A

THESIS FOR THE DEGREE OF

MASTER OF SCIENCE

RANDOM NUMBER GENERATORS AND THEIR APPLICATION IN CRYPTOGRAPHY (BLUM, BLUM, SHUB REALIZATION)

BY

USMAN ABRAHAM USMAN

(209043011)

DEPARTMENT OF ELECTRICAL/ELECTRONICS ENGINEERING

UNIVERSITYOF LAGOS, AKOKA,

NIGERIA.

APRIL, 2003.

DEDICATION.

To God, my all sufficient one who kept my hope alive and to my parents, Mr. and Mrs. Samuel N. Usman for bearing this long with me.

CERTIFICATION

I hereby certify that Usman Abraham Usman carried out this research work under my supervision.

Prof. O. Adegbenro	17th. April 2003	[illegible]
Supervisor	Date	Signature

ACKNOWLEDGEMENT.

My sincere thanks goes to my supervisor, Professor O. Adegbenro for constructively criticizing the work and giving thorough guidance all through the work.

To all relations, friends , colleagues and the brethren of the faith most of which gave of their time, prayers, encouragement, substance, homes, etc, I say thank you. You are too numerous to mention. You have done all these as unto the Lord who is also your exceeding great reward.

I give all the thanks to God my maker without whom I can do NOTHING and through whom I can do ALLTHINGS. His name be praised forevermore.

ABSTRACT.

The need for information security is the need for generation of secret quantities around which the efficiency of these security systems are tied. This study covers a review of the concept of randomness with the stringent randomness requirement in cryptography.

Some of the techniques for random number generation are reviewed. The cryptostrength of the Blum Blum Shub generator is discussed. Various Visual Basic codes to aid the choice of parameters necessary for the design of such a generator are designed. In the end, a source code is designed for the BBS RNG, and the output for different choice of parameters analyzed. The source code is compiled and can be installed on any cryptosystem.

LIST OF FIGURES

TABLE OF CONTENTS.

CHAPTER ONE.

1.1.0 INTRODUCTION.

The requirements of information security within an Organization have undergone two major changes in the last several decades. Before the wide spread use of data processing equipment, the security of information felt to be valuable to an organization was provided primarily by physical (e.g. the use of rugged filing cabinets with a combination lock for storing documents that are sensitive) and administrative (e.g. personal screening procedures used during the hiring process) means.

With the introduction of the computer, the need for automated tools for protecting files and other information stored on the computer become evident. This is especially the case of a shared system, such as a time – sharing system, and the need is even more acute for systems that can be accessed over a public telephone or data network. The generic name for collection of tools designed to protect data and to thwart hackers is computer security [2].

The second major change that affected security is the introduction of distributed systems and the use of network and communication facilities for carrying data between terminal user and computer, and between computer and computer. Clearly, security is required in any environment where information, data or items are not intended to be freely available to all during their transmission.

Today, security systems are built on increasingly strong cryptographic algorithms that foil pattern analysis attempts. However, at the heart of all cryptographic systems is the generation of secret, Unpredictable (i.e. random) numbers [17, 2]. In other words good cryptography requires good random numbers. For example, random number generators are required to generate public/private key pairs for asymmetric (public key) algorithms including the Rivest – Shamir – Adelman (RSA) [17, 3], Digital signature Algorithm (DSA) [17,4,] and Diffie – Hellman [17,4] . Keys for symmetric and hybrid cryptosystems are also generated randomly.

Random number generators are easily over looked and can thus become the weakest point of a cryptosystem. For any chain is only as strong as its weakest link [17]. Even a strong confusion generator can made irrelevant if the system supports only a small number of keys, since simply trying all the keys (a key search attack), would be sufficient to penetrate such a cipher. Thus any real system based on keys must support enough keys to prevent this attack and that is an issue for the random number generator.

Actually Random numbers find frequent applications in several other fields of life. Researchers use random numbers for tackling a wide range of problems [5,15,3]. From modeling molecular behavior and sampling opinion to solving certain equations and testing the efficiency of algorithms. Such numbers also play crucial roles in a wide variety of games, including electronic versions of slot machines, lotteries, and other forms of gambling. [16, 15, 3, 5]. Ivars [15] and Knuth [3] present good materials on the History and advancement of such random numbers. But of particular interest in this study is the application of random numbers in cryptography.

Since security protocols rely on the unpredictability of the keys they use, random numbers for cryptographic applications must meet stringent requirements. The most important requirement is that attackers, including those who know the random number generator design, must not be able to make any meaningful predictions about the random number generator outputs. D.Eastlake, S.Croker and J.Schiller [8] makes a good recommendation of such stringent requirements.

Unfortunately, there are so many random number generator designs and so many claims for them [3, 18, and 20], that it is difficult even to compare the claims, let alone the designs. The most common approach is to compute random numbers by means of an algorithm or a formula. These normally result in what is commonly named pseudo-random numbers. As a matter of fact, in the past, the Random number generation was mostly done by software mostly based on those algorithms. The resulting sequences from such systems typically don't meet all the criteria that establish randomness. Patterns often still remain in the sequence. After all, the computer simply follows a set procedure to generate the numbers, and restarting the process produces the same sequence. Moreover, the sequences eventually begin by repeating themselves of course.

However, as digital systems becomes faster and denser, it is possible, and sometimes necessary, to implement the generator directly in Hardware. Serious research is carried out in increasing measure on a truly (real) random source of data for random (unpredictable) numbers, such as values based on radioactive decay, atmospheric noise, thermal electrical noise and a fast, free running oscillator etc. This is the hardware approach to random number generation which is beyond the scope of this work.

Today, some computers have a hardware component that functions as real random value generator [8, 14]. However, most computers still don't have a hardware that generates random numbers that is sufficiently random that an adversary cannot predict. This study present miscellaneous thought on random number generation.

What follows in Chapter Two is a background discussion of theories that establishes randomness and the kind of randomness requirement for secure data communication.

Chapter Three is a look at some of the existing random number generators. Their cryptographic strength and weaknesses are considered.

In chapter four, the Blum Blum shub (x^2 MOD N) generator that is claimed to be cryptographically strong is designed and implemented using Microsoft Visual Basic version 6.0.

Chapter five is the summarization, with the conclusions presented and recommendations for further research.

CHAPTER TWO

2.1.0 BASIC DESCRIPTION OF PHYSICAL DATA

Any observed data representing a physical phenomenon can be broadly classified as being either deterministic or nondeterministic(stochastic). Deterministic data are those that are describable by an explicit mathematical relationship. There are many physical phenomena in practice which produce data that are represent able with reasonable accuracy by explicit mathematical relationship e.g. the motion of a satellite in orbit about the earth, the potential across a condenser as it discharges through a resistor, the vibration response of an unbalanced rotating machine, or the temperature of water as heat is applied, are basically deterministic. However, there are many other physical phenomena which produce data that are not deterministic e.g. the height of waves in a confused sea, the acoustic pressure generated by air rushing through a pipe, or the electrical output of a noise generator, represent data which cannot be described by explicit mathematical relationships. There is no way to predict an exact value at a future instant of time. These data are "random" in character and necessarily are described in terms of probability statements and statistical averages rather than by explicit equations

In practical terms, the decision as to whether or not physical data are deterministic or random is usually based upon the ability to reproduce the data by controlled experiments. If an experiment producing specific data of interest can be repeated many times with identical results (within the limits of experimental error), then the data is generally considered deterministic, otherwise, the data is usually considered "random" in nature. In essence, a random physical Phenomenon is indescribable by an explicit mathematical relationship as each observation of the phenomenon is unique.

2.1.1 RANDOMNESS DEFINED.

The Dictionary meaning of the term random is that which is done, chosen, etc without method or conscious choice. This points to haphazardness i.e. that which is pattern less.

According to Thierry Moreau [5], there are many subjective perceptions about the term "randomness" but the exact definition has been debated depending on the application. Actually, it appears there can never be a straightforward definition of a random process. This is because, if a finite set of random events could be defined, then at least in principle, it could be built (or chosen) such as to comply with the definition. If so, it would not be random, because it would obey a certain rule used to build it. Therefore, the definition of randomness is intrinsically not possible.

In practice, this result in pseudo – definitions which all suffer from various hidden logical and functional defects. Some of these are circular, that is they use the term "random" or something perceptive equivalent or similar in order to define the term "randomness". For example Ritter [7] define randomness as "an attribute of the process which generates or selects "random" number rather than the number themselves. Other definitions merely emphasize one or more statistical property of true random numbers.

