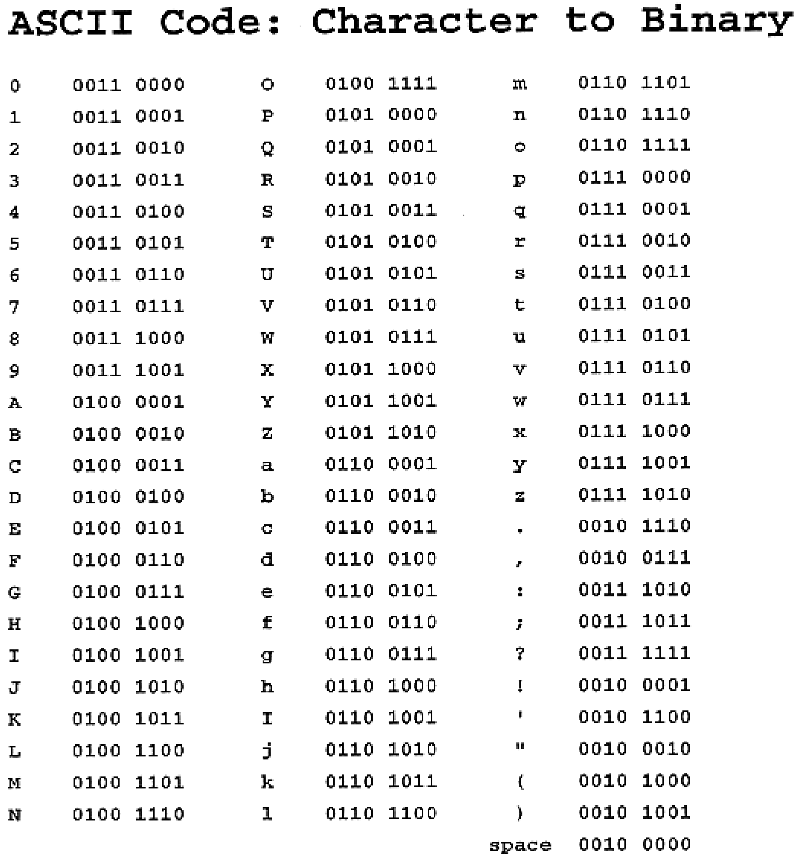
The Caesar cipher follows Shannon's third method of cryptography, because one cannot understand the ciphertext unless one knows that the key is to shift the letters three places to the right. Yet, this cipher is quite lacking, and only worked well in 100 BC when most people could not even read the plaintext. From this ciphertext it is obvious that the plaintext is three words long, and letters that repeat in the plaintext will repeat in the ciphertext. This is a clear problem. In addition, if one understood the system was simply to shift the alphabet, it would be easy to check all 25 possible (nontrivial) shifts.

1.2.3 Perfect Secrecy

Every cryptosystem that uses Shannon's True Secrecy method has a key. An adversary may try to guess this key. If an adversary does not have a plaintext/ciphertext pair, guessing can be difficult. Yet, if an adversary can obtain these pairs, they can analyze the encryption and look for patterns, as well as do analysis on the frequency of commonly used letters. However, this is only possible if keys are reused or repeated. These flaws motivate the idea of perfect secrecy.

Definition 1.2.1 *Perfect Secrecy: Let $H(K)$ be the probability that an adversary is able to guess the key of a message without a plaintext/ciphertext pair. Let $H(K|C)$ be the probability that an adversary is able to guess the key given an unlimited amount of plaintext/ciphertext pairs to analyze. A cryptosystem is considered perfectly secure if $H(K) = H(K|C)$* [4].

Such a system would solve the problem of secrecy. To begin designing such a system, it helps to understand how a message is represented. We cannot perform mathematical operations easily on letters, thus we need a system that changes letters into numbers that a computer can operate on. Yet we must make sure that the system we use is uniquely decipherable. We call this coding theory. One simple example of this encoding technique is ASCII. Figure 1.1 shows a simple conversion table of letters and numbers into binary using ASCII [6].



ASCII Code: Character to Binary

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

Figure 1.1: ASCII Code Chart

Now referring to the previous example, the ASCII message "ready the troops" can be represented in binary as 01010010011001010110000101100100011110010010000001110100011010000110010100 10000001010100011100100110111101101111011100000111001100001010.

If we wanted to encrypt this message so that an adversary did not know that we were indeed readying the troops, we would have to do some operation on this string of numbers. To utilize the Caesar cipher, simply add in binary "11" to each ASCII character, since "11" is the binary representation of the decimal number three. This is equivalent to adding "11" to each string of eight binary numbers. Thus, the change in format from letters to binary does not change the way we encrypt. It simply changes the way we represent our characters, putting them into a format that a computer can operate on.

Rather than use the Caesar cipher, which is rather trivial, a better encryption would be what we call a one-time pad. A one-time pad requires a key, which consists of bits that are equal in length to the message. The encryption is simply adding plaintext to the key, reducing the result modulo 2. This operation is the “exclusive or,” or the XOR, as it is commonly called.

With these tools, we can finally give a theoretically perfectly secure cipher. A one-time pad, where the key that is used consists of perfectly random bits equal to the length of the message, will be perfectly secure. This perfect secrecy is due to the fact that if the key is entirely random and unpredictable, it holds no patterns. Since its length matches the length of the message, we know it never repeats. Without knowledge of the key, any plaintext could be possible.

If we want to utilize random bits for encryption, we need a process for creating bits that we can rely on. The question that arises: Is it possible to generate truly random bits in a deterministic fashion? Much research has been done in this area. These generators can be implemented by chaotic processes that are viewed to be random. Stojanovski and Kocarev write a thorough analysis on this topic, using examples of generators that are implemented by “measuring radioactive decay, integrating dark current from a metal insulator semiconductor capacitor, detecting locations of photo-events, and sampling a stable high-frequency oscillator with an unstable low-frequency clock” [7]. Also, generators have been created that utilize minute differences in hardware processes, that are considered complex and hard to control. An example being Physical Unclonable Functions, which measures responses in silicon physical systems. These are thought to have minute differences that are impossible to predict ahead of time [8]. All of these are examples of random number generators that are thought to be truly random and unpredictable, but for the purposes of this thesis, we will not focus on these methods and turn our attention instead to deterministic generators, namely, pseudorandom number generators.

1.2.4 Pseudorandom Numbers

Pseudorandom number generators are a method of generating bits that can be used for encryption with a deterministic mathematical process. Intuitively, one should know that this will not be enough to satisfy our guidelines for a perfectly secure cipher. Yet, it is worth noting that our assumptions about our adversary in perfect secrecy are not realistic. Thus, pseudorandom bits may function in the same way as truly random bits, assuming an adversary does not have infinite time, infinite resources, and infinite plaintext/ciphertext pairs to work with.

Since we have mentioned that there are methods for generating truly random bits, a valid question is, why would we want to use pseudorandom bits instead? Mainly, a PRNG can be implemented much quicker and at less cost.

The security of a PRNG is dependent upon its use of computationally "hard" mathematical problems. For instance, given a very large number, n, the product of two primes, p and q, it is computationally hard to factor n without knowing p or q. This is the foundation of RSA. Another such problem is the discrete logarithm problem, which provides security for Diffie-Hellman key exchange, among other cryptographic tools.

In BBS the security is based on the difficulty of the quadratic residuosity problem as well as the difficulty to factor large numbers n, made up of Blum primes (defined below).

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Blum-Blum-Shub

Chapter 2 explains how pseudorandom sequences are generated using the original BBS as well as the cubic modification proposed in this paper. Second, this chapter introduces a public key cryptosystem based upon BBS as well as the mathematics behind this system. Finally, it presents a new public key cryptosystem based upon the modified cubic generator.

2.1 BBS Generator and Modification

In BBS, random bits are created by

1. Taking a random seed x_0.
2. Taking 2 large prime numbers p and q such that:
   - $p, q \equiv 3 \pmod 4$ (Such a prime is called a Blum prime)
   - $n = p * q$
3. Creating a sequence of numbers by squaring x_i:
   - $x_{i+1} \equiv x_i^2 \pmod n$
4. Taking the parity of each element in the sequence to create a binary string:
   - $y_i \equiv x_i \pmod 2$

This binary string provides our pseudorandom bits, which will be used for encryption. The two modifications proposed in this thesis are quite simple.

1. Modification 1: Sequences are created by cubing x_i.
   - $x_{i+1} \equiv x_i^3 \pmod n$
2. Modification 2: We take the last 10, 50, and 100 bits of each x_i to create a binary string (for our considered size of the modulus). Additionally, we concatenate the binary representations of each x_i to create a (potential pseudorandom) binary string.

2.2 BBS Public Key Cryptosystem

Chapter 1 explained the need for random numbers and pseudorandom number generators, as well as the basics of a one-time pad. BBS could be used as a one-time pad, but that is not its primary purpose. As we already mentioned, a one-time pad is a private key cipher, requiring both the sender and the recipient to know a pre-shared secret key in order to encrypt and decrypt. One the other hand, the BBS crytosystem is a public key crytosystem, where both parties need not know the key in order to communicate securely. This section will explain how communication takes place with BBS in a public key environment.

Consider Alice attempting to send Bob an encrypted message of length m using BBS. Let the plaintext message be denoted by $M = \{m_1, m_2, ..., m_m\}$ in binary. The crytosystem is a simple four step process, and will enable Alice to send a secure message to Bob.

1. Bob chooses two large primes p, q (which he keeps secret) and publishes their product, the modulus $n = p \cdot q$.
2. By using the BBS PRNG with modulus n, Alice will then use a random seed x_0 to create a random sequence of m numbers by squaring each term and reducing it modulo n to create the next number in the sequence. Let this sequence be $X = \{x_1, x_2, ..., x_m\}$. Alice then takes the parity of each element of X to create a bit string, which we will call the key: $K = \{k_1, k_2, ..., k_m\}$.
3. Alice XORs her message with the key to create a ciphertext, $C = \{c_1, c_2, ..., c_m\}$. $M \oplus K = C$. Alice sends C to Bob, along with x_{m+1}.
4. Bob repeatedly takes the square root of x_{m+1} to recover $\{x_m, x_{m-1}, ..., x_1\}$. After Bob recovers this sequence, he can easily recover K and compute $C \oplus K = M$.

