NEW TYPES OF PSEUDORANDOM NUMBER GENERATORS

By

Qi Jiang

Bachelor of Science
Fudan University
Shanghai, China
1996

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1999

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements of
the Degree of
MASTER OF SCIENCE
December, 1999

NEW TYPES OF PSEUDORANDOM

NUMBER GENERATORS

Thesis Approved:

J Chandler

Thesis Adviser

[illegible]

Blayne E. Mayfield

Wayne B. Powell

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my thesis advisor, Dr. J. P. Chandler for his guidance, encouragement and inspiration. I also appreciate my other committee members Dr. B. E. Mayfield and Dr. H. K. Dai, whose encouragement and assistance were invaluable. I would like to thank the Department of Computer Science for providing me with this learning opportunity.

I would like to give my special appreciation to my parents and my brother, who always support me in any circumstance. Had been without you, I cannot finish this tenuous task by myself.

TABLE OF CONTENT

LIST OF TABLES

CHAPTER ONE

Introduction

When making a difficult decision or just trying to win the state lottery, lots of people sometimes get their lucky numbers by flipping a coin, by drawing a card from the deck, or simply from fortune cookies. Those are some realistic examples of using random numbers. Scientifically, random numbers are widely used in cryptography, stochastic simulation, and many other fields. Random numbers are also a vital utility for today's recreational entertainment business. As an important part of computational science, random numbers are used to test the effectiveness of computer algorithms. Random numbers are also used in the operation of randomized algorithms.

Usage of random numbers can be traced to before the birth of modern electronic computers. In ancient times, the best source of a random number was by rolling dice, or by drawing balls out of a "well-stirred urn", as is done today in the lottery [1]. In 1927, a table of over 40,000 random digits was published by L. H. C. Tippett. Upon the heated discussion stirred by the publication of this random digit table, M. G. Kendall and B. Babington-Smith introduced the first random number generation machine in 1939 [2,3].

This machine was capable of generating 100,000 random digits using several rotating discs. Other machines capable of generating random numbers were introduced later through the 1950s. Among those machines, the Ferranti Mark I first contained a hardware random number generator. Another famous machine called ERNIE has been used for many years picking the winning number in the British Premium Saving Bonds lottery [4]. ERNIE contains a number of random digit generators, which relied for their randomness on electronic noise in neon tubes. It was able to generate more than 10^8 random numbers and the distribution of those numbers has been proven to be close to perfect randomness.

In the 1990s, the once nearly extinct random digits table has made strong a comeback supported by new technologies. The biggest problem for random digit tables was the size limitation of the table. To supply a useful number of random numbers for meaningful simulations on printed media is next to impossible. With the maturation of CD-ROM technology, George Marsaglia helped the rebirth of a random digit table, which occupied the whole capacity of a single CD-ROM --- 650 megabytes [7]. This table was generated by a noise-diode circuit.

After the invention of electronic computers, the awesome computational power of computers led people to search for effective algorithms to produce random numbers using computers. Since then, many algorithms have been introduced by different mathematicians. The first such algorithm, which was suggested by John von Neumann, was later called the "middle-square" method [1]. The most popular algorithm nowadays is the linear congruential method [1]. This method can be generalized to the quadratic congruential method [1]. Other methods are also available. The inversive congruential method [14], the lagged Fibonacci method [1], and combinations of two or more pseudorandom number generators [1] are examples of the variety of available random number generators.

All of the software solutions listed above use a deterministic algorithm to generate the next number depending on one or more numbers existing earlier in the series, and thus are not truly random. The word "pseudorandom" is used to describe those generated sequences. Because of the deterministic nature of the algorithms used to generate pseudorandom numbers, the pseudorandom sequence will show some internal correlation. Those internal correlations may or may not show their effect on a particular application depending on the application itself. It is not possible to find a perfect algorithm whose sequence fits the need for all the applications. Rather, it is very important to analyze a pseudorandom number generator before applying its sequence to one specific application in order to avoid the misuse of this sequence. Using a sequence whose internal correlation showed up in an application can yield very misleading results.

To use a software pseudorandom number generator correctly and effectively, it is important to know the advantages and disadvantages of the different methods available. It is also important to know the theoretical and empirical support for various pseudorandom number generators. Finally, it is important to know how to test a pseudorandom sequence and the resources available for testing such a sequence.

Cryptography is a very important field for both security and privacy [20]. The idea of cryptography has existed for hundreds or even thousands of years, but only during the last 20 years has public research on cryptography exploded [20]. Random numbers have been used widely in encoding and decoding messages. Real random numbers are used mostly in one-time pads, which provide a truly unbreakable encryption [20]. However, to use real random numbers, both the encoder and the decoder must have access to the same random sequence. This reason made the usage of real random numbers rare, although they are used for extremely important diplomatic messages. Pseudorandom numbers, on the other hand, can be generated by a certain algorithm and are easy to reproduce. However, some certain patterns and structures exist in pseudorandom sequence which could also help the attacker to decrypt the cipher-text. Thus, the choice of pseudorandom number generator becomes very important for encryption usage.

As we know, there is no unbreakable encryption algorithm other than one-time pads, especially when a brute-force attack is employed. The problem relies on how many resources would be involved in an attack. If the resources required for an attack outweigh the value of the message, then the encryption algorithm is probably safe to use. Resources involved in an attack depend on the data complexity, the processing complexity and the storage requirement of the attack. In a good encryption algorithm, those three factors should be maximized and be made practically impossible for an attacker. It is always assumed that the attacker knows the encryption system being used; only the "key" used to encrypt a given message is unknown [20].

In this paper, a new pseudorandom number generator using a modified shuffling method is proposed. The modification made to the shuffling method is to improve the unpredictability of this new pseudorandom number generator.

CHAPTER TWO

Literature Review

Overview of PseudoRandom Number Generators

Software pseudorandom number generators (PRNGs) use deterministic algorithms to generate pseudorandom sequences. One might ask what is the real meaning for the word "random" here. In fact, the random sequence we are talking about is "a sequence of independent random numbers with a specified distribution" [1]. Every number in the sequence is taken by chance, has no relationship with other numbers, and has a specified probability to appear anywhere in the sequence. In a uniform distribution over a finite range, each possible number has equal probability to appear on any position of the sequence. Other distributions of random sequences are usually transformed from a uniform distribution sequence by applying certain restrictions.

A sequence generated by software PRNGs is called a pseudorandom sequence. The term itself clearly states that such a sequence is not truly random, rather, it just appears to be random. There is a theoretical definition of randomness in statistics [1]. It is probably enough to say here that a pseudorandom number sequence is random enough if this sequence is able to pass a number of statistical tests.

Statistical and empirical tests

1. χ^2 test

The "chi-square" test (χ^2 test) [1] is a well known statistical test and serves as a foundation of lots of other tests. The basic idea behind the χ^2 test is that the actual distribution should not be far away from the expected distribution. For example, in a uniform random distribution of 10,000 numbers in the range of 1 to 1,000 inclusive, it is easy to know that each number is expected to appear 10 times. The number of 10 is obtained by multiply the probability of one specific number appeared in one position, denoted by p, by the total number, denoted by n. Here, p is 1/1,000, and n is 10,000. The actual times a certain number occurred in the sequence, denoted by Y, should not be too far away from the expected number, np. Although for a single number, there is a possibility that the deviation is significant, the summary of the square deviations $\sum(Y-np)^2$ should be small for a good distribution. Since the deviation could be either positive or negative, the simple summary will offset a positive deviation with a negative one. The square summary is thus a better description than the simple summary. If the distribution is not uniform, in order to eliminate the weight factor, we should divide each term in the summary by *np*.

In general, suppose that the observation could fall into k categories and we take n independent observations, we can obtain

$$V = \sum_{s=1}^{k} \frac{(Y_s - np_s)^2}{np_s},$$

where p_s is the probability that each observation falls into category *s*.

The calculated *V* value can be compared to a χ^2 distribution table and the probability of yielding a particular sequence could be obtained. If the probability is smaller than 1% or larger than 99%, the sequence deviates too much or is too close to ideal distribution, which means the sequence itself is not random. On the other hand, if the percentage is between 10% to 90%, we are able to say that the sequence passes the χ^2 test. Remarkably, the table entry has nothing to do with *n*, the total number of items in the sequence. The only attribute that affects the table entry is the number of degrees of freedom, which is equal to the number of categories minus one.

Suppose we have a sequence of numbers that has blocks of non-random sequences. Since those blocks might only count as small portions of the whole sequence, the entire sequence could still be able to pass the χ^2 test. In order to keep those bad sequences from passing the χ^2 test, a series of tests with different values of *n* should be performed. We should also do several duplicates of tests with the same value of *n*.

2. The Kolmogorov-Smirnov test

The Kolmogorov-Smirnov test (KS test) [1] is used when the observations distributed over the range could be infinitely many values. The χ^2 test requires the observation values fall into a finite number of categories or "bins".

Suppose we have a random number *X*; the probability that *X* is smaller than *x* can be defined as the distribution function *F(x)*. If we have a sequence of numbers of length *n*, the observation value $X_1, X_2, \ldots, X_n$ can be formed into the empirical distribution function $F_n(x)$, where

$$F_n(x) = \frac{number\_of\_X \leq x}{n}.$$

The difference between *F(x)* and $F_n(x)$ is the basis of the KS test. A good sequence of random number will have a $F_n(x)$ that approximates *F(x)*, while a bad sequence could have significant deviations.

We calculate K_n^+ and K_n^- for the sequence by applying the following equation:

$$K_n^+ = \sqrt{n} \max_{-\infty < x < +\infty} \left(F_n(x) - F(x)\right);$$

$$K_n^- = \sqrt{n} \max_{-\infty < x < +\infty} \left(F(x) - F_n(x)\right).$$

As in the χ^2 test, we then look up in a percentage table to determine if the value of K_n^+ and K_n^- are reasonable.

The problem of the choice of value of n also exists here. On one hand, we need n to be large enough to determine the true distribution of the sequence; on the other hand, a large n tends to average out locally nonrandom distributions. A good compromise is to choose a moderate n, while calculating a large number of K_n^+ over different parts of the sequence. This compromise will tend to detect both locally and globally nonrandom distributions [1].

3. The spectral test

The spectral test [1] is a very important test for any congruential pseudorandom number generator (see pg. 14) in the sense that all good generators have passed this test, while all bad generators have failed.