As an exercise, an attempt is made to define a black-box random number generator, which produces random bits. Random bits are events that may have only two possible outcomes: "0" and "1". One practical realization of such events is by tossing a fair coin, heads denotes "1", and tails denotes "0". One could define "a sequence of bits as random if there is an equal chance that the next bit will be 0 or 1". This is the classical probability approach as seen shortly. This appears an illogical definition since for example, the obviously non-random sequence

01010101010101 -----------------------

Satisfies the above definition. Clearly it followers a particular pattern of 01 repeated 7 times

An improvement to the definition is to add "- - - and there is no way to predict the next bit in a sequence". This improved definition shows no objective intelligence, as somebody very stupid would still be unable to predict the next bit even in the above example whereas a clever person would be able to predict a more complicated sequence, which has some internal structure. Actually, these definitions really want to claim that someone infinitely intelligent should not be able to predict the truly random sequence.

A further improved view says a sequence of m bits(i.e m-tuples) is random if the number of 1s in the sequence is distributed according to binomial distribution with average value of m/2 and variance $m^2/4$, for m > 0". Looks better, but "distributed according to binomial distribution" is a practical issue that cannot be addressed in an experimentally verifiable way without setting some artificial, arbitrary criterion which define when a distribution is followed and when it is not. Still this "definition" is just making sure that this particular type of statistical property would be satisfied, but there are many others, which may still be not.

One other way of improving the definition would be to consider a sequence of m bits coming out of the process. The process is random if this m bits thought of as being integer numbers in binary notation, are distributed uniformly, for any m > 0. Again the problem of distribution arises. Imagine the "coin flipper" gives the following sequence of 7 – tuples: - 0,1,2,3,4,5,6, - - -, 126,127,128,0,1,2,3, - - - then, it satisfies the above definition at least for m = 7 (But it does for other m's as well). The numbers from 0 – 128 form a nice "Flat" uniform distribution, but the problem is that they appear in an ascending ordered sequence. Then comes the temptation to add a requirement that this "uniformly distributed" numbers should appear at random. But that is exactly what is not allowed when defining the term random.

In essence, where the magic word "random" is not used to define randomness, an infinite definition that tries to eliminate all statistical defects thought of result.

2.1.2 CLASSICAL PROBABILISTIC PERSPECTIVE OF RANDOMNESS

From the perspective of classical probability, any sequence of equally – probable events is equally likely and thus equally "random" [E.g. 3, 18]. The sequence 1001 and 0000 or 1111 are equally probable with the probability of any of four bit outcome being 1/16, because there is 16 possible combinations namely; (0000,0001,0010, - - -, 1111). Thus if origin in a probabilistic event were made the sole criterion of randomness, then both series would have to be considered random; and indeed so would all others, since the same mechanism generate all possible series.

The classical definition above allows one to speak of a process, such as the tossing of coin as being random. It does not allow one to call a particular outcome or string or sequence of outcome like obtaining ten heads in a row with twenty tosses of a fair coin, random. Ashish[24] gives the classical probability notions of randomness based on Shannon's concept of entropy.

Deviating a little from this classical probability definition of randomness, a more sensible definition, which focused on the individual outcomes rather than on the generating process of strings, is considered. It established hierarchy of degrees of randomness. The limitation that the definition cannot help to determine, except in a very special cases whether or not a given sequence is random is closely related to Kurt Gödel's incompleteness theorem devised and proved in 1931 [23,7,24]

2.1.3 ALGORITHMIC PERSPECTIVE.

This new definition of randomness has its heritage in information theory dating black to World War II [23]. More often situations arise of providing as input a program, or string of instructions to a computer so that it produces a desired string as output. Such instructions given to computer must be complete and step without requiring that it comprehend the result of any part of the operations it performs.

The possibility of reducing redundancy by compressing such input string will mean a shorter sequence can serve as a code to reproduce the original string. Gregory[23] and Ashish[24], provides a practical example of compressible and incompressible data to be transmitted to a friend in another galaxy. A result of this is the canonical definition of randomness based on incompressibility, which was proposed independently about 1965 by A.N. Kolmogorov of the Academy of science of the U.S.S.R and G.J. Chaitin and stated thus;

"A series of number is random if the smallest algorithm capable of specifying it to a computer has about the same number of bits of information as the series itself."

It is on the basis of this definition that the two series of digits presented below are examined;

10101010101010101010

01100010011001110010

Whereas based on the classical probabilistic notions the two series can be considered "random", it is observed that the first sequence consist of patterns of 10 repeated ten times and as such could be specified to a computer by a very simple algorithm, such as "Print 10 ten times" Extending the series by following the same pattern, might just change the programmer to say, "Print 10 a thousand times." The number of bits in such an increased algorithm is a small fraction of the number of bits in the series it specifies, and as the series grows larger the size of the program increases at a slower rate

The second series of binary digits generated by flipping a coin 20 times and writing a "1" when the outcome was heads and a "0" when it was tails appears pattern less. There is no shortcut to reproduce it, as the shortest most economical algorithm for introducing the series into a computer would be "print 01100010011001110010." Which means necessarily the algorithm has to be expanded to the corresponding size of a much longer pattern less series. This incompressibility is a property of all random numbers [23, 24]

2.1.4 MINIMAL PROGRAM, ALGORITHMIC COMPLEXITY & RANDOMNESS

Ashish [24] defined the algorithmic complexity of an object as a measure of the difficulty of specifying that object. Based on Alan Turing's work, which distinguish between computable and non-computable numbers, it is possible to characterize different numbers by the length of the program required to compute them. A relatively short program can be written for computable numbers even if they are infinitely long. For example, the rational number 2. 4494897427831780 - - - has an infinitely long number of digit that never repeats. By recognizing the number as the square root of 6 expressed in decimal form, an algorithm to compute this number becomes "compute square root of six and print the results" this is relatively short. For the non-computable, random numbers, the only algorithm that can describe the number is about as long as the number.

Closely linked (related) to the concept of complexity is the minimal program to generate a series of numbers [24]. There is actually an infinite number of algorithms to generate any number but of particular interest here is the minimal program (the smallest one) that yield a given numerical series. For example, the number 144 can be obtained from the algorithms; "Add 2 to 142". "Multiply 12 by 12", "subtract 4 from 148", "divide 288 by 2", or an infinite number of other programs. For a given series, however, there may be only one minimal program or there may be many.

'A random series of digits is thus one whose complexity is approximately equal to its size in bits. Chaitin[23] and Ashish[24] explains the use of the notion of complexity to measure randomness. On what degree of randomness really constitutes randomness, Ashish[24] suggests the value to be set low enough so that numbers with obviously random properties are not excluded and high enough so that numbers with conspicuous patterns are disqualified.
Summarily, Randomness of a string of numbers can be understood through three points of view ;

i) A string is random because it is completely unexpected; thus its entropy is maximal.
ii) A string is random if each number in the string is generated by an unpredictable process. This randomness stems from the disorder in the generating process.
iii) A string is random because no prescribed program of shorter length can generate its successive digits.

Accordingly, randomness implies the absence of any compression possibility. Thus, the string has maximum information content. Being maximally complex in algorithmic sense, the string can only be reproduced by explicitly specifying the string itself.

This understanding of the term randomness leads to the review of some existing random number generators form the next chapter and the dependence of the numbers they produce on the generating processes.

CHAPTER THREE

3.1.0 RANDOM NUMBER SOURCES.

3.1.1BACKGROUND

Most "random" number sources actually utilize a PseudoRandom Number Generator (PRNG). A PRNG is an algorithm with some state information that is the sole input. The generator is exercised by steps, and two things occur consistently during each step. There is a transformation of the state information, and the generator outputs a fixed size bit string. The generator seed is simply the initial state information. In other words it is a deterministic algorithm which, given a truly-random binary sequence of length n, outputs a binary sequence of length k(n) > n which appears to be random, k() being a polynomial.

With any PRNG, after a sufficient number of steps, the generator comes back to some sequence of states that was already visited before (a cycle). Then, the period of the generator is the number of steps required to do one full cycle through the visited states. The actual entropy (unpredictability) of the output can never be greater than the entropy of the seed.

Ari J., Makin J. ,Elizabeth S. & Bruce K.H. [1] referenced a large body of literature on the design and properties of PRNGS. Knuth [3] has a classic exposition on pseudo-random numbers. Applications he mentions include simulation of natural phenomena, sampling, and numerical analysis, testing computer programs, decision making, aesthetics and recreation. None of these have the same characteristics as the sort of security uses in cryptography. Only in the last three could there be an adversary trying to find the random quantity in use.

Thierry Moreau [5], classified random number generators based on their applications to include;

- The toy generators that are provided by most programming language, and many software pages, which he said, are to be considered suspicious for most serious application;
- The serious generator, that is generators with internal state information using at least 64 bits and with empirical and /or theoretical justification and;
- The truly random generators. i.e. electronic circuits that employ measurements of a natural phenomenon to provide totally unpredictable draws.

