NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DIVIDE AND CONQUER MODIFICATION OF THE BLUM-MICALI PSEUDORANDOM NUMBER GENERATOR

by

Daniel K. Gillespie

June 2023

Thesis Advisor: Pantelimon Stanica
Second Reader: Thor Martinsen

THIS PAGE INTENTIONALLY LEFT BLANK

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

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2023	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
DIVIDE AND CONQUER MODIFICATION OF THE BLUM-MICALI PSEUDORANDOM NUMBER GENERATOR

5. FUNDING NUMBERS

6. AUTHOR(S) Daniel K. Gillespie

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 Blum-Micali pseudorandom number generator (PRNG) outputs cryptographically secure sequences of pseudorandom bits. One of the primary drawbacks of the Blum-Micali PRNG is that it can be computationally expensive and slow to run. This thesis proposes a modification to the Blum-Micali PRNG that allows for more pseudorandom bits to be extracted per iteration of the algorithm while retaining cryptographic security. We use the National Institute of Standards and Technology (NIST) Statistical Test Suite for PRNGs to evaluate the performance of sequences produced using this modification. In addition we compare the computational run time of our modification with that of the original Blum-Micali PRNG. Previous research indicates that there is an upper limit on the number of bits that may be extracted per iteration while retaining cryptographic security. Our test data suggest that our modification to the Blum-Micali PRNG performs just as well as the original version when extracting up to 11 bits per iteration. Furthermore, our data suggest that our modification can speed up computational run times in direct proportion to the number of bits extracted per iteration. We consider additional possible extensions that could further increase the speed of computation while still retaining randomness and cryptographic security.

14. SUBJECT TERMS Blum, Micali, Blum-Micali, pseudorandom, generator, cryptography, PRNG, pseudorandom number generator	**15. NUMBER OF PAGES** 73
	16. PRICE CODE

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

NSN 7540-01-280-5500

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

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

DIVIDE AND CONQUER MODIFICATION OF THE BLUM-MICALI PSEUDORANDOM NUMBER GENERATOR

Daniel K. Gillespie
Lieutenant Junior Grade, United States Coast Guard
BS, United States Coast Guard Academy, 2019

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2023

Approved by: Pantelimon Stanica
Advisor

Thor Martinsen
Second Reader

Francis X. Giraldo
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Blum-Micali pseudorandom number generator (PRNG) outputs cryptographically secure sequences of pseudorandom bits. One of the primary drawbacks of the Blum-Micali PRNG is that it can be computationally expensive and slow to run. This thesis proposes a modification to the Blum-Micali PRNG that allows for more pseudorandom bits to be extracted per iteration of the algorithm while retaining cryptographic security. We use the National Institute of Standards and Technology (NIST) Statistical Test Suite for PRNGs to evaluate the performance of sequences produced using this modification. In addition we compare the computational run time of our modification with that of the original Blum-Micali PRNG. Previous research indicates that there is an upper limit on the number of bits that may be extracted per iteration while retaining cryptographic security. Our test data suggest that our modification to the Blum-Micali PRNG performs just as well as the original version when extracting up to 11 bits per iteration. Furthermore, our data suggest that our modification can speed up computational run times in direct proportion to the number of bits extracted per iteration. We consider additional possible extensions that could further increase the speed of computation while still retaining randomness and cryptographic security.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

BBS	Blum-Blum-Shub
DLP	discrete logarithm problem
LFSR	linear feedback shift register
NIST	National Institute of Standards and Technology
NPS	Naval Postgraduate School
PRNG	pseudorandom number generator

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

This thesis proposes a modification to the Blum-Micali pseudorandom number generator (PRNG) that we term the Divide and Conquer Blum-Micali PRNG. The intention of this modification is primarily to speed up the generation of pseudorandom sequences while retaining statistical randomness and cryptographic security. This thesis compares the performance of the Classical Blum-Micali PRNG with the Divide and Conquer Blum-Micali PRNG. This is done by evaluating the randomness of the sequences produced by both PRNGs using the National Institute of Standards and Technology (NIST) test suite for assessing statistical randomness. Additionally, the computational run times required to generate pseudorandom sequences are used to assess the relative performance of the Divide and Conquer Blum-Micali PRNG.

The Divide and Conquer Blum-Micali PRNG extracts multiple bits per iteration by extending the basic principle used for bit extraction in the Classical Blum-Micali PRNG. In the Classical version, if the residue in a given iteration is greater than $\frac{p-1}{2}$ then a 1 is extracted; otherwise, a 0 is extracted. In the Divide and Conquer modification, we extract more bits by subdividing the interval of interest and then repeating the same process.

We observe in our experimental results that there are clear performance advantages to using the Divide and Conquer Blum-Micali PRNG. Under our experimental conditions, up to 11 bits per iteration can be extracted while producing outputs that are comparably random to those generated under the same conditions using the Classical Blum-Micali PRNG. When implemented, the Divide and Conquer modification can thus produce comparable pseudorandom sequences in approximately 1/11th of the time required by the Classical Blum-Micali PRNG. However, it appears that there is a strong upper limit to the number of bits that can be extracted per iteration while still producing statistically random outputs. We observe that when extracting 12 or more bits per iteration, the sequences produced by the Divide and Conquer modification are non-random with high probability. Therefore, we conclude that our modification, if implemented under similar conditions, should not be set to extract more than 11 bits per iteration.

Incidental to our investigation of both versions of the Blum-Micali PRNG, we find that the selection of an appropriate prime modulus is critical to the generation of random and cryptographically secure outputs. We find that fewer than half of the prime moduli we initially tested using the Classical Blum-Micali PRNG resulted in random or near-random sequences. While further research is needed, it initially appears that certain prime moduli result in short cycles when the Blum-Micali algorithm is applied.

Finally, we suggest some possible further extensions to our proposed modification to the Blum-Micali PRNG that could further reduce computational run times while retaining statistical randomness and cryptographic security.

Acknowledgments

I would like to thank my thesis advisor, Dr. Pantelimon Stanica, for both his ongoing support throughout this thesis process as well as for inspiring in me an interest in the mathematics of secure communications. I would also like to thank Captain Thor Martinsen for his support throughout my time here at Naval Postgraduate School (NPS). I am grateful to the Coast Guard and the American taxpayer for affording me the opportunity to study at NPS. I should remind myself that it is now my turn to fulfill my end of the deal by well and faithfully discharging the duties of the office in which I am about to enter upon graduation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Motivation

For centuries there has been a need for methods of secure communication given the presence of an adversary. Cryptography is a field of research involving mathematics and computer science that attempts to develop and implement such methods for modern applications. One important sub-field of cryptography is the development of cryptographically secure pseudorandom number generators (PRNGs). These generators are an important component in many cryptographic implementations since they can introduce randomness into an algorithm relatively efficiently. One such generator that is secure under certain conditions is the Blum-Micali PRNG [1]. One challenge with the Blum-Micali PRNG is that it is computationally expensive and can be slow. In this thesis, we develop a modification to the Blum-Micali PRNG with the intent of speeding it up while still retaining statistical randomness. We apply the National Institute of Standards and Technology (NIST) test suite for statistical randomness to the outputs produced by this new PRNG, which strengthens our claim that the modification we propose is worthy of further investigation.

1.2 Background

Random numbers have a variety of uses in cryptography that include generating public keys, one-time pads, and session keys [2]. There are naturally occurring phenomena such as radiation counters or radio-frequency noise that can act as sources of randomness; however, in practice these methods are implemented as PRNGs [2]. PRNGs are initialized with a short random seed and then use some deterministic algorithm to create a long random bit stream. The outputs of PRNGs are in fact periodic, but with such long periods that for practical purposes this fact is irrelevant. Additionally, it is important that an appropriate random seed is chosen to initialize the generator. For example, many PRNGs will output a bit stream of all 0's if the initial seed chosen is 0. While PRNGs trade off provable security for increased practicality, they are still considered secure for cryptographic purposes if properly designed and implemented [2]. The use of PRNGs is widespread today in a world

that relies heavily on cryptographically secure information exchange protocols; the internet (among other things) would not be the same without them.

1.2.1 Cryptographic Security of Pseudorandom Sequences

In many cases it is difficult to determine whether a PRNG is provably secure. In such cases, we resort to determining whether the output of a PRNG is experimentally secure. The pseudorandom output of a PRNG is considered experimentally secure if it passes either the next-bit test or all polynomial time statistical tests [3].

Definition 1.2.1. Passing the Next Bit Test

A PRNG passes the next bit test if "there exists no polynomial-time algorithm which, given as an input the first $n-1$ bits of some sequence produced from a random seed by the PRNG, can predict the nth bit of the string with a probability significantly greater than $\frac{1}{2}$" [2], [3].

Definition 1.2.2. Passing All Polynomial-Time Statistical Tests

A PRNG passes all polynomial-time statistical tests if "no polynomial-time algorithm can distinguish the output of the generator from a truly random bit string of the same length with a probability significantly greater than $\frac{1}{2}$" [2], [3].

In Chapter 3 we discuss the methodology used to show that our modification to the Blum-Micali PRNG passes these tests for various prime moduli. NIST has a standard suite of tests for statistical randomness for PRNGs.

1.2.2 Discrete Logarithm Problem

The security of many PRNGs is based on the computational intractability of a certain mathematical problem. One such problem is the discrete logarithm problem. The difficulty of this problem, given a sufficiently large input, is the source of the security of the Blum-Micali PRNG.

Definition 1.2.3. The discrete logarithm problem (DLP)

Given $\alpha \in$ group $G, \beta \in \langle\alpha\rangle$, find the least positive $x \in \mathbb{Z}$ such that $\alpha^x = \beta$ [4].

The computational difficulty of the discrete logarithm problem depends on the choice of the group G. If the group G is chosen to be $\mathbb{Z}_p^*$ where $p-1$ is the product of small primes, then the discrete logarithm problem can be solved efficiently using the Pohlig-Hellman algorithm [5]. Therefore, the choice of an appropriate prime modulus in any implementation of the Blum-Micali PRNG is necessary for cryptographic security.

1.3 Number Theory Concepts

While it is assumed that the reader is already familiar with the basic constructs that form the mathematical foundation for PRNGs, we briefly define a few key concepts here for clarity.

Definition 1.3.1. Congruences Modulo n
$\forall a, b, n \in \mathbb{Z}, n \neq 0$, if $\exists k \in \mathbb{Z}$ such that $a = b + nk$, then $a \equiv b \pmod{n}$ [6].

In implementations of the Blum-Micali PRNG, we seek to avoid congruent residues modulo p, since these produce repeating periods in a sequence. If $\exists w \in \mathbb{Z}$ such that $x_{i+w} \equiv x_i \pmod{p}$, then the period of a sequence produced by Equation (2.1) is $\leq w$.

Definition 1.3.2. Prime Fields
An algebraic structure that is isomorphic to the finite field $\mathbb{Z}_p^*$ for some prime number p is called a prime field [7].

Prime fields are the algebraic environment for the exponentiation algorithm used in the Blum-Micali PRNG. Furthermore various computational tricks (such as Fermat's Little Theorem) are based around the use of these structures, and allow for computations of large exponentiations modulo n that would otherwise result in overflow errors.

Definition 1.3.3. Primitive Roots/Generators
An element $g \in \mathbb{Z}_p^*$ is a primitive root (or generator) for $\mathbb{Z}_p^*$ if $\forall a \in \mathbb{Z}_p^*$, $\exists k \in \mathbb{Z}$ such that $g^k \equiv a \pmod{p}$ [8].

Often angular brackets are used to denote the set of elements generated by some element. If the element α is a primitive root modulo p, then $\langle \alpha \rangle = \mathbb{Z}_p^*$.

1.4 Quadratic Residues

In order to ensure that we are analyzing cryptographically secure residues, this thesis chooses prime moduli congruent to $3 \pmod 4$ with the intention of only analyzing prime moduli that appear to be more cryptographically secure. From the Prime Number Theorem, we know that asymptotically, about half of the primes of some fixed length are $\equiv 3 \pmod 4$, and about half are $\equiv 1 \pmod 4$ [9]. We do not have a theoretical reason to suspect that Blum primes, that is, primes that are $\equiv 3 \pmod 4$, play any significant role here, as they do in the Blum-Blum-Shub (BBS) PRNG [10]. However, we find that they do appear to play a role in our analysis, and thus treat this distinction as important. This may stem from the fact that half the residues modulo p are even and half are odd, so squaring, like in the BBS PRNG, does occur.