The security of BBS relies upon the fact that step four of this process is impossible without knowing the factors of n. The next section will explain the mathematics behind this security.

2.3 Number Theoretic Background

This section focuses on the theorems and mathematics that secure BBS. Pascal Junod, a well-known cryptographer, writes a summary of many key theorems in his paper on BBS [9]. This thesis uses the relevant theorems and proofs he has written to provide a base understanding of why BBS is secure, given that the factors of n are kept secret.

2.3.1 Quadratic Residues and the Legendre Symbol

First, we define the concept of a quadratic residue found in [9].

Definition 2.3.1 *For $n \in \mathbf{N}$, Then $a \in \mathbf{Z}_{\mathbf{n}}^{*}$, is called a quadratic residue modulo n, if there exists $b \in \mathbf{Z}_{\mathbf{n}}^{*}$ such that*

$$a \equiv b^2 \pmod{n}$$

The set of quadratic residues modulo n is denoted by QR_n. Furthermore, the set of non quadratic residues modulo n is denoted by QNR_n.

To make matters easier we define the Legendre Symbol as follows from [9]:

Definition 2.3.2 *Let p be an odd prime, and $a \in \mathbf{Z}_{\mathbf{p}}^{*}$. The* Legendre symbol *is defined by*

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p|a \\ 1 & \text{if } a \in QR_p \\ -1 & \text{if } a \in QNR_p \end{cases}$$

Theorem 2.3.1 (commonly known as Euler's Theorem) from [9] will allow us to easily compute the Legendre symbol.

Theorem 2.3.1 *We have $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \pmod{p}$.*

Proof.

Let $a \in QR_p$. Then, $a \equiv b^2$ (mod p) for some $b \in \mathbf{Z_p^*}$. By Fermat's Little Theorem, it follows that

$$a^{\frac{p-1}{2}} \equiv (b^2)^{\frac{p-1}{2}} \equiv b^{p-1} \equiv 1 \pmod{p}. \tag{2.1}$$

Next, assume $a \in QNR_p$. Since p is prime, we can find g such that g is a generator of $\mathbf{Z_p^*}$, that is, $\mathbf{Z_p^*} = \{1, g, g^2, \ldots, g^{p-1}\} = \langle g \rangle$.

Since g is a generator, repeated multiplication will generate the entire structure, $\mathbf{Z_p^*}$. Thus our $a = g^t$, for some odd t. If t is even, then $t = 2s$ for some $s \in \mathbf{Z_p^*}$. Thus $g^t = g^{2s} = (g^s)^2$.

Then for odd $t = 2s + 1$,

$$a^{\frac{p-1}{2}} \equiv g^{t\frac{p-1}{2}} \equiv g^{2s\frac{p-1}{2}} \cdot g^{\frac{p-1}{2}} \pmod{p}. \tag{2.2}$$

By (2.1) we know a quadratic residue raised to the power $\frac{p-1}{2}$ reduces to 1, so we are left with

$$g^{\frac{p-1}{2}} \pmod{p}.$$

Note that $g^{\left(\frac{p-1}{2}\right)^2} \equiv 1 \pmod{p}$, by Fermat's Little Theorem. Then it is clear that $g^{\frac{p-1}{2}}$ is either 1 or -1. Finally, since g is a generator of the group, its order is precisely $p-1$, and so, $g^{\frac{p-1}{2}} = -1$ [9]. □

These theorems are significant, as they provide a grounds for the difficulty of the Quadratic Residuosity Problem, which is key to the security of BBS. By Theorem 2.3.1 we also infer that given a prime finite field $\mathbf{Z_p^*}$, half the elements will be quadratic residues, half will be quadratic non residues. As proof, consider a generator, g, of the field $\mathbf{Z_p^*}$. As stated in the proof of Theorem 2.3.1, g^t will be a quadratic residue for all even t. There are exactly $\frac{p-1}{2}$ such t's that are even. Thus, $|QR_p| = |QNR_p| = \frac{p-1}{2}$ [9].

Example 2.3.1 *Let $p = 7$. By squaring each element in $\mathbf{Z_7^*}$ we can compile a list of the quadratic residues and non residues:*

1. $\{1,2,4\} \in QR_7$.

2. $\{3,5,6\} \in QNR_7$.

As expected, half the elements are quadratic residues and half are non-residues.

Example 2.3.2 *Is* 2 *a quadratic residue in* $\mathbf{Z}^*_{\mathbf{29}}$*?*
If we have a p for which we do not want to list out the squares of each element, we can use Theorem 2.3.1 *to calculate the value of the Legendre symbol:*

$$\left(\frac{2}{29}\right) \equiv 2^{\frac{29-1}{2}} = \equiv 16384 \equiv 28 \equiv -1 \pmod{29}$$

Thus 2 *is a quadratic non residue in* $\mathbf{Z}^*_{\mathbf{29}}$*.*

2.3.2 Jacobi Symbol and Blum primes

Quadratic residues of $\mathbf{Z}^*_{\mathbf{p}}$ where p is prime is important, but recall that the BBS algorithm does not use $\mathbf{Z}^*_{\mathbf{p}}$ to generate sequences. Rather, BBS uses $\mathbf{Z}^*_{\mathbf{n}}$, where $n = p \cdot q$ and p, q are Blum primes.

This subsection will explain what happens to our quadratic residuosity theorems when we extend our environment to something other than $\mathbf{Z}^*_{\mathbf{p}}$, as well as explain the significance of Blum primes.

Definition 2.3.3 ***Jacobi Symbol***
The Jacobi symbol is used to evaluate the Legendre function $\left(\frac{a}{n}\right)$*, where n is not necessarily prime. To cite Junod again, let n be an odd integer with prime factorization* $n = \prod_i (p_i)^{e_i}$*. Then the Jacobi symbol* $\left(\frac{a}{n}\right)$ *is defined by*

$$\left(\frac{a}{n}\right) = \prod_i \left(\frac{a}{p_i}\right)^{e_i} \tag{2.3}$$

Junod goes on to prove that half the elements of $\mathbf{Z}^*_{\mathbf{n}}$ have Jacobi symbol $+1$ and half have Jacobi Symbol -1. These sets are denoted $\mathbf{Z}^*_{\mathbf{n}}(+1)$ and $\mathbf{Z}^*_{\mathbf{n}}(-1)$. None of the elements of $\mathbf{Z}_{\mathbf{n}}^*(-1)$ are quadratic residues, and exactly half the elements of $\mathbf{Z}_{\mathbf{n}}^*(+1)$ are quadratic residues [9].

Next we recall once again the definition of a Blum prime.

Definition 2.3.4 *A prime number p where $p \equiv 3 \pmod 4$ is called a Blum prime.*

The following theorem from [9] shows the importance of Blum primes.

Theorem 2.3.2 *We have that*

$$-1 \in QNR_p \textit{ if and only if } p \textit{ is a Blum prime.}$$

Proof. By Theorem 2.3.1

$$\left(\frac{-1}{p}\right) \equiv -1^{\frac{p-1}{2}} \pmod p \tag{2.4}$$

Note that p is odd by assumption, so p must be congruent to 1 or 3 modulo 4. If $p \equiv 1 \pmod 4$, then $\left(\frac{-1}{p}\right) = 1$. If $p \equiv 3 \pmod 4$, then $\left(\frac{-1}{p}\right) = -1$. □

If -1 were a quadratic residue then a BBS sequence could contain $x_i = -1$. For any n-bit long BBS sequence, if any $x_i = -1$ then the key K would have the property $\{k_{i+1}, k_{i+2}, \ldots, k_n\} = 1$. Obviously a key with this property will not be considered random, so Blum primes are a necessity.

2.3.3 Square Roots and the Chinese Remainder Theorem

Recall from Section 2.2 that in step four of the BBS cryptosystem, Bob was able to compute the square roots of x_{m+1}, which allowed him to re-create the key, K. This subsection will explain how this step takes place.

In the BBS cryptosystem, the sequence X is generated by squaring each term so that $x_{i+1} \equiv {x_i}^2 \pmod n$. Since BBS sends x_{m+1} unencrypted, if one could determine the square roots, they could recover the key. Jeremy Booher gives a polynomial-time algorithm for determining square roots in Blum prime fields, $\mathbf{Z_p^*}$:

$$\text{If } a \in QR_p \text{, then a square root is given by } a^{\frac{p-1}{4}} \text{ [10].}$$

For this reason, we do not use $\mathbf{Z_p^*}$ for BBS, as it would be easy for anyone to recover the key. However, when using $\mathbf{Z_n^*}$, where n is the product of two Blum primes, it is not as easy to determine square roots. In a prime field $\mathbf{Z_p^*}$ there are obviously two distinct square roots for every element. Junod goes on to show that if $a \in QR_n$ there are exactly 2^k distinct square roots of a, where k is the number of distinct prime factors of n [9]. In BBS, n is made up of two distinct prime factors, so each $a \in QR_n$ will have four distinct square roots. Of these square roots, only one will be a quadratic residue of n. If one knows the factors of the modulus n, the Chinese Remainder Theorem (CRT) helps in computing the square roots of each term. This is done by using the CRT to solve the system:

$$\begin{cases} x_m \equiv {x_{m+1}}^{\frac{p+1}{4}} \pmod{p} \\ x_m \equiv {x_{m+1}}^{\frac{q+1}{4}} \pmod{q}. \end{cases}$$

The result of this system allows the recipient to calculate the proper square root with certainty. Thus, the factors of the modulus are necessary in order to recover the proper sequence, and the correct sequence is required to recover the key. For a full proof of security, see [9] and [2].