A close examination of a few most popular pseudo-random number generators is what follows in the rest part of this chapter.

3.2.0 LINEAR CONGRUENTIAL GENERATORS (LCG).

The Linear Congruential Generator (LCG) is by far the most widely used technique for random number generation. The LCG is one of the oldest, and still the most common type of RNG implemented for programming language commands (e.g. the standard rand ( ) function for C and ANSI C used by VAX C, RANDU introduced by IBM in 1963, MTHSRANDOM used by VAXFOTRAN, and VAX Basic etc). These are all built of LCG algorithm. David W. D. [18] presents a good description of techniques for analyzing output of such RNGs.

The LCG algorithm is parameterized with four numbers, as follows:

m	the modulus	$m > 0$
a	the multiplier	$0 \leq a < m$
c	the Increment	$0 \leq c < m$
V_0	the starting value or seed	$0 \leq V_0 < m$

It is a modular arithmetic where the (n + 1) th value is obtained via the following iterative equation:

$V_{n+1} = (a \cdot V_n + c) \bmod m$

If m, a, c and V_0 are integers, then this technique will produce a sequence of integers with each integer in range $0 \leq V_0 < m$.

The selection of values for a, c and m is critical in developing a good LCG random number generator. For example with m = c = 1, the sequence produced is obviously not satisfactory

Now with

a=7

c=0

m=32 and V_0 =1, this generates the

Sequence {1, 7, 17, 23, 1, 7, etc}. i.e.

V_1=7(1) mod 32=7

V_2=7(7) mod 32=17

V_3=7(17) mod 32=23

V_4=7(23) mod 32=1

V_5=7(1) mod 32=7 - - - - - which is also unsatisfactory. The reason is that of the 32 possible values only four is used; thus the sequence is said to have a period of 4.

A change in the value of "a" to 5, result in the following sequence; {1,5,25,29,17,21,9,13,1,5,- - - - -}, which increases the period to 8.

A larger m results in an increased potential for producing a long series of distinct random numbers. A common criterion is that m is nearly equal to the maximum represent able nonnegative integer for a given computer [2, 18]. This value of m near to or equal to 2^{31} is typically chosen.

William Stallings [2] proposes three criteria to use in evaluating a random number generator, Viz a Viz;

X_1: That the function is a full period generating function. That is, the function should generate all the numbers between 0 and m before repeating.

X_2: That the generated sequence appears random. The sequence is not purely random because it is generated deterministically, but there is a variety of statistical test that can be used to asses the degree to which a sequence exhibit randomness. Knuth[3] Suggests a variety of measures including statistical and spectral tests. These tests check things like autocorrelation between different parts of a "random" sequence or distribution of its value, which according to D. Eastlake [8] could be met by a constant stored random sequence, such as the "random" sequence printed in the CRC standard mathematical table.

X_3: that the function implement efficiently with 32 – bit arithmetic.

With appropriate values of a, c, and m, these tests can be passed. With respect to X_1, it can be shown that if m is prime and c = 0, then for certain values of a, the period of the generating function is m -1, with only the value 0 missing. For 32 –bit arithmetic, a convenient prime value of m is 2^{31} -1. Thus the generating function becomes:

$$V_{n+1} = (a \cdot V_n) \bmod (2^{31}-1).$$

This is the approach of Guinier as cited by Ritter [7] to mean making LCG design almost trivial as this assures that the result is always computable without overflow. Actually the strength of the linear congruential algorithm is that if the multiplier and modulus are properly chosen, the resulting sequence of numbers will be statistically indistinguishable from a sequence drawn at random (but without replacement) from the set 1,2, - - - - -, m-1. But there is really nothing random at all about the algorithm, apart from the choice of the initial value V_0. Once that value is chosen, the remaining numbers in the sequence follow deterministically. This of course has implication for cryptanalysis.

The real problem with most LCGs is their tiny amount of internal state, simple step formula, and complete exposure. If an enemy knows that LCG is being used, and if the parameters are known (e.g.) a =1024, c = 0, m = 2^{31} -1), then once a single number in the series is discovered, all subsequent numbers are known. Even if a few values from the sequence becomes available for analysis, the formula (i.e. the parameters of the algorithm) can be deduced, the sequence reproduced, and the cryptosystem penetrated [7].

For example, suppose an enemy is able to determine values for V_0, V_1, V_2 and V_3. Then:

$V_1= (a \cdot V_0 +C) \bmod m$ - - - - - - - -(1)

$V_2= (a \cdot V_1+C) \bmod m$ - - - - - -- -- (2)

$V_3= (a \cdot V_2 +C) \bmod m$ - - - - - - - - (3)

These three simultaneous equation can be solved for the three unknowns a, c, and m. Consequently, LCGs can not be considered secure, unless, strongly isolated or perhaps combined with other generators [18, 7].

To make the actual sequence used "non– reproducible", so that the knowledge of part of the sequence on the part of an opponent is insufficient to determine future elements of the sequence Bright H.& Enison R. [9] suggest using an internal system clock to modify the random number stream. One way to use the clock would be to restart the sequence after every N numbers, using the current clock values (mod m) as the new seed. Another way would be to simply add the current clock value to each random number (mod m) [2].

But designing such portable application code to generate unpredictable numbers based on such system clocks is particularly challenging because the system designer does not always know the properties of the system clocks that the code will execute on . So if the code is to be employed across a variety of computer platforms and systems, it becomes a problem.

Because it is difficult to find a good set of LCG parameters, LCGs are normally difficult to customize – Not only have linear congruential generators been broken, but all polynomial congruent generators, such as quadratic generators and cubic generators, have also been broken [8,4].

3.3.0 LINEAR FEEDBACK SHIFT REGISTER (LFSR).

A feedback shift register consist of an ordinary shift register made up of m flip-flops (two- state memory stages) and a logic circuit that are interconnected to form a multiloop feedback circuit. The flip-flops in the shift register are synchronously clocked. At each pulse of the clock, the state of each flip – flop is shifted to the next one down the line. With each clock pulse the logic circuit computes a Boolean function of the states of the flip- flops. The result is thereby fedback as the input to the first flip – flop, there by preventing the shift register from emptying (figure 3.0). The sequence so generated is determined by the length m of the shift register, its internal state, and the feed back logic [10, 11].

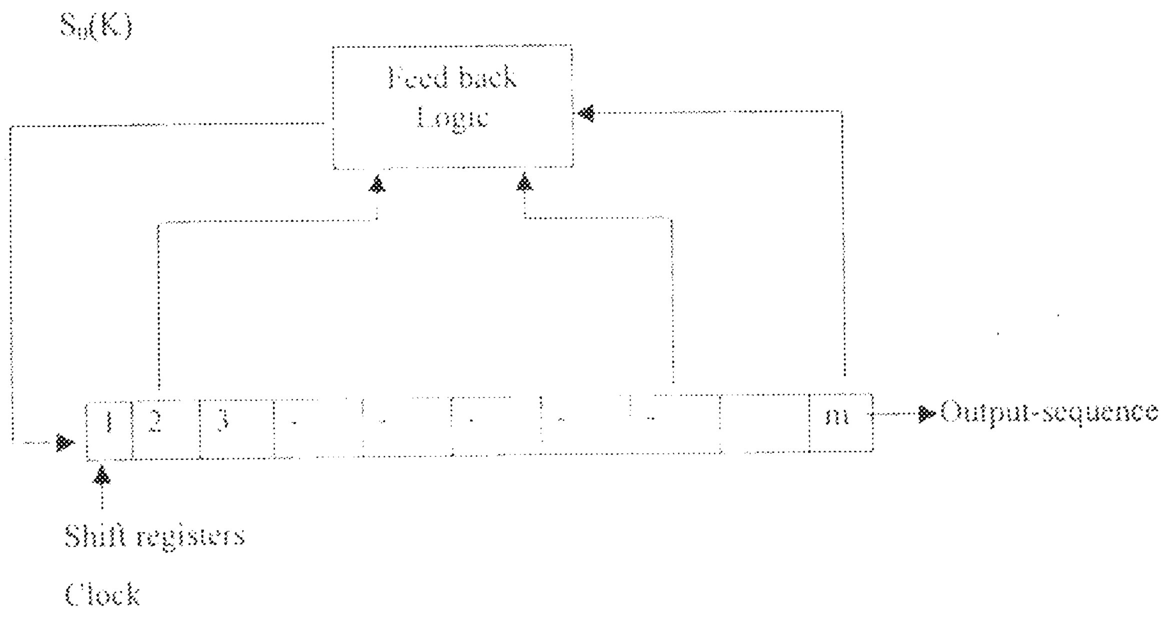


Figure 3.0 LFSR Pseudorandom number generators.