We remind the reader why we make the choice to use Blum primes [10]. If you take $y = x^2 \pmod p$, and have y, if $p \equiv 1 \pmod 4$, a solution x would simply be $x \equiv \pm y^{\frac{p-1}{4}}$. If p is a Blum prime there are two observations here. The first is that -1 is not a quadratic residue modulo p, and the second is that if x and $-x$ are the quadratic residues of y, then *exactly one of them* is a quadratic residue itself. Regarding BBS, exactly one of the four roots of the Blum integer $n = pq$ (both primes are $\equiv 3 \pmod 4$) is a quadratic residue. That is, the squaring is a *one-to-one map* from quadratic residues into quadratic residues.

1.5 Intuition for Modification

Long and Wigderson discuss in their 1988 paper the possibility of extracting k bits from each iteration of the Blum-Micali PRNG by dividing the interval $[0, p]$ into 2^k sub-intervals [11]. They state that it is "natural... to extend the Blum-Micali pseudorandom bit generator to output k bits per step" [11]. While Long and Wigderson discuss the possibility of such an extension and explore its theoretical effectiveness, they conduct no experimental or computational tests of their idea [11]. We take their theoretical basis for an extension and formalize the idea into the concept of 'buckets' and develop a decision rule mechanism to assign unique bit strings to each bucket. This thesis serves as a preliminary experimental exploration of the theoretical ideas posited by Long and Wigderson.

1.6 Berlekamp-Massey and Linear Complexity Profiles

One good measure of the randomness of a sequence is to examine its linear complexity profile. The linear complexity of a given sequence can be determined using the Berlekamp-Massey algorithm [12]. The Berlekamp-Massey algorithm "find[s] the shortest linear feedback shift register (LFSR) for a given binary output sequence" [12]. The order of the polynomial used to represent this shortest LFSR is the linear complexity of the sequence. For a random sequence, the order the polynomial representing this shortest LFSR should be approximately equal to $n/2$, where n is the number of bits in the sequence [13]. The linear complexity profile of a sequence is generated by applying the Berlekamp-Massey algorithm to the first k bits of a sequence, and taking the resulting linear complexities at every interval $mk \leq n$ where $m \in \mathbb{Z}$ and plotting these versus n [13]. A favorable linear complexity profile will closely track the function $n/2$ such as the profile displayed in Figure 1.1.

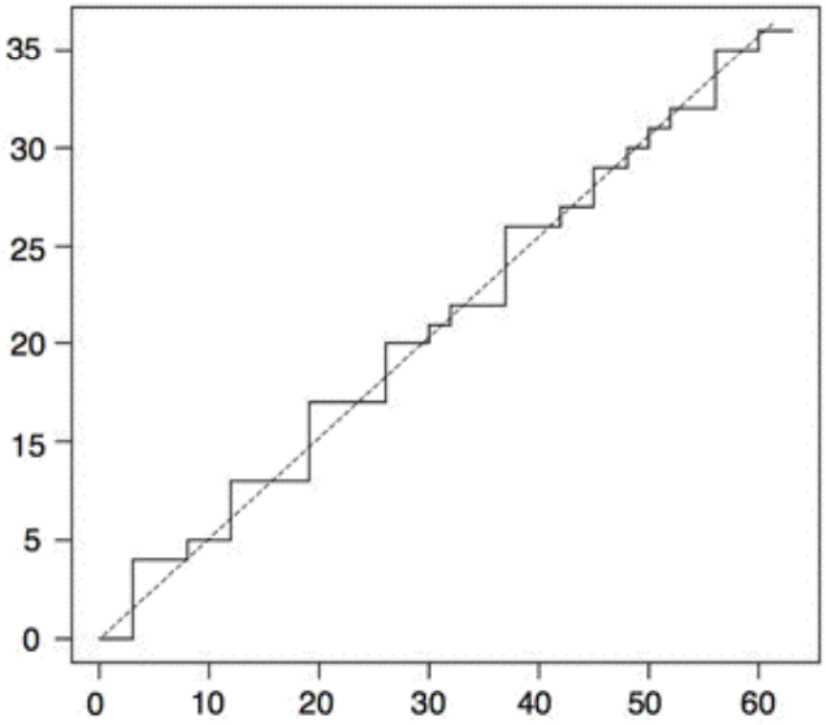

Figure 1.1. Example of a Favorable Linear Complexity Profile. Source: [13]

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Blum-Micali PRNG

2.1 Classical Blum-Micali PRNG

In this chapter, we explore the basic formulation of the Blum-Micali PRNG as well as our proposed modification. The Blum-Micali PRNG was originally proposed by Manuel Blum and Silvio Micali in their 1982 paper [1]. For the purposes of this thesis, we will refer to the version described in [1] as the Classical Blum-Micali PRNG.

Definition 2.1.1. The Classical Blum-Micali PRNG

1. Choose an appropriate prime number p that is sufficiently large to render the DLP intractable in the algebraic environment $\mathbb{Z}_p^*$.
2. Choose a primitive root g such that $\langle g \rangle = \mathbb{Z}_p^*$.
3. Choose an appropriate random seed x_0.
4. Calculate x_{i+1} using the following formula:

$$x_{i+1} = g^{x_i} \pmod{p}. \tag{2.1}$$

5. If $x_{i+1} < \frac{p-1}{2}$, extract a 0; otherwise, extract a 1.
6. Repeat steps four and five until you generate a bit string of the desired length.

Example 2.1.1. Two Iterations of the Classical Blum-Micali PRNG

Note: The prime modulus chosen for this example is too small for the output to be cryptographically secure or sufficiently random; it was selected for illustrative purposes only.

1. Let $p = 101$.
2. Let $g = 2$; $\langle g \rangle = \mathbb{Z}_{101}^*$.
3. Let $x_0 = 14$.
4. $x_1 = 2^{14} \pmod{101} = 16384 \pmod{101} = 22$.
5. $\frac{101-1}{2} = 50$.

6. $22 < 50$ so extract a 0 for the first bit of the pseudorandom bit string.
7. $x_2 = 2^{22} \pmod{101} = 4194304 \pmod{101} = 77$.
8. $77 \geq 50$ so extract a 1 for the second bit of the pseudorandom bit string.
9. Bit string = 01; Continue iterations until bit string is of desired length.

During each iteration of the Classical Blum-Micali algorithm, only one pseudorandom bit is extracted. With this bit extraction rule, n iterations of the algorithm are required to generate a sequence of length n.

2.2 Divide and Conquer Blum-Micali PRNG

With the intention of increasing the speed of the Classical Blum-Micali PRNG, this thesis uses the following modification to the bit extraction rule as proposed by Long and Wigderson [11]. We call this modified version the Divide and Conquer Blum-Micali PRNG in reference to the divide and conquer search algorithm used in its construction [14].

Definition 2.2.1. The Divide and Conquer Blum-Micali PRNG

1. Choose d, the number of bits to be extracted at each iteration.
2. Let bucket size $b = \lfloor \frac{p}{2^d} \rfloor$. There are 2^d equally sized buckets.
3. Use the divide and conquer search algorithm to determine which bucket x_{i+1} falls in. The divide and conquer search algorithm for finding the bucket that x_{i+1} falls in provides the process for extracting a bit string of length d.
4. Repeat this process until the pseudorandom bit string is of the desired length.

Note: Step three of this process requires the use of the divide and conquer search algorithm.

Given there are 2^d buckets, where d is the number of bits extracted per iteration, the number of bits that can be uniquely extracted per iteration using this method is upper bounded by the prime modulus p. Specifically:

$$2^d < p. \tag{2.2}$$

Additionally, given that p is prime, 2^d cannot be a divisor of p. Therefore, the buckets

cannot all be of exactly equal size and must be made to be approximately equal in size. There are several ways to deal with this issue. This thesis uses the following procedure to assign bucket sizes:

1. The baseline bucket size is $b = \lfloor \frac{p}{2^d} \rfloor$.
2. The remainder $r = p - b \cdot 2^d$.
3. The ith bucket size will be adjusted to be $b + 1$, where $i = x \cdot \lfloor \frac{p}{r} \rfloor$. Repeat $\forall x \in [1, r]$.

Definition 2.2.2. The Divide and Conquer Binary Search Algorithm

Let L be a list of n terms in increasing order. Let x be the element to located. Compare x to kth term of the list where $k = \lfloor \frac{n}{2} \rfloor$. If $x < k$, discard all terms greater than or equal to the comparison term. Otherwise, discard all terms less than the comparison term. Take the remaining terms and form a new list M with the $\lfloor \frac{n}{2} \rfloor$ remaining terms and repeat the process until x is located. For a list with n terms, this algorithm has the complexity class $O(\log_2 n)$ [14].

Example 2.2.1. Two Iterations of the Divide and Conquer Blum-Micali PRNG

Note: The prime modulus chosen for this example is too small for the output to be cryptographically secure or statistically random; it was selected for illustrative purposes only.

1. Let $p = 101$.
2. Let $g = 2$; $\langle g \rangle = Z^*_{101}$.
3. Let $x_0 = 14$.
4. Let the number bits extracted per iteration, $d = 2$.
5. Calculate bucket size $b = \lfloor \frac{101}{2^d} \rfloor = 25$.
6. Buckets $= [[0, b), [b, 2b), \ldots, [((2^d)-1)b, p]] = [[0, 24], [25, 49], [50, 74], [75, 101]]$.
7. $x_1 = 2^{14} \pmod{101} = 16384 \pmod{101} = 22$.
8. 22 falls in the first bucket $[0, 24]$ so extract 00 for the first two bits of the pseudorandom bit string.
9. $x_2 = 2^{22} \pmod{101} = 4194304 \pmod{101} = 77$.
10. 77 falls in the fourth bucket $[74, 101]$ so extract 11 for the third and fourth bits of the pseudorandom bit string.
11. Bit string = 0011; Continue iterations until bit string is of desired length.

2.3 Periodic Cycles with the Blum-Micali PRNG

The prime field $\mathbb{Z}_p^*$ has $p-1$ elements. In the Blum-Micali algorithm, x_{i+1} is directly determined by the value of x_i. Therefore, any sequence generated by the Blum-Micali algorithm will have a period of length at most p. Furthermore, in many cases the period of the sequence will be significantly less than p; this occurrence is called a short cycle in this thesis. To generate a pseudorandom sequence of length n using the Classical Blum-Micali PRNG, n iterations are required. Therefore, under the best case scenario with the Classical Blum-Micali PRNG, when generating a sequence of length n, the prime modulus must be large enough that $p > n$. If $p < n$, then periodicity is guaranteed within the output sequence. Since the presence of short cycles is always possible when p is randomly chosen within a given range, it is highly recommended that p be chosen such that $p \gg n$ (at least several orders of magnitude larger). Doing so ensures the probable length of any short cycle, s, is still sufficiently large such that $s > n$.

Since the Divide and Conquer Blum-Micali algorithm extracts b bits per iteration, to generate a sequence of length n, only $d = \lceil \frac{n}{b} \rceil$ iterations are required. This means that when implemented, the modified algorithm requires a modulus large enough that $p \gg d$. Thus, a smaller prime modulus is required when compared to the classical version in direct proportion to the number of bits extracted per iteration. Depending on the nature of the implementation, this can greatly reduce the computational run time of the algorithm.

2.4 A Note About Implementing the Blum-Micali PRNG

When implementing the Blum-Micali algorithm computationally, much care should be taken to choose an appropriate expression for $x_{i+1} = g^{x_i} \pmod{p}$ within the code. Some expressions use number theoretic tricks to increase the speed of this very large exponentiation while others do not. In Python implementations, a good function to use to address this issue is the *pow(root, power, modulus)* function that is built in to most Python distributions. We find in Python implementations there is approximately a $1000:1$ run time difference between the following two expressions:

Slow: *x1 = (g**x0)%p*

Fast: *x1 = pow(g, x0, p)*

CHAPTER 3:
Methodology

3.1 Testing Sequences for Statistical Randomness

In this thesis, we use the "Statistical Test Suite for Pseudorandom Number Generators for Cryptographic Applications" published by NIST to determine whether sequences generated by the Classical Blum-Micali and the Divide and Conquer Blum-Micali PRNGs are statistically random [15]. Detailed explanations of the purpose, method, and structure of each test can be found in the NIST documentation [15]. In this thesis, we use the following 16 statistical tests from Bassham et al. to assess the performance of each bit sequence:

1. The Frequency (Monobit) Test.
2. The Frequency Test within a Block.
3. The Runs Test.
4. The Longest-Run-of-Ones in a Block Test.
5. The Binary Matrix Rank Test.
6. The Discrete Fourier Transform (Spectral) Test.
7. The Non-Overlapping Template Matching Test.
8. The Overlapping Template Matching Test.
9. Maurer's 'Universal Statistical' Test.
10. The Linear Complexity Test.
11. The Serial Test.
12. The Approximate Entropy Test.
13. The Cumulative Sums Test (Forward).
14. The Cumulative Sums Test (Reverse).
15. The Random Excursions Test.
16. The Random Excursions Variant Test. [15]

It is important to note that several of the statistical tests within the NIST test suite require sequences that are at least one million bits in length [15]. Therefore, in this thesis we use sequences with a length of one million bits for all tests.