Suppose we have a sequence $X_1, X_2, \ldots, X_n$ of period m; the spectral test will test the distribution of all m points

$$\left\{ \frac{1}{m}\left(x, s(x), s(s(x)), \ldots, s^{[t-1]}(x)\right) \mid 0 \le x < m \right\}$$

in t-dimensional space. Here, $s(x)$ stands for the successor of x.

We name the maximum distance between two numbers $1/\nu$. Similarly, the maximum distance between points $\left\{\left(x/m, s(x)/m \right)\right\}$ could be named as $1/\nu_2$ in a 2-dimensional test.

In general, $1/\nu_t$ is the maximum distance between the hyperplanes covering all points $\left\{\left(x/m, s(x)/m, \ldots, s^{[t-1]}(x)/m\right)\right\}$ in t-dimensions.

Suppose we have a truly random sequence between 0 and 1. If we round or truncate the numbers to a finite accuracy and put this sequence into a one-dimensional spectral test, the distribution will be regular. A pseudorandom sequence with a period of m will also display regularity in the test. The difference between a pseudorandom sequence and a truly random sequence is that the accuracy of the truly random sequence will remain the same in all dimensions, while that of the pseudorandom sequence will decrease as t increases. For most applications, it is enough to test a sequence for $2 \le t \le 6$, and $\nu^t \ge 2^{30/t}$ appears to be adequate for passing the spectral test [1].

4. Other empirical tests

There are a number of other empirical tests available to test randomness of a sequence.

The frequency test (equidistribution test) [1] applies either the KS test or χ^2 test to the sequence. Since the uniform distribution is necessary for pseudorandom number sequences, this test is always needed.

The serial test [1] inspects the distribution of pairs of successive numbers. To perform this test, we first divide the range into d groups. We then throw pairs of successive numbers into d^2 categories and perform a χ^2 test on those d^2 categories.

The poker test [1] considers groups of five successive numbers. We set up 5 categories from the card games called poker: all different, one pair, two pairs or three of a kind, full house or four of a kind, and five of a kind. Then we classify different groups into different categories and perform a χ^2 test on them.

The permutation test [1] divides the pseudorandom sequence into n groups. Each group has t elements. The categories in this test will be all the possible combination of t elements. We then use a χ^2 test to determine the randomness of the sequence.

In the runs test [1], we examine the monotone parts of the input sequence. There are at least two ways to make the test. The simpler one needs to throw away the number immediately after a run and a χ^2 test is engaged thereafter.

The collision test [1] is designed for the situation in which the numbers of categories far outweigh the numbers of observations. This test counts the number of collisions that happened in all categories. To pass this test, the number of collisions should be neither too large nor too small.

The birthday spacing test [1] is important because the lagged Fibonacci generators constantly fail it although they behave well on other tests [5]. It choose m birthdays in a "year" of n days and lists the spacing between the birthdays. The spacings should be asymptotically Poisson distributed.

The monkey tests were introduced by George Marsaglia [6,7]. We suppose that the PRNG is a monkey sitting in front of a typewriter and pressing the keys randomly. We then count in successive n strokes how many words of length d were missing. Monkey tests include OPSO (overlapping-pairs-sparse-occupancy), OQSO (overlapping-quadruples-sparse-occupancy), DNA test, and count the 1's test. Those tests use overlapping d-tuples as the word, and thus are not subject to the χ^2 test. Some feedback shift register generators failed these tests miserably [6].

Of all the tests above, the runs test, the collision test and the monkey tests may be the most valuable ones. This is because they are either very strong or specially designed to detect deficiencies hard to detect otherwise.

There is also a new set of tests developed by Vattulainen around 1995 [8,9]. Those tests are specially designed for testing random numbers for stochastic simulation purposes. The cluster test and the autocorrelation test are based on the known properties of the two-dimensional Ising model. The random walk test and the n-block test are based on random walks on lattices. The first two tests are designed to test the randomness at the bit level. The ith bit of every successive number was extracted and put into a two-dimensional square lattice. By counting 1 as "up" and 0 as "down" as in the Ising model, the result of a random sequence can be compared to the expected values. More, we can use the expected distribution instead of the expected average value to achieve a more meaningful result. The random walk test divides a two-dimensional lattice into four squares and puts the finish point of a walk of a random number sequence of arbitrary length into the four categories. A χ^2 test of three degrees of freedom is performed thereafter. The n-block test is essentially a random walk test on a one-dimensional lattice.

Pseudorandom number generators

1. Linear congruential generators

Linear congruential generators are probably the most widely used PRNGs today. They were first introduced by D. H. Lehmer in 1949 [10]. The sequence can be obtained using

$$X_{n+1} = (aX_n + b) \bmod m, \qquad n \geq 0.$$

Here, m is the modulus, which should be greater than 0; a is the multiplier, which should between 1 and m; b is the increment, which should be between 0 and m; and X_0 is the starting value, which should also be between 0 and m-1. We denote this PRNG as LCG (m, a, b, X_0). The originally proposed sequence [10] was

$$X_n = X_0 \times 23^n \bmod (10^8 + 1).$$

Lehmer also proposed to use for m the Mersenne Prime, 2^{31}-1, in a binary machine [10].

There is a whole family of LCGs being used. To name a few, there are the ANSI-C system generator LCG(2^{31}, 1103515245, 12345, 12345), the SIMSCRIPT generator LCG(2^{31}-1, 630360016, 0), and Maple's LCG(10^{12}-11, 427419669081, 0, 1) [11].

The behavior of a LCG depends on the choice of its parameters. The modulus m should be large enough to allow a useful period, because at most m values can be generated before the sequence repeats itself. It is common to choose m as the word size of the computer. This, however, will lead to an unfortunate result: the lower digits of X_n

will be much less random than the higher digits. To avoid this situation, m should be the word size plus 1 or the word size minus 1. Better, the m should be the largest prime number less than the word size [1].

The choice of multiplier a affects the period of the sequence. If the modulus m is the product of distinct primes, the only choice for a to achieve a period of m is 1, which is undesirable. However, when m is divisible by a high power of some prime, the period of m-1 could be achieved when a is chosen as follows [1]:

a) The increment c is relatively prime to modulus m;

b) let $b = a - 1$; then b must be a multiple of p, for each prime p dividing m;

c) b is a multiple of 4, if m is a multiple of 4.

Despite the fact that LCGs are widely used, they suffer from the fact that all of them form a lattice structure in a spectral test of d-dimensions [11]. In some extreme cases, all points of 3-tuples in a 3-dimensional test can be included on merely 15 parallel planes [12].

A number of PRNGs have been developed by combining two or more LCGs in various ways. However, some combination generators have been proved to inherit the lattice structure [13]. Those combination generators should also be tested for lattice structure before application.

2. Lagged Fibonacci generators

We can denote the lagged Fibonacci generator [1] in a general form as $F(r, s, m, \cdot)$, where the X_n can be obtained by

$$X_n = (X_{n-r} \cdot X_{n-s}) \bmod m,$$

and r and s are called the lags. The symbol "$\cdot$" represents a binary operation. It can be either plus, minus, multiply, or bitwise exclusive OR (XOR). Lagged Fibonacci generators were first introduced in order to extend the period of a linear congruential method and have been used successfully in a lot of situations. Lagged Fibonacci generators are also faster than LCGs if the binary operation is plus or minus. In the 1990s, people discovered that lagged Fibonacci generators consistently fail the birthday spacing test unless r is more than 500 [5].

3. Inversive congruential generators

Inversive congruential generators (ICGs) [14] can be described in the following form:

$$X_{n+1} = a\overline{X}_n + c(\bmod p).$$

The modulus p is usually a prime. Let $\overline{X}_n = X_n^{-1}$ if $X_n \neq 0$, $\overline{X}_n = 0$ if $X_n = 0$. In other words, $\overline{X}_n$ equals the number X_n^{p-2} modulo p [14]. One significant aspect of ICGs is that they lack the lattice structure of d-tuples of consecutive random numbers. This is very different from the LCGs [12,14]. This made ICGs very useful in simulations to verify the result yielded by LCGs. However, the ICGs are usually slower than LCGs because of the inversion operation involved.

4. Shuffling method

The effort to use shuffling methods combining two different RNGs to provide a better random sequence is quite common nowadays. The introduction of a second RNG can sometimes show significant improvement over the original sequence.

In a shuffling method [16,17], we use the second generator to choose a random order for the numbers produced by the first generator. Generally, we first generate a vector of pseudorandom number of given size using the first generator. We then use the second generator to choose a number from the vector randomly. After the number been extracted, we fill the same slot with the next number in the sequence generated by the first RNG [15].

The idea of a shuffling method was first introduced by George Marsaglia [16,17]. The original shuffling method proposed was

$$U_{k+1} = (2^{17} + 3) \times U_k \bmod 2^{35},$$

$$V_{k+1} = ((2^7 + 1) \times V_k + 1) \bmod 2^{35}.$$

The sequence of U_k was used as the first sequence, while V_k is used as the second sequence. A vector of length 128 was used. The index of the vector was obtained from the V_k by using its first 7 bits. The pseudorandom numbers generated by this shuffling method were put through a series of tests, including the equidistribution test, the spectral test at dimensions 2 and 3, and the maximum and minimum of n. In contrast to LCGs, the shuffling random number generator passed all the tests [15]. Of course, the involvement of two pseudorandom number generators instead of one made the computational time twice as long.

5. Add-with-carry and Subtract-with-borrow generators

Add-with-carry (AWC) and Subtract-with-borrow (SWB) generators were proposed by Marsaglia and Zaman [18]. The AWC generators can be described as below:

$$x_i = (x_{i-s} + x_{i-r} + c_i) \bmod b\,;$$

$$c_{i+1} = I(x_{i-s} + x_{i-r} + c_i \geq b)\,;$$

where b, r, s are positive integers, b is called the base, and $r > s$ are called the lags. The carry c_i is calculated from the indicator function I, whose value is 1 if its argument is true, and 0 otherwise. SWB generators have similar formulas. These types of generators are extremely fast, since no multiplication is involved. These generators also have very long periods. These generators have been proved equivalent to LCGs with very large modulus and inherit the lattice structure in high dimensions [19]. These types of generators are currently used by G. Marsaglia since they are fast and have very long periods. (Marsaglia is perhaps the leading authority on PRNGs.)