Based on the configuration described in figure 3.0 above $S_0(K)$ is a Boolean function of the individual states $S_1(K)$, S_2 (K), - - - - - , S_m (K). For a specified length m, this Boolean function uniquely determines the subsequent sequence of state and therefore the sequence produced at the output of the final flip-flop in the register. With a total of m flip-flops, the number of possible state of the shift register is at most 2^m.

A feedback shift register is said to be linear when the feedback logic consists entirely of modulo – 2 adders. (Modulo-2 addition is explained in [13]. In such a case the zero state (e.g. the state for which all the flip – flops are in state 0) is not permitted. (The all zero's state will lock up in a degenerate cycle.) . As a result a maximum – length sequence of 2^m - 1 is produced by a LFSR.

Maximal – length sequences have many of the properties possessed by a truly random binary sequence. But a maximal length sequence will occur if the shift – register "taps" from a polynomial that is primitive [7]. (A primitive is a special kind of irreducible prime of a given finite field). The statistical performance of an LFSR with a primitive feedback polynomial really is quite interesting. Observation of bits produced by the feedback presented in Horowitz & Hill [12] shows that;

i. In each period of a sequence, the number of 1s is always one more than the number of 0s. Which implies 1s or 0s are almost equally likely.

ii. Among the runs of 1s and of 0s in each period of a sequence, one half the runs of each kind are of length one, one – fourth are of length two, one –eight are of length three, and so on as long as these fractions represent meaningful numbers of runs. By a "run" we mean a subsequence of identical symbols (1s or 0s) within one period of the sequence. The length of this subsequence is the length of the run. So each of these sequence is also equally likely; and,

iii. There is only a period of length (2m- 1) steps. The all –zeros state is an isolated and degenerate cycle.

With this understanding, the design of a maximal – length sequence generators reduces to finding the feedback logic for a desired period. The task is made particularly easy by virtue of the extensive tables of the necessary feedback connection for varying shift – register length that have been computed in the literature. [11, 12].

As the length of shift register m, or equivalently, the period of the maximal sequence is increased, the sequence becomes increasingly similar to the random binary sequence [11]. Indeed in the limit, the two sequences become identical when m is made infinitely large. However, the price paid for making m large is an increasing storage requirement, which imposes a practical limit on how large m can actually be made. But of course, the LFSR can be made arbitrarily large (and thus more difficult to solve), and is also easily customized. Yet Ritter [7] recommends additional isolation in cryptographic use.

The mathematical basis for these sequence was described by Tauworthe and is cited by Ritter [7] as being a "linear recursion relation".

a(k) = c (1) * a [k-1] + c(2) * a[k-2] + - - - + c(m)* a[k-m] (mod 2) where a(k) is the latest bit in the sequence, c the coefficients or binary numerical factors a mod 2 polynomial, and m the degree of the polynomial and thus the required number of storage elements (Flip – flops). This formula is particularly interesting, for relatively minor generalization of the same formula produced the Generalized Feedback Shift Register (GFSR) and additive generators described in [7].

3.4.0 CRYPTOGRAPHYCALLY GENERATED RANDOM NUMBERS.

For cryptographic applications, it makes sense to take advantage of the encryptions logic available to produce random numbers. A number of means have been used, and a few representative examples are considered below.

3.4.1 CYCLIC ENCRYPTION.

Figure 3.1 illustrates an approach suggested in [25]. In this case, the procedure is used to generate session keys from a master key. A counter with period N Provides input to the encryption logic. For example, if 56 – bit DES keys are to be produced, then a counter with period 256 is used. After each key is produced, the counter is incremented by one. Thus the pseudo-random numbers produced by this scheme cycle through a full period: Each output V_0, V_1, - - - - -V_{N-1} is based on a different counter value, and therefore $V_0 \neq V_1 \neq$ - - - $\neq V_{N-1}$.

Since the Master key is protected, it is not computationally feasible to deduce any of the secret keys through knowledge of one earlier key.

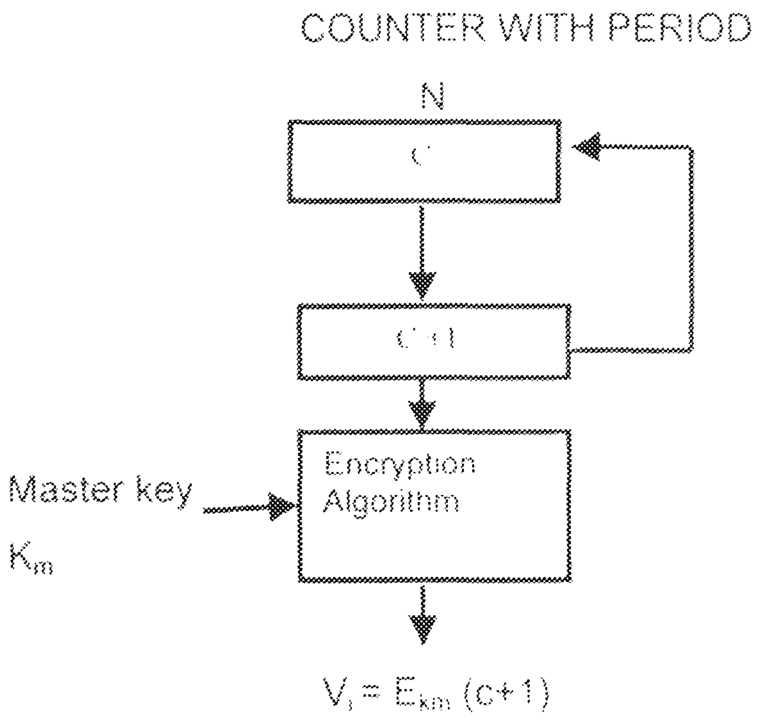


Figure 3.1: pseudorandom number generation from a counter.

A further way to strengthen the algorithm will be to take the input from the output of a full period pseudorandom number generator, rather than a simple counter.

3.4.2 DES OUTPUT FEEDBACK MODE

During 1968-1975, IBM developed a cryptographic procedure that enciphers a 64 – bit block of plaintext into a 64 –bit block of cipher text

under the control of a 56-bit key. The National Bureau of Standards (now the National Institute of standards and Technology (NIST) accepted this algorithm as a Federal Information Processing Standard 46 (FIPS PUB 46) and it became effective on July, 15 1997 [13, 2].
The algorithm for this most widely used encryption scheme; the Data Encryption Standard (DES) is described in detail in [8, 13, 2, and 11]. Stallings [2] presents the permutation table for the scheme. The four operation modes defined for different applications of DES are; Electronic codebook (EBC), Cipher Block chaining (CBC), Cipher feedback (CFB) and the output feedback (OFB) modes.

The output feedback (OFB) mode of DES illustrated in Figure 3.2 below can be used for key generation as well as for stream encryption. In the figure, it is assumed that the unit of transmission is n bits; a common value is n = 8 [13]. The input to the encryption function is a 64-bit shift register that is initially set to some initialization vector (IV).

It is seen that the output of each stage of operation is a 64-bit value, of which the n leftmost bits are fed back for encryption. Successive 64-bit outputs constitute a sequence of pseudo-random numbers with good statistical properties [13,11]. Here also, the use of a protected master key protects the generated session keys.

A particular implementation of the cipher feedback (CFB) mode recommended by the national bureau of standards for the generation of cryptographic bit stream is shown in [13]. Deley [18] claims the implementation of this scheme in software is too slow.