We use a Python script created by Kho-Ang and Churchill to implement this suite of tests computationally [16]. For each test, we use a p-value benchmark of 0.01. In this thesis, we consider a sequence random if it passes all 16 NIST tests with a p-value of 0.01 or higher. We consider a sequence near-random if it passes 15 of the 16 NIST tests with a p-value of 0.01 or higher. We consider a sequence non-random if it passes 14 or fewer NIST tests. While not an industry standard, this scheme was chosen to provide a more detailed analysis.

3.2 Generating Hard Prime Numbers and Choosing Primitive Roots

3.2.1 Choice of Prime Moduli

When implementing either version of the Blum-Micali PRNG, an appropriate prime modulus must be chosen. Prime moduli are appropriate candidates for use in implementation if they are sufficiently large hard prime numbers. It is worth noting that not all sufficiently large hard prime numbers are guaranteed to generate random or near-random sequences when implemented under every set of initial conditions. In our analysis, we only generated sequences using sufficiently large hard prime numbers as moduli. For the purposes of this thesis, a hard prime number, p, is one that has the following properties:

1. The prime factorization of $p - 1$ includes at least one large prime factor, q. We consider a prime factor, q, large if it has at least 16 bits such that $q > 2^{16} = 65,536$.
2. The prime $p \equiv 3 \pmod 4$.

We consider a hard prime p sufficiently large if it is large enough that it is probable that the period of a pseudorandom sequence of 1 million bits generated using the Classical Blum-Micali PRNG with modulus p is longer than 1 million bits. We observe experimentally that this means the hard prime should be chosen such that $p > 10,000,000$.

3.2.2 Generation of Prime Moduli

In this thesis we used a Python script titled "generateHardPrimeList.py" (Appendix A.3.1) to generate and select appropriate pairs of hard primes and associated primitive roots. Each sufficiently large hard prime modulus was generated using the following process:

1. Randomly generate a large prime factor, q, of at least 16 bits using the Python script published by Tabmir [17].
2. Generate a candidate hard prime, c, by using the formula $c = (2 \cdot x \cdot q) + 1$. Start with $x = 1$.
3. If c is prime and $> 10,000,000$, stop and let the candidate c be the hard prime number, p.
4. If not, let $x = x + 1$ and repeat steps two and three until an appropriate candidate is found.

Once each sufficiently large hard prime, p, was generated, we chose an associated primitive root, g, using the following process:

1. Select a candidate primitive root, r, from the set of the prime numbers in ascending order, $H = [2, 3, 5, 7, 11, \ldots]$.
2. If $r^{\frac{p-1}{2}} \equiv -1 \pmod{p}$, then let the candidate r be the primitive root g associated with the hard prime p.
3. Otherwise, select the next candidate r from the set H and return to step two.

3.3 Generating Pseudorandom Sequences

This thesis implements both the classical Blum-Micali and the Divide and Conquer Blum-Micali PRNGs using a series of Python scripts. These Python scripts are included in Appendix A.1 for reference. The following procedure was used to generate the pseudorandom sequences for our analysis:

1. Generate a list of sufficiently large hard prime numbers and their associated primitive roots.
2. Conduct one million iterations of the Classical Blum-Micali PRNG and output the resulting bit sequences for each hard prime.
3. Test these sequences using the 16 NIST tests. Only retain pairs of hard primes and primitive roots that produce random or near-random sequences using the Classical Blum-Micali PRNG. Repeat steps one and two until 100 such pairs of hard primes and primitive roots are found.

4. Now implement the Divide and Conquer Blum-Micali PRNG for 2 to 15 bits per iteration on each of the 100 pairs of hard primes and primitive roots. Generate pseudorandom sequences with a length of one million bits.
5. Test all sequences generated using the NIST test suite.

Note: With a list of hard primes where each hard prime is > 10 million, one will typically require an initial list of approximately 200-250 hard primes to end up with 100 hard primes that produce sequences that are random or near-random when used as the moduli for the Classical Blum-Micali PRNG.

3.4 Seed Selection

Both the Classical Blum-Micali and the Divide and Conquer Blum-Micali PRNGs require a random seed to be initialized. While there are $p - 1$ possible choices for the seed, and much analysis could be done to investigate the effects of different seed choices, this thesis chooses to use the same seed for all test sequences. This is done with the intention of standardizing the test sequences so that the choice of seed is not a relevant factor in our analysis. We use the seed $x_0 = 12345$ when generating all test sequences. This seed is large enough that for any primitive root, g, and hard prime modulus, p, between 10 million and 20 million, $y = g^{x_0} > p$. Since $y > p$, $y \neq x_1 = g^{x_0} \pmod{p}$; this ensures that the DLP is intractable from the first iteration of the implementation.

3.5 Linear Complexity Profile Analysis

This thesis uses a Python script in order to determine the linear complexity profiles of sequences generated by the Classical and Divide and Conquer Blum-Micali PRNGs. The Python script for the implementation of the Berlekamp-Massey algorithm was taken from work by Reid [18]. The Python script that ultimately generates and outputs the linear complexity profiles was written by the author and is included in Appendix A.4. Due to challenges with long computational run times, we only test the first $100,000$ bits of a sequence for its linear complexity profile and assume that the results are representative of the full $1,000,000$ bits. Additionally, we apply the Berlekamp-Massey algorithm in increments of 5000 bits, since this increment provides sufficient detail without prohibitive run times.

CHAPTER 4:
Results and Analysis

In this chapter we discuss the results of our analysis. Our analysis is split into two parts. In the first part, we generate sequences using various hard primes with the Classical Blum-Micali PRNG and examine the results using the NIST test suite. In the second part, we use 100 prime moduli known to produce random or near-random sequences with the Classical Blum-Micali PRNG, and then use them to generate sequences with the modified Divide and Conquer Blum-Micali PRNG. We examine the resulting sequences using the NIST tests, computational run times, and linear complexity profiles.

4.1 First Test Results: Classical Blum-Micali PRNG

We generated hard primes and used each as the modulus for a run of the Classical Blum-Micali PRNG. We then tested each sequence of one million bits to determine if the sequences were random or near-random (passed all, or all but one of the NIST tests). We repeated this process until we found 100 prime moduli that produce random or near-random sequences. Appendix B.1 contains the list of hard primes and primitive roots used for this portion of the analysis.

In total, 210 prime moduli were required to produce 100 random/near-random sequences. Interestingly, the sequences produced either passed all or nearly all of the NIST tests, or they failed most of them. There were very few sequences that failed some, but not most, of the NIST tests. Of the 210 hard prime moduli, only 26 failed between two and ten of the 16 NIST tests; 84 prime moduli failed 11 or more of the NIST tests. Figure 4.1 shows a summary of the results of this analysis.

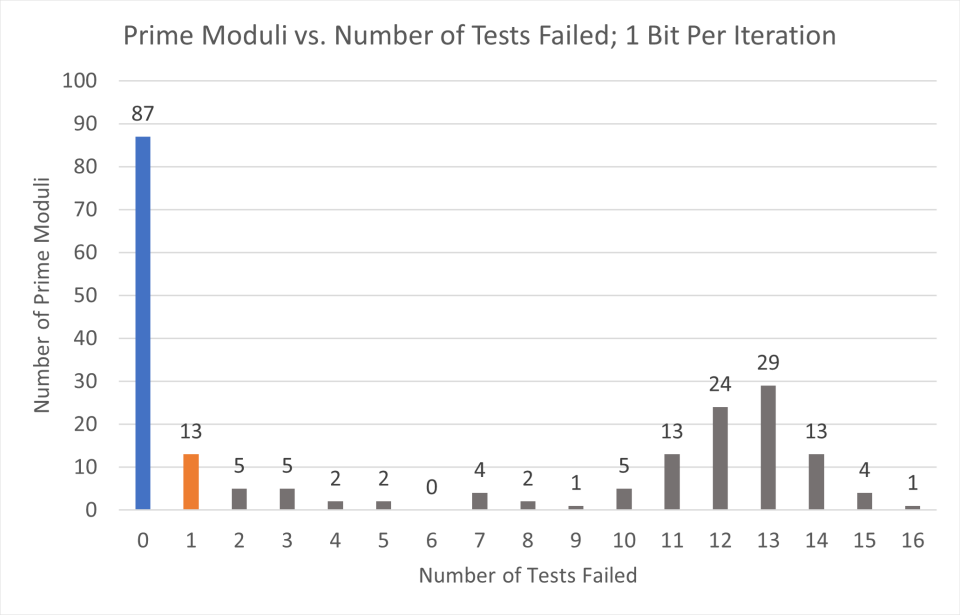

Figure 4.1. Frequency of Number of NIST Tests Failed Using the Classical Blum-Micali PRNG

When examining the 100 hard prime moduli that produced random/near-random sequences with respect to each specific NIST test, we find that some NIST tests were failed much more frequently than others. Specifically, the Spectral test, the Serial test, and the Approximate Entropy test were each failed more than 100 times. At the same time, some of the tests were significantly underrepresented in terms of test failures. Specifically, these were the Block Frequency test, Binary Matrix Rank test, Random Excursions test, and Random Excursions Variant test. Figure 4.2 presents a visual summary of these findings.

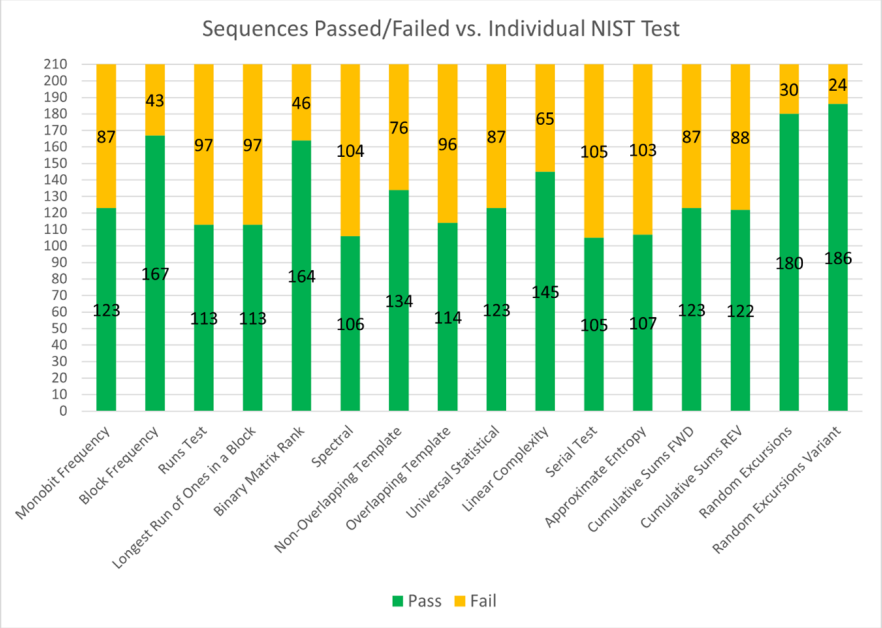

Figure 4.2. Proportion of Sequences Passed/Failed vs. NIST Test

The average run time to generate a one million bit sequence using the Classical Blum-Micali PRNG was 3.032 seconds. With the sample of hard primes ($p > 10,000,000$) used, we found that 47.6% were random or near-random. If a strict criterion for randomness was used (i.e., the sequence must pass all NIST tests), then we found that there is a 41.4% success rate. Furthermore, we found that given a sequence fails, it is probable that it failed quite significantly (i.e., failed more than 10 of the 16 NIST tests). While further investigation and research is required in this area, upon cursory visual inspection of some of the sequences that failed a majority of the NIST tests, we found that they appeared to be periodic. Intuitively, this suggests that there is a short cycle occurring with these prime moduli. Indeed, when examining the .txt files containing the bit sequences, we found that the sequences that failed most of the NIST tests exhibit a consistent patterned structure that can be observed visually by scrolling through the file. Sequences that are random or near-random do not appear to have the same property. Of course, this is not an acceptable mathematical method of analysis, and it is mentioned simply to inform our intuition. Figure 4.3 provides an example of this patterned property for a sequence that failed 15 of the 16 NIST tests.