2.4 A New Public Key Cryptosystem Based upon the Cubic Generator

While the BBS cryptosystem is well defined above, we found that there has been no previous research done on a cubic cryptosystem. This section addresses how the cubic modification proposed in this thesis can be implemented as a public key cryptosystem.

Previous research from [11] gives algorithms for computing cube roots in the finite field $\mathbf{Z_p^*}$, where p is prime. In addition, this research extends the algorithm to the field $\mathbf{Z}_\mathbf{n}^*$, where n is a composite number, by using the Chinese Remainder Theorem.

By using the CRT, our problem of finding cube roots in $\mathbf{Z}_\mathbf{n}^*$ becomes easier, as we must simply determine cube roots in $\mathbf{Z}_\mathbf{p}^*$, for all p that make up the prime factorization of n. For the cubic generator proposed in this thesis, n consists of two prime factors, $p * q$. From [11], if we assume that $p \equiv -1 \pmod 3$, then the cube root can be found as follows:

$$\sqrt[3]{a} \equiv a^{\frac{2p-1}{3}}. \tag{2.5}$$

Using this equation along with the CRT, it is possible to solve the following system and determine the cube root for the modulus n:

$$\begin{cases} \sqrt[3]{a} \equiv a^{\frac{2p-1}{3}} \pmod p \\ \sqrt[3]{a} \equiv a^{\frac{2q-1}{3}} \pmod q. \end{cases}$$

If we use our example of Alice sending Bob a message again, Alice can generate a sequence $X = \{x_0, x_1, x_2, \ldots, x_m\}$ with the cubic generator. She uses this sequence to generate her key, $K = \{k_0, k_1, k_2, \ldots, k_m\}$. She once again sends Bob the cipher text, $C = \{c_0, c_1, c_2, \ldots, c_m\}$, along with x_{m+1}. Using equation (2.5), Bob can solve the system given above to determine the cubic root of x_{m+1} and recover the entire sequence, X.

It is worth noting that it is necessary for Bob to choose his modulus n, wisely. The restrictions we place upon this cryptosystem are that $n = p \cdot q$, where

$$\begin{cases} p, q \equiv 3 \pmod 4 \\ p, q \equiv -1 \pmod 3. \end{cases}$$

This can be simplified again using the CRT to yield

$$p, q \equiv 11 \pmod{12}.$$

CHAPTER 3: Methodology

This chapter explains the procedure used to generate and test pseudorandom sequences. Next, it lays out the experiment that is performed in this thesis. Finally, this chapter gives a brief, yet useful, summary of each NIST test that is used.

3.1 Generating and Testing Sequences

This thesis uses a Python script to generate and test sequences. The source code for the codes referenced below can all be found in the Appendix. The method used is as follows:

- Generate Blum Prime Numbers p and q with Code 1.
- Specify a number of sequences, a length of each sequence, and how many bits per iteration. Generate the sequence and record it as a string with Code 2.
- Feed the string into each of the NIST randomness tests given in Code 3.

The sequences generated in this thesis will be roughly one million bits in length.

3.2 Experiment Procedure

As stated previously, the purpose of this thesis is twofold. First, we focus on implementing a cubic generator instead of the quadratic generator found in the original BBS. Second, we focus on the number of bits that we can take per iteration in order to improve the efficiency and test the bounds given by [1].

3.2.1 Research Objectives

The original BBS simply takes the parity bit during each iteration, which is probabilistically secure [2]. By using Vazirani and Vazirani's proof, we can safely take up to $\log(\log(n))$ bits and guarantee randomness (with high probability) [1]. While Vazirani and Vazirani guaranteed randomness up to that bound, it is an interesting idea to push these limits. In his own work, he limits the decryption to a probabilistic polynomial-time algorithm, and shows that with these assumptions the bound could be pushed up to $\sqrt{n}$. This thesis uses a modulus of size $n = 2^{512}$, thus $\log(\log(n)) = 9$. We will test this bound on the original quadratic generator used in BBS as well as the modified cubic generator by taking the last 10, 50, and even 100 bits per iteration. Additionally, we test the actual sequence $X = \{x_0, x_1, x_2, \ldots, x_m\}$ for randomness. The actual sequence means the binary representation of each x_i in the iteration. It is worth noting that some iterations will not give the desired amount of bits. For example, when generating a sequence from the last 100 bits per iteration, there can exist a number, x_i, within the sequence, X that will not be 100 bits in length. When this happens, the program simply takes the entire binary number, x_i. We could have used a padding technique, but we chose this simple approach for convenience.

3.3 Testing Sequences

This section focuses on explaining the NIST tests for randomness that are used in this thesis. They are statistical tests, so they do not determine whether a sequence is or is not random, but rather, they determine the statistical confidence we have in the randomness of a sequence. The details on the tests will not be exhaustive, since the statistics used to derive them are beyond the scope of this thesis. However, an understanding of what each test does and what "success" looks like is necessary.

The following tests have been written on extensively in Rukhin's et al. paper "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications" [12]. For all of the tests, the null hypothesis, H_0, will be the assumption that the sequence is random. The alternate hypothesis, H_a, will be that the sequence is not random. Each test yields a p-value, which is used to determine whether we reject, or accept the null hypothesis. Rukhin et al. recommends using $\alpha = 0.01$. Thus, if the p-value is below 0.01, we reject the null hypothesis [12].

3.3.1 Monobit Frequency Test

The Monobit Frequency Test is the simplest of all the statistical tests. The purpose of the test is to determine if there are an equal (or approximately equal, since the length may be odd) number of 0's and 1's in a given sequence, S_n. This is done by creating a new sequence, by converting the bit b into the integer $2b - 1$ (this is often called the sign-sequence, or its complement, of the original one), that is, 0's are mapped into -1 and 1's are left alone. Next, the test takes the sum of each element in the sequence, and computes the absolute value, $|S_n|$. After computing the test statistic, the test uses the complementary error function to determine the p-value.

It is worth noting that while a sequence may pass this test, that does not mean the sequence is random. The Monobit Frequency Test can be thought of as the first step in a refining process. It simply acts as a filter, which rules out the sequences that are quite clearly not random.

3.3.2 Block Frequency Test

The Block Frequency Test is a more specialized version of the Monobit Frequency Test. Instead of testing the entire sequence to find the proportion of 1's and 0's, the block frequency tests the proportion of 1's and 0's within a certain block size. The assumption of the test is that in an M bit block size, there will be approximately $M/2$ 1's and $M/2$ 0's. Note that if the block size is one, the test is nothing more than the Monobit test [12].

The purpose of the test is to reject sequences that appear to be random due to an equal proportion of 1's and 0's, but are not random due to the distribution of those numbers within the sequence. For example, the 10 bit sequence: 0000011111 passes the Monobit frequency test, but fails the block frequency test, as it is clearly not random.

3.3.3 Runs Test

A run is defined as a series of either 1's or 0's in a row. The purpose of the test is to determine if the length of runs in the sequence is an expected value for a random sequence.

The test is done by computing a test statistic $V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$, where $r(k) = 0$ if $\epsilon_k = \epsilon_{k+1}$ and $r(k) = 1$, otherwise [12].

Put simply, each bit is compared to the next bit in the sequence to check if they are the same or not, and then given a zero if they are the same, or a one if they differ. Taking the summation gives us a measurement on the amount of runs in the sequence.

The $V_n(obs)$ value is then put into the error function to determine the p-value.

3.3.4 Longest Runs Test

The Longest Runs Test measures the longest run within blocks of M-bits. The test determines if the length of this run is consistent with a value that would be expected of a random sequence.

The NIST standard uses three common block sizes for this test, $M = 8, 128$, and $10, 000$.

The test divides the sequence into blocks, and looks for the longest run of ones within the blocks. The test then computes the test statistic $\chi^2(obs)$, which is used to compute the p-value [12].

3.3.5 Spectral Test

The Spectral Test measures the peak heights of the Discrete Fourier Transform of the sequence and analyzes it for repetition and periodic behavior. Theoretically, in a random sequence, 95 percent of the values will not exceed T, where T is given by the equation $T = \sqrt{\log(1/0.05)n}$.

From this we compare the difference between the observed values and our theoretical bound, and derive d. This value is used in the error function to create our p-value.

3.3.6 Non-overlapping Template Matching Test

The Non-overlapping Template Matching Test divides the sequence into blocks. It then checks for the occurrence of a pre-specified, non-repeating template within these blocks. The test uses a sliding window equal to the length of the template. The purpose is to see if a particular template string occurs too often or not frequently enough within a sequence.

Once again the number of observed matches is compared to the theoretical number of matches in order to produce a test statistic, $\chi^2(obs)$, which is used to compute the p-value [12].

3.3.7 Overlapping Template Matching Test

The Overlapping Template Matching Test uses the same procedure as the Non-overlapping Template Matching Test. However, when a match to the template is found, then the window slides one bit over and continues its search [12].

3.3.8 Maurer's Universal Statistical Test

The Maurer's Universal Statistical Test focuses on determining whether the sequence can be compressed without significant loss of information. If a sequence is easily compressed, we determine the sequence is non random. The testing procedure is quite involved and an explanation is beyond the scope of this thesis. For more information on this test, we recommend the reader refer to [12].

3.3.9 Linear Complexity Test

Many pseudorandom sequences used in light-weight ciphers today are generated by Linear Feedback Shift Registers (LFSR). LFSR's can generate long strings of pseudo-random sequence quickly. The Linear Complexity Test is a measure of the complexity of such pseudorandom sequences.

The test divides the sequence into blocks, and uses the Berlekamp-Massey algorithm to determine the Linear Complexity of each block and compares it to a theoretical mean under the assumption of randomness [12].

3.3.10 Approximate Entropy Test

The Approximate Entropy test records the bits of overlapping blocks of the sequence of size M. Given the fact that there are 2^M possible sequences for a block of size M, the test creates a probability distribution of these recorded blocks. The purpose of the test is to calculate the approximate entropy by comparing these block distributions to what an acceptable value would be for a random sequences [12].