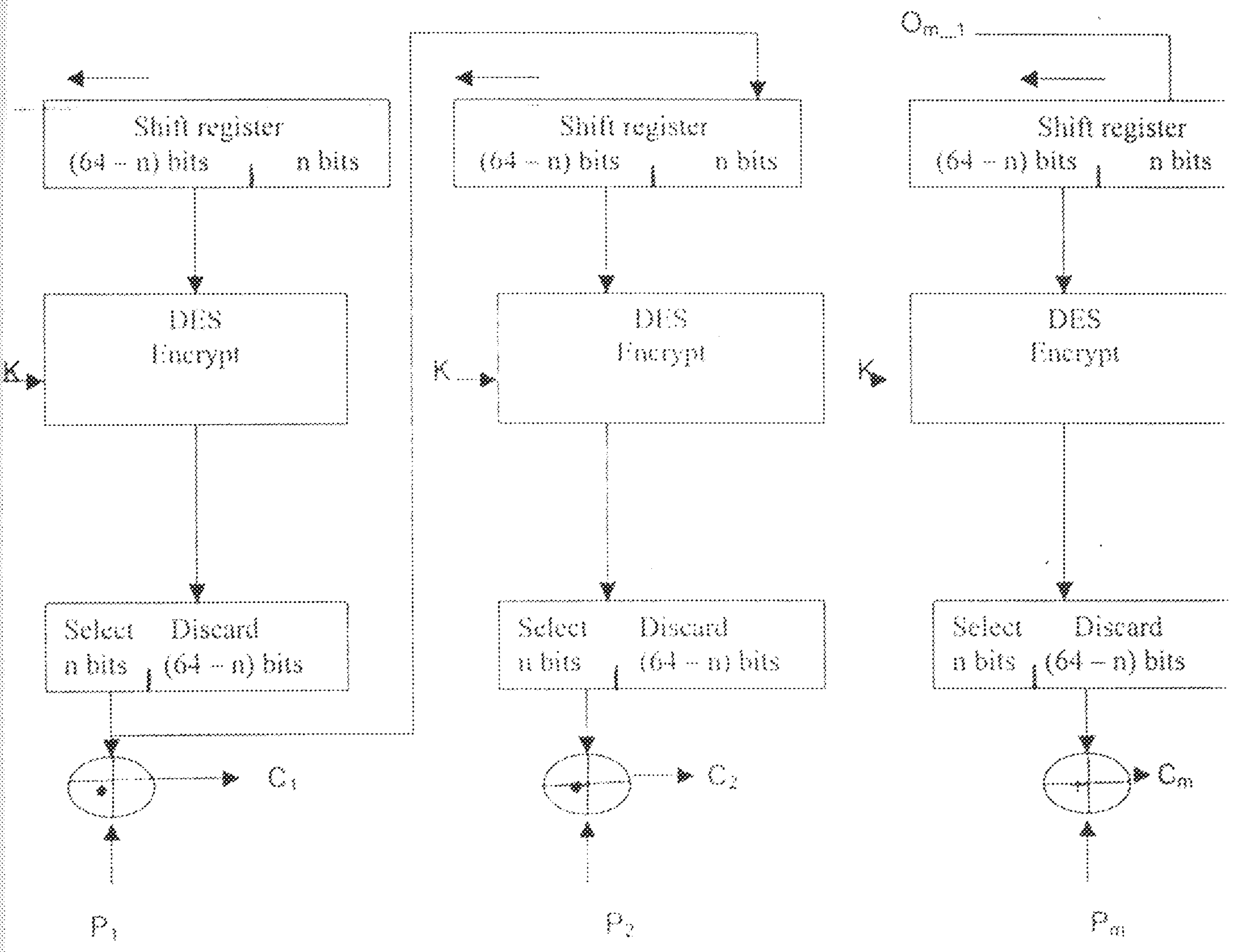


Figure 3.2: n bit output feedback (OFB) Mode.

(C_1 is the result of X0Ring the left most (most significant) n bit of the output of the encryption with the first unit of plaintext.)

3.4.3 ANSI X9.17 PSEUDO-RANDOM NUMBER GENERATOR.

Specified in the ANSI financial institution key management standard ANSI X9.17, is one of the strongest (cryptographically speaking) pseudo-random number generator [8]. A number of applications employ this technique, including financial security applications and Pretty Good Privacy (PGP). PGP is the effort of Phil Zimmer man described in [2] figure 3.3 illustrates the algorithm, which makes use of triple DES for encryption. The scheme involves;

- **Input**: Two pseudorandom inputs that drive the generator. One is a 64-bit representation of the current date and time, which is updated on each number generation. The other is a 64-bit seed value; this is initialized to some arbitrary value and is updated during the generation process.
- **Keys**: The generator makes use of three triple DES encryption modules. All the three make use of the same pair of 56-bit keys, which must be kept secret and are used only for pseudorandom number generation.
- **Output**: The output consists of a 64-bit pseudorandom number and a 64-bit seed value.

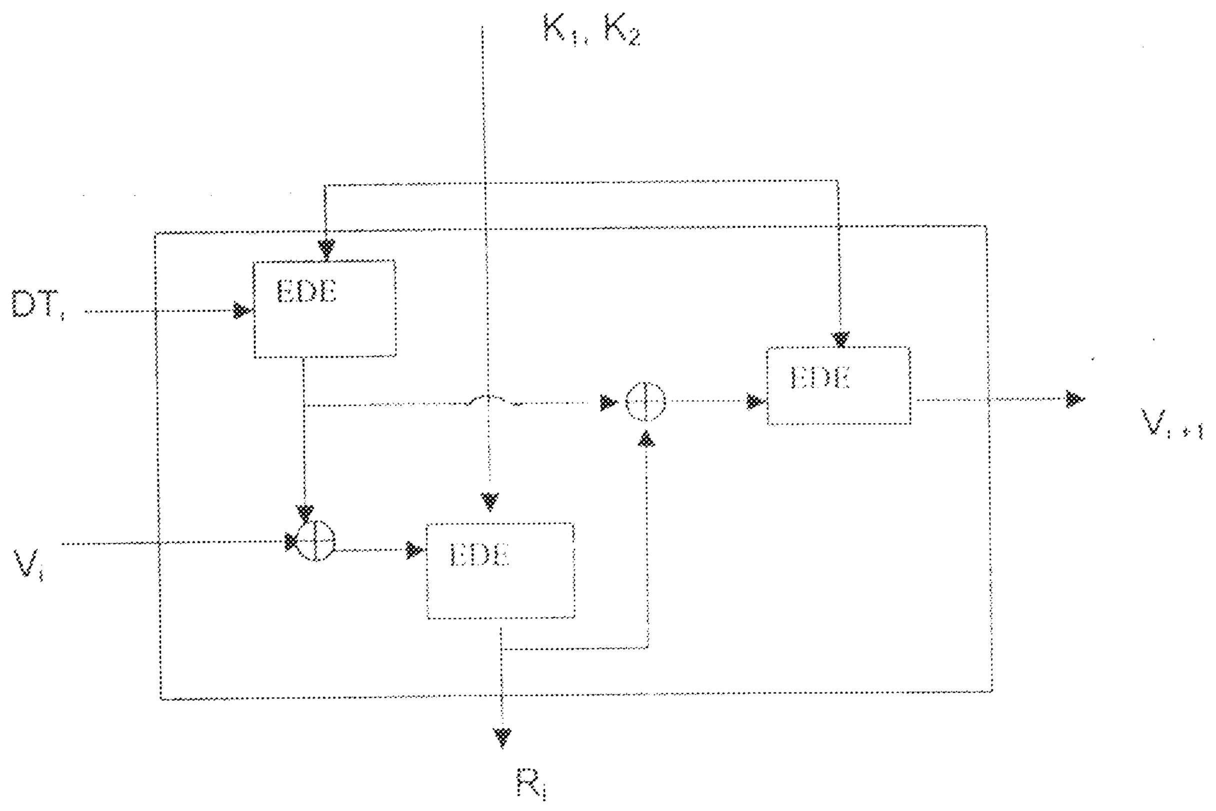


Figure 3.3 : ANSI X9. 17 PSEUDO –RANDOM NUMBER GENERATOR

By definition;

DT_i is the Date & Time value at the beginning of i th generation stage;

V_i is the seed value at the beginning of i th generation stage;

R_i the pseudorandom number produced by the i th generation stage;

K_1, K_2 are the DES keys used for each stage .

Therefore;

$R_i = EDE_{K1,K2}\,[EDE_{K1,K2}\,[DT_i] \oplus V_i]$

$V_{i+1} = EDE_{K1,K2}\,[EDE_{K1,K2}\,[DT_i] \oplus R_i]$

The cryptographic strength of this method depends on several factors. The technique involves a 112-bit key and 3 Encrypt- Decrypt – Encrypt (EDE) [2] encryptions for a total of 9 DES encryptions. The scheme is driven by 2 pseudorandom inputs, the Date and time value, and a seed produced by the generator that is distinct from the pseudorandom number produced by the generator because an additional EDE operation is used to produce the V_{i+1}, the knowledge of R_i is still not enough to produce the V_{i+1}, from R_i. Thus the amount of material that must be compromised by an opponent is overwhelming.

What follows in chapter four is the design and implementation of the Blum, Blum shub RNG which is claimed to be suitable for most cryptographic applications.

CHAPTER FOUR.

4.1.0 THE BLUM, BLUM SHUB RANDOM NUMBER GENERATOR

The Blum, Blum shub (BBS) generator is a random number generator named after its inventors, which is based on quadratic residues. The initial seed for the generator S (0) and the method for calculating subsequent values are based on the following iterations;

$S(0) = (X^2) \bmod N$

$S(i+1) = (S(i))^2 \bmod N$

Where N is the product of two large primes. X is chosen at random to be relatively prime to N and the output is the least significant bit of S(i) or the parity of S(i) . Or, the output can be several of the least significant bits of S (i) up to $\log_2(\log_2 N)$ bits[7,8,28].