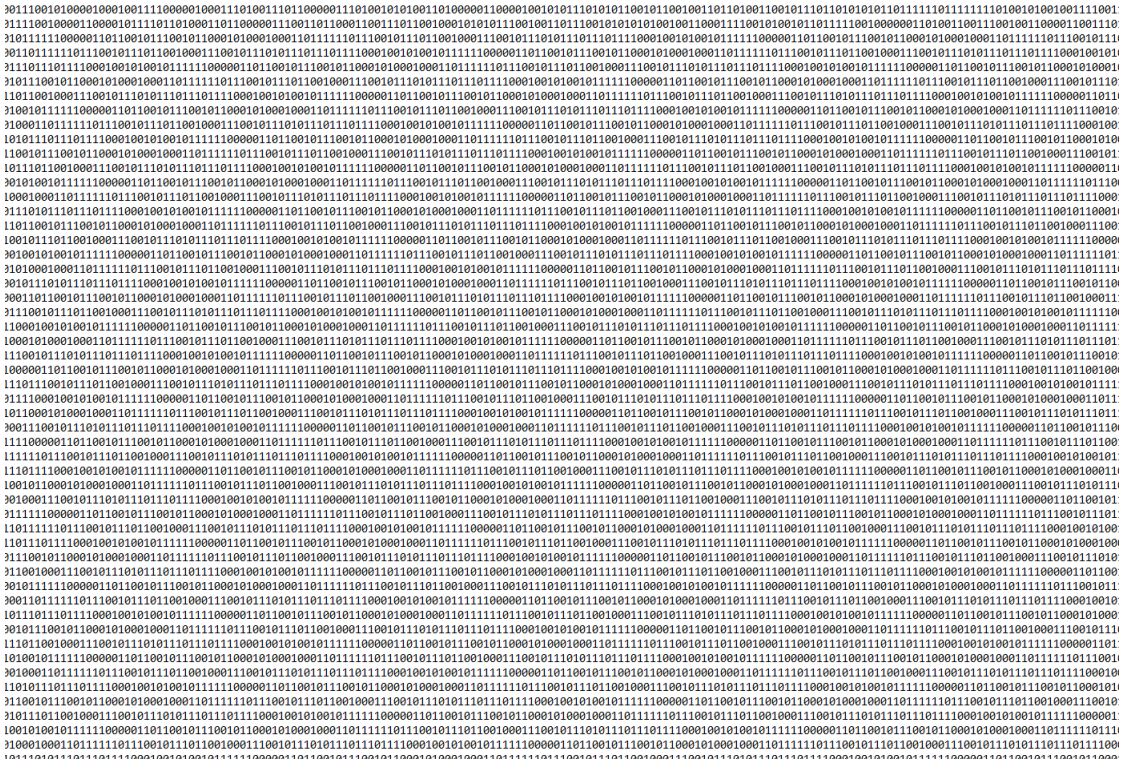

Figure 4.3. Screenshot of Classical Blum-Micali Sequence with $p = 12877951$ and $g = 3$

4.2 Second Test Results: Divide and Conquer Blum-Micali PRNG

We conducted this portion of the analysis by taking the 100 hard prime moduli from the previous results that yielded random/near-random sequences (see Appendix B.2) and then used them with the Divide and Conquer modified Blum-Micali PRNG. We chose to generate sequences by extracting 2 to 15 bits per iteration for each of the 100 hard prime moduli. We compare the performance of the sequences generated with the Divide and Conquer Blum-Micali PRNG to those generated with the Classical Blum-Micali PRNG in terms of both the proportion of the sequences that were random/near-random, as well as the computational run times of each implementation.

4.2.1 2 to 11 Bits Per Iteration

We found that extracting between 2 and 11 (inclusive) bits per iteration using the Divide and Conquer Blum-Micali PRNG yielded performance comparable to the sequences generated by the Classical Blum-Micali PRNG. For the sample of 100 hard prime moduli, we saw relatively stable and consistent performance in terms of number of random and near-random sequences; we did not observe any significant or anomalous decreases in performance. The worst performing case in this range was 6 bits per iteration; there were 79 random and 17 near-random sequences, with 4 non-random sequences. Even in this case, the results were still comparable to the Classical Blum-Micali baseline of 87 random and 13 near-random sequences. The overall performance was still at 96% of baseline, with 90.8% as many random sequences and 130.7% as many near-random sequences. Furthermore, it is worth noting that the fact that there were no non-random sequences in the Classical Blum-Micali baseline is an artifact of the selection process that discarded any non-random sequences in phase one of our analysis. At 11 bits per iteration, there were still 87 random and 11 near-random sequences, with only 2 non-random sequences. For reference, when extracting 11 bits per iteration, there were 2048 buckets. Since all hard prime moduli used were greater than 10 million, the ratio of the prime modulus to the number of buckets was at least 4882:1.

4.2.2 12 to 15 Bits Per Iteration

We found that once we extracted 12 or more bits per iteration, there was a precipitous decline in overall performance. While the number of near-random sequences remained roughly constant between 16 and 28, as expected, the number of random sequences fell rapidly and was displaced by non-random sequences. Figure 4.4 displays these findings in graphical form.

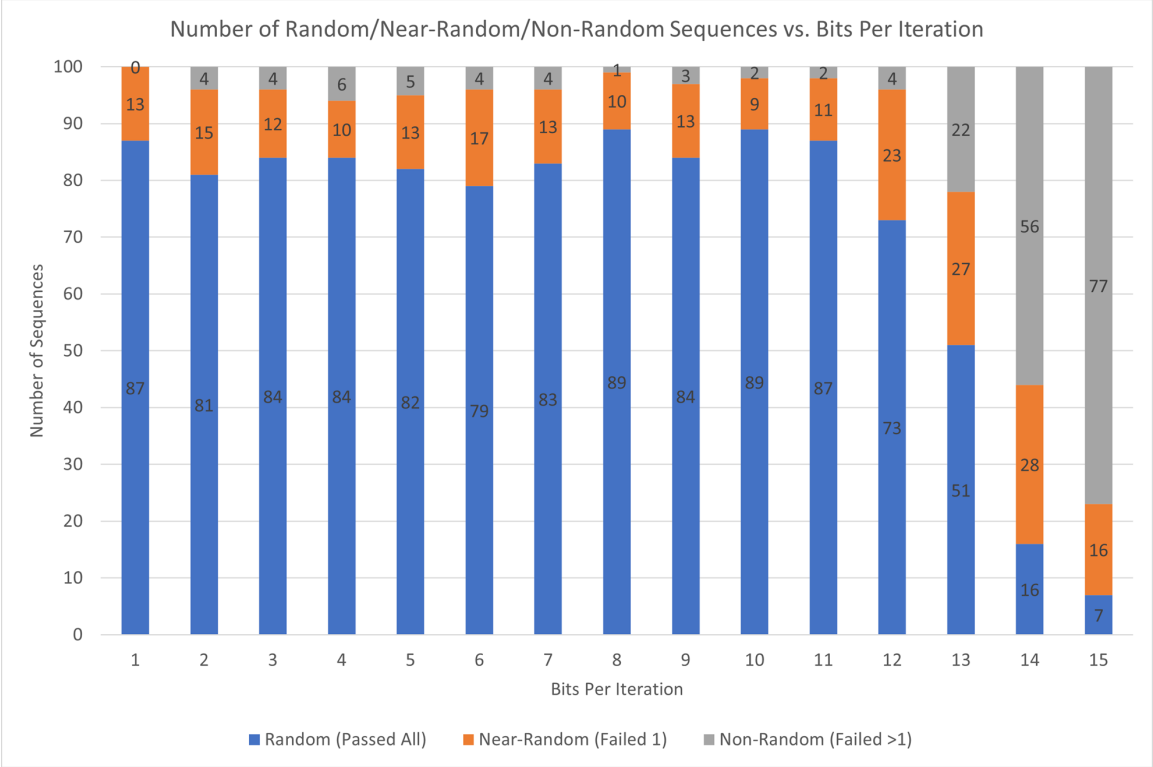

Figure 4.4. Overall Performance of Classical and D&C PRNGs from 1-15 Bits Per Iteration

The data clearly suggest that there is an upper bound to the number of bits per iteration that can be extracted using the Divide and Conquer Blum-Micali algorithm to produce a pseudorandom sequence. Furthermore, it appears that this bound is significantly less than the trivial upper bound of p (the prime modulus). It appears that a ratio of approximately 5000:1 for the prime modulus to the number of buckets (2^b where b = bits per iteration) is a reasonable estimate for this upper bound when the prime modulus is between $10,000,000$ and $20,000,000$. Further research is needed to determine whether specific properties of the prime modulus (such as the factors of $p-1$) and its primitive root have an effect on this upper bound.

Examining the results of the 12-15 bits per iteration sequences in isolation and looking at the individual NIST tests, we found that there were several tests that are failed with high probability. Specifically, these were the Longest Run-Of-Ones in a Block test, the Overlapping Template Matching test, the Approximate Entropy test, and the Serial test. On the other hand, most of the other NIST tests failed with relatively low probability even at 15 bits per iteration. Figure 4.5 displays the results by specific NIST test for the implementation extracting 15 bits per iteration and is quite illustrative of these findings.

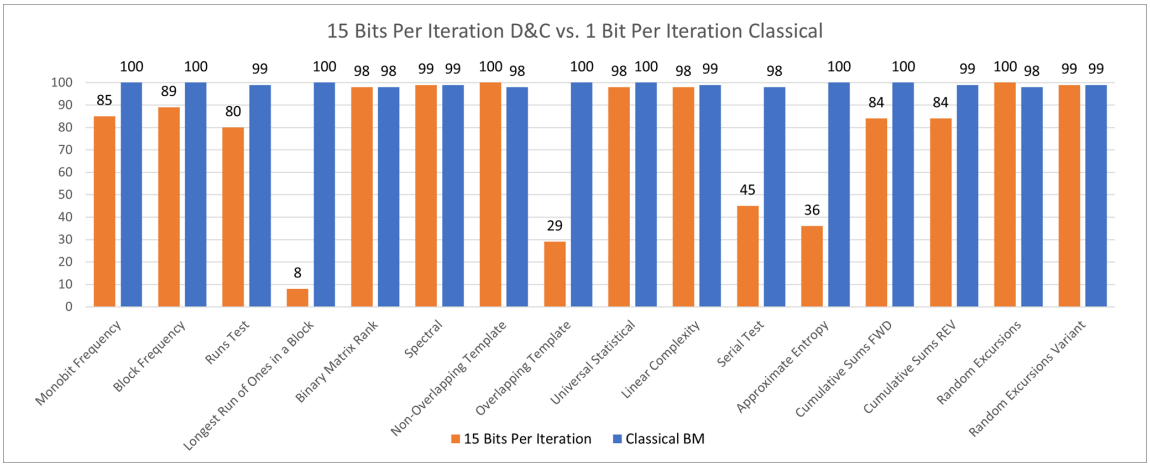

Figure 4.5. Performance of D&C PRNG w/ 15 Bits Per Iteration by Test

Looking at the Longest Run-Of-Ones in a Block test in isolation, we found that a sequence's individual test performance appears to mirror (and perhaps even drive) the sequence's overall performance across all implementations from 1-15 bits per iteration. While further research and investigation is needed in this area, it seems that Divide and Conquer Blum-Micali PRNG implementations that extract more than 11 bits per iteration generate sequences that have long runs of ones resulting from an iteration residue value that is close to p. Recalling the bit extraction method from Chapter 2, we know that if a residue was for example $x_i = p-1$ in a given iteration, when extracting 11 bits per iteration the output would be 11111111111. Perhaps the previous residue x_{i-1} resulted in the output 10001011111 and the next residue x_{i+1} resulted in the output 11111100100. This would result in the concatenated sequence

1000101111111111111111111111100100 which has a run of 22 consecutive ones in it. This is obviously a contrived example; however, the data seem to suggest that occurrences similar to this were not uncommon. Figure 4.6 shows the individual test performance of the Longest Run-Of-Ones in a Block test from 1-15 bits per iteration.