3.3.11 Cumulative Sums Test

The purpose of the Cumulative Sums Test is to determine the greatest excursion of random walks within a sequence. A random walk can be defined in terms of a graph. First, this test changes all zeroes within the tested sequence to -1. Consider a graph formed by taking the partial sums of this sequence. This graph will begin at the point 0 and at each step along the x-axis will either move up or down one unit based on whether the next term of the sequence is a one or a zero. The excursion of the graph is defined as the greatest distance such a graph strays from zero. In a random sequence the excursion of a random walk will be close to zero. If a sequence is non random however, the excursion will be large [12].

3.3.12 Random Excursions Test

Similar to the Cumulative Sums Test the Random Excursions Test is a measure of the excursions of random walks. It is a series of eight tests with eight results for each test. The purpose is to determine if there is a deviation from the expected value of a random sequence [12].

3.3.13 Random Excursions Variant Test

Similar to the Random Excursions Test the Random Excursions Variant Test is used to test the number of times a particular state is seen in a random walk. The test is a series of 18 tests, and each produces a specific p-value. The results determine if the number of deviations in states in a random walk is consistent with the expected value [12].

CHAPTER 4:
Results and Analysis

This chapter will focus on the results of the NIST tests, and an analysis of the data. The chapter will give data comparing the quadratic generator to cubic generator. It will also show results for the NIST tests on sequences that were generated by taking more than the last $\log(\log(n))$ bits from each iteration.

4.1 Results and Analysis

As explained earlier, this thesis focuses on two research objectives. First, we attempt to discover how well the cubic generator performs. Second, we attempt to discover how well sequences that take more than the parity bit perform. Below, data is shown for five cubic modifications, with quadratic data as a comparison. Data is gathered from a sample size of 200 sequences, each roughly one million bits in length. Each section below will give the success rates per NIST test, as well as a success rate per sequence.

4.1.1 Parity Bit Samples

Below, we show results for cubic and quadratic sequences that were generated by taking the parity bit each iteration.

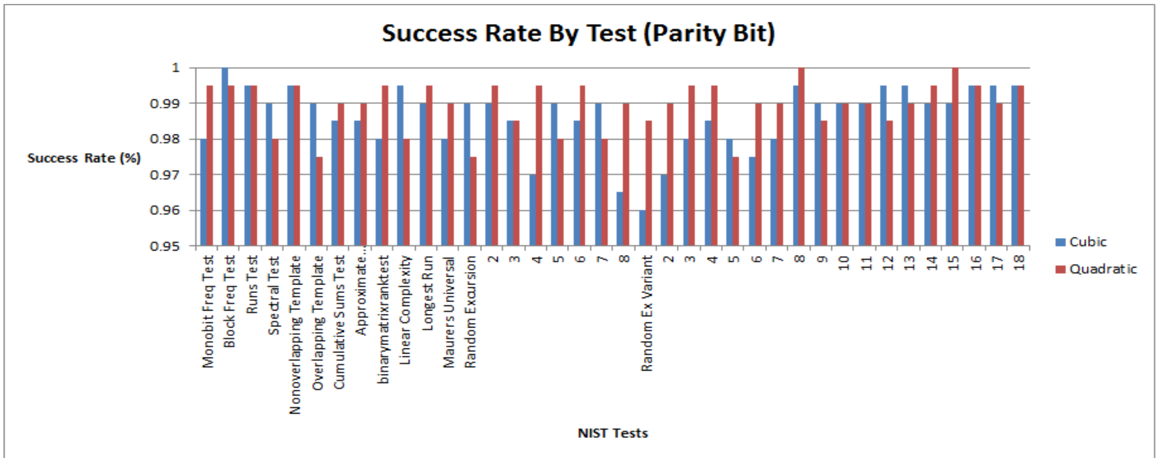
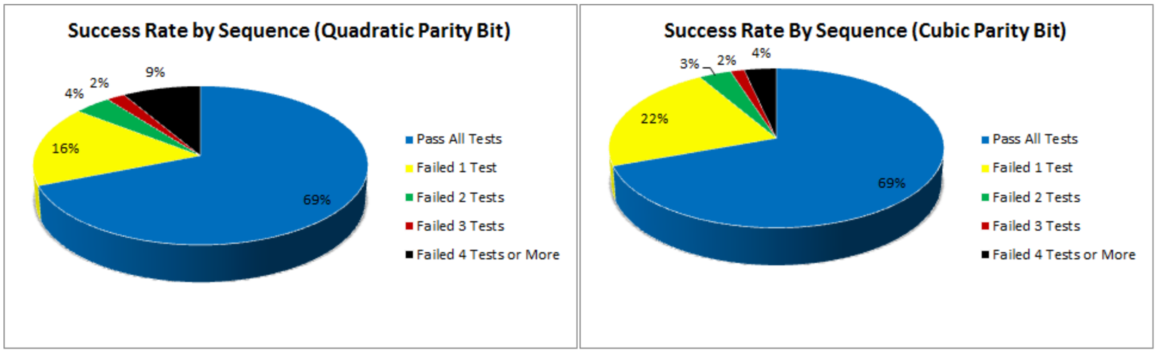


Figure 4.1: Parity Bit NIST Test Results

Overall, the success rate per NIST test was above 95 percent for both the cubic and quadratic generator. Recall that the random excursion test and random excursion variant test are multiple part tests, so each of the results are shown above.

Looking at success rate per sequence rather than per test is a stricter standard. The following graphs show the percentage of sequences which passed every NIST test, comparing the quadratic generator to the cubic one.



(a) Quadratic

(b) Cubic

Figure 4.2: Quadratic versus Cubic Parity Sequence Results

This allows us to see that while 95 percent of the parity sequences may pass a certain NIST test, the number of sequences that pass every NIST test is lower. In this sample 69 percent of the parity sequences generated with both quadratic and cubic generators pass every single test. The results seem to show no discernible difference between the original quadratic generator and the modified cubic generator.

4.1.2 Last 10 Bits per Iteration

Below, we show results for cubic and quadratic sequences that were generated by taking the last ten bits per iteration. Recall that our theoretical bound for the number of bits that can safely be taken per iteration is $\log(\log(n))$ where n is the size of the modulus and the log is base 2. Our modulus size is 512 bits, so $\log(\log(512)$ is roughly 9. Thus, our sequences from this point on all exceed the bound.

The following graph shows the success rate for each NIST test.

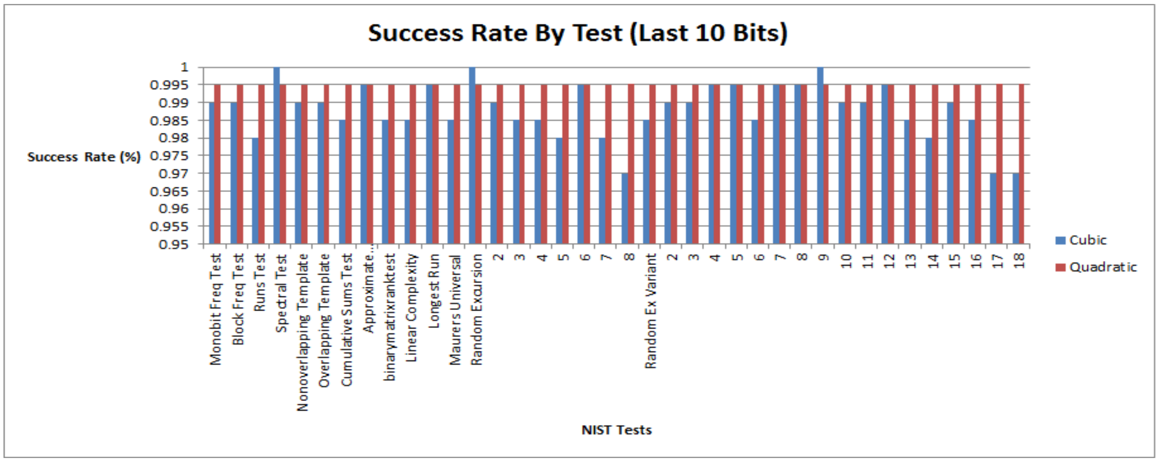
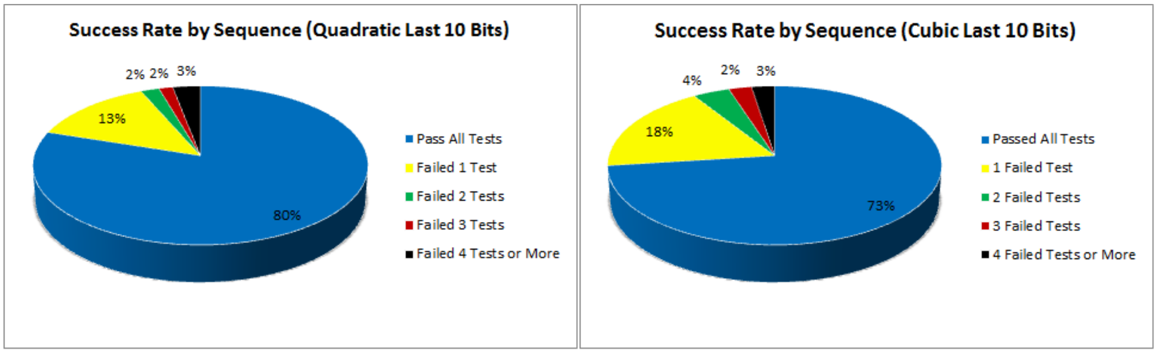


Figure 4.3: Last 10 Bit NIST Test Results

The success rate per test is above 95 percent for all NIST tests that were used in this experiment, with little noticeable difference between the two generators.

The graphs below compare the success rate per sequence for both quadratic and cubic sequences generated with the last ten bits.



(a) Quadratic

(b) Cubic

Figure 4.4: Quadratic versus Cubic Last 10 Bit Sequence Results

Quadratic sequences generated by taking the last ten bits per iteration tend to perform slightly better than the modified BBS generator. Similar cubic sequences remain nearly unchanged from the parity generator.

4.1.3 Last 50 Bits per Iteration

Below, we show results for cubic and quadratic sequences that were generated by taking the last 50 bits per iteration.