This generator seems unique in that it is claimed to be "polynomial – time unpredictable" and cryptographically strong though not suitable for use in simulations because of its slow nature [7,28]. The BBS generator is not a permutation generator like some digital computer RNGs discussed in chapter 3.

All digital computer RNGs including the BBS necessarily repeat eventually, and may well include many short or degenerate cycles. Thus the generator requires some fairly – complex design procedures, which are apparently intended to assure long cycle operation. It has an unusually strong security proof, which relates the quality of the generator to the difficulty of integer factorization [28, 7, and 8].

Given two large prime numbers, it is easy to multiply them together. However, given their product, it appears to be difficult to find the nontrivial factors of a large integer. This is relevant for many modern systems in cryptography. If a fast method were found for solving the integer factorization problem, then several important cryptographic systems would be broken, including the RSA public-key algorithm, and the Blum Blum Shub random number generator.

Although fast factoring is one way to break these systems, there may be other ways to break them that don't involve factoring. So it is possible that the integer factorization problem is truly hard, yet these systems can still be broken quickly. A rare exception is the Blum Blum Shub generator. It has been proved [29] to be exactly as hard as integer factorization. There is no way to break it without also solving integer factorization quickly.

If a large, *n*-bit number is the product of two primes that are roughly the same size, then no algorithm is known that can factor the number in polynomial time. When the primes are chosen appropriately and care is taken that only a few lowest order bits of each S (i) are output, then in the limit as N grows large, distinguishing the output bits from random will be at least as difficult as factoring N.

4.2.0 DESIGN REQUIREMENTS FOR THE BBS GENERATOR

The basic BBS requirement for $N = P \cdot Q$ is that P and Q each be primes congruent to 3 mod 4 (this guarantees that each quadratic residue has one square root which is also a quadratic residue) [28]. This looks exceeding easy. But to guarantee a particular cycle length, there are two more conditions:

Condition 1 defines that a prime P is "special" if;

$P = 2\ P1 + 1$ and

$P1 = 2 \cdot P2 + 1$ where P1 and P2 are odd primes. Both P and Q are required to be "special".

The original BBS paper as cited by Ritter [7] gives 2879, 1439, 719, 379, and 89 as examples of special primes (but 179 and 89 appear not to be special while 167, 47, and 23 should be). Namely;

If P = 179 from the above condition

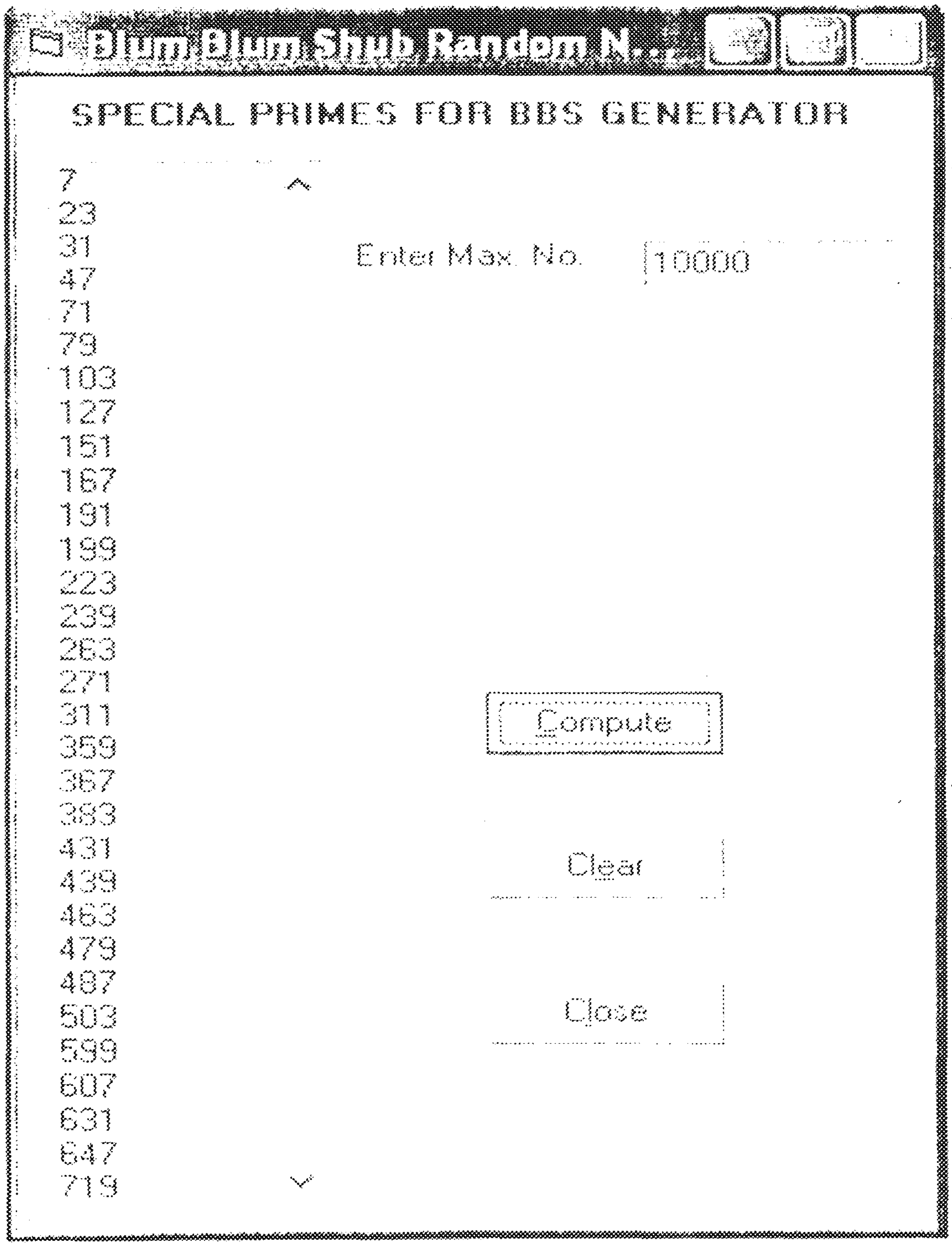
$179 = 2 \cdot P1 + 1$

Thus $P1 = 178 / 2 = 89$

$89 = 2 \cdot P2 + 1$ hence $P2 = 88 / 2 = 44$ Thus whereas 89 is an odd prime, 44 is not.

Similarly for $P = 89 = 2 \cdot P1 + 1 \Rightarrow P1 = 88 / 2 = 44$ which is not an odd prime.

Appendix two contains a module that generate primes that meet these requirements of P and Q being primes congruent to 3 mod 4 and being "special" primes. Figure 4.0 is the result of running the program.



```
SPECIAL PRIMES FOR BBS GENERATOR

7
23
31
47
71
79
103
127
151
167
191
199
223
239
263
271
311
359
367
383
431
439
463
479
487
503
599
607
631
647
719
```

10000

Figure 4.0 The BBS special primes.

Condition 2 says that, only one of P1, Q1 may have 2 as a quadratic residue (P1,Q1 are the intermediate values computed during the "special" certification). Appendix two also contains a module that generates such "particular" special primes. Accordingly, N = 719 . 47 fails this additional condition, because the intermediate values of both special primes have 2 as a quadratic residue.

Each of the special conditions provides additional structure in the generator, which presumably was needed for some aspect of mathematical proof. Currently, it is thought that the BBS special primes construction is sufficient, provided X is chosen not on a degenerate cycle, which is easily checked. On whether the special primes guarantee that the RNG will not have short cycles, Ritter [30] points out that with public key size special primes, the "short" cycles will either be "long enough" to use, or degenerate (i.e. single – cycle loops).

As an illustration, the BBS system of P = 23, Q = 47 (N = 108) is considered, a system which according to Ritter [7] was specifically given as an example of the prescribed form" in the original BBS paper. Starting with X = 46 ;

S (0) = $(46)^2$ mod 1081 = 1035

S (1) = $(1035)^2$ mod 1081 = 1035 --- a degenerate cycle.

With X = 47, 47 is repeated which is also a degenerate cycle.

Starting with X = 48, result in;

S (0) = $(48)^2$ mod 1081 = 142.