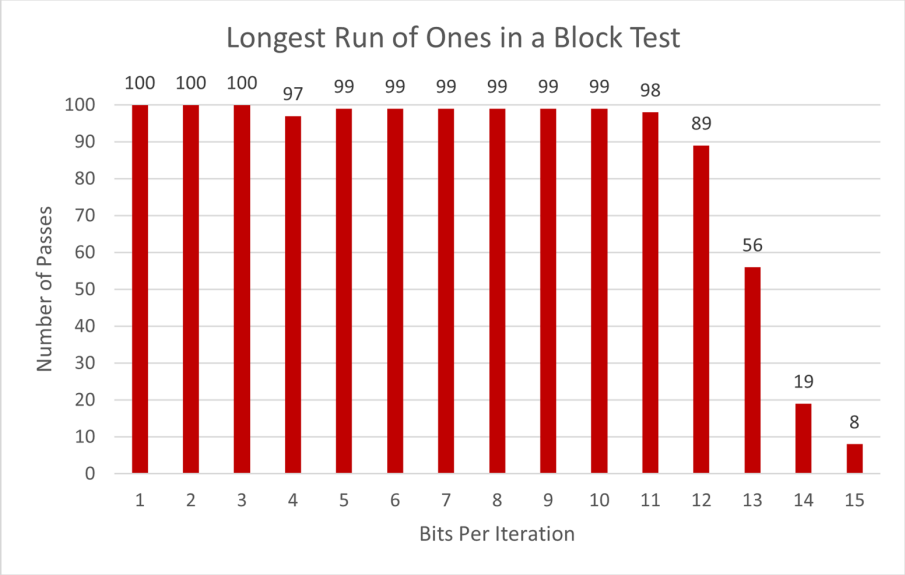

Figure 4.6. Longest Run-Of-Ones In a Block Test Performance for 1-15 Bits Per Iteration

When examining the worst performing case, that is the sequences generated by extracting 15 bits per iteration, we found that most sequences only failed a few NIST tests. In fact, 69 of the 100 sequences tested only failed four or fewer NIST tests. Notably none of the sequences tested failed more than nine NIST tests. Figure 4.7 shows the frequency of the number of NIST tests failed among the sequences generated by extracting 15 bits per iteration.

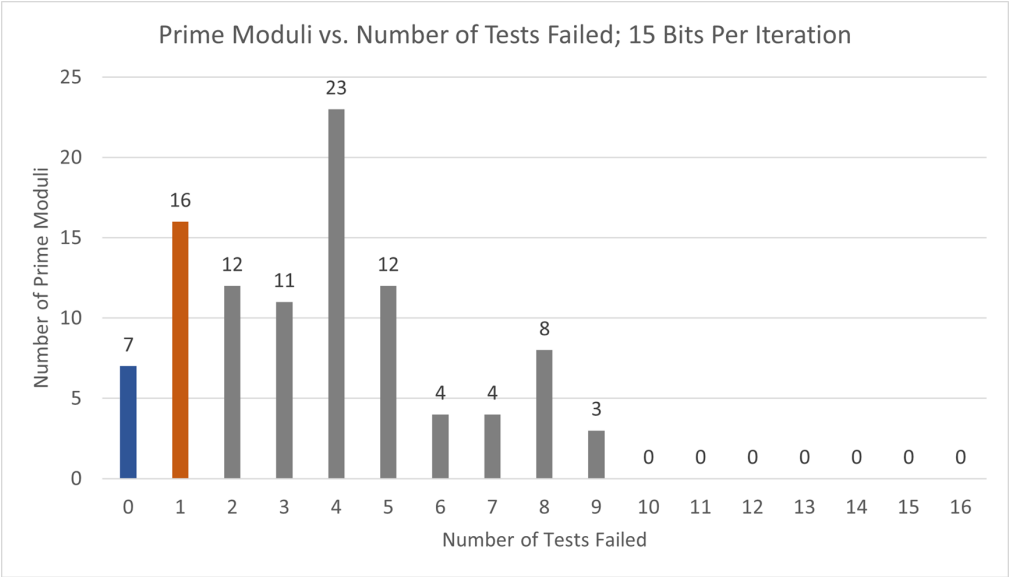

Figure 4.7. Frequency of Number of NIST Tests Failed When Extracting 15 Bits Per Iteration

In the second worst case of 14 bits per iteration, we found that the frequency of number of NIST tests failed was much more favorable. In this case, 98 of the 100 sequences failed four or fewer NIST tests. None of the sequences failed more than six NIST tests. This suggests that although the number of random/near-random sequences declined rapidly after 12 bits per iteration were extracted, there are still many sequences that *almost* met the criteria for randomness even with 14 bits per iteration. The appropriateness of using such sequences will depend on the specific application and there is indeed a trade-off between the number of bits extracted per iteration and the probability of statistical randomness. Figure 4.8 shows the frequency of the number of NIST tests failed among the sequences generated by extracting 14 bits per iteration.

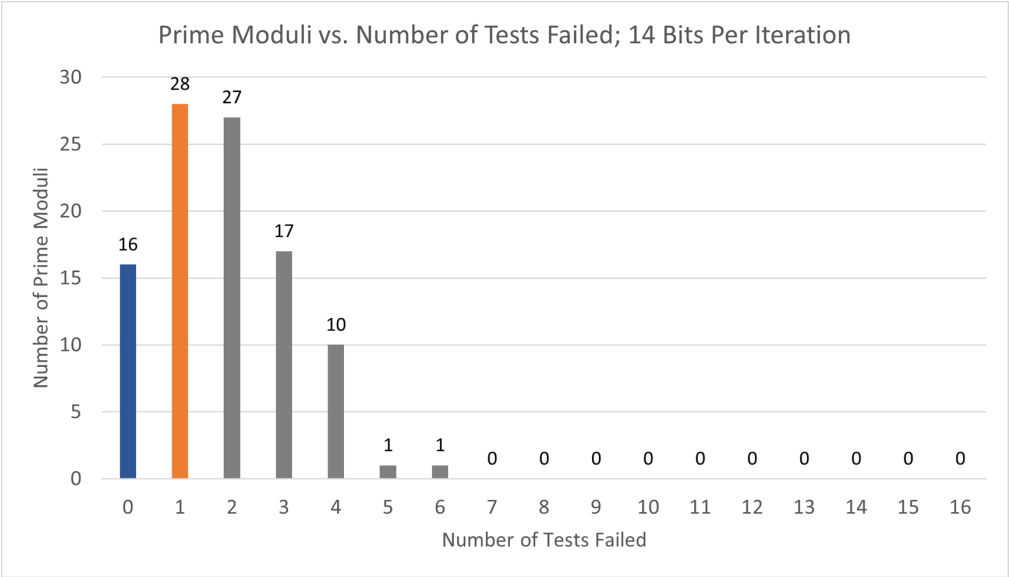

Figure 4.8. Frequency of Number of NIST Tests Failed When Extracting 14 Bits Per Iteration

4.2.3 Run Time Performance

The Divide and Conquer Blum-Micali PRNG is indeed faster computationally than the Classical Blum-Micali PRNG. There is a linear improvement in run time between the Classical Blum-Micali PRNG and the Divide and Conquer PRNG for d bits per iteration; that is, the Divide and Conquer version will run d times faster than the Classical version. We observed that in our Python implementations of the Blum-Micali PRNG, run times were not prohibitive to generate one million bit sequences for many applications. The average run time for a one million bit sequence was just over three seconds. The run time increases linearly with the length of the sequence, so a 10 million bit sequence will take approximately 30 seconds to generate using our implementation of the Classical Blum-Micali PRNG. Figure 4.9 displays the observed average computational run times collected during our analysis.

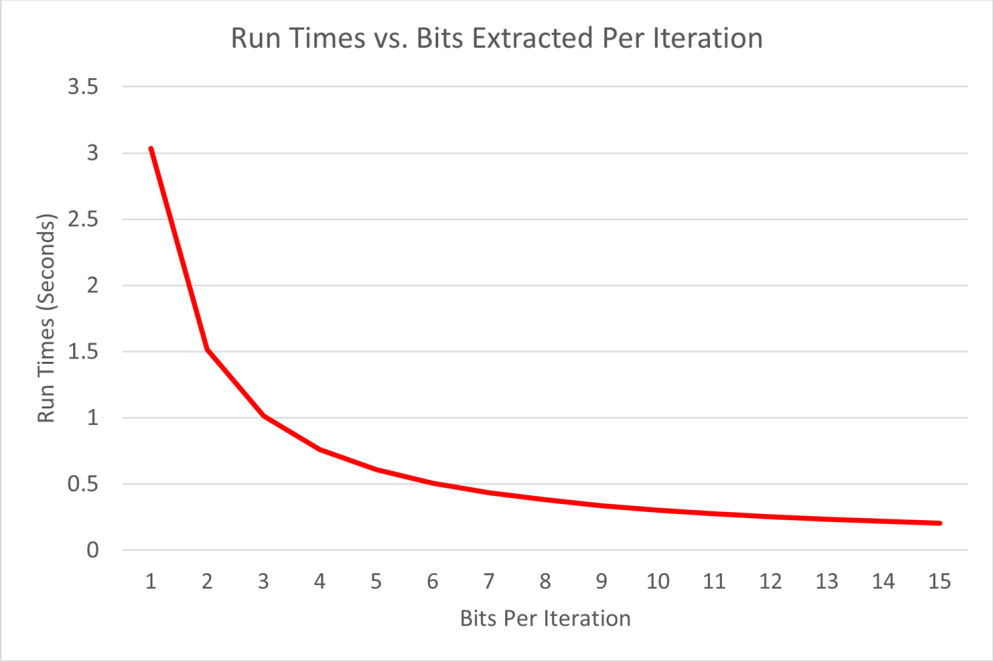

Figure 4.9. Average Run Times of Classical and D&C Blum-Micali PRNGs for a One Million Bit Sequence

Testing Equipment Used:
MSI GT70 (2013 model)
OS: Windows 10; Python 3.9 running on Spyder 5.5
Processor: Intel i7-3630QM @ 2.40GHz
RAM: 16GB DDR3
Graphics: Nvidia GeForce GTX675MX; 4GB DDR5

4.2.4 Linear Complexity Profiles

We conducted some preliminary analysis of the linear complexity profiles of sequences generated by the Divide and Conquer Blum-Micali PRNG. We found that all sequences that we tested appeared to have very favorable linear complexity profiles. Though we only tested a small (and not necessarily representative) subset of sequences, we found that the presence of universally favorable linear complexity profiles among those tested suggests

that most or all sequences generated using the Divide and Conquer Blum-Micali PRNG performed very well even when they failed multiple NIST tests. The case of the prime modulus $p = 10057699$ with primitive root $g = 2$ is illustrative in this regard. When using the Classical Blum-Micali PRNG, the sequence generated with this modulus and root passed all NIST tests and had the linear complexity profile shown in Figure 4.10. When using the Divide and Conquer Blum-Micali PRNG extracting 15 bits per iteration with this modulus and root, the sequence generated failed 9 of the NIST tests. This was the worst performing sequence among all sequences generated for the analysis in this thesis in terms of the number of NIST tests failed. However, even this sequence possessed a very favorable linear complexity profile as seen in Figure 4.11. It is worth noting that these linear complexity profiles are near-identical despite the fact that the sequence produced with the Divide and Conquer Blum-Micali PRNG with 15 bits per iteration failed a majority of the NIST tests.