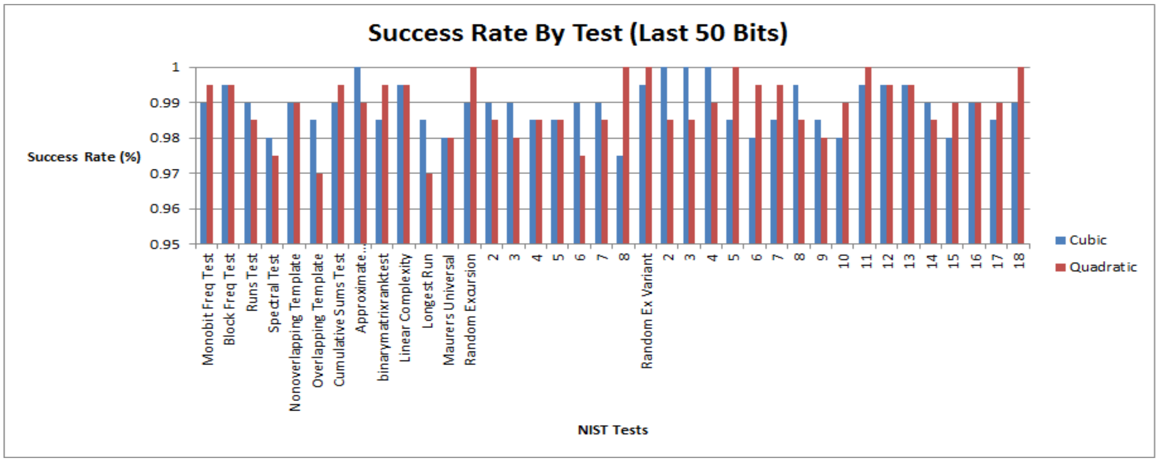
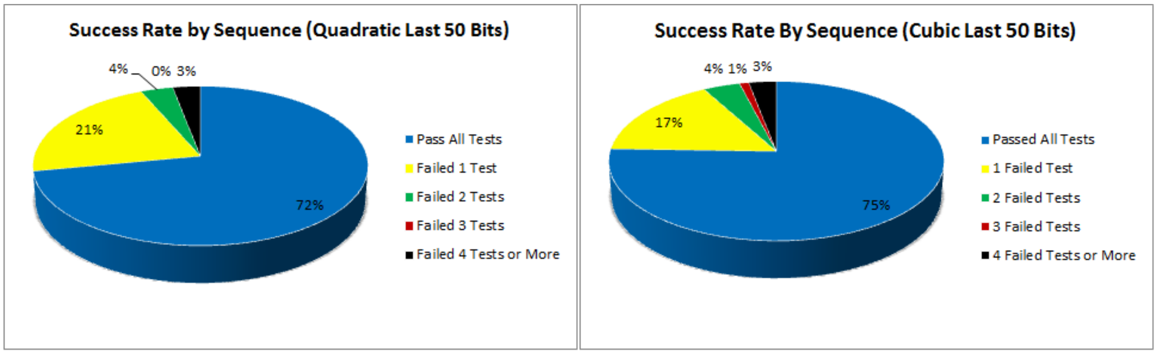


Figure 4.5: Last 50 Bit NIST Test Results

The results once again show all tests passed at above a 95 percent success rate. The graphs below again compare success rate per sequence.



(a) Quadratic

(b) Cubic

Figure 4.6: Quadratic versus Cubic Last 50 Bit Sequence Results

The two generators have similar results. It is interesting to note that we are far beyond the theoretical bound of 9, yet the percentage of sequences that pass all tests has actually increased slightly.

4.1.4 Last 100 Bits per Iteration

Below, we show results for cubic and quadratic sequences that were generated by taking the last 100 bits per iteration.

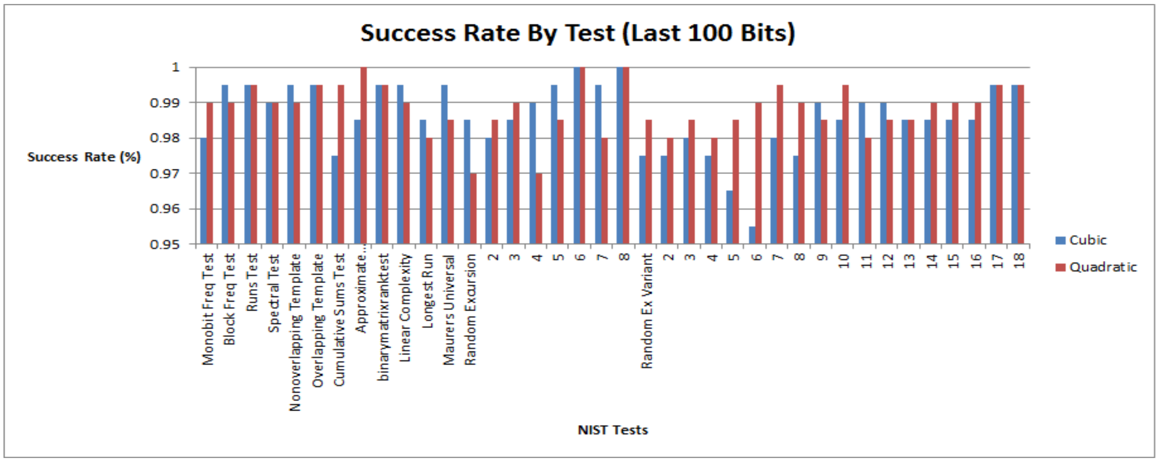
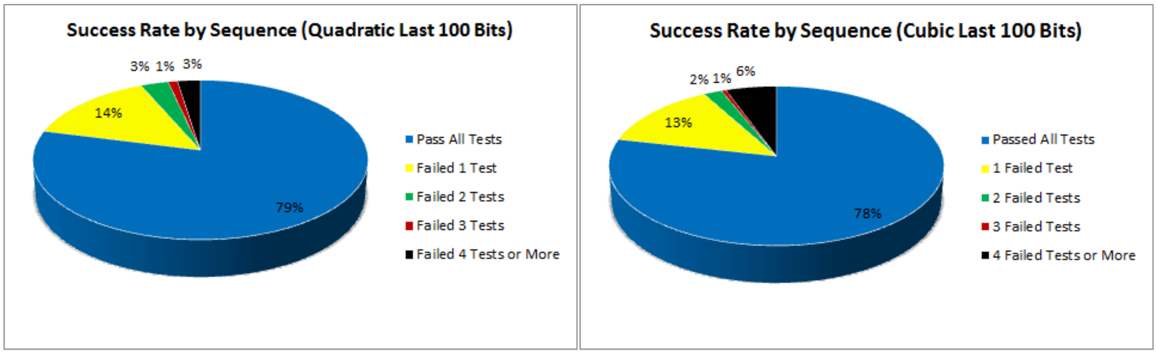


Figure 4.7: Last 100 Bit NIST Test Results

Even by increasing the generators to 100 bits per iteration the NIST tests still pass over 95 percent of the time. The following graphs compare success rate per sequence.



(a) Quadratic

(b) Cubic

Figure 4.8: Quadratic versus Cubic Last 100 Bit Sequence Results

Data suggests that both of these generators perform better than the parity generator by around 10 percent. However, it is roughly 100 times faster to generate sequences in this way than by using the parity generator.

4.1.5 Full Sequence Generator

Up until this point, data suggests that sequences performed well beyond the $\log(\log(n))$ bound. The following sequences maximize the number of bits that can be taken per iteration by simply taking the binary representation of the sequence number.

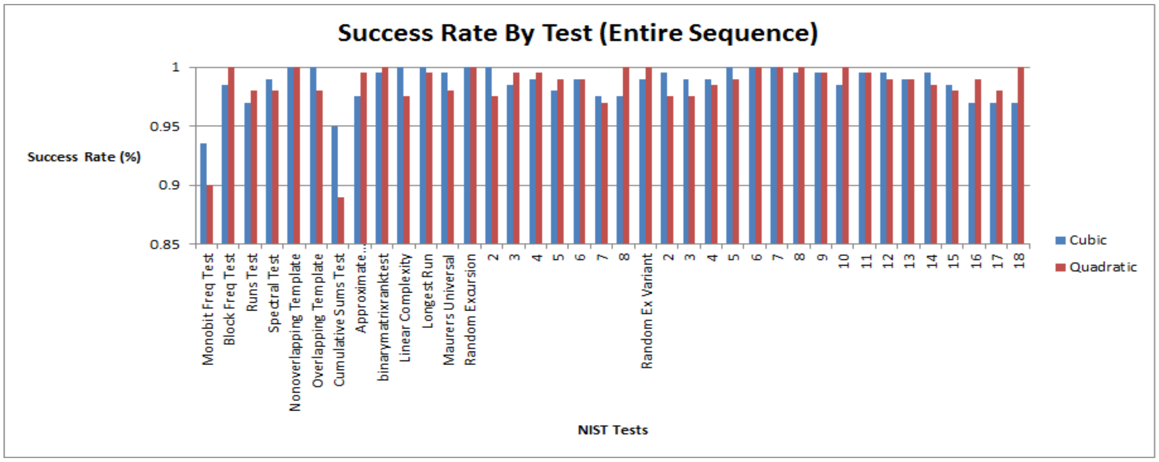
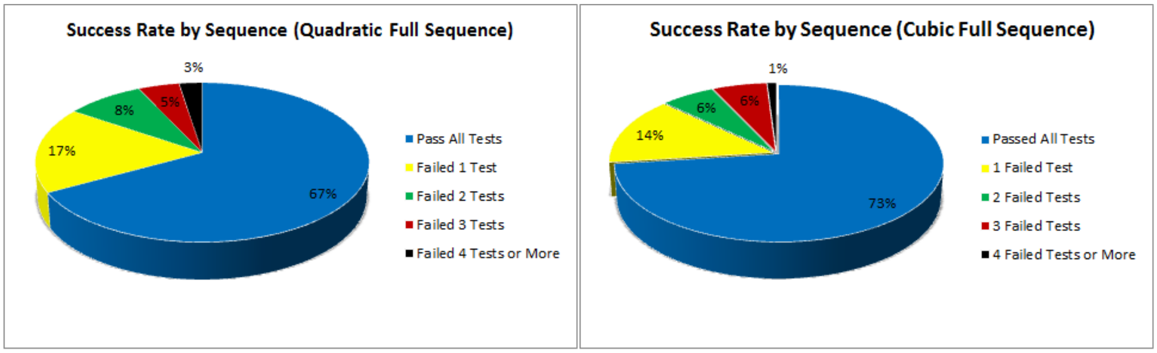


Figure 4.9: Full Sequence NIST Test Results

All NIST tests perform similarly to the previous data, with the exception of the cumulative sums test, which has a 89 percent success rate for the quadratic generator. The following graphs compare the success rate per sequence.