$S(1) = (142)^2 \bmod 1081 = 176,$

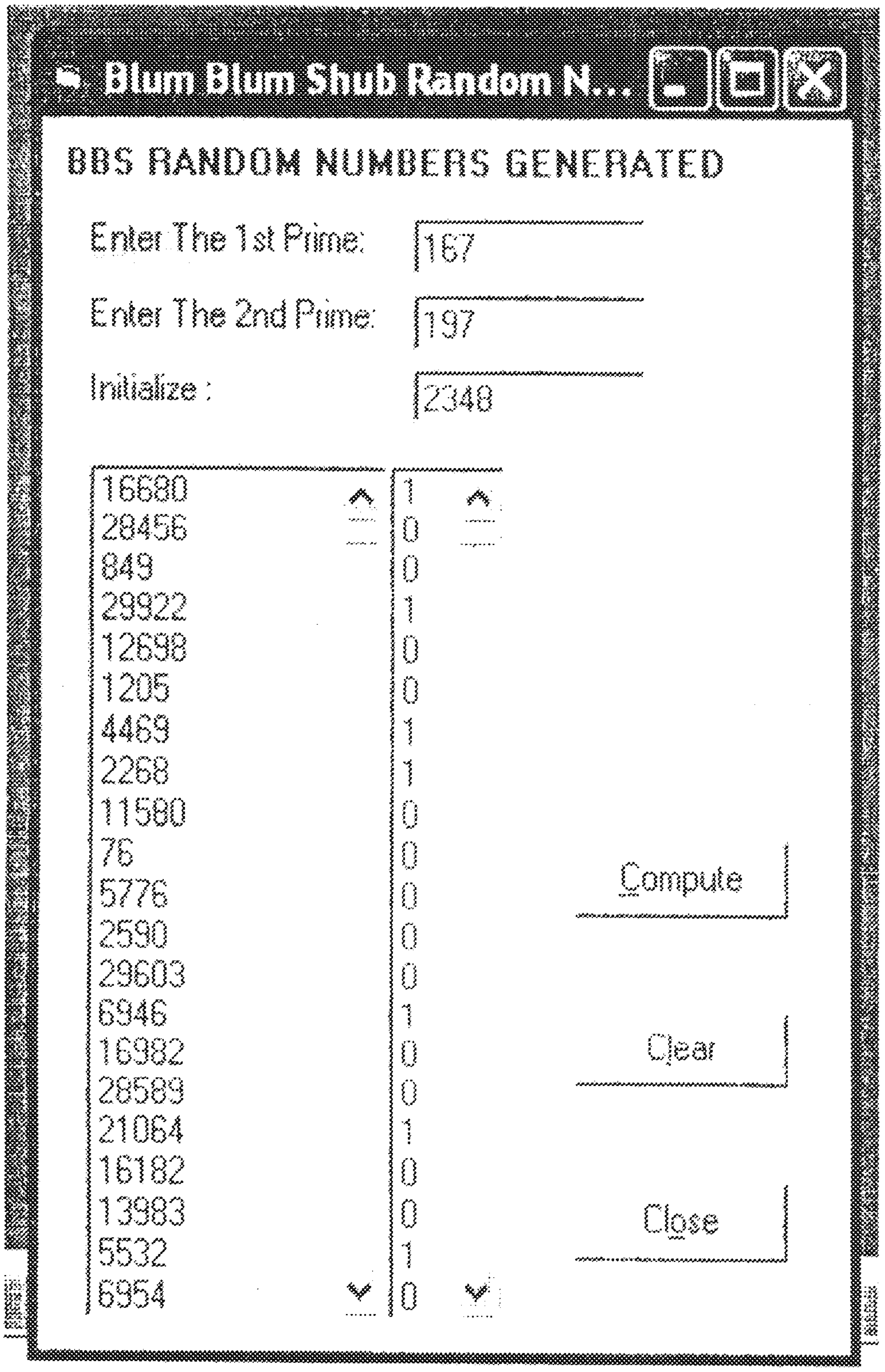
$S(2) = 95$, $S(3) = 377$, $S(4) = 518$ - - - and begin to repeat after the tenth cycle.

Because BBS generators generally define multiple cycles with various numbers of states, the initial value X must be specially selected as well. According to D. Eastlake, S. Crocker, J. Schiller [8] it must be relatively prime to N.

Appendix two shows source code for the BBS generator. The random bit returned is;

$F[S_{(0)}] = (Z_0, Z_1, \cdots, Z_t)$, where $Z_i = S_i$ MOD 2.

Figure 4.1 is a result of running the program with P = 167, Q = 197 and X =2348



Blum Blum Shub Random N...

BBS RANDOM NUMBERS GENERATED

Enter The 1st Prime: 167

Enter The 2nd Prime: 197

Initialize : 2348

```
16680   1
28456   0
849     0
29922   1
12698   0
1205    0
4469    1
2268    1
11580   0
76      0
5776    0
2590    0
29603   0
6946    1
16982   0
28589   0
21064   1
16182   0
13983   0
5532    1
6954    0
```

igure 4.1: Random numbers generated by BBS generator.

CHAPTER FIVE.

5.0 CONCLUSION.

A description of what randomness entails is given with the stringent randomness requirement in cryptography explored. A few random number generators were reviewed and a particular realization of the Blum Blum Shub generator undertaken.

The BBS construction is *not* a maximal length RNG, but instead defines a system with multiple cycles, including degenerate, short and long cycles. With large integer factors, state values on short cycles are very rare, but do exist. Short cycles are dangerous with any RNG, because when a RNG sequence begins to repeat, it has just become predictable, despite any theorems to the contrary. Consequently, if the BBS is keyed by choosing X at random, unknowingly a short cycle (a weak key) may be selected, which would make the sequence predictable as soon as the cycle starts to repeat.

BBS is said to be "proven secure" in the sense that if factoring is hard, then the sequence is unpredictable. And factoring large composite of public key size is thought to be hard. Yet when a short cycle is selected and used, BBS is obviously insecure. That of course is a direct contradiction to the idea that BBS is proven to be universally secure. Just knowing the length of a cycle (by finding sequence repetition) should be enough to expose the factors. This is evidence that the assumption that factoring is hard is not universally true. Of course, factoring is not hard -- when the factors are given away.

With the special prime's construction, apparently all "short" (but not degenerate) cycles are "long enough" for use. Thus, X is simply chosen at random, and then easily tested that it is not on a degenerate cycle. By getting some S(0), which is stepped to S(1), S(1) is saved and stepped to S(2). S(2) and S(1) are compared and if they are the same the process is repeated. The result is a guarantee that the selected cycle is "long enough" for use.

The BBS is very slow in comparison to other RNG's, thus selecting BBS RNG clearly implies a decision to pay a heavy price with the expectation of getting an RNG which is "proven secure" in practice. (That actually misrepresents the BBS proof, which apparently allows weakness to exist provided it is not an easy way to factor N; short cycles are very rare.) The obvious goal is to get a practical RNG which has no known weakness at all.

No mere proof guarantees protection when a weak key is chosen and used, even if doing that is shown to be statistically very unlikely. And if a weak key is used, the "proven secure" RNG is clearly insecure, which surely contradicts the motive for using BBS in the first place. In contrast, simply using the special prime's construction and checking for degenerate cycles can eliminate weak keys at modest expense. Eliminating a known possibility of weakness, even if that possibility is very small, seems entirely consistent with the goal of achieving a practical RNG with no known weakness, even if the result is not an RNG proven to have absolutely no weakness at all.

5.1 RECOMMENDATIONS.

Typical RNG's found in most computer libraries are not sufficiently strong for security purposes. Whereas hardware RNG's produce truly (real) random numbers, they are not easy to come by. So in most applications Pseudo- random number generators are used.

For a practical application of the BBS generator realized in this work, it is recommended to try a more random way (like using the system clock) of seeding the generator. Also, a hardware implementation of the algorithm may be considered for further research.

REFERENCES.

1. Ari J., Marcus J., Elizabeth S., and Bruce K. H., How to turn loaded dice into fair coins, IEEE Transaction on Information theory, 46(3), May 2000.
2. Williams S., Network and Internetwork security (Principles and practice), Prentice Hall Inc., 1995.
3. Knuth D., The art of computer programming, Semi numeric algorithms, Addison Wesley Publishing Company, 2nd Edition, Chapter 3, 1982.
4. Secure Programming for Linux and Unix HOWTO., Chapter 10, Special Topics.
5. Thierry M., Pseudo-Random Generators, a High level survey in progress, CONNOTECH Experts-Conseils Inc., March 1997.
6. An Overview of Random Number Generators, The Random Numbers home page, paul@npac.syr.edu
7. Ritter T., The efficient generation of Cryptographic confusion sequences, Cryptologia, 15(2): 1991, Pgs 81-139.
8. D. Eastlake, S. Croker, J.Schiller, Randomness recommended for security, Dec., 1994.
9. Bright H., and Enison R., Quasi-Random Number Sequences from Long-period TLP Generator with Remarks on Application to Cryptography, Computing Surveys, De., 1979.
10. William Schweber, Electronic Communication System (A Complete Course)., !996, Pgs 407-413.
11. Simon H., Communication Systems,3rd Edition, John Wiley and Sons Inc., 1994, Pgs 578-586; 815-836.
12. Horwitz and Hill, The Art of Electronics, 2nd Edition, Pgs 655-664.