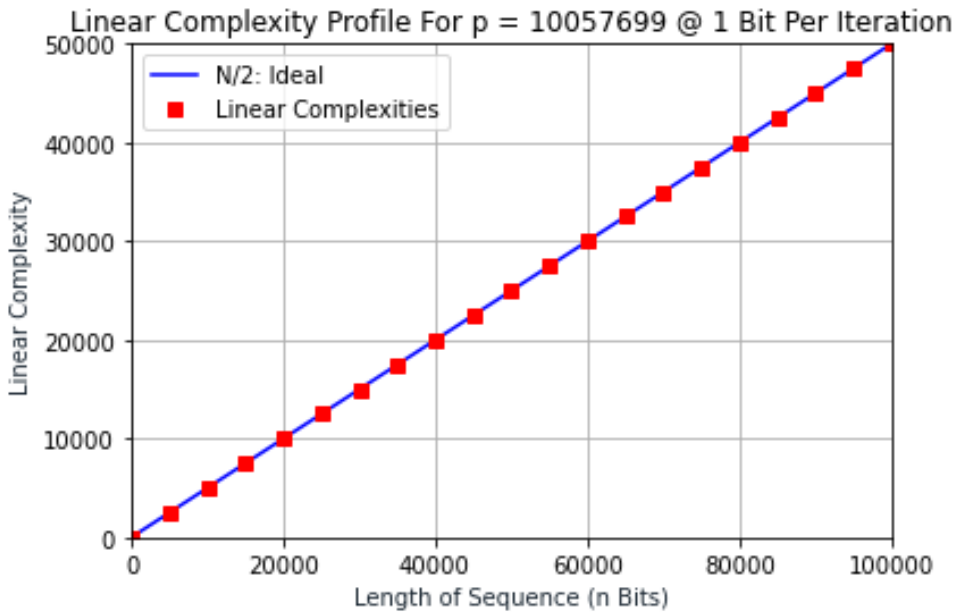

Figure 4.10. Linear Complexity Profile for $p = 10057699$ with 1 Bit Per Iteration

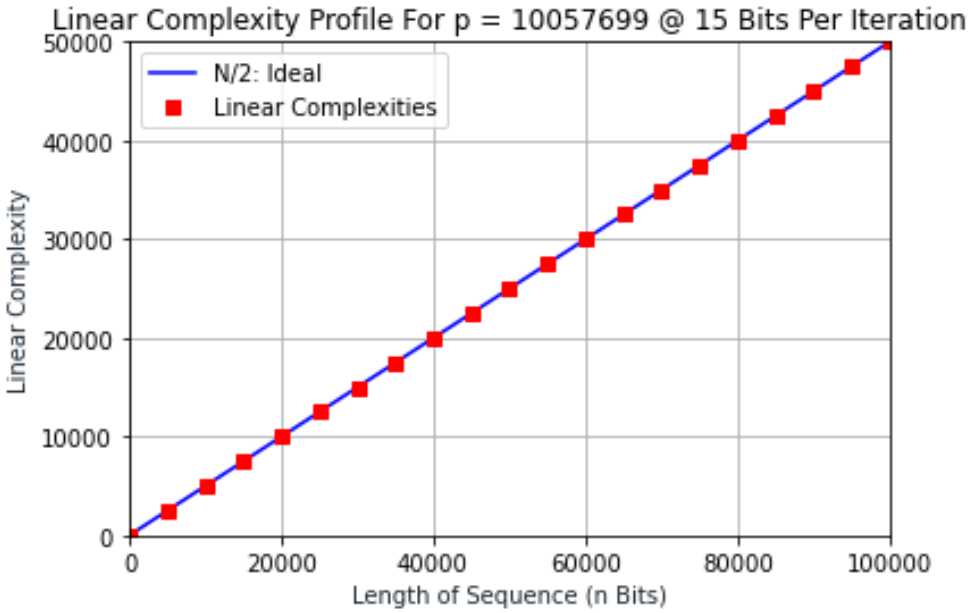

Figure 4.11. Linear Complexity Profile for $p = 10057699$ with 15 Bits Per Iteration

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusions and Future Work

5.1 Conclusions

This thesis compares the performance of the Classical Blum-Micali PRNG with a modification we call the Divide and Conquer Blum-Micali PRNG. When extracting 11 or fewer bits per iteration using the Divide and Conquer version, we found that overall performance was comparable to the classical version. Furthermore, in some cases where a short cycle occurs due to the choice of prime modulus, the Divide and Conquer version is preferable because it requires fewer iterations and therefore it is less probable that the output will be periodic for a given sequence length. When extracting 12 or more bits per iteration, the performance of the Divide and Conquer Blum-Micali PRNG rapidly declined. The NIST test most often failed by sequences generated under these conditions was the Longest Run-Of-Ones in a Block test.

The Divide and Conquer modification of the Blum-Micali PRNG provides a significant run time improvement when compared to implementations of the Classical Blum-Micali PRNG. This run time improvement increases in approximately linear proportion to the number of bits extracted per iteration. That is, a sequence generated using the Divide and Conquer modification with d bits per iteration will require approximately $1/d$ the number of operations as the Classical Blum-Micali PRNG. Our results strongly suggest that the choice of prime modulus is critical to the successful implementation of both the Classical and Divide and Conquer Blum-Micali PRNGs. Much care should be taken to ensure that the prime modulus chosen is a hard prime and does not produce short cycles when implemented. Without a formal decision rule, or look-up table of acceptable prime moduli, some trial and error is needed to select an appropriate modulus for implementation. Additionally, we found that the size of the prime modulus must be sufficiently large to ensure randomness; this requirement is determined by the length of the sequence desired. Among prime moduli specifically selected to meet these criteria, we found that still only roughly 45% of them will produce sequences of one million bits that are random or near-random when used as moduli for the Classical Blum-Micali PRNG.

5.2 Future Work

While our test data indicate that the performance of the Divide and Conquer Blum-Micali PRNG and the Classical Blum-Micali PRNG are comparable in terms of the randomness of sequences produced for 11 or fewer bits per iteration, many areas of future investigation and research remain.

There could be further modifications made to the Divide and Conquer decision rule construction to attempt to retain comparable performance while increasing the number of bits per iteration. One possible method we think will be effective in this pursuit is interlacing parity bits in the binary outputs during each iteration of the Divide and Conquer algorithm. We suspect that it may be possible to modify the decision rule of the Divide and Conquer algorithm (even without incorporating interlacing of parity bits) to avoid failing the Longest Run-Of-Ones in a Block test with high probability when extracting 12 or more bits per iteration.

The analysis done in this thesis is purely statistical in nature and there is no formal proof or theoretical investigation of the cryptographic security of the Divide and Conquer Blum-Micali PRNG. While it appears that the Divide and Conquer Blum-Micali PRNG performs comparably to the Classical version, certain applications may require stronger assurances than mere intuition. There is further work to be done linking the experimental results of this thesis to the $O(\log(\log n))$ bound proposed by Long and Wigderson regarding the number of bits extracted per iteration [11]. Using the smallest prime modulus 10033759 as an example, we see that $\log(\log(10033759)) \approx 4.54$. This suggests that, with a constant coefficient $c \geq 2.42$, our experimentally observed upper bound of 11 bits per iteration supports Long and Wigderson's claim [11].

There is much potential for future work regarding the selection of prime moduli for implementations of the Divide and Conquer Blum-Micali PRNG. Ideally, one would be able to formalize a decision rule or algorithm that enables the user to select an appropriate prime modulus that is, at minimum, guaranteed to produce random sequences of a given length using the Classical Blum-Micali PRNG. If devising a formal decision rule is not possible, generating a look-up table of prime moduli that are known to perform well when implemented in the Classical or Divide and Conquer Blum-Micali PRNGs would be of

great utility. While it would be acceptable for some applications to have a fixed modulus (and primitive root) and simply vary the random seed, other applications may require that a prime modulus be selected uniquely for each specific instance of the implementation.

In addition to the potential future work regarding prime moduli mentioned above, there is much room for investigation into the nature of the short cycles produced by some prime moduli. It would be of great interest to us to know what determines the occurrence and length of these cycles, and whether these properties can be formally stated.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: Python Scripts

A.1 Blum-Micali PRNG

This section shows the Python scripts used to implement both the Classical and Divide and Conquer Blum-Micali PRNGs. The main file is called "blumMicaliIterable.py" and is set up to implement the PRNG for multiple different prime moduli so as to expedite sequence generation for analysis. All user defined functions are also included in this section.

A.1.1 Main: "blumMicaliIterable.py"

```
import time
from modulo import reduce
from readPrimeList import readPrimeList
from defineBounds import generateBounds
from binaryOutputs import generateOutputs
from boundSearch import searchBounds


globalStartTime = time.time()

### How Long Do You Want Your Bit Streams to Be and How Many Bits Per Iteration
lengthOfStream = 1000000
maxBitsPerIteration = 15 ### Number of buckets = 2**bitsPerIteration
notificationRate = 100000


### Define Your Seed
seed = 12345 # The standard seed


### Generate List of Primes
listOfHardPrimes = readPrimeList("listOfHardPrimesAndRootsFinal.txt") ### Adjust the
    readPrimeList function file based on your prime list

### Initialize List of Outputs
runTimes = []
fileNamesList = []
fileNames2List = []

### THE MAIN LOOP!
for number in range(len(listOfHardPrimes)):
```

```
31
32      p = listOfHardPrimes[number][0]
33      g = listOfHardPrimes[number][1]
34      x0 = seed
35
36      listOfResidues = []
37
38      startTime = time.time()
39
40      for calculation in range(lengthOfStream):
41
42          x1 = reduce(g,x0,p)
43          listOfResidues.append(x1)
44          x0 = x1
45          if calculation%notificationRate == 0 and calculation !=0:
46              print(str(calculation) + " Base Iterations Complete")
47
48      endTime = time.time() - startTime
49      runTimes.append([p,endTime])
50
51      for bits in range(1,maxBitsPerIteration+1):
52
53          fileName = f"{bits:02d}" + "bits" + "P" + str(p) + "G" + str(g) + ".txt"
54          fileNamesList.append(fileName)
55          fileName2 = "Data" + f"{bits:02d}" + "bits" + "P" + str(p) + "G" + str(g) + ".txt"
56          fileNames2List.append(fileName2)
57
58          boundsList = generateBounds(bits, p)
59          outputs = generateOutputs(bits)
60
61          #Output File for Each Bit Stream
62          with open(fileName, 'w') as f:
63              print("Writing Bit Stream for Prime = " + str(p) + " and Root = " + str(g) + "
     with " + str(bits) + " bits per iteration")
64
65              numberOfIterations = int(lengthOfStream/bits)
66
67              for iteration in range(numberOfIterations):
68
69                  ### Search Algorithm to determine which bounds the outputs is between
70                  boundIndex = searchBounds(listOfResidues[iteration], boundsList)
71
72                  ### Write the string to a text file
73                  bitExtraction = str(outputs[boundIndex-1])
74                  f.write(bitExtraction)
75
76          # Info File for Each Bit Stream
77          with open(fileName2, 'w') as h:
78              h.write("--- Run Time: " + str(endTime/bits) + " seconds")
79              h.write("\n")
```

```
            h.write("--- Modulus: " + str(p))
            h.write("\n")
            h.write("--- Root: " + str(g))
            h.write("\n")
            h.write("--- Seed: " + str(seed))
            h.write("\n")
            h.write("--- Bits Per Iteration: " + str(bits))
            h.write("\n")
            h.write("--- Iterations: " + str(numberOfIterations))
            h.write("\n")
            h.write("--- Length of Stream: " + str(lengthOfStream) + " bits")


    # Completed Round
    print()
    print("ROUNDS COMPLETE: " + str(number+1))
    print("Rounds Remaining: " + str(len(listOfHardPrimes)-number-1))
    print()


with open("listOfFileNames.txt", 'w') as j:
    for name in range(len(fileNamesList)):
        j.write(str(fileNamesList[name]))
        j.write("\n")

with open("listOfRunTimes.txt",'w') as k:
    for runTime in range(len(runTimes)):
        k.write(str(runTimes[runTime]))
        k.write("\n")


print("Finally Complete")
globalEndTime = time.time() - globalStartTime
print("--- Run Time: " + str(globalEndTime) + " seconds")
```

A.1.2 "modulo.py"

```
def reduce(root,power,modulus):
    remainder = pow(root,power,modulus)
    return remainder
```

A.1.3 "readPrimeList.py"

```
def readPrimeList(listFileName):

    with open(listFileName) as f:
        lines = f.readlines()

```

```
    listOfHardPrimesAndRoots = []

    for line in range(len(lines)):
        tempList = []
        tempString = lines[line]
        prime = int(tempString[1:9])
        tempList.append(prime)
        root = int(tempString[11])
        tempList.append(root)
        listOfHardPrimesAndRoots.append(tempList)

    return listOfHardPrimesAndRoots
```

A.1.4 "defineBounds.py"

```
def generateBounds(numberOfBuckets,primeModulus):
    from math import floor
    boundsList = []
    for i in range(2**numberOfBuckets):
        bound = i*floor((primeModulus)/2**numberOfBuckets)
        boundsList.append(bound)
    return boundsList
```

A.1.5 "binaryOutputs.py"

```
def generateOutputs(exp):
    binOutput = []
    for j in range(2**exp):
        string = '0'+str(exp)+'b'
        bitStream = format(j,string)
        binOutput.append(bitStream)
    return binOutput
```

A.1.6 "boundSearch.py"

```
def searchBounds(xValue,bounds):

    previous = int(len(bounds))
    current = int(previous/2)
    update = 256 #dummy

    while update >= 1:
        if xValue <= bounds[current]:
            update = int(abs(previous-current)/2)
            previous = current
            current = current - update
        else:
```

```
            update = int(abs(previous-current)/2)
            previous = current
            current = current + update

    if xValue <= bounds[current]:
        current = current
    else:
        current = current + 1

    return current
```

A.2 NIST Test Suite

This section shows the Python scripts used to apply the NIST statistical test suite to the sequences generated using the "blumMicaliIterable.py" file. It is designed to work without any need for the user to move files to a different directory. This script will NOT work if the user changes the location of any of the files generated in the previous step. All user defined functions are included in this section with the exception of those sourced from Kho-Ang & Churchill [16].