The standard for a good PRNG

It is very difficult to tell what a good pseudorandom number generator should be. In fact, for different applications, the standard is different. For stochastic simulation, a good pseudorandom generator must have a long period, a uniform distribution, and the internal correlation of the RNG should not disturb the simulation. Thus, before any usage of a PRNG for simulation, the PRNG should be thoroughly tested. Better, a task-specific test should be developed for each simulation to make sure that the internal correlation between pseudorandom sequences would not affect the result. For cryptography, the unpredictability of a random sequence is also very important [20]. The cracking of an encryption method based on a PRNG is essentially to predict the next number from a short section of the same sequence. Other attributes a good pseudorandom number generator should have include the efficiency of the algorithm, the portability and the homogeneity of the pseudorandom numbers, which means that every bit in a pseudorandom number should have the same randomness.

Stream Ciphers and PRNGs

Pseudorandom numbers can be used in cryptography in conjunction with stream ciphers [20]. A symmetric encryption algorithm uses the same key in both encryption and decryption. A stream cipher is a kind of symmetric encryption algorithm that usually works on one byte each time. A block cipher is a symmetric algorithm that works on a block of information at one time.

In a very simple way, we can encrypt a message using a key and the exclusive OR (XOR) operation. Suppose we have a key of "icecream" and the plain text is "FOOTBALL", the result of the XOR operation is "/,*70$-!" in a UNIX system. This may seem good enough for ordinary encryption. In fact, this kind of encryption is nothing more than plain text itself in the eyes of a cryptanalyst, if we use the same key "icecream" repeatedly in the entire message. The idea of a one-time pad is to use different keys chosen at random for each byte in the message and never use the same sequence of keys again. This method has been proved unbreakable if the stream is truly random. One-time keys are in fact used for diplomatic messages of the highest security. However, a one-time pad method requires a large amount of random keys to be stored and distributed between the sender and receiver. The storing and distribution procedures are practically difficult to be secure, quick, and convenient at same time. This situation leads us to a third solution.

An intermediate method is to use a pseudorandom sequence as the key for encryption. An encryption depending on a pseudorandom sequence essentially makes an attack on the encryption method an attack on the pseudorandom generator. If the pseudorandom sequence displays a certain pattern, like the lattice structure displayed by all LCGs in *d*-tuples, it is proved breakable [21].

The most popular form of pseudorandom number generator used in cryptography is the feedback shift register [20, 30]. The hardware implementation of a feedback shift register is very simple and quick. It consists of only two parts. The first part is a shift register; the second part is a feedback function. The feedback shift register produces one bit at a time. After the rightmost bit been taken as the produced bit, the register shifts right one bit. The leftmost bit is determined by the "feedback function" $f(x_1, x_2, \ldots, x_n)$, where x_i $(1 \le i \le n)$ denotes the *i*th bit in the register. If the "feedback function" $f(x_1, x_2, \ldots, x_n)$ can be expressed as

$$f(x_1, x_2, \ldots, x_n) = c_1 x_1 \oplus c_2 x_2 \oplus \ldots \oplus c_t x_n ,$$

where each of the constants c_i $(1 \le i \le n)$ is either 0 or 1, and where the symbol $\oplus$ denotes addition modulo 2 (1 for odd sum and 0 for even sum), the shift register is called a linear shift feedback register (LSFR) [30]. The Linear feedback shift register has the same mathematical formula as lagged Fibonacci generators [20]. The difference is the modulus *m* here is 2. To generate a number, the bits are collected into words. Linear shift register sequences are easily cracked and are therefore never used in cryptography. Nonlinear feedback shift registers are used instead.

An additional text compression step is usually performed before encryption, which greatly increases the complexity of cryptanalysis and reduces the size of the message.

However, there are ways to crack a sequence encrypted by PRNGs. For a sequence encrypted by a LCG, only the length of the key and the same length of the plain text are necessary to reconstitute the LCG and its original value, which serves as the key [21]. The key to crack a sequence encrypted by a PRNG is to get a segment of the pseudorandom number sequence and predict the next number from this segment. Thus, the unpredictability of a pseudorandom number sequence is more important in cryptology than other aspects of the sequence, like the uniform distribution.

There are also reports on cracking the shuffling method sequence [22,23]. The basic idea of cracking the shuffling method is to separate the effect of the two generators involved in the algorithm. As we noticed, the sequence generated by the shuffling method is totally from one of the generators. The other one was only used to change the order of the number generated by the first generator. This change in order, however, is bounded to the size of the table used in the shuffling method. This is the basic weakness of this encryption. To improve the security, a double-encryption method was introduced [22]. This method uses two numbers from the first generator in shuffling method instead of one. This method can somewhat improve the time and resources involved in an attack but is also thought to be breakable. Later we will discuss another method suggested by Chandler [24], which may be much more difficult to break.

Stream ciphers have a limitation that the same key(s) (initial seed(s)) must never be used to encipher two different messages [20]. If two lengthy messages are enciphered using the same key(s), a simple frequency analysis can recover both of the messages and the pseudorandom stream, if not the seeds themselves. Block ciphers do not share this limitation. For greater security, a message can be compressed, then added to a pseudorandom sequence (stream cipher), then enciphered with a block cipher.

DNA and pseudorandom number generators

DNA (deoxynucleic acid) is the genetic material for all living cells in the world. It is also one of the genetic materials for viruses. A single stranded DNA chain can be considered as a linear sequence consisted of only four types of nucleic acid residues, which can be represented by A, G, C, and T. The double stranded DNA, which is the actual genetic material for human beings, consists of two strands of DNA winding around each other, forming a double helix structure. The two strands of DNA are complementary to each other while going the opposite direction. In the double helix structure, A is always complemented by T, and G with C. A pair of A-T or C-G residue is thus called a base pair, since all nucleic acid residues have a basic (alkali) part. The length of DNA is counted in base pairs. In living cells, DNA is organized into a large structure called a "chromosome". A single copy of the full DNA sequence is called the "genome" for that organism. The total length for the human genome is estimated around 4×10^9 base pairs. The sequences of the DNAs are transcripted into RNAs, and then translated into proteins, which performs all kinds of biological activities.

Since the 1970s, sequences of all kinds of different DNA pieces were reported. After the beginning of the genome project, more and more sequences were reported by different groups around the world. The current genome projects are focused on human beings, mice, fruit flies, yeast, bacteria, etc. The sequence of DNA discovered so far cannot be predicted in any mathematical way. Thus, the sequence of DNA can be considered as a partially random mixture of the four types of nucleic acid residues.

Since DNA sequences are fixed and anybody can retrieve them from the Internet, DNA themselves cannot be considered as a PRNG. Rather, combining the DNA sequence and a PRNG would be a better choice. An obvious way to crack the combined PRNG is to remove the effect of DNA by testing all the DNA sequences. After removing the DNA effect, one can try to crack the PRNG. The problem with this approach is that the amount of resources and times needed for a successful attack are increased dramatically, because the length of DNA sequence and thus the number of possible combinations is simply huge.

CHAPTER THREE

Program Listing

1. Mshuffle and Mimped

The Mshuffle method and Mimped method were implemented in C language. The 'M' in the method name stands for "masking". In both of the methods, a third generator was used to mask the result from a shuffling method by a bit-wise XOR operation. The C language provided a convenient way to do the XOR operation and thus was the choice of language. The first generator used in both methods is the standard ANSI C generator [12]; the second generator is the SIMSCRIPT generator [12]; and the third generator used in both methods is the Maple generator [12]. All generators have a period of 2^{31}-2 [12]. The first generator is used to provide the output sequence of the shuffling method. The second generator is used as the index generator for the 97-entry table. The sequence generated by the first generator is stored in the table first. The output of the shuffling method is selected from the table using the index generated by the second generator. The third generator generates a separate sequence. The sequence generated by the third generators is bitwise XORed with the sequence generated by the shuffling method. This is the output of Mshuffle method. Thus, all three LCGs are used to generate every number in the output sequence. The Mimped method then took the middle two bytes of the number generated by the Mshuffle method as its output. The output value was between 0 and 65535. To fulfill the requirement of some tests involved, the Mimped method called the Mshuffle twice and combined the two two-byte numbers into one four-byte number. Both methods use the double precision arithmetic operation provided by standard C library. Since all three LCGs used in the Mshuffle and Mimped methods have the same period (2^{31}-2), it is safe to say that Mshuffle and Mimped methods have periods at least as long as 2^{31}-2. After generating 2^{31}-2 numbers, those three LCGs will begin to repeat. However, since the table content probably will not be the same as it was in the beginning, the generated sequence should be different. Thus, these two generators should have a period at least as long as 2^{31}-2. The ANSI C generator was initialized to 12345; the other two generators were initialized to 1.

Mshuffle.h

```
#include <math.h>

/*  Table is designed to be of length 97.  Normally, the
 *  table used in the shuffling method will be of size
 *  near 100 for both convenience and randomness.
 */
#define TABLELENGTH 97
unsigned long table[TABLELENGTH];
const double MODNUMBER1 = 2147483648.0; // 2**31
const double MODNUMBER2 = 2147483647.0; // 2**31-1
```