(a) Quadratic

(b) Cubic

Figure 4.10: Quadratic versus Cubic Full Bit Sequence Results

Data suggests that the cubic generator performs 6 percent better than the quadratic.

Surprisingly, the cubic generator also performs better than the original parity generator.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

This thesis proposes a cubic modification of BBS, a modification to the number of bits taken per iteration, and gives a new public key cryptosystem for the cubic modification.

Data gathered through NIST testing suggests that the cubic generator performs just as well as the quadratic generator. In addition, this research suggests that quadratic and cubic sequences generated by taking greater than $\log(\log(n))$ bits per iteration retain randomness according to NIST tests. One major issue with BBS is that bit generation takes a long time compared to other PRNGs; however, the data from this thesis could speed up bit generation significantly.

While the data gathered in this thesis suggests that the modifications to BBS retain randomness, many opportunities for future research in this area remain. The data gathered in this thesis is statistical, and no formal proof of security is given to the cubic generator. Areas of future work may include research into cubic residues and roots to complement [11]. Additionally, research could be done on the bound given in [1] to determine the theoretic safety of extracting large numbers of bits per iteration. Finally, this thesis provides a new public key cryptosystem. However, future research could be done on the computational feasibility of implementing such a system, as well as a comparison of this system to other common cryptosystems.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Programming Codes

This Appendix contains the programming codes used to generate and test sequences.

Code 1: Generating Large Primes

The following code is used to generate prime numbers which will be used as the p and q for our BBS generator. The code is taken from an open source website, 4dsolution.net [13].

```
import random

def bigppr(bits=256):
    """
    Randomly generate a probable prime with a given
    number of hex digits
    """

    candidate = random.getrandbits(bits) | 1  # Ensure odd

    prob = 0
    while 1:
        prob=pptest(candidate)
        if prob>0:
            return candidate
        candidate += 2

def pptest(n):
    """
    Simple implementation of Miller-Rabin test for
    determining probable primehood.
    """

    if n<=1:
        return 0
    # if any of the primes is a factor, we're done
    bases  = [random.randrange(2,50000) for x in xrange(90)]


    for b in bases:
        if n%b==0:
            return 0

    tests,s  = 0L,0
    m        = n-1

    # turning (n-1) into (2**s) * m

    while not m&1:  # while m is even
        m >>= 1
        s += 1

    for b in bases:
        tests += 1
        isprob = algP(m,s,b,n)
        if not isprob:
            break

    if isprob:
        return (1-(1./(4**tests)))

    return 0

def algP(m,s,b,n):
    """
    based on Algorithm P in Donald Knuth's 'Art of
    Computer Programming' v.2 pg. 395
    """
    y = pow(b,m,n)
    for j in xrange(s):
        if (y==1 and j==0) or (y==n-1):
            return 1
        y = pow(y,2,n)

        return 0
```

The code takes in the length of the prime number in bits. It then uses the Miller-Rabin test to determine primehood. It will increment the number by two until it finds one that passes the test. Once it passes, it returns this prime number.

Code 2: Main BBS

```
import csv
import random
import time
import numpy as np



randsequence = input("How many sequences?")
DesiredBits = input("How long do you want the bits sequence to be?")
divisor= input("How many of last n bits do you want to take
                to make bit sequence?")
IterationsNeeded = DesiredBits/divisor #this gives you the desired
                   bits eg. if you want 1 million bits sequence,
                   and wanna take last 4 bits, algorithm
                   will divide the iterations into 250000

file1 = open( "Name of File for the result.csv", ’w’)

file1.write("Monobit,Blockfreq, Runstest, Spectraltest,
            Nonoverlapping Template, Overlapping Template,
            Cumulative Sums Test,Approximate Entropy, binarymatrixranktest,
            Linear Complexity, Longest Run, Maurers Universal,
            Random Excursion1,2,3,4,5,6,7,8,
            Random Excursion Variant1,2,3,4,5,6,7,8,9,10,
            11,12,13,14,15,16,17,18,p,p1,q,q1,r,r1,xi, \n")


from PrimeGen import bigppr, pptest, algP

from randtest import monobitfrequencytest, blockfrequencytest, runstest,
                    spectraltest, nonoverlappingtemplatematchingtest,
                    overlappingtemplatematchingtest,cumultativesumstest,
                    aproximateentropytest, binarymatrixranktest,
                    linearcomplexitytest,longestrunones10000,
                    maurersuniversalstatistictest, randomexcursionstest,
                    randomexcursionsvarianttest




def rng():
    global xi, yi

    xi = (xi * xi) % n

    bin (xi)
    yi= bin(xi)[2:] #it eliminates the first two characters 0b...
    zi = str (yi)
    ai = str (zi[-divisor:])
    return ai

for i in range(randsequence):
    p = bigppr(256)
    q = bigppr(256)
    r = bigppr (256)

    n = p*q*r
    #n = p*q
    xi = bigppr (300)

    output = ’’

    result = ’’

    for i in range(IterationsNeeded):
        output += str(rng())

    p1 = p%4 #Double check if Blum-prime
    q1 = q%4 #Double check if Blum-prime
    r1 = r%4 #Double check if Blum-prime

    file1.write(str(monobitfrequencytest(output)) + ’,’ +
                str(blockfrequencytest(output)) + ’,’ +
                str(runstest(output)) + ’,’ + str(spectraltest(output))
                + ’,’ + str(nonoverlappingtemplatematchingtest(output))
                + ’,’ + str(overlappingtemplatematchingtest(output))
                + ’,’ + str(cumultativesumstest(output)) + ’,’ +
                str(aproximateentropytest(output)) + ’,’ +
                str(binarymatrixranktest(output)) + ’,’ +
                str(linearcomplexitytest(output))  +  ’,’ +
                str(longestrunones10000(output)) + ’,’ +
                str(maurersuniversalstatistictest(output)) + ’,’ +
                str(randomexcursionstest(output)) + ’,’ +
                str(randomexcursionsvarianttest(output))
                 + ’,’ + str(p)+ ’,’+ str(p1)+ ’,’ + str(q) + ’,’+ str(q1)
                 + ’,’ + str(r)  + ’,’+ str(r1) + ’,’ + str(xi) + ’\n’)

file1.close()

#toc= time.clock()
#print (toc - tic)
print p
print q
print xi
#print output
#print toc
import os
os.system('say "Your test is done" ')
```

This program was written by myself and a colleague in order to quickly and efficently implement BBS and modifications to BBS. The user specifies the number of sequences to generate, the length of the sequences, and how many bits to take per iteration. The sequences are generated as a string, and immediately tested using Code 3. The results are compiled in a CSV.