A.2.1 Main: "testandWriteFile.py"

```
from iterableTest import nistTest
from readFileList import readFileList
import xlwt


fileNameList = readFileList("listOfFileNames.txt")
fileNameList.sort()
numOfFiles = len(fileNameList)

maxBitsPerIteration = 15
primesPerSheet = int(numOfFiles/maxBitsPerIteration)


resultList = []

for file in fileNameList:
    testResult = nistTest(file)
    resultList.append(testResult)
    print("NIST Tests Completed for P="  + str(file[7:15]) + " with " + str(file[0:2]) + "
     bits per iteration")


### Set up Headers
```

```
titles = ['PRIME NUMBER','PRIMITIVE ROOT','Bits Per Iteration','Overall Pass','Number
    Failed',\
          'Monobit', 'BlockFreq', 'RunsTest',\
          'LongestOneBlock', 'BinaryMatrixRank', 'Spectral', 'NonOverlapping',\
        'Overlapping', 'UniversalStatistical', 'LinearComplexity', 'SerialTest',\
        'ApproximateEntropy', 'CumulativeSumFWD', 'CumulativeSumREV',\
        'RandomExcursion', 'RandomVariant']

### Initialize Workbook
wb = xlwt.Workbook()
for i in range(1,maxBitsPerIteration+1):
    sheet = wb.add_sheet(str(i)+" Bits")

    # Add titles
    for j in range(len(titles)):
        sheet.write(0,j,titles[j])

    ## Input the Results
    for l in range(primesPerSheet): # Range = number of primes
        sheet.write(l+1,0,int(fileNameList[l][7:15]))    # Prime
        sheet.write(l+1,1,int(fileNameList[l][16]))   # Root
        sheet.write(l+1,2,int(fileNameList[l][0:2])) # Bits Pear Iteration
        sheet.write(l+1,3,int(resultList[l][1]))    # Overall Pass
        sheet.write(l+1,4,int(resultList[l][2]))    # Number Failed
        for k in range(len(resultList[l][0])):
            sheet.write(l+1,k+5,int(resultList[l][0][k])) # Individual Test Results

    for l in range(primesPerSheet): # Range = number of primes
        resultList.pop(0)
        fileNameList.pop(0)

### Save the Workbook
wb.save("Results.xls")
```

A.2.2 "iterableTest.py"

All of the user defined functions in this script, with the exception of "boolTrue.py", were sourced from Kho-Ang & Churchill [16].

```
from ApproximateEntropy import ApproximateEntropy as aet
from Complexity import ComplexityTest as ct
from CumulativeSum import CumulativeSums as cst
from FrequencyTest import FrequencyTest as ft
from Matrix import Matrix as mt
from RandomExcursions import RandomExcursions as ret
from RunTest import RunTest as rt
from Serial import Serial as serial
from Spectral import SpectralTest as st
```

```
from TemplateMatching import TemplateMatching as tm
from Universal import Universal as ut
from boolTrue import testPass


### Declaring Test Functions
testFunction = {
    0:ft.monobit_test,
    1:ft.block_frequency,
    2:rt.run_test,
    3:rt.longest_one_block_test,
    4:mt.binary_matrix_rank_text,
    5:st.spectral_test,
    6:tm.non_overlapping_test,
    7:tm.overlapping_patterns,
    8:ut.statistical_test,
    9:ct.linear_complexity_test,
    10:serial.serial_test,
    11:aet.approximate_entropy_test,
    12:cst.cumulative_sums_test,
    13:cst.cumulative_sums_test,
    14:ret.random_excursions_test,
    15:ret.variant_test}

monobitFreq = testFunction[0]
blockFreq = testFunction[1]
runTest = testFunction[2]
longestSingleBlock = testFunction[3]
binaryMatrixRank = testFunction[4]
spectral = testFunction[5]
nonOverlappingTemplate = testFunction[6]
overlappingTemplate = testFunction[7]
universal = testFunction[8]
linearComplexity = testFunction[9]
serial = testFunction[10]
approximateEntropy = testFunction[11]
cumulativeSumForward = testFunction[12] # Mode = 0
cumulativeSumReverse = testFunction[13] # Mode = 1
randomExcursion = testFunction[14]
randomVariant = testFunction[15]


def nistTest(fileName):

    ### Read In your Input
    with open(fileName) as f:
        contents = f.read()

    results = []

```

```
    test1 = monobitFreq(contents)
    results.append(test1[1])

    test2 = blockFreq(contents)
    results.append(test2[1])

    test3 = runTest(contents)
    results.append(test3[1])

    test4 = longestSingleBlock(contents)
    results.append(test4[1])

    test5 = binaryMatrixRank(contents)
    results.append(test5[1])

    test6 = spectral(contents)
    results.append(test6[1])

    test7 = nonOverlappingTemplate(contents)
    results.append(test7[1])

    test8 = overlappingTemplate(contents)
    results.append(test8[1])

    test9 = universal(contents)
    results.append(test9[1])

    test10 = linearComplexity(contents)
    results.append(test10[1])

    test11 = serial(contents)
    results11 = [test11[0][1],test11[1][1]]
    pass11 = testPass(results11)
    results.append(pass11)

    test12 = approximateEntropy(contents)
    results.append(test12[1])

    test13 = cumulativeSumForward(contents)
    results.append(test13[1])

    test14 = cumulativeSumReverse(contents, mode=1)
    results.append(test14[1])

    test15 = randomExcursion(contents)
    results15 = []
    for i in range(len(test15)):
        results15.append(test15[i][4])
    pass15 = testPass(results15)
    results.append(pass15)
```

```
110
111     test16 = randomVariant(contents)
112     results16 = []
113     for i in range(len(test16)):
114         results16.append(test16[i][4])
115     pass16 = testPass(results16)
116     results.append(pass16)
117
118
119     ### Prepare Output
120     output = []
121     overallPass = testPass(results)
122     numFailures = len(results)-sum(results)
123
124     output.append(results)
125     output.append(overallPass)
126     output.append(numFailures)
127
128     return output
```

A.2.3 "readFileList.py"

```
1  def readFileList(fileName):
2
3      with open(fileName) as f:
4          lines = f.readlines()
5
6      listOfFileNames = []
7
8      for line in lines:
9          tempString = line.replace("\n", "")
10         listOfFileNames.append(tempString)
11
12     return listOfFileNames
```

A.2.4 "boolTrue.py"

```
1  def testPass(testList):
2      if len(testList) == 0:
3          raise Exception("There is nothing in your test list")
4      for item in testList:
5          if item == False:
6              return False
7      return True
```

A.3 Hard Prime and Primitive Root Generator

This section shows the Python scripts used to generate the list of sufficiently large hard prime numbers and associated primitive roots for each. The list output by the main script here is a primary input for the "blumMicaliIterable.py" file.

A.3.1 Main: "generateHardPrimeList.py"

```
from generateLargePrime import generateLargePrime
from generateHardPrime import generateHardPrime
from checkPrimitiveRoot import checkPrimitiveRoot


### Generate List of Primes
setOfPrimes = set()
for prime in range(200):
    generator = generateLargePrime(17) ### Keep an eye on the size of your generators!
    setOfPrimes.add(generator)
print("done1")
listOfHardPrimes = []
for prime in setOfPrimes:
    listOfHardPrimes.append(generateHardPrime(prime, 12584660)) ### Hard primes will be at
     least 10,000,000
listOfHardPrimes.sort()
print("done2")
### Generate List of Potential Roots
listOfPossibleRoots = [2,3,5,7,11,13,17,19]


### Generate List of Primitive Roots
listOfRoots = []
tempG = 0

for number in range(len(listOfHardPrimes)):

    p = listOfHardPrimes[number]

    for index in range(len(listOfPossibleRoots)):
        g = listOfPossibleRoots[index]
        if checkPrimitiveRoot(g, p) == True:
            tempG = g
            listOfRoots.append(tempG)
            break
    if g >= 17:
        tempG = 17
        listOfRoots.append(tempG)
        print("done3")
### Thin Out Primes with Large Roots (>11)
```

```
for index in range(150):
    if listOfRoots[index] > 9:
        listOfRoots.pop(index)
        listOfHardPrimes.pop(index)


### Take Only the First Hundred Primes
listOfHardPrimes = listOfHardPrimes[0:110]
listOfRoots = listOfRoots[0:110]

### Generate Combined List
listOfPrimesAndRoots = []
for index in range(len(listOfHardPrimes)):
    tempList = []
    tempList.append(listOfHardPrimes[index])
    tempList.append(listOfRoots[index])
    listOfPrimesAndRoots.append(tempList)

print(listOfPrimesAndRoots)
```

A.3.2 "generateLargePrime.py"

This script was sourced from [17].

```
# Large Prime Generation
import random



def nBitRandom(n):
    return random.randrange(2**(n-1)+1, 2**n - 1)

def getLowLevelPrime(n):

    # Pre generated primes
    first_primes_list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
                         31, 37, 41, 43, 47, 53, 59, 61, 67,
                         71, 73, 79, 83, 89, 97, 101, 103,
                         107, 109, 113, 127, 131, 137, 139,
                         149, 151, 157, 163, 167, 173, 179,
                         181, 191, 193, 197, 199, 211, 223,
                         227, 229, 233, 239, 241, 251, 257,
                         263, 269, 271, 277, 281, 283, 293,
                         307, 311, 313, 317, 331, 337, 347, 349]
    '''Generate a prime candidate divisible
    by first primes'''
    while True:
        # Obtain a random number
        pc = nBitRandom(n)
```

```
26
27          # Test divisibility by pre-generated
28          # primes
29          for divisor in first_primes_list:
30              if pc % divisor == 0 and divisor**2 <= pc:
31                  break
32          else: return pc
33
34
35  def isMillerRabinPassed(mrc):
36      '''Run 20 iterations of Rabin Miller Primality test'''
37      maxDivisionsByTwo = 0
38      ec = mrc-1
39      while ec % 2 == 0:
40          ec >>= 1
41          maxDivisionsByTwo += 1
42      assert(2**maxDivisionsByTwo * ec == mrc-1)
43
44
45      def trialComposite(round_tester):
46          if pow(round_tester, ec, mrc) == 1:
47              return False
48          for i in range(maxDivisionsByTwo):
49              if pow(round_tester, 2**i * ec, mrc) == mrc-1:
50                  return False
51          return True
52
53      # Set number of trials here
54      numberOfRabinTrials = 20
55      for i in range(numberOfRabinTrials):
56          round_tester = random.randrange(2, mrc)
57          if trialComposite(round_tester):
58              return False
59      return True
60
61
62  def generateLargePrime(bits):
63
64      if 1 == 1:
65          while True:
66              n = bits
67              prime_candidate = getLowLevelPrime(n)
68              if not isMillerRabinPassed(prime_candidate):
69                  continue
70              else:
71                  break
72      return prime_candidate
```

A.3.3 "generateHardPrime.py"

```
import math

def isPrime(n):
  for i in range(2,int(math.sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True

def generateHardPrime(primeCandidate,minimum):
    multiplier = 2
    while multiplier <=1000:
        hardPrimeCandidate = (multiplier*primeCandidate) + 1
        if hardPrimeCandidate > minimum:
            if isPrime(hardPrimeCandidate) == True and hardPrimeCandidate%4==3:
                break
        multiplier = multiplier + 1
    return hardPrimeCandidate
```

A.3.4 "checkPrimitiveRoot.py"

```
def checkPrimitiveRoot(root,prime):
    exp = int((prime-1)/2)
    mod = pow(root,exp,prime)
    if mod == prime-1:
        return True
    else:
        return False
```

A.4 Linear Complexity Profiles

This section shows the Python scripts used to generate the linear complexity profile of a given sequence.