Mshuffle.c / Mimped.c

```
#include "Mshuffle.h"

/*  First generator used in the shuffling method.  This generator will
generate
 *  numbers which will be put into the table.  This is a linear
congruential
 *  generator: LCG (2**31, 1103515245, 12345, 12345) ANSI C generator
 */
unsigned long rand1 ()
{
    static double seed=12345;
    while ((seed = fmod (1103515245.0*seed+12345.0,MODNUMBER1))<0);
    return (unsigned long)seed;
}

/*  Second generator used in the shuffling method.  This generator
 *  generates numbers that are used to permutate the sequence of the
 *  table.  This is a linear congruential generator: LCG (2**31-1,
 *  630360016,0) SIMSCRIPT generator.
 *  The seed is arbitrary.
 */
unsigned long rand2 ()
{
    static  double seed = 33456109 ;
    while ((seed = fmod (630360016.0*seed, MODNUMBER2))<0);
    return (unsigned long)seed;
}

/*  This generator is used to mask the result of the shuffling method.
 *  The result of the shuffling method will be exclusive ORed with the
 *  result of this generator.
 *  This is also a linear congruential generator: LCG (2**31,65539,0)
 *  Maple's LCG. The seed is arbitrary.
 */
unsigned long rand3 ()
{
    static double seed = 7789098.0 ;
    while ((seed = fmod (65539.0*seed,MODNUMBER1))<0);
    return (unsigned long)seed;
}

/*  This function takes two numbers and exclusive ORs them and returns
 *  the value.  This function is used to mask the pseudorandom number
 *  sequences.
 */
unsigned long mask ( unsigned long number1, unsigned long number2)
{
    return number1^number2;
}

/*  This function is used to extract the middle two bytes which will be
 *  the output of the complete pseudorandom number generator.
 */
unsigned int extract (unsigned long original)
{
    return (original&0x00ffff00)>>8;
}

/*  This function is the shuffling method itself.  It uses rand1,
 *  rand2 and the table to get the next number generated by the
 *  shuffling method.
 */
unsigned long shuffling ()
{
    unsigned long number1;
    unsigned long number2;
    unsigned long result;
    unsigned int index;

    number1 = rand1();
    number2 = rand2();
    index = number2 % TABLELENGTH;
    result = table[index];
    table[index] = number1;

    return result;
}

/*  This function is used to initialize the table.  It repeatedly calls
 *  rand1 to fill the table with the pseudorandom sequence.  This
 *  function should be called only once.
 */
void init ()
{
    register short i;
    for ( i=0; i<TABLELENGTH; i++)
        table[i] = rand1();
}

/*  This function MyRand is the complete pseudorandom number generator.
 *  The procedure of this generator is as followed:
 *  1. initialize shuffling method;
 *  2. call shuffling() to get the next number in the shuffling
 *     sequence;
 *  3. call rand3() to get the next number in the LCG ();
 *  4. bitwise exclusive OR two pseudorandom numbers to get a temporary
 *     result;
 *  5. extract the middle two bytes from the temporary result such that
 *     the final result is between 0 and 65535, inclusively;
 *  6. goto step 2 to get the next number in the sequence.
 */

/*  This function is being called when the original version of mshuffle
 *  is needed. It will return an unsigned integer value between 0 and
 *  2**32-1, inclusively.
 */
unsigned int MShuffle()
{
      return mask ( shuffling(), rand3() );
}

/*  This function is called when the improved version of mshuffle is
 *  needed. It will return an unsigned integer value between 0 and
 *  65535, inclusively.
 */
unsigned int Mimped()
{
      return extract ( mask ( shuffling(), rand3() ) );
}

/*  This function is called when the improved version of mshuffle is
 *  needed. It will return an unsigned integer value between 0 and
 *  2**32-1, inclusively.
 */
unsigned int Mimped32()
{
      unsigned int temp;
      temp = extract ( mask ( shuffling(), rand3() ) );
      temp = temp<<16; // generate left half
      temp += extract ( mask ( shuffling(), rand3() ) );
      // generate right half
      return temp;
}
```

2. Dshuf3

Dshuf3 was implemented in Fortran 77 by Dr. J. P. Chandler of Oklahoma State University [24]. This generator uses three simple LCGs and a table. The first LCG involved is based on the LCG (1048583, 1997, 0, 1). The second LCG involved is based on the LCG (1048589, 1993, 0, 1). The third LCG involved is based on the LCG (1048601, 1973, 0, 1). The first and second LCGs are involved in a shuffling method. The third LCG is used to decide the role of the first and the second generators. There are some modification made to the LCGs used in this method. Ten elements are thrown out from the first generator. Similarly, six and twelve elements are thrown out from the second and third generator, respectively. This thrown-out action ensures that the period for each generator is prime, and therefore relatively prime to the other generators' periods. The three relatively prime, yet different periods give Dshuf3 a period at least as long as the product of those three periods. Only after the product of those three periods of times, could Dshuf3 possibly begin to repeat. However, the table at the cycling point will almost centainly not be same as the table at the starting point, which means a much longer period for Dshuf3. This generator must to be initialized before its first usage. The initialization is to fill a table of 127 elements. All three generators are involved in the generation of each pseudorandom number, no matter whether it is in the initialization step or in later steps. This generator first calls all three LCGs to get the next numbers in the LCG's sequences. Then it compares the number generated by the third LCG to an arbitary cutoff number. If the number generated by the third LCG is greater than the cutoff value, the number generated by the first LCG serves as the index. The number generated by the second LCG would be inserted to the indexed table entry; the original number in this table entry would be the next number generated by Dshuf3. If the number generated by third LCG is smaller than the cutoff value, the second LCG serves as the index generator and the first LCG served as the number generator. By switching the role of the first and second generators using the value of third generator, this generator provides excellent unpredictability. The output from the current number generating LCG, which is a double precision number without the decimal part, is used as an intermediate result. Double precision numbers in FORTRAN have a resolution about 2^{-55}. To achieve an even better resolution, the third LCG is used to provide the fractional part of the number. This intermediate number is then divided by the period of current number generating LCG to yield the final output, a double precision number in the range of (0,1). To yield an integer number between $-2^{31}+1$ to 2^{31}, we multiplied the double precision number by 2^{32} and subtracted $2^{31}-1$ from it. An alternative form of this method generates one pseudorandom byte each time. Thus, to get a 32-bit integer, we needed to call the generator four times and assemble the four bytes into one number. For the Dshuf3 method, the first and the third LCG were initialized to 1 and the second generator was initialized to 2.

Otpt.c

```
/* This function provides file output for generators written in
 * FORTRAN 77. The FORTRAN77 function should make calls to an
 * external C function which is called "otpt" and takes one integer
 * argument.
 */
#include <stdio.h>
#include <stdlib.h>

void otpt_ (int * in)
{
    static int ff = 1;
    static FILE *fp;
    if (ff)
    {
        // open output file
        if (!(fp=fopen("ofile","w"))) exit (0);
        ff = 0;
    }
    // use fwrite() to output unformatted binary file.
    // this is the requirement for DIEHARD test suite.
    fwrite (in,sizeof(int),1,fp);
}
```

Dshuf3.f

```
C
C  CHALLENG.CNTL       PSEUDORANDOM NUMBER GENERATOR
C                         JULY 1999
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
      INTEGER JSEED,KUT3,INIT3,N,J
      INTEGER JTABLE,JTINDX,JTSIZE,M,KA,KMAX
      INTEGER RSLT
C
C     EXTERNAL FUNCTION OTPT WRITTEN IN C HANDLES FILE OUTPUT
C
      EXTERNAL OTPT !$PRAGMA C(OTPT)
C
      DOUBLE PRECISION DENOM
      DOUBLE PRECISION DD,DSHUF3
C
      DIMENSION JSEED(3),RBYTE(24000)
C
      COMMON /CDSHU3/ DENOM(3),
     *    JTABLE(127),JTINDX(127),JTSIZE,M(3),KA(3),KMAX(3),
     *    KMODE,JDIVIS,JRBYTE
C
C     GENERATE 1,240,000 BYTES
C
      N=2600000
C
C     CUTOFF VALUE
C
      KUT3=500000
C
C     SET KMODE TO 1 IF PSEUDORANDOM BYTE IS DESIRED
C     SET KMODE TO 2 IF DOUBLE PRECISON PSEUDORANDOM NUMBER
C     BETWEEN O AND 1 IS DESIRED
C
      KMODE=2
      JDIVIS=67
C
      INIT3=1
C
C     SET SEEDS HERE
C
      JSEED(1)=1
      JSEED(2)=1
      JSEED(3)=1

      DO 30 J=1,N
   20    DD=DSHUF3 (JSEED,KUT3,INIT3)

         RSLT = DD * 4294967296.D0 - 2147483648.D0
         IF (RSLT.GT.-1.D0 .AND. RSLT.LE.0.0D0) GOTO 20
         CALL OTPT(RSLT)

C        RBYTE(J)=JRBYTE
C
   30 CONTINUE
C
C     SUM=0.0D0
C     DO 100 J=1,N
C        SUM=SUM+RBYTE(J)
C 100 CONTINUE
C     AVE=SUM/N
C     PRINT 110,N,SUM
C 110 FORMAT(/' N =',I11,5X,'SUM =',G15.7)
C
      STOP
C
C  END CHALLENG (MAIN PROGRAM)
C
      END
      DOUBLE PRECISION FUNCTION DSHUF3 (JSEED,KUT3,INIT3)
C
C  DSHUF3 1.0          JULY 1999
C
C  PSEUDORANDOM NUMBER GENERATOR --
C     SHUFFLING GENERATOR WITH THE ROLES INTERCHANGING
C
C  J. P. CHANDLER, COMPUTER SCIENCE DEPARTMENT,
C     OKLAHOMA STATE UNIVERSITY
C
C     JSEED()  --  ARRAY OF THE CURRENT INTEGER PSEUDORANDOM
C                     DEVIATES, ONE FROM EACH CONGRUENTIAL GENERATOR.
C                     INITIALLY, THE USER MUST SET EACH JSEED(J)
C                     TO AN INTEGER BETWEEN 1 AND 2^20, INCLUSIVE.
C
C     KUT3     --  CUTOFF USED IN DECIDING WHEN TO INTERCHANGE
C                     THE ROLES OF GENERATORS NUMBER ONE AND TWO
C
C     INIT3    --  =1 ON THE FIRST CALL TO DSHUF3 FOR A GIVEN
C                     PROBLEM, TO FORCE INITIALIZATION.
C                     INIT3 IS RESET TO ZERO BY DSHUF3, AND MUST
C                     NOT BE CHANGED BY THE USER UNTIL THE NEXT
C                     PROBLEM IS TO BE STARTED, AND PERHAPS NOT
C                     EVEN THEN.
C
      IMPLICIT REAL*8 (A-H,O-Z)
      INTEGER JSEED,KUT3,INIT3
      INTEGER JTABLE,JTINDX,JTSIZE,M,KA,KMAX
      INTEGER INDX,JFILL,JHOLD,JTYPE,J,JINDX,JRNG,MOD
C
      DOUBLE PRECISION DENOM
      DOUBLE PRECISION TEMP
C
      DIMENSION JSEED(3)
C
      COMMON /CDSHU3/ DENOM(3),
     *   JTABLE(127),JTINDX(127),JTSIZE,M(3),KA(3),KMAX(3),
     *   KMODE,JDIVIS,JRBYTE
```

```
C
C
      IF(INIT3.EQ.0) GO TO 40
C
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
C  INITIALIZE.
C  SET THE PARAMETERS FOR THE THREE MULTIPLICATIVE CONGRUENTIAL
C  GENERATORS.
C  THE M(J) ARE THE MODULI.
C  THE KA(J) ARE THE MULTIPLIERS.
C  THE KMAX(J) ARE LIMITS ON THE SIZES OF THE JSEED(J), TO CAUSE
C  THE PERIODS OF THE THREE CONGRUENTIAL GENERATORS TO BE PRIME.
C
      M(1)=1048583
      KA(1)=1997
      KMAX(1)=M(1)-10
C
      M(2)=1048589
      KA(2)=1993
      KMAX(2)=M(2)-6
C
      M(3)=1048601
      KA(3)=1973
      KMAX(3)=M(3)-12
C
      DENOM(1)=KMAX(1)
      DENOM(2)=KMAX(2)
      DENOM(3)=KMAX(3)+1
C
C  FILL JTABLE() INITIALLY.
C
      JTSIZE=127
      DO 30 J=1,JTSIZE
C
C  CYCLE ALL THREE CONGRUENTIAL GENERATORS.
C
         DO 20 JRNG=1,3
   10       JSEED(JRNG)=MOD(KA(JRNG)*JSEED(JRNG),M(JRNG))
            IF(JSEED(JRNG).GT.KMAX(JRNG)) GO TO 10
   20    CONTINUE
C
C  USE THE VALUE OF JSEED(3) TO SELECT WHICH GENERATOR OF
C  THE FIRST TWO TO USE TO FILL THE NEXT SLOT IN THE TABLE.
C
         JFILL=1
         IF(JSEED(3).GT.KUT3) JFILL=2
C
         JTABLE(J)=JSEED(JFILL)
C
C  STORE IN JTINDX(J) THE NUMBER OF THE GENERATOR THAT WAS USED
C  TO FILL THE SLOT JTABLE(J).
C
         JTINDX(J)=JFILL
   30 CONTINUE
C
      INIT3=0
C
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
C  GENERATE AND RETURN THE NEXT PSEUDORANDOM NUMBER.
C  FIRST, CYCLE ALL THREE CONGRUENTIAL GENERATORS.
C
   40 DO 60 J=1,3
   50     JSEED(J)=MOD(KA(J)*JSEED(J),M(J))
          IF(JSEED(J).GT.KMAX(J)) GO TO 50
   60 CONTINUE
C
C  CHOOSE THE ROLES OF GENERATORS NUMBER ONE AND TWO,
C  DEPENDING ON THE VALUE FROM GENERATOR NUMBER THREE.
C
      JFILL=1
      IF(JSEED(3).GT.KUT3) JFILL=2
      JINDX=3-JFILL
C
C  SELECT AN ELEMENT FROM THE ARRAY, AND REFILL THAT SLOT.
C
      INDX=1+JSEED(JINDX)*JTSIZE/KMAX(JINDX)
C
      IF(INDX.GT.JTSIZE) THEN
         PRINT 70,JINDX,JSEED(JINDX),JTSIZE,KMAX(JINDX),INDX
   70    FORMAT(/' JINDX =',I2,5X,'JSEED(JINDX) =',I11,
     *      5X,'JTSIZE =',I4/
     *      5X,'KMAX(JINDX) =',I8,5X,'INDX =',I4)
         GO TO 40
      ENDIF
C
      JHOLD=JTABLE(INDX)
      JTABLE(INDX)=JSEED(JFILL)
      JTYPE=JTINDX(INDX)
      JTINDX(INDX)=JFILL
C
      IF(KMODE.EQ.1) THEN
         DSHUF3=-999.0D0
C
C  GENERATE A PSEUDORANDOM BYTE.
C
         JRBYTE=MOD((JHOLD+JSEED(3))/JDIVIS,256)
      ELSE
C
C  GENERATE THE DOUBLE PRECISION DEVIATE, USING GENERATORS
C  NUMBER JINDX AND NUMBER THREE TO FILL IN THE GAPS
C  AND GIVE HIGH RESOLUTION.
C
         TEMP=(JSEED(JINDX)-1)+JSEED(3)/DENOM(3)
         DSHUF3=((JHOLD-1)+TEMP/DENOM(JINDX))/DENOM(JTYPE)
         IF(DSHUF3.LE.0.0D0 .OR. DSHUF3.GE.1.0D0) GO TO 40
C
         JRBYTE=-999
      ENDIF
C
      RETURN
C
C  END DSHUF3
C
      END
```