Code 3: NIST Tests

This program, written by Ilja Gerhardt [14], executes NIST tests on binary strings.

```
#!/usr/bin/env python

import numpy as np
import scipy.special as spc
import scipy.fftpack as sff
import scipy.stats as sst

def sumi(x): return 2 * x - 1
def su(x, y): return x + y
def sus(x): return (x - 0.5) ** 2
def sq(x): return int(x) ** 2
def logo(x): return x * np.log(x)

def pr(u, x):
    if u == 0:
        out=1.0 * np.exp(-x)
    else:
        out=1.0 * x * np.exp(2*-x) * (2**-u) * spc.hyp1f1(u + 1, 2, x)
    return out

def stringpart(binin, num):
    blocks = [binin[xs * num:num + xs * num:] for xs in xrange(len(binin)
             / num)]
    return blocks

def randgen(num):
    ’’’Spits out a stream of random numbers like ’1001001’
    with the length num’’’

    rn = open(’/dev/urandom’, ’r’)
    random_chars = rn.read(num / 2)
    stream = ’’
    for char in random_chars:
        c = ord(char)
        for i in range(0, 2):
            stream += str(c >> i & 1)
    return stream

def monobitfrequencytest(binin):

    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
    sn = reduce(su, sc)
    sobs = np.abs(sn) / np.sqrt(len(binin))
    pval = spc.erfc(sobs / np.sqrt(2))
    return pval

def blockfrequencytest(binin, nu=128):

    ss = [int(el) for el in binin]
    tt = [1.0 * sum(ss[xs * nu:nu + xs * nu:]) / nu for xs in
         xrange(len(ss) / nu)]
    uu = map(sus, tt)
    chisqr = 4 * nu * reduce(su, uu)
    pval = spc.gammaincc(len(tt) / 2.0, chisqr / 2.0)
    return pval

def runstest(binin):

    ss = [int(el) for el in binin]
    n = len(binin)
    pi = 1.0 * reduce(su, ss) / n
    vobs = len(binin.replace('0', ' ').split()) +
           len(binin.replace('1' , ' ').split())
    pval = spc.erfc(abs(vobs-2*n*pi*(1-pi)) / (2 * pi * (1 - pi)
           * np.sqrt(2*n)))
    return pval

def longestrunones8(binin):

    m = 8
    k = 3
    pik = [0.2148, 0.3672, 0.2305, 0.1875]
    blocks = [binin[xs*m:m+xs*m:] for xs in xrange(len(binin) / m)]
    n = len(blocks)
    counts1 = [xs+'01' for xs in blocks] # append the string 01 to
    guarantee the length of 1
    counts = [xs.replace('0',' ').split() for xs in counts1] # split into
    all parts
    counts2 = [map(len, xx) for xx in counts]
    counts4 = [(4 if xx > 4 else xx) for xx in map(max,counts2)]
    freqs = [counts4.count(spi) for spi in [1, 2, 3, 4]]
    chisqr1 = [(freqs[xx]-n*pik[xx])**2/(n*pik[xx]) for xx in xrange(4)]
    chisqr = reduce(su, chisqr1)
    pval = spc.gammaincc(k / 2.0, chisqr / 2.0)
    return pval

def longestrunones128(binin):  # not well tested yet
```

```
    if len(binin) > 128:
        m = 128
        k = 5
        n = len(binin)
        pik = [ 0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124 ]
        blocks = [binin[xs * m:m + xs * m:] for xs in
                 xrange(len(binin) / m)]
        n = len(blocks)
        counts = [xs.replace('0', ' ').split() for xs in blocks]
        counts2 = [map(len, xx) for xx in counts]
        counts3 = [(1 if xx < 1 else xx) for xx in map(max, counts2)]
        counts4 = [(4 if xx > 4 else xx) for xx in counts3]
        chisqr1 = [(counts4[xx] - n * pik[xx]) ** 2 / (n * pik[xx])
        for xx in xrange(len(counts4))]
        chisqr = reduce(su, chisqr1)
        pval = spc.gammaincc(k / 2.0, chisqr / 2.0)
    else:
        print 'longestrunones128 failed, too few bits:', len(binin)
        pval = 0
    return pval

def longestrunones10000(binin):  # not well tested yet

    if len(binin) > 128:
        m = 10000
        k = 6
        pik = [0.0882, 0.2092, 0.2483, 0.1933, 0.1208, 0.0675, 0.0727]
        blocks = [binin[xs * m:m + xs * m:] for xs in xrange(len(binin)
                 / m)]
        n = len(blocks)
        counts = [xs.replace('0', ' ').split() for xs in blocks]
        counts2 = [map(len, xx) for xx in counts]
        counts3 = [(10 if xx < 10 else xx) for xx in map(max, counts2)]
        counts4 = [(16 if xx > 16 else xx) for xx in counts3]
        freqs = [counts4.count(spi) for spi in [10,11,12,13,14,15,16]]
        chisqr1 = [(freqs[xx] - n * pik[xx]) ** 2 / (n * pik[xx]) for xx in
                  xrange(len(freqs))]
        chisqr = reduce(su, chisqr1)
        pval = spc.gammaincc(k / 2.0, chisqr / 2.0)
    else:
        print 'longestrunones10000 failed, too few bits:', len(binin)
        pval = 0
    return pval

# test 2.06
def spectraltest(binin):

    n = len(binin)
    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
    ft = sff.fft(sc)
    af = abs(ft)[1:n/2+1:]
    t = np.sqrt(np.log(1/0.05)*n)
    n0 = 0.95*n/2
    n1 = len(np.where(af<t)[0])
    d = (n1 - n0)/np.sqrt(n*0.95*0.05/4)
    pval = spc.erfc(abs(d)/np.sqrt(2))
    return pval

def nonoverlappingtemplatematchingtest(binin, mat="000000001", num=8):

    n = len(binin)
    m = len(mat)
    M = n/num
    blocks = [binin[xs*M:M+xs*M:] for xs in xrange(n/M)]
    counts = [xx.count(mat) for xx in blocks]
    avg = 1.0 * (M-m+1)/2 ** m
    var = M*(2**-m -(2*m-1)*2**(-2*m))
    chisqr = reduce(su, [(xs - avg) ** 2 for xs in counts]) / var
    pval = spc.gammaincc(1.0 * len(blocks) / 2, chisqr / 2)
    return pval

def occurances(string, sub):
    count=start=0
    while True:
        start=string.find(sub,start)+1
        if start>0:
            count+=1
        else:
            return count

def overlappingtemplatematchingtest(binin,mat="111111111",num=1032,numi=5):

    n = len(binin)
    bign = int(n / num)
    m = len(mat)
    lamda = 1.0 * (num - m + 1) / 2 ** m
    eta = 0.5 * lamda
    pi = [pr(i, eta) for i in xrange(numi)]
    pi.append(1 - reduce(su, pi))
    v = [0 for x in xrange(numi + 1)]
    blocks = stringpart(binin, num)
    blocklen = len(blocks[0])
    counts = [occurances(i,mat) for i in blocks]
    counts2 = [(numi if xx > numi else xx) for xx in counts]
    for i in counts2: v[i] = v[i] + 1
    chisqr = reduce(su, [(v[i]-bign*pi[i])** 2 / (bign*pi[i])
                  for i in xrange(numi + 1)])
    pval = spc.gammaincc(0.5*numi, 0.5*chisqr)
    return pval


def maurersuniversalstatistictest(binin,l=7,q=1280):

    ru = [
        [0.7326495, 0.690],
        [1.5374383, 1.338],
        [2.4016068, 1.901],
        [3.3112247, 2.358],
        [4.2534266, 2.705],
        [5.2177052, 2.954],
        [6.1962507, 3.125],
        [7.1836656, 3.238],
        [8.1764248, 3.311],
        [9.1723243, 3.356],
        [10.170032, 3.384],
        [11.168765, 3.401],
        [12.168070, 3.410],
        [13.167693, 3.416],
        [14.167488, 3.419],
        [15.167379, 3.421],
        ]
    blocks = [int(li, 2) + 1 for li in stringpart(binin, l)]
    k = len(blocks) - q
    states = [0 for x in xrange(2**l)]
    for x in xrange(q):
        states[blocks[x]-1]=x+1
    sumi=0.0
    for x in xrange(q,len(blocks)):
        sumi+=np.log2((x+1)-states[blocks[x]-1])
        states[blocks[x]-1] = x+1
    fn = sumi / k
    c=0.7-(0.8/l)+(4+(32.0/l))*((k**(-3.0/l))/15)
    sigma=c*np.sqrt((ru[l-1][1])/k)
    pval = spc.erfc(abs(fn-ru[l-1][0]) / (np.sqrt(2)*sigma))
    return pval

def lempelzivcompressiontest1(binin):

    i = 1
    j = 0
    n = len(binin)
    mu = 69586.25
    sigma = 70.448718
    words = []
    while (i+j)<=n:
       tmp=binin[i:i+j:]
       if words.count(tmp)>0:
           j+=1
       else:
           words.append(tmp)
           i+=j+1
           j=0
    wobs = len(words)
    pval = 0.5*spc.erfc((mu-wobs)/np.sqrt(2.0*sigma))
    return pval

def lempelzivcompressiontest(binin):

    i = 1
    j = 0
    n = len(binin)
    mu = 69586.25
    sigma = 70.448718
    words = []
    while (i+j)<=n:
       tmp=binin[i:i+j:]
       if words.count(tmp)>0:
           j+=1
       else:
           words.append(tmp)
           i+=j+1
           j=0
    wobs = len(words)
    pval = 0.5*spc.erfc((mu-wobs)/np.sqrt(2.0*sigma))
    return pval


# test 2.11
def serialtest(binin, m=4):

    n = len(binin)
    hbin=binin+binin[0:m-1:]
    f1a = [hbin[xs:m+xs:] for xs in xrange(n)]
    oo=set(f1a)
    f1 = [f1a.count(xs)**2 for xs in oo]
    f1 = map(f1a.count,oo)
    cou =f1a.count
    f2a = [hbin[xs:m-1+xs:] for xs in xrange(n)]
    f2 = [f2a.count(xs)**2 for xs in set(f2a)]
    f3a = [hbin[xs:m-2+xs:] for xs in xrange(n)]
    f3 = [f3a.count(xs)**2 for xs in set(f3a)]
    psim1 = 0
    psim2 = 0
    psim3 = 0
    if m >= 0:
        suss = reduce(su,f1)
        psim1 = 1.0 * 2 ** m * suss / n - n
    if m >= 1:
        suss = reduce(su,f2)
        psim2 = 1.0 * 2 ** (m - 1) * suss / n - n
    if m >= 2:
        suss = reduce(su,f3)
        psim3 = 1.0 * 2 ** (m - 2) * suss / n - n
    d1 = psim1-psim2
    d2 = psim1-2 * psim2 + psim3
    pval1 = spc.gammaincc(2 ** (m - 2), d1 / 2.0)
    pval2 = spc.gammaincc(2 ** (m - 3), d2 / 2.0)
    return [pval1, pval2]
```

```
def cumultativesumstest(binin):

    n = len(binin)
    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
    cs = np.cumsum(sc)
    z = max(abs(cs))
    ra = 0
    start = int(np.floor(0.25 * np.floor(-n / z) + 1))