A.4.1 "linearComplexity.py"

```
import numpy
import copy
import time
import matplotlib.pyplot as plt
from berlekampMassey import berlekamp_massey_algorithm

upperLimit = 100000
interval = 5000
startTime = time.time()
fileName = '15bitsP10057699G2.txt'
```

```
with open(fileName) as f:
    contents = f.read()

xValues = []
yValues = []

for i in range(interval, upperLimit + interval, interval):
    string = contents[0:i]
    linComp = berlekamp_massey_algorithm(string)
    xValues.append(i)
    yValues.append(linComp)

w = numpy.linspace(0,upperLimit,10)
z = w/2
plt.plot(w, z, '--b', label='N/2')
plt.plot(xValues, yValues, '-r', label='Linear Complexities')

plt.axis([0, upperLimit, 0, upperLimit/2])
plt.title('Linear Complexity Profiles')
plt.xlabel('x', color='#1C2833')
plt.ylabel('y', color='#1C2833')
plt.legend(loc='upper left')
plt.grid()
plt.show()

endTime = startTime - time.time()
print(endTime)

with open('outputs.txt', 'w') as g:
    g.write(str(xValues))
    g.write('\n')
    g.write(str(yValues))
```

A.4.2 "berlekampMassey.py"

This script was sourced from Reid [18].

```
def berlekamp_massey_algorithm(block_data):

    n = len(block_data)
    c = numpy.zeros(n)
    b = numpy.zeros(n)
    c[0], b[0] = 1, 1
    l, m, i = 0, -1, 0
    int_data = [int(el) for el in block_data]
    while i < n:
        v = int_data[(i - l):i]
        v = v[::-1]
        cc = c[1:l + 1]
```

```
        d = (int_data[i] + numpy.dot(v, cc)) % 2
        if d == 1:
            temp = copy.copy(c)
            p = numpy.zeros(n)
            for j in range(0, l):
                if b[j] == 1:
                    p[j + i - m] = 1
            c = (c + p) % 2
            if l <= 0.5 * i:
                l = i + 1 - l
                m = i
                b = temp
        i += 1
    return l
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Lists of Prime Moduli and Primitive Roots

This section shows the lists of sufficiently large hard prime numbers and associated primitive roots used for the analysis presented in Chapter 4 of this thesis. Each pair is listed in the following format: (Hard Prime Modulus, Primitive Root).

B.1 First Results: Classical Blum-Micali PRNG

1. 10033759, 3
2. 10033831, 3
3. 10057699, 2
4. 10099147, 2
5. 10130839, 3
6. 10157767, 3
7. 10161911, 7
8. 10162571, 2
9. 10181947, 2
10. 10183463, 5
11. 10189651, 2
12. 10195051, 2
13. 10195543, 3
14. 10208851, 2
15. 10211087, 5
16. 10235047, 3
17. 10267051, 2
18. 10288727, 5
19. 10303247, 5
20. 10304579, 2
21. 10316743, 3
22. 10318423, 3
23. 10360447, 2
24. 10369627, 3
25. 10379119, 2
26. 10392983, 3
27. 10424851, 5
28. 10458419, 2
29. 10465379, 2
30. 10470791, 2
31. 10497511, 7
32. 10537679, 3
33. 10564187, 5
34. 10593923, 2
35. 10603823, 2
36. 10627039, 5
37. 10640947, 3
38. 10652063, 2
39. 10735903, 5
40. 10745363, 3
41. 10773731, 2
42. 10779827, 2
43. 10831207, 2
44. 10840147, 3
45. 10856071, 2
46. 10877627, 3
47. 10894843, 2
48. 10902107, 2
49. 10933831, 2
50. 10945871, 3
51. 11012087, 3
52. 11041951, 5
53. 11085983, 5
54. 11093171, 3
55. 11095739, 5
56. 11099771, 2
57. 11118911, 2
58. 11138299, 2
59. 11185919, 2
60. 11211071, 7
61. 11288623, 7
62. 11295631, 3
63. 11317259, 2
64. 11370839, 3
65. 11398939, 3
66. 11432591, 2
67. 11435027, 7
68. 11453467, 2
69. 11459447, 7

70. 11460871, 2
71. 11483191, 2
72. 11499071, 5
73. 11524231, 3
74. 11549467, 3
75. 11608907, 3
76. 11682943, 2
77. 11692091, 2
78. 11714707, 2
79. 11759131, 3
80. 11778607, 2
81. 11794723, 2
82. 11848787, 2
83. 11867027, 3
84. 11873483, 2
85. 11888719, 2
86. 11913179, 2
87. 11979167, 2
88. 12055811, 3
89. 12079999, 2
90. 12091103, 5
91. 12121547, 2
92. 12146527, 3
93. 12163031, 5
94. 12188107, 2
95. 12188863, 3
96. 12226351, 2
97. 12241679, 3
98. 12300203, 3
99. 12343631, 3
100. 12378859, 7
101. 12403939, 2
102. 12467551, 2
103. 12490507, 2
104. 12514043, 7
105. 12575527, 3
106. 12576943, 2
107. 12584659, 2
108. 12590111, 7
109. 12657859, 2
110. 12659819, 2
111. 12682987, 2
112. 12689099, 2
113. 12692027, 2
114. 12717259, 2
115. 12720427, 2
116. 12728503, 3
117. 12740579, 2
118. 12757439, 7
119. 12774787, 2
120. 12776207, 5
121. 12793663, 3
122. 12827239, 3
123. 12850051, 2
124. 12862651, 2
125. 12877951, 3
126. 12900227, 2
127. 12907571, 2
128. 12910087, 3
129. 12910279, 3
130. 12923243, 2
131. 12934619, 2
132. 12955759, 3
133. 12965027, 2
134. 13009027, 2
135. 13016299, 2
136. 13026707, 2
137. 13056011, 2
138. 13065883, 2
139. 13076603, 2
140. 13081339, 2
141. 13107803, 2
142. 13156063, 3
143. 13165639, 3
144. 13170343, 3
145. 13223087, 5
146. 13226123, 2
147. 13231651, 2
148. 13264963, 2
149. 13273739, 2
150. 13303811, 2
151. 13313647, 3
152. 13314163, 2
153. 13322531, 2
154. 13325387, 2
155. 13337603, 2
156. 13356103, 3
157. 13366963, 2
158. 13409303, 5
159. 13411303, 3
160. 13450219, 2
161. 13489699, 2
162. 13490227, 2
163. 13496383, 3
164. 13525907, 2
165. 13532551, 3
166. 13542743, 5
167. 13543963, 2
168. 13549519, 3

169. 13570103, 5
170. 13572971, 2
171. 13593007, 3
172. 13610899, 2
173. 13625419, 2
174. 13656859, 2
175. 13673783, 5
176. 13761203, 2
177. 13791263, 5
178. 13815691, 2
179. 13828211, 2
180. 13828351, 3
181. 13914151, 3
182. 13972811, 2
183. 13979027, 2
184. 13988551, 3
185. 13991479, 3
186. 13993787, 2
187. 14009731, 2
188. 14029723, 2
189. 14051879, 7
190. 14080279, 3
191. 14142439, 7
192. 14143639, 3
193. 14181359, 3
194. 14234063, 2
195. 14251547, 5
196. 14298283, 2
197. 14330747, 2
198. 14347279, 2
199. 14358899, 3
200. 14390339, 2
201. 14391011, 2
202. 14393299, 2
203. 14400839, 2
204. 14443343, 5
205. 14487311, 5
206. 14649347, 2
207. 14732743, 2
208. 14837687, 2
209. 14908319, 3
210. 15051451, 5

B.2 Second Results: Divide and Conquer Blum-Micali PRNG

1. 10033759, 3
2. 10033831, 3
3. 10057699, 2
4. 10157767, 3
5. 10161911, 7
6. 10162571, 2
7. 10181947, 2
8. 10183463, 5
9. 10189651, 2
10. 10195051, 2
11. 10208851, 2
12. 10211087, 5
13. 10304579, 2
14. 10316743, 3
15. 10318423, 3
16. 10458419, 2
17. 10564187, 5
18. 10593923, 2
19. 10773731, 2
20. 10779827, 2
21. 10902107, 2
22. 11085983, 5
23. 11211071, 7
24. 11295631, 3
25. 11317259, 2
26. 11435027, 7
27. 11524231, 3
28. 11759131, 3
29. 11848787, 2
30. 11913179, 2
31. 12091103, 5
32. 12121547, 2
33. 12146527, 3
34. 12188107, 2
35. 12188863, 3
36. 12403939, 2
37. 12575527, 3
38. 12590111, 7
39. 12657859, 2
40. 12659819, 2
41. 12682987, 2
42. 12689099, 2

43. 12692027, 2
44. 12720427, 2
45. 12740579, 2
46. 12757439, 7
47. 12776207, 5
48. 12793663, 3
49. 12862651, 2
50. 12900227, 2
51. 12907571, 2
52. 12910087, 3
53. 12910279, 3
54. 12923243, 2
55. 12934619, 2
56. 12955759, 3
57. 12965027, 2
58. 13009027, 2
59. 13016299, 2
60. 13026707, 2
61. 13076603, 2
62. 13107803, 2
63. 13156063, 3
64. 13170343, 3
65. 13223087, 5
66. 13231651, 2
67. 13264963, 2
68. 13273739, 2
69. 13303811, 2
70. 13325387, 2
71. 13337603, 2
72. 13356103, 3
73. 13409303, 5
74. 13411303, 3
75. 13450219, 2
76. 13489699, 2
77. 13525907, 2
78. 13542743, 5
79. 13543963, 2
80. 13570103, 5
81. 13572971, 2
82. 13593007, 3
83. 13625419, 2
84. 13791263, 5
85. 13828211, 2
86. 13828351, 3
87. 13914151, 3
88. 13972811, 2
89. 13979027, 2
90. 13988551, 3
91. 13993787, 2
92. 14009731, 2
93. 14051879, 7
94. 14251547, 5
95. 14298283, 2
96. 14330747, 2
97. 14390339, 2
98. 14391011, 2
99. 14443343, 5
100. 14649347, 2

List of References

[1] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudorandom bits," *SIAM Journal on Computing*, vol. 13, no. 4, pp. 850–864, 1984.

[2] P. Stanica, "Pseudorandom bit/number generators part 1," unpublished.

[3] A. C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, 1982, pp. 160–164.

[4] "Discrete logarithm," *Wikipedia*. Accessed October 12, 2022 [Online]. Available: https://en.wikipedia.org/wiki/Discrete_logarithm

[5] "Pohlig-hellman algorithm," *Wikipedia*. Accessed November 16, 2022 [Online]. Available: https://en.wikipedia.org/wiki/Pohlig-Hellman_algorithm

[6] W. Trappe and L. Washington, *Introduction to Cryptography with Coding Theory*, 2nd ed. Saddle River, NJ, USA: Pearson, 2006.

[7] J. Fraleigh, *A First Course in Abstract Algebra*, 7th ed. Saddle River, NJ, USA: Pearson, 2003.

[8] B. Lynn, "Number theory - generators," Stanford University, Accessed Dec. 9, 2022 [Online]. Available: https://crypto.stanford.edu/pbc/notes/numbertheory/gen.html

[9] D. Shanks, *Solved and Unsolved Problems in Number Theory*, 2nd ed. New York, NY, USA: Chelsea Publishing Company, 1978.

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

[11] D. Long and A. Wigderson, "The discrete logarithm hides o(log n) bits," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 363–372, 1988.

[12] "Berlekamp-massey algorithm," *Wikipedia*. Accessed December 6, 2022 [Online]. Available: https://en.wikipedia.org/wiki/Berlekamp-Massey_algorithm

[13] P. Stanica, "Pseudorandom bit/number generators part 2," unpublished.

[14] K. Rosen, *Discrete Mathematics and Its Applications*, 8th ed. New York, NY, USA: McGraw-Hill, 2018.

[15] L. Bassham et al., “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” National Institute of Standards and Technology, December 2010 [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final

[16] S. Kho-Ang and S. Churchill, “Python nist randomness test suite,” GitHub, Accessed Nov. 15, 2022 [Online]. Available: https://github.com/stevenang/randomness_testsuite

[17] Tabmir, “How to generate large prime numbers for rsa algorithm,” GeeksforGeeks, Accessed Nov. 12, 2022 [Online]. Available: https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm

[18] S. Gordon-Reid, “Python implementation of the berlekamp-massey algorithm,” GitHub, Accessed Dec. 3, 2022 [Online]. Available: https://gist.github.com/StuartGordonReid/a514ed478d42eca49568

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California