3. DNA and DNAimp

We use DNA to denote the number sequence translated from actual human genes [26]. The translation was described below. Nucleic acid residue A was translated into 0, G was translated into 1, C into 2, and T into 3. Each residue was considered as a two-bit number. Four residues were thus translated into one byte. Four bytes (16 residues) were then organized into a thirty-two-bit number. For any test required an range of [0,1), the numbers were divided by 2^{32}. DNAimp was used to denote the method in which a random number obtained from DNA was bitwise XORed with a pseudorandom number generated by Mimped. DNAimp was implemented in C.

DNA.c

```
#include <stdio.h>

/* * This function takes an EMBL format DNA sequence file and transforms
   * the DNA sequence into numbers.  The EMBL format DNA sequence is a
   * sequence without linebreaks preceeded by an online description. In
   * the description is the number of the sequence in EMBL.
   */

int main () {
    FILE *fp, *fp2;
    char inchar;
    int inflag;
    int count;
    unsigned long output;
    unsigned long temp;
    count = 1;
    inflag = 0;
    output = temp = 0;
    fp = fopen ("hum.dat","r"); // open input file
    fp2 = fopen ("humout","w"); // open output file
    while ((inchar = fgetc(fp))!=EOF)
    {
        /* The only linebreak is at the end of a sequence or at the end
         * of a description.  Since a sequence is always preceeded by a
         * description, the linebreak can serve as a switching sign.
         */
        if (inchar == '\n') inflag = !inflag;
        if (inflag)
        {
            switch (inchar)
            {
            case 'a': temp = 0; break;
            case 'g': temp = 1; break;
            case 'c': temp = 2; break;
            case 't': temp = 3; break;
            default:  temp = 4; break;
            }
            if (temp>=4) continue; // ignore all other residues
            output |=temp;
            output = output<<2; // constuct number
            if (!(count%=16))  // every 16 base pairs consist one number
            {
                fwrite (&output, sizeof(unsigned long),1,fp2);
            }
            count ++;
        }
    }
    fclose (fp);
    fclose (fp2);
    return 1;
}
```

DNAimp.c

```
#include "Mimped.c"
int main ()
{
    unsigned short ran, hum;
    unsigned long i;
    FILE *fp2;
    FILE * fp;

    if (!(fp=fopen("myresult","w"))) exit (0); // open output file
    if (!(fp2=fopen("humout","r"))) exit (1); // open input file
    /* Input file should be the output file of dna.c */
    init();
    // generate 10 million of random bytes
    for (i =0; i<5000000; i++)
    {
        ran = Mimped();
        fread (&hum,sizeof(unsigned short), 1, fp2);
        ran = ran ^ hum;
        fwrite (&ran,sizeof(unsigned short),1,fp); // write unformatted
bytes
    }
}
```

4. Dshuf2

A simple shuffling method Dshuf2 was implemented in C in order to compare with Dshuf3. Dshuf2 used the first and second LCG in Dshuf3 without the thrown-out action to implement a typical shuffling method. All of the numbers generated by Dshuf2 are from the first LCG. The second LCG is used to generate the index into the 127-entry table. Both the first and second LCG are initialized to 1.

Dshuf2.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define TABLESIZE 127

unsigned long table[TABLESIZE] ={0};

unsigned long LCG1 ()
{
    /* LCG1 = LCG (1048583,1997,0,1) */
    static unsigned long seed = 1;
    unsigned long range = 1048573; // a prime
    do {
        seed = (unsigned long) fmod ( 1997.0*seed, 1048583.0);
    } while (seed >= range);
    return seed;
}

unsigned long LCG2 ()
{
    /* LCG2 = LCG (1048589,1993,0,1) */
    static unsigned long seed = 1;
    unsigned long range = 1048583; // a prime
    do {
        seed = (unsigned long) fmod ( 1993.0*seed, 1048589.0);
    } while (seed >= range);
    return seed;
}

void init()
{
    int i;
    for (i=0; i<TABLESIZE; i++)
        table[i] = LCG1();
}
```

```
unsigned long RANDO()
{
    unsigned long index;
    unsigned long number;
    unsigned long outnumber;

    index = LCG2()%TABLESIZE;
    number = LCG1();

    outnumber = table[index];
    table[index]=number;

    return outnumber;
}


int main()
{
    FILE *fp;
    int i;
    long ran;
    char temp;

    if (!(fp=fopen("dshuf2result","w"))) exit(0);
    init();