    stop = int(np.floor(0.25 * np.floor(n / z) - 1))
    pv1 = []
    for k in xrange(start, stop + 1):
        pv1.append(sst.norm.cdf((4 * k + 1) * z / np.sqrt(n)) -
        sst.norm.cdf((4 * k - 1) * z / np.sqrt(n)))
    start = int(np.floor(0.25 * np.floor(-n / z - 3)))
    stop = int(np.floor(0.25 * np.floor(n / z) - 1))
    pv2 = []
    for k in xrange(start, stop + 1):
        pv2.append(sst.norm.cdf((4 * k + 3) * z / np.sqrt(n)) -
        sst.norm.cdf((4 * k + 1) * z / np.sqrt(n)))
    pval = 1
    pval -= reduce(su, pv1)
    pval += reduce(su, pv2)

    return pval

def cumultativesumstestreverse(binin):

    pval=cumultativesumstest(binin[::-1])
    return pval

def pik(k,x):
    if k==0:
        out=1-1.0/(2*np.abs(x))
    elif k>=5:
        out=(1.0/(2*np.abs(x)))*(1-1.0/(2*np.abs(x)))**4
    else:
        out=(1.0/(4*x*x))*(1-1.0/(2*np.abs(x)))**(k-1)
    return out

def randomexcursionstest(binin):

    xvals=[-4, -3, -2, -1, 1, 2, 3, 4]
    ss = [int(el) for el in binin]
    sc = map(sumi,ss)
    cumsum = np.cumsum(sc)
    cumsum = np.append(cumsum,0)
    cumsum = np.append(0,cumsum)
    posi=np.where(cumsum==0)[0]
    cycles=([cumsum[posi[x]:posi[x+1]+1] for x in xrange(len(posi)-1)])
    j=len(cycles)
    sct=[]
    for ii in cycles:
        sct.append(([len(np.where(ii==xx)[0]) for xx in xvals]))
    sct=np.transpose(np.clip(sct,0,5))
    su=[]
    for ii in xrange(6):
        su.append([(xx==ii).sum() for xx in sct])
    su=np.transpose(su)
    pikt=([([pik(uu,xx) for uu in xrange(6)]) for xx in xvals])
    # chitab=1.0*((su-j*pikt)**2)/(j*pikt)
    chitab=np.sum(1.0*(np.array(su)-j*np.array(pikt))**2
            /(j*np.array(pikt)),axis=1)
    pval=([spc.gammaincc(2.5,cs/2.0) for cs in chitab])
    return pval

def getfreq(linn, nu):
    val = 0
    for (x, y) in linn:
        if x == nu:
            val = y
    return val

def randomexcursionsvarianttest(binin):

    ss = [int(el) for el in binin]
    sc = map(sumi, ss)
    cs = np.cumsum(sc)
    li = []
    for xs in sorted(set(cs)):
        if np.abs(xs) <= 9:
            li.append([xs, len(np.where(cs == xs)[0])])
    j = getfreq(li, 0) + 1
    pval = []
    for xs in xrange(-9, 9 + 1):
        if not xs == 0:
            # pval.append([xs, spc.erfc(np.abs(getfreq(li, xs) - j) /
            np.sqrt(2 * j * (4 * np.abs(xs) - 2)))])
            pval.append(spc.erfc(np.abs(getfreq(li, xs) - j) /
            np.sqrt(2 * j * (4 * np.abs(xs) - 2))))
    return pval

def aproximateentropytest(binin, m=10):

    n = len(binin)
    f1a = [(binin + binin[0:m - 1:])[xs:m + xs:] for xs in xrange(n)]
    f1 = [[xs, f1a.count(xs)] for xs in sorted(set(f1a))]
    f2a = [(binin + binin[0:m:])[xs:m + 1 + xs:] for xs in xrange(n)]
    f2 = [[xs, f2a.count(xs)] for xs in sorted(set(f2a))]
    c1 = [1.0 * f1[xs][1] / n for xs in xrange(len(f1))]
    c2 = [1.0 * f2[xs][1] / n for xs in xrange(len(f2))]
    phi1 = reduce(su, map(logo, c1))
    phi2 = reduce(su, map(logo, c2))
    apen = phi1 - phi2
    chisqr = 2.0 * n * (np.log(2) - apen)
    pval = spc.gammaincc(2 ** (m - 1), chisqr / 2.0)
    return pval

def matrank(mat): ## old function, does not work as advertized -
gives the matrix rank, but not binary
    u, s, v = np.linalg.svd(mat)
    rank = np.sum(s > 1e-10)
    return rank

def mrank(matrix): # matrix rank as defined in the NIST specification
    m=len(matrix)
    leni=len(matrix[0])
    def proc(mat):
        for i in xrange(m):
            if mat[i][i]==0:
                for j in xrange(i+1,m):
                    if mat[j][i]==1:
                        mat[j],mat[i]=mat[i],mat[j]
                        break
            if mat[i][i]==1:
                for j in xrange(i+1,m):
                    if mat[j][i]==1: mat[j]=[mat[i][x]^mat[j][x]
                    for x in xrange(leni)]
        return mat
    maa=proc(matrix)
    maa.reverse()
    mu=[i[::-1] for i in maa]
    muu=proc(mu)
    ra=np.sum(np.sign([xx.sum() for xx in np.array(mu)]))
    return ra

def binarymatrixranktest(binin,m=32,q=32):

    p1 = 1.0
    for x in xrange(1,50): p1*=1-(1.0/(2**x))
    p2 = 2*p1
    p3 = 1-p1-p2;
    n=len(binin)
    u=[int(el) for el in binin] # the input string as numbers,
                                  to generate the dot product
    f1a = [u[xs*m:xs*m+m:] for xs in xrange(n/m)]
    n=len(f1a)
    f2a = [f1a[xs*q:xs*q+q:] for xs in xrange(n/q)]
    # r=map(matrank,f2a)
    r=map(mrank,f2a)
    n=len(r)
    fm=r.count(m);
    fm1=r.count(m-1);
    chisqr=((fm-p1*n)**2)/(p1*n)+((fm1-p2*n)**2)/(p2*n)+
           ((n-fm-fm1-p3*n)**2)/(p3*n);
    pval=np.exp(-0.5*chisqr)
    return pval

def lincomplex(binin):
    lenn=len(binin)
    c=b=np.zeros(lenn)
    c[0]=b[0]=1
    l=0
    m=-1
    n=0
    u=[int(el) for el in binin] # the input string as numbers, to generate
                                  the dot product
    p=99
    while n<lenn:
        v=u[(n-l):n] # was n-l..n-1
        v.reverse()
        cc=c[1:l+1] # was 2..l+1
        d=(u[n]+np.dot(v,cc))%2
        if d==1:
            tmp=c
            p=np.zeros(lenn)
            for i in xrange(0,l): # was 1..l+1
                 if b[i]==1:
                     p[i+n-m]=1
            c=(c+p)%2;
            if l<=0.5*n: # was if 2l <= n
                 l=n+1-l
                 m=n
                 b=tmp
        n+=1
    return l

# test 2.10
def linearcomplexitytest(binin,m=500):

    k = 6
    pi = [0.01047, 0.03125, 0.125, 0.5, 0.25, 0.0625, 0.020833]
    avg = 0.5*m + (1.0/36)*(9 + (-1)**(m + 1)) - (m/3.0 + 2.0/9)/2**m
    blocks = stringpart(binin, m)
    bign = len(blocks)
    lc = ([lincomplex(chunk) for chunk in blocks])
    t = ([-1.0*(((-1)**m)*(chunk-avg)+2.0/9) for chunk in lc])
    vg=np.histogram(t,bins=[-9999999999,-2.5,-1.5,-0.5,0.5,1.5,2.5,
       9999999999])[0][::-1]
    im=([((vg[ii]-bign*pi[ii])**2)/(bign*pi[ii]) for ii in xrange(7)])
    chisqr=reduce(su,im)
    pval=spc.gammaincc(k/2.0,chisqr/2.0)
    return pval

def testall(bits):
    print 'Length:\t\t\t\t\t', len(bits)
    print
    print 'monobitfrequencytest\t\t\t', monobitfrequencytest(bits)
    print 'blockfrequencytest\t\t\t', blockfrequencytest(bits, 3)
    print ’runstest\t\t\t\t’, runstest(bits)
    print ’spectraltest\t\t\t\t’, spectraltest(bits)
    print ’nonoverlappingtemplatematching\t\t’,
    nonoverlappingtemplatematchingtest(bits, ’1001’, 10)
    print ’overlapingtemplatematching\t\t’,
           overlappingtemplatematchingtest(bits, ’100’, 12, 5)
    print ’serialtest\t\t\t\t’, serialtest(bits, 10)
    print ’cumulativesumstest\t\t\t’, cumultativesumstest(bits)
    print ’aproximateentropytest\t\t\t’, aproximateentropytest(bits, 4)
    print ’randomexcursionsvarianttest\t\t’,
           randomexcursionsvarianttest(bits)
    print "linearcomplexitytest\t\t\t",linearcomplexitytest(bits,10)
    print "binarymatrixranktest\t\t\t",binarymatrixranktest(bits,3,4)
    print "lempelzivcompressiontest\t\t",lempelzivcompressiontest(bits)
    print "longestrunones10000\t\t",longestrunones10000(bits)
    print "maurersuniversalstatistictest\t\t",
           maurersuniversalstatistictest(bits,12,5)
    print "randomexcursionstest\t\t\t",randomexcursionstest(bits)
    return
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] U. V. Vazirani and V. V. Vazirani, “Efficient and secure pseudo-random number generation,” in *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, 1984, pp. 193–202.

[2] 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.

[3] K. Hansen, T. Larsen, and K. Olsen, “On the efficiency of fast rsa variants in modern mobile phones,” *IJCSIS (International Journal of Computer Science and Information Security*, vol. 6, no. 3, 2009.

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

[5] R. F. Churchhouse, *Codes and Ciphers: Julius Caesar, the Enigma, and the Internet*. Cambridge University Press, 2002.

[6] R. Parker, “Ascii code: Character to binary,” 2007. [Online]. Available: http://rossparker.org/bwa/

[7] T. Stojanovski and L. Kocarev, “Chaos-based random number generators-part i: analysis [cryptography],” *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 48, no. 3, pp. 281–288, 2001.

[8] C. W. O’Donnell, G. E. Suh, and S. Devadas, “Puf-based random number generation,” *In MIT CSAIL CSG Technical Memo*, vol. 481, 2004.

[9] P. Junod, “Cryptographic secure pseudo-random bits generation: The blum-blum-shub generator,” 1999.

[10] J. Booher, “Square roots in finite fields and quadratic nonresidues,” 2012.

[11] C. Padró and G. Sáez, “Taking cube roots in $\mathbf{Z_m}$,” *Applied Mathematics letters*, vol. 15, no. 6, pp. 703–708, 2002.

[12] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” Booz-Allen and Hamilton Inc, Mclean, VA, Tech. Rep., 2001.

[13] S. Tardieu, “primes.py.” [Online]. Available: https://github.com/VSpike/BBS/blob/master/primes.py

[14] I. Gerhardt, “randtest.py.” [Online]. Available: https://gerhardt.ch/random.php

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California