    for (i=0;i<12000000;i++)
    {
        ran = RANDO();
        temp = ran%256;
        fwrite(&temp, sizeof(char), 1, fp);
    }
    fclose(fp);
    return 1;
}
```

CHAPTER FOUR

Results

1. Runs test

In the runs test, we counted the runs up and runs down. For a sequence of number X_1, X_2, ..., X_n in the range [0,1), if $X_1<X_2<\ldots<X_m$, and $X_{m+1}>X_m$, we counted it as one runs up. We then discarded X_m and continued the counting from X_{m+2}. After testing a sequence of length of 10,000, we got the number of all runs up in the sequence. This number was then compared to the expected distribution, and a probability value was obtained. The test was repeated 10 times, and the ten obtained probability values were subjected to a χ^2 test. Tests on runs down were performed similarly. Two sets of four tests total were performed for each generator. The result listed in the table below is the probability value of the runs test. Values in boldface meant that this value is out of the 5% limitation, which is greater than 0.95 or smaller than 0.05. Too many boldface values suggest a failure in the particular test. Note that boldface values do happen sometimes. A 10% boldface value to all values ratio is acceptable. The results suggested that all generators passed the runs test.

Generator		MShuffle	Mimped	DShuf3[24]	Smith[25]	Dshuf2	DNAimp
Set1	Runs up	0.160863	0.703222	0.083466	**0.015760**	0.185621	0.540155
	Runs down	0.913231	0.724113	0.300338	0.401432	0.412830	0.275385
Set2	Runs up	0.792193	0.816320	0.854419	0.106908	0.415503	0.859278
	Runs down	0.259628	0.065763	0.248263	0.552016	0.346444	0.676084

Table 1. Results of the runs test. The values listed in the table are the possibility value. A possibility value between 0.5 (5% limit) and 0.95 (95% limit) can be considered as passed the particular test. All the following tables will follow this convention unless specifically denoted.

2. Permutation test

The permutation test takes *n* consecutive numbers. This number sequence has *n*! possible combinations. We characterize each one of those combinations as a state. After *m* overlapping sequences of length *n* have been processed, cumulative counts were made of the number of occurrences of each state. The χ^2 value was then calculated on the set of cumulative counts. In this particular experiment, *n* equals 5 and *m* equals 1,000,000. The result listed in the table below is the probability value of the permutation test. Values in boldface mean that this value is out of the 5% limitation, which is greater than 0.95 or smaller than 0.05. Too many boldface values suggest a failure in the particular test.

Generator	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
Set1	0.292171	0.167450	0.914650	0.315338	0.758860	0.830703
Set2	0.532492	0.622684	0.109340	0.054743	**0.975648**	**0.000021**

Table 2. Results of permutation test

3. Birthday spacing test

The birthday spacing test supposes we have *m* birthdays in a "year" of *n* days. After throwing *m* numbers into *n* categories, we list the spacing between *m* numbers. Let *j* be the times of occurrence of spacing values. The distribution of *j* should be asymptotically a Poisson distribution with mean $m^3/4n$. In this test, *m* is 2^{10} and *n* is 2^{24}. The *j* values are subsequently subjected to a χ^2 test. The result listed in the table below is the probability value of the permutation test. Values in boldface meant that this value is out of the 5% limitation, which is greater than 0.95 or smaller than 0.05. Results suggest that all the tests except the Mshuffle method and DNAimp passed this test.

Generator	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
	0.000000	0.123547	0.328946	0.133603	0.071341	**0.999394**

Table 3. Results of birthday spacing test

4. Monkey test

The monkey tests [6] are a set of tests that share the same idea. We first define an alphabet for the "monkey", then let the monkey type randomly on a typewriter. After a certain number of words have been typed, we compare the numbers of missing words to the expected values. In a bit stream test, the alphabet is '0' and '1' and the length of the words is twenty. The alphabet size of the OPSO test is 2^{10} and the length of the words is two. The alphabet size of the OQSO test is 2^{5} and the length of the words in four. The DNA test has an alphabet of C, G, A, and T. Each letter is represented by two bits. The length of the words in the DNA test is ten. The result listed in the table below is the probability value of the monkey tests. Values in boldface meant that this value is out of the 5% limitation, which is greater than 0.95 or smaller than 0.05.

In the bit stream test, the probability value of all 20 bitstreams for Mshuffle are 0.000000, which means these two generators failed on all bitstreams. The Harry Smith generator also failed this test but at a less severe level. The Mimped method passed this test. In the OPSO test, the Harry Smith generator performed badly. The Mimped, Dshuf2 method and Dshuf3 methods passed this test. The Mshuffle method's results were mixed. The right half of the number generated by Mshuffle failed this test badly while the left half geared through. This result suggested that the left half side was more random than the right half side for the Mshuffle method. In the OQSO test, methods Mimped, Dshuf2 and Dshuf3 passed without difficulties. The Harry Smith generator behaved somewhat better than in the OPSO test while still not passing. The Mshuffle method failed this test, too. The righthand side was again less random than the lefthand side. The DNA test performed on those four generators gave the similar result. Methods

Mimped, Dshuf2 and Dshuf3 passed the test again. In contrast to the OPSO and OQSO tests, the Harry Smith generator passed the DNA test, too. The Mshuffle method failed again with its left half behaved better than its right half. In these four tests, it was clear that Dshuf3 and Mimped behaved the best. The lefthand side of the numbers generated by Mshuffle was significantly better in tests than the righthand side.

Counting the 1's test is also a part of the monkey test with some twists. We count the 1's in a byte and designate an alphabet of 'A', 'B', 'C', 'D', and 'E'. 'A' is associated with a byte with zero, one, or two 1's. 'B' is three, 'C' is four, 'D' is five, and the rest are 'E'. The length of the words is five. The first test counting the1's treated all bytes as a stream and tests the overall behavior, while the second test uses a specific byte (8 bits). The results showed that the Mimped method, Dshuf2 method and Dshuf3 method passed the first test, which is not surprising, considering their better behavior in the other monkey tests. The Mshuffle method failed both tests. In the second test, it showed the "left better than right" pattern again.

4.1 Monkey tests on 20-bit words

Bit-stream	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
1	**0.000000**	0.436266	0.741497	**0.000036**	0.794138	0.570772
2	**0.000000**	0.507764	0.907503	**0.002966**	0.152698	**0.015642**
3	**0.000000**	0.199702	0.148330	**0.000194**	0.291745	0.666649
4	**0.000000**	0.415202	0.309632	**0.000088**	0.816642	0.798108
5	**0.000000**	0.092752	0.097495	**0.000598**	0.882981	0.495647

6	**0.000000**	**0.981776**	0.239729	**0.000127**	0.819726	0.268160
7	**0.000000**	0.865618	0.208977	**0.000333**	0.062194	0.765078
8	**0.000000**	0.816642	0.752880	**0.000000**	0.767940	**0.987118**
9	**0.000000**	0.667498	0.684295	**0.000216**	0.934791	0.423419
10	**0.000000**	0.037380	0.374741	**0.000004**	0.799421	0.431669
11	**0.000000**	0.696650	**0.029418**	**0.000000**	0.083783	0.249267
12	**0.000000**	**0.016201**	0.453806	**0.000039**	0.680129	0.393481
13	**0.000000**	0.061624	0.143517	**0.000106**	0.287748	0.212350
14	**0.000000**	0.870609	0.304707	**0.000004**	0.112339	0.688437
15	**0.000000**	0.224753	0.434426	**0.000372**	0.753439	0.723039
16	**0.000000**	0.081645	0.114133	**0.003393**	**0.031017**	0.327983
17	**0.000000**	0.217125	**0.000401**	**0.000017**	**0.032858**	0.683464
18	**0.000000**	0.917170	0.417937	**0.000001**	0.624071	0.234678
19	**0.000000**	0.312935	0.717536	**0.000429**	0.812268	0.380957
20	**0.000000**	0.277478	0.429833	**0.006474**	0.870609	**0.043034**

Table 4. Results of bit stream test

4.2 OPSO test

Bits used	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
23 to 32	**0.000000**	0.834860	0.534801	**0.000000**	0.899189	**0.000805**
22 to 31	**0.000000**	0.445597	0.319954	**0.000000**	0.239098	**0.023568**

21 to 30	**0.000000**	0.098185	0.402401	**0.000000**	0.804087	0.326144
20 to 29	**0.000000**	0.567536	0.829670	**0.000000**	0.553939	0.414453
19 to 28	**0.000000**	0.757314	0.709821	**0.000000**	0.559386	0.949401
18 to 27	**0.000000**	0.086224	0.121438	**0.000000**	0.351371	0.252122
17 to 26	**0.000000**	**0.975979**	**0.033891**	**0.000000**	0.593136	0.125662
16 to 25	**0.000000**	0.341194	0.171794	**0.000000**	0.169162	0.076869
15 to 24	0.452419	0.628818	0.867560	**0.000000**	0.434716	0.634019
14 to 23	0.810692	0.377271	0.678392	**0.000000**	0.409086	0.751890
13 to 22	0.242319	0.658376	0.558026	**0.000000**	0.268913	0.834860
12 to 21	0.731837	0.619659	0.732972	**0.000000**	0.947948	0.739731
11 to 20	0.835715	0.726126	0.826152	**0.000000**	0.825266	**0.025553**
10 to 19	0.299288	0.195589	0.814404	**0.000000**	0.478454	0.863075
09 to 18	0.859260	0.603804	**0.985641**	**0.000000**	0.333635	0.437432
08 to 17	0.777330	0.417143	0.154790	**0.000000**	0.117315	0.514208
07 to 16	0.253224	0.098783	0.667192	**0.000000**	0.170907	0.882454
06 to 15	0.448324	0.115291	0.441512	**0.000000**	0.736363	0.422534
05 to 14	0.860029	0.449688	0.949401	**0.000000**	**0.999182**	0.434716
04 to 13	0.777330	0.883133	0.754067	**0.000000**	0.193690	0.464733
03 to 12	0.308938	0.863830	0.338667	**0.000000**	0.833142	0.101203
02 to 11	0.572953	0.527944	0.675917	**0.000000**	0.715696	**0.011775**
01 to 10	**0.000000**	0.812553	0.760538	**0.000000**	**0.965759**	0.936405

Table 5. Results of overlapping pairs sparse occupancy test

4.3 OQSO test

Bits used	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
28 to 32	**0.000000**	0.230298	0.901933	**0.004277**	0.313123	0.311923
27 to 31	**0.000000**	**0.025705**	0.731735	0.072159	0.340008	0.918903
26 to 30	**0.000000**	0.127589	0.930893	**0.023180**	0.924359	0.862725
25 to 29	**0.000000**	0.909342	0.719305	**0.000557**	0.693643	0.828150
24 to 28	**0.000000**	0.864208	0.519371	0.135541	0.604698	0.393554
23 to 27	**0.000000**	0.794307	0.383162	**0.007216**	0.488277	0.635660
22 to 26	**0.000000**	0.599466	0.832435	0.117277	0.658326	0.682854
21 to 25	**0.000000**	0.936983	0.816688	**0.026947**	0.501799	0.680435
20 to 24	0.117945	0.203442	**0.963076**	**0.006754**	0.189365	0.804760
19 to 23	**0.963349**	0.223151	0.676792	0.229270	0.241784	0.379286
18 to 22	0.051258	0.936563	0.316735	0.229270	0.598156	0.813976
17 to 21	0.389648	0.353773	0.574389	0.146921	0.848081	**0.996309**
16 to 20	**0.962802**	0.294166	0.855913	0.087200	0.199628	0.809402
15 to 19	0.599466	0.818481	0.577045	0.050548	0.356298	0.398779
14 to 18	**0.041417**	0.624140	0.094972	0.157289	0.657081	0.562398
13 to 17	0.898968	0.430480	0.066295	**0.009523**	0.648317	0.426488
12 to 16	0.726119	0.859725	0.532865	**0.000110**	0.244969	0.170695
11 to 15	**0.024125**	0.320365	0.591586	**0.002967**	0.353773	**0.007633**
10 to 14	0.461289	0.718162	0.251407	**0.027158**	0.060807	0.088280

09 to 13	0.246036	**0.956642**	0.772432	**0.005966**	0.946810	**0.011914**
08 to 12	0.852032	0.449195	0.400088	**0.030038**	**0.024905**	0.803823
07 to 11	0.069858	0.446514	0.347494	0.139265	0.765207	**0.952096**
06 to 10	**0.043252**	0.866413	0.704267	**0.032176**	0.209243	0.261230
05 to 09	0.247105	0.599466	0.871460	**0.008072**	0.074996	0.310725
04 to 08	**0.986840**	0.474768	0.780535	0.737291	0.926744	0.056447
03 to 07	0.100828	0.613807	0.890321	0.068059	0.607306	**0.007633**
02 to 06	0.504503	0.302395	**0.044831**	**0.002119**	0.669454	0.735076
01 to 05	**0.000000**	0.840795	0.733964	0.332587	0.081946	0.932678

Table 6. Results of overlapping quadruples sparse occupancy test

4.4 DNA test

Bits used	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
31 to 32	**0.000000**	**0.023449**	0.649679	0.555518	0.920005	**0.003978**
30 to 31	**0.000000**	0.255641	0.860067	0.144351	0.543842	0.868423
29 to 30	**0.000000**	0.430140	0.137104	0.110068	0.525085	0.198056
28 to 29	**0.000000**	0.236135	0.915984	0.768079	0.379874	0.391154
27 to 28	**0.000000**	0.778737	0.391154	0.886940	0.557848	0.939740
26 to 27	**0.000000**	0.329424	0.936501	0.128866	0.249983	0.352137
25 to 26	**0.000000**	0.242544	0.934644	**0.014777**	0.190751	0.663780
24 to 25	**0.000000**	0.593657	0.870301	0.691286	0.836537	0.536817
23 to 24	0.281903	0.629813	0.426667	0.163944	0.151851	0.676812

22 to 23	0.166141	0.083861	0.800017	0.434780	0.409390	0.907490
21 to 22	0.365363	0.185185	0.153242	0.832872	**0.005655**	0.217478
20 to 21	0.213165	0.376510	0.186765	0.268124	0.812172	0.595944
19 to 20	0.343417	0.581024	0.823096	**0.018554**	0.349949	0.641998
18 to 19	0.447583	0.812966	0.899421	0.220090	**0.974108**	**0.998257**
17 to 18	0.936867	**0.045582**	0.669144	0.824623	0.733487	0.523910
16 to 17	0.329424	**0.013917**	0.835077	0.905022	0.133258	0.345589
15 to 16	0.294995	0.093817	0.132624	0.909911	0.076401	**0.990210**
14 to 15	0.835808	0.734455	0.130110	0.156757	0.677655	0.015799
13 to 14	0.208902	0.451085	0.789109	0.867160	0.495681	0.175118
12 to 13	0.455760	0.578717	0.375390	0.253747	0.716760	0.048182
11 to 12	0.326228	0.476865	0.190751	0.798361	0.399105	0.806557
10 to 11	0.872161	0.776981	**0.976821**	0.071045	0.241623	**0.037478**
09 to 10	0.583327	0.858750	0.862675	0.697491	0.363147	0.559012
08 to 09	0.780486	0.626466	0.861375	0.431299	0.832872	0.850672
07 to 08	0.503919	0.189150	0.327292	0.470993	0.919565	**0.950220**
06 to 07	0.369809	0.063742	0.636476	0.357627	0.466301	0.523910
05 to 06	0.593657	0.820016	0.486269	0.858088	0.708719	0.303189
04 to 05	0.756218	0.880595	0.308361	0.303189	0.799190	0.163944
03 to 04	0.812966	0.482741	0.355427	0.795862	0.601646	**0.954038**
02 to 03	0.109514	0.492151	0.881182	**0.031208**	0.203854	0.391154
01 to 02	**0.000000**	0.050592	0.102495	0.499211	0.466301	0.193977

Table 7. Results of DNA test

4.5 Count the 1's in a stream of bytes

Generator	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
Set1	**0.000000**	0.219021	0.116777	**0.000000**	0.634400	0.220701

Table 8. Results of count the 1's in bytes test

4.6 Count the 1's in specific bytes

Bits used	MShuffle	Mimped	Dshuf3	Smith	Dshuf2	DNAimp
25 to 32	**0.000000**	0.123408	**0.973099**	0.366552	0.769704	0.888372
24 to 31	**0.000000**	0.576830	0.611456	0.433805	0.332958	0.818292
23 to 30	**0.000000**	0.493151	0.720227	0.752541	0.535383	**0.043703**
22 to 29	**0.000000**	0.689718	0.485419	0.822143	0.735728	0.493513
21 to 28	**0.000000**	0.216088	0.489146	0.057368	0.098578	**0.031674**
20 to 27	**0.000000**	0.495211	0.520268	0.264894	0.781242	0.555019
19 to 26	**0.000000**	0.364342	0.123086	0.799487	0.406989	0.179143
18 to 25	**0.000000**	0.399651	0.425996	**0.995670**	0.763872	0.481004
17 to 24	0.154335	0.691977	0.811911	**0.974886**	0.309174	0.168689
16 to 23	0.275023	0.396283	**0.026894**	0.065395	**0.019572**	0.512074
15 to 22	0.142204	0.210622	0.564815	0.718395	0.742663	0.666865
14 to 21	0.706495	0.446023	0.259722	0.168268	0.333931	0.855187
13 to 20	0.500360	0.633576	0.427987	0.126223	0.437817	0.214845
12 to 19	0.403946	0.902999	**0.956481**	0.096766	**0.044850**	0.477072
11 to 18	0.925530	**0.968591**	0.143360	**0.961834**	0.538469	0.139057
10 to 17	**0.042823**	0.198665	**0.015321**	0.558488	0.314934	0.783452
09 to 16	0.544964	0.845644	0.517026	0.918330	0.773595	0.468725
08 to 15	0.689033	**0.043349**	0.193690	0.222800	0.651871	0.731186
07 to 14	0.585551	0.752019	0.500130	0.241756	0.583573	**0.004309**
06 to 13	0.534703	0.609942	0.303118	**0.995490**	0.070391	0.602733

05 to 12	**0.007059**	0.057582	0.738741	0.546244	**0.008683**	0.459820
04 to 11	**0.998394**	0.208006	0.276706	0.569758	0.574135	0.210272
03 to 10	0.599927	0.453185	0.376765	0.929855	0.766192	0.829111
02 to 09	0.624419	0.936965	0.092553	0.295041	0.512580	0.653603
01 to 08	**0.000000**	0.393100	0.591665	0.719330	0.829522	0.753569

Table 9. Results of count the 1's in specific byte test

5. N-block test

The N-block test [8,9] was similar to a two-dimensional random walk test (see below). The difference was that the N-block test is in one dimension. The result listed in the table below is the probability value of the N-block test. Values in boldface meant that this value is out of the 5% limitation, which is greater than 0.95 or smaller than 0.05. The results suggest that all four generators passed both tests.

Generator	MShuffle	Mimped	Dshuf3	Smith
χ^2	0.841000	0.112360	0.686440	0.243360

Table 10. Results of n-block test. The value obtained by this test was a χ^2 value with degrees of freedom equal to one (1). The 5% limit in this test was 0.00393. The 95% limit in this test was 3.841. All four generators subjected to this test passed.

6. Two-dimensional random walk test

In the two-dimensional random walk test [8,9], we divide the two-dimensional plane into four equal size blocks, using x-axis and y-axis as the limits of each block. When a series of floating point numbers between [0,1) are passed through the test, we take a sequence of length *n* and define a random walk. The starting point of the walk is the origin. We also define the value of the number greater than 0.5 as movement along the y-axis, while any value less than 0.5 is the movement along the x-axis. Further, we define a value greater than 0.75 as a positive increment along the y-axis while a value between 0.5 and 0.75 is a negative increment along the y-axis. Similarly, the increments along the x-axis were also defined. We then let the walk begin and record the finish point of the walk. After a certain number of steps, the count of finishing walks in four different blocks were subjected to a χ^2 test with degrees of freedom equal to three. The result listed in the table below is the probability value of the two-dimensional random walk test. The results suggest that all four generators passed this test.

Generator	MShuffle	Mimped	Dshuf3	Smith
χ^2	1.998900	3.472400	2.287200	2.411400

Table 11. Results of two-dimension random walk test. The value obtained by this test was a χ^2 value with degrees of freedom equal to three (3). The 5% limit in this test was 0.3518. The 95% limit in this test was 7.815. All four generators subjected to this test passed.

CHAPTER FIVE

Discussion

1. Implementation details

The Harry Smith generator used nine different simple generators. Five of them are LCGs, the other four are shift-feedback registers. In order to improve the unpredictability, this generator cycles through those nine simple generators. Each simple generator is used as the index generator for the simple generator before it. Thus, each simple generator is both an index generator and generates numbers for the Harry Smith generator. This method can be considered as an extremely complicated shuffling method. This generator was also implemented in C. The output range is –32768 to 32767. For further information, please check the reference [25]. The Harry Smith generator required an input key and would utilize this input key to initialize the nine simple generators it used. The input key was "ajkK15k9kk599A9071136dg[q83f.w43". The corresponding seeds were -121385432, -2109937680, -1447464168, 184364026, -7426668, 2146034726, 1590708279, -542013146, and -491220481.

The Dshuf3 method was the answer to a previous report on cracking the shuffling method. As mentioned before, the basic idea of cracking a shuffling method is to separate the functionality of the first generator and the second generator. In a shuffling method, all of the sequence generated is from the first generator. The second generator only generated the index into the table. In the Dshuf3 method, however, both the first and second generators were involved in the number sequence and the index generation, and thus eliminated the possibility to separate the functionality of these two LCGs. Even better, the role switching was decided pseudorandomly by a third LCG and a cutoff value. A different third generator provides a different mixture pattern. Changing the cutoff value will also change the behavior of this method. In our tests, the cutoff value was 500,000, roughly equal to half the size of the third generator's period. This meant that in our tests, the numbers in Dshuf3 output were about equally provided by both first and second generators. Just by simply increasing or decreasing the cutoff value, we can easily get an output sequence that has more or fewer numbers from one generator. This feature provides a lot of flexibility and more unpredictability. Dshuf3 is easily portable to any computer and any language offering 32-bit integers. It has very high resolution ($\sim 2^{-60}$), and a very long but unknown period. The period is guaranteed to be greater than 2^{60}, and probably very much greater.

The two-dimensional random walk test and n-block test was implemented in Fortran 77. The source codes were downloaded from the web [27]. Other tests were from the test package "diehard", which was downloaded from G. Marsaglia's web site [28]. The "diehard" test package was implemented in C.

All interfaces between C programs and Fortran 77 programs were implemented as below. The C functions were called as subroutines from Fortran 77 programs. The definition and calling followed the Fortran 77 convention [29]. The value passed from a Fortran 77 program to a C subroutine and from a C subroutine to a Fortran 77 program was passed as parameters by reference. The C subroutine had no return value. The "diehard" test package needed a stream of unformatted binary data of about 10,000,000 bytes to 20,000,000 bytes. This is achieved for a C program by calling the fwrite() function in the standard library. For generators implemented in Fortran 77, a C subroutine was called for file I/O.

2. Tests and results

The Harry Smith generator performed poorly in the monkey tests. One might think that since this generator is very complicated, it should perform better. The reason that the Harry Smith generator performed poorly in all monkey tests is that it uses four shift feedback registers. Shift feedback registers are known to be very poor in monkey tests [6]. Since in the Harry Smith generator, four out of nine numbers are generated by shift feedback registers, it inherits the poor performance in monkey tests from those shift feedback registers.

For the test result of DNA (not shown) and DNAimp, we concluded that DNAimp was able to pass most of the tests while DNA itself was not able to pass most tests, especially for those tested on a bit level. This result was not a surprise. This is because the genes usually consisted of smaller functional groups. Those small functional groups were not capable of complex biological activity. Rather, it was the combination of the small functional groups that provided the very complex biological activity needed by different proteins. Those small functional groups can be considered as the alphabet of the biological world, while the gene itself resembled the words. Those alphabets were reused in many different ways and appeared at different positions on different genes. Thus, DNA has some regularity in it. However, since certain functionality of those sequences only needs a loose resemblance, the DNA sequences itself still maintain a certain degree of randomness. The DNAimp generator, on the other hand, incorporated the sequence of DNA and the sequence of numbers generated by Mimped. We already knew that Mimped was able to pass all the tests. It was not surprising to see that DNAimp did much better in all tests than DNA itself.

3. Comparison among different generators

Both the Mshuffle and Mimped methods used the same mechanism to generate pseudorandom numbers. The difference between these two methods was small. The Mimped method simply extracted the middle two bytes and merged two generation results into one. This improvement was made after noticed the pattern existed in Mshuffle method, that is, the left half consistently performed better than the right half. This result was directly affected by the fact that both generators involved in number generation in the Mshuffle method, the ANSI C generator and the Maple generator, used a modulus equals to 2^{31}. Changing one or both modulus to a different number will effectively minimize the unfortunate patterns. The choice of simple LCGs for the Mshuffle method is obviously not good. However, the approach used in Mimped was demonstrated to be effective in limiting this pattern. In all test results for Mimped, we do not notice any unbalancedness of the randomness between left half and right half. Thus, the improvement of Mimped over Mshuffle was significant and resulted in a better generator.

An alternative form of Dshuf3, which took four bytes generated by Dshuf3 and assembled those four bytes into one number, was also tested. The detailed results were not listed above. The alternative form of Dshuf3 performed well in all tests.

We can consider Mshuffle and Mimped as siblings and thus had three generator types. Those three types of generators had one thing in common. They all combined some simple generators to overcome certain faults of those simple generators. These three methods should be slower than simple generators, because it took at least two generators to yield one number. The times consumed to generate 10 million bytes (2.5 million 32-bit numbers) are listed in table 12.

Generator	Dshuf3	Dshuf3 Alternative	Harry Smith	Mshuffle	Mimped	ANSI C
Time used (seconds)	8.96	31.71	1.49	11.89	24.27	0.66

Table 12. Time consumed by different generators to generate 10 million bytes

From Table 12, we can see that the Harry Smith generator is the fastest compound generator. It took about twice as much time as the ANSI C generator, which is a simple LCG generator. Dshuf3 method alternative form took about four times the time of Dshuf3. This is because the alternative form needs to call Dshuf3 four times to get one 32-bit number. The situation is similar in the Mshuffle and Mimped methods. Since Mimped called Mshuffle twice to get a number, it took about twice as long as the Mshuffle method. These results were obtained without any type of I/O operation and should be considered as pure calculation time.

DNAimp should be a good generator for a stream cipher, not only because it performed well in statistical tests, but it has inherited the unpredictability of DNA as well. We cannot say how good the unpredictability this generator will have. Actually, one can never prove unpredictability, only predictability. The unpredictability of DNAimp should be good in the sense that if we can predict the sequence generated by DNAimp, then in turn, we should be able to predict the sequence of an actual DNA sequence, which is a very unlikely situation. Since the sequences of the genes can be to downloaded from hundreds of web sites, the availability of a DNA sequence should not be an obstacle for the application of this generator. Even better, most protein has a standard numerical name. Since proteins are derived from DNA, most DNA sequences (genes) can share this numerical name.

Dshuf3 should be a very good generator for any purpose. It not only provided excellent unpredictability, but it is fast and portable as well. It also has a very long period and very fine resolution. Although we cannot provide evidence that Dshuf3 is unbreakable, it is safe to say that Dshuf3 is extremely difficult to crack.

CHAPTER SIX

Summary and Future Work

Pseudorandom number generators are an important topic in computer science. We have discussed several new types of compound pseudorandom number generators above. Among those new types of pseudorandom number generators, the Dshuf3 method performed very well in all statistical tests. It is also a very promising pseudorandom number generator for cryptography usage. The DNAimp and Mimped methods are able to pass most statistical tests and should be good enough for daily usage. The Harry Smith generator needs some improvement to yield better results in statistical tests. Further investigation of the property of these generators should be followed. The effect of using different LCGs, even different types of pseudorandom number generators (i.e. ICGs, lagged Fibonacci generators, etc.) in the Dshuf3 method should be investigated. Also, the effect of changing the cutoff value should be examined. An improved version of Mshuffle using different starting LCGs should be examined. In this improved version, the period of the three LCGs involved should be carefully chosen so that they are relatively prime. For the Harry Smith generator, a newer version using 32-bit arithmetic operations and have changed some of its feedback shift registers to other types of PRNGs should be further examined.

BIBLIOGRAPHY

1. D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms Vol.* **2**, *Second edition* (1981) Addison-Wesley, Reading, MA;

2. Randomness and random sampling numbers, M. G. Kendall and B. Babington-Smith, *J. Royal Stat. Soc.* **A101** (1938) 147-166;

3. Second paper on random sampling numbers, M. G. Kendall and B. Babington-Smith, *J. Royal Stat. Soc.* **B6** (1939) 51-61;

4. ERNIE --- a mathematical and statistical analysis, W. E. Thomson, *J. Royal Stat. Soc.* **A122** (1959) 301-333;

5. Toward a universal random number generator, G. Marsaglia, A. Zaman, W.W. Tsang, *Stat. And Prob. Letters* **8** (1990), 35-39;

6. Monkey tests for random number generators, G. Marsaglia, *Computers & Mathematics with Applications* **9** (1993) 1-10;

7. G. Marsaglia, web site: stat.fsu.edu/~geo;

8. Physical models as tests of randomness, I. Vattulainen, T. Ala-Nissila, and K. Kankaala, *Physical Review E* **52** (1995) 3205-3214;

9. Physical tests for random numbers in simulations, I. Vattulainen, T. Ala-Nissila, and K. Kankaala, *Physical Review Letters* **73** (1994) 2513-2516;

10. Mathematical methods in large-scale computing units, D. H. Lehmer, *Proc. 2nd Symp. On Large-Scale Digital Calculating Machinery*, (Cambridge, Mass.: Havard University Press, 1951) 141-146;

11. Random numbers fall mainly in the planes, G. Marsaglia, *Proceedings National Academy Science* **61** (1968) 25-28;

12. Good random number generators are (not so) easy to find, P. Hellekalek, *Math. and Computers in Simulation* **46** (1998) 485-505;

13. Structural properties for two classes of combined random number generators, P. L'Ecuyer and S. Tezuka, *Math. Of Computation* **57** (1991) 735-746;

14. Web site: random.mat.sbg.ac.at;

15. A review of pseudorandom number generators, F. James, *Computer Phys. Comm.* **60** (1990) 329-344;

16. Uniform random number generators, M. D. MacLaren and G. Marsaglia, *JACM* **12** (1965) 83-89;

17. One-line random number generators and their use in combinations, G. Marsaglia and T. A. Bray, *CACM* **11** (1968) 757-759;

18. A new class of random number generators, G. Marsaglia and A. Zaman, *Ann. Appl. Prob.* **1** (1991) 462-480;

19. On the lattice structure of the add-with-carry and subtract-with-borrow random number generators, S. Tezuka, P. L'Ecuyer, and R. Couture, *ACM Transactions on Modeling and Computer Simulation* **3** (1993) 315-331;

20. *Applied Cryptography, Second edition*, B. Schneier (John Wiley & Sons, 1996);

21. Cracking random number generator, J. A. Reeds, *Cryptologia*, **1** (1977), 20-26;

22. Cryptanalysis of a MacLaren-Marsaglia system, C. T. Retter, *Cryptologia*, **8** (1984), 97-108;

23. A key-search attack on MacLaren-Marsaglia systems, C. T. Retter, *Cryptologia*, **9** (1984), 114-130;

24. Personal communication, Dr. J. P. Chandler, Department of Computer Science, Oklahoma State University;

25. The Harry Smith generator, Website: www.netcom.com/~hjsmith;

26. European Molecular Biology Laboratory, Website: www.embl-heidelberg.de

27. Website: www.netlib.org/random

28. Diehard test package, Website: stat.fsu.edu/pub/diehard/diehard.c.tar.gz

29. *Fortran 77 4.0 User's Guide*, website: docs.sun.com;

30. *Shift Register Sequences*, Solomon W. Golomb (Holden-Day, Inc., 1967)

VITA

Qi Jiang

Candidate for the Degree of

Master of Science

Title of Study: NEW TYPES OF PSEUDORANDOM NUMBER GENERATOR

Major Field Computer Science

Biographical.

Personal Data: Born in Shanghai, P.R. China, on April 25, 1973, the son of Dehua Jiang and Yazhen Zhang.

Education: Graduated from the No.2 middle school attached to East China Normal University, Shanghai, P R. China in July, 1991; Received a Bachelor of Science degree in Genetics and Genetic Engineering from Fudan University, Shanghai, P R. China in July, 1996 Completed the requirements for Master of Science degree with a major in Biochemistry and Molecular Biology at Oklahoma State University in May, 1999. Completed the requirements for Master of Science degree with a major in Computer Science at Oklahoma State University in December 1999.

Experience Employed by Oklahoma State University as a graduate research assistant; Oklahoma State University, Department of Biochemistry and Molecular Biology, August, 1996 to 1998; Employed by Oklahoma State University as part-time technician; Oklahoma State University. Department of Entomology and Plant Physiology, August 1998 to present