13. Terry R., Ritter's CryptoGlossary and Dictionary of Technical Cryptography, 1995, 2003.
14. Benjamin J. and Paul K., The INTEL Random Number Generator, Cryptography Research Inc., April 1999.
15. Ivars Peterson's Mathland, A catalog of Random bits, April 22, 1996.
16. R.B.Davies, True Random Number Generators, http://www.robertnz.net/true-rng.html
17. Cryptography A- Z, www.SSH-Tech-corner-cryptographic-Algorithms.htm
18. David W. D., Computer Generated Random Numbers, 1991.
19. M. Haahr, Introduction to Randomness and Random Numbers, June 1999.
20. Terry R., Research Comments from Ciphers by Ritter, Random Number Machines, A literature Survey.
21. Robert D., Hardware Random Number Generators, Research Associates Limited, 14 October,2000.
22. J. S.Bendat and A. G. Piersol, Measurement and analysis of Random data., John Wiley and Sons, Inc., 1996.
23. Gregory J. C., Randomness and Mathematical proof, Scientific American 232(5), May 1975, Pgs 47-52.
24. Ashish P. Information, Uncertainty and Randomness Algorithmic Perspectives, IEEE Potentials, Oct/Nov. 2002.
25. Meyer C., and Matyas S., Cryptography: A new Dimension in Computer Data Security. New York: Wiley, 1982.
26. Peter Hellekalek, pLab project, University of Salzburg, 2002.
27. Dierks T., and Allen C., Lessons from doing Source Code Reviews of Commercial Products, Consensus Development, 1997.

28. Wikipedia, The Free Encyclopedia.

29. Pascal J., Cryptographic Secure Pseudo-Random Number Generation: The Blum-Blum-Shub Generator, August 1999.

30. Ritter T., Simple Seed Selection in BB&S, June 2001.

APPENDIX ONE.

DEFINITIONS.

SEED:

The value placed into the state of an RNG. It is the start of an RNG.

QUADRATIC RESIDUES:

If there is an integer y such that;

$$y^2 = p \pmod q$$

then p is a quadratic residue (mod q), otherwise p is said to be a nonquadratic residue (mod q). E.g., $3^2 \equiv 9 \pmod{10}$ implies that 9 is a quadratic residue (mod 10).

INTEGER FACTORIZATION:

The Integer factorization problem is this: given a positive integer, write it as a product of prime numbers. For example, given the number 45, the answer would be $3^2 \cdot 5$.

GREATEST COMMON DIVISOR:

In mathematics, the greatest common divisor (abbreviated GCD), or highest common factor (HCF) of two integers which are not both zero is the largest integer that divides both numbers. The GCD of *a* and *b* is often written as gcd(*a*,*b*). For example, gcd(42,56) = 14

RELATIVE PRIME:

In simple terms, two (or more) numbers are said to be relatively prime if their greatest common factor (gcd) is 1. Every time a fraction is reduced to its lowest terms, they are actually relatively prime. E.g., 7/15 is in its lowest terms because there is no number that is a common factor (other than 1) by which we can divide the numbers 7 and 15.

DEGENERATE CYCLE:

In finite state machine, a cycle with only one state, which thus does not cycle through different states, and so does not change its output value is called a degenerate cycle.

DETERMINISTIC:

A process whose sequence of operations is fully determined by its initial state. A mechanical or clockwork-like process whose outcome is inevitable, given its initial setting. Computational random number generators are deterministic.

PERMUTATION GENERATORS:

A permutation generator is a program which produces permutations of a set of distinct objects. Permutation is the mathematical term for a particular arrangement or a re-arrangement of symbols, objects, or other elements.

MODULAR ARITHMETIC:

This is a modified system of arithmetic for integers, sometimes referred to as 'clock arithmetic', where numbers 'wrap around' after they reach a certain value (the modulus). If x is an integer and p is a positive integer, we write x mod p for the remainder in (0,...,p-1) that occurs if x is divided by p.

CONGRUENT MODULO:

We call two integers x, y congruent modulo q, written as : x = y (mod q) if one of the following equivalent conditions holds:

1. their difference is divisible by q;
2. they leave the same remainder when divided by q, i.e. if x mod q = y mod q;

CRYPTOGRAPHY:

Greek for "hidden writing." The art and science of transforming information into an intermediate form which secures the information while in storage or in transit. Cryptography seeks to render a message unintelligible even when the message is completely exposed

PSEUDORANDOM :

Something which *appears* to be random, but in fact is not is called pseudorandom . Typically, a sequence of values produced by an RNG, or any other completely deterministic computational mechanism. As opposed to really random.

The usual random number generator is *pseudo*random. Given the initial state or seed, the entire subsequent sequence is completely pre-determined, but nevertheless exhibits many of the expected characteristics of a random sequence. Pseudo-randomness supports generating the exact same sequence repeatedly at different times or locations. Pseudo-randomness is generally produced by a mathematical process, which may provide good assurances as to the resulting statistics, assurances which a really random generator generally cannot provide.

POLYNOMIAL-TIME – UNPREDICTABLE:

A Pseudo- random number is Polynomial – Time –Unpredictable if and only if for every finite initial segment of a sequence produced by such a generator, but with any element deleted from the segment a computer can do no better at trying to guess the missing element (in polynomial time in seed length) than by flipping a coin.

CIPHERS:

A secrecy mechanism or process which operates on individual characters or bits independent of semantic content. A good cipher can transform secret information into a multitude of different intermediate forms, each of which represents the original information.

APPENDIX TWO.

SOURCE CODE FOR THE DIFFERENT MODULES OF THE BBS GENERATOR

```
******************Main Menu program

Private Sub Mn2_Click()
   Me.Hide
   Form2.Show
End Sub

Private Sub Mn3_Click()
   Me.Hide
   Form3.Show
End Sub

Private Sub Mn4_Click()
   Me.Hide
   Form4.Show
End Sub

Private Sub Mn5_Click()
  Me.Hide
  Form1.Show
End Sub

Private Sub Mn6_Click()
   End
End Sub


*****************Special Prime Numbers' generator

Private Sub Command1_Click()

Dim num, k, count, max,  i, p1, p2, prime As Integer

   num = 2
```

```
max = Val (Text1)
For count = 0 To max ' keep track of primes found, and stop at
  'a maximum value input ( max)
  prime = 1 ' prime keeps track of if a number is found to be
       'prime or not--1=prime 0=not prime
  k = num ^ (0.5)
  For i = 2 To k
   If num Mod i = 0 Then prime = 0 ' test to see if num is prime
  Next i
  If prime <> 0 Then
    If num Mod 4 = 3 Then ' test to see if the prime number is
       'congruent to 3 mod 4
     p1 = (num - 1) / 2 ' A test for specialty
     If p1 Mod 2 = 1 Then
      p2 = (p1 - 1) / 2
       If p2 Mod 2 = 1 Then
       List1.AddItem CStr(num)
       End If
      End If
    End If
```

```
        End If

        num = num + 1 ' Increase the  value of the number being tested

     Next count ' Increase the count of the primes found

End Sub

Private Sub Command2_Click()
        List1.Clear
End Sub

Private Sub Command3_Click()

        Me.Hide

         Form5.Show
End Sub
```

*****************Test for BBS Condition Two

```
Private Sub Command1_Click()

Dim m, x, p, p1 As Integer

'   cout<<"Enter p:";
 p = Val(Text1)

   p1 = (p - 1) / 2

   For x = 1 To (p1 - 1)

      m = (x * x) Mod p1

      List1.AddItem CStr(m)
   Next x

End Sub
```

```
Private Sub Command2_Click()

      Text1 = ""

      List1.Clear

      Text1.SetFocus
End Sub

Private Sub Command3_Click()
   Me.Hide

   Form5.Show
End Sub


*****************Factors of  a Number to check for relative primality

Private Sub Command1_Click()

 Dim num, x, count As Integer


 x = Val(Text1)

      num = 1

      For count = 0 To x

      num = num + 1


       If x Mod num = 0 Then

       List1.AddItem CStr(num)

       End If

       Next count
```

```
End Sub

Private Sub Command2_Click()
        Text1 = ""

        Text1.SetFocus

        List1.Clear

End Sub

Private Sub Command3_Click()
        Me. Hide

        Form5.Show
End Sub



******************* The Blum Blum Shub generator

Private Sub Command1_Click()

 Dim S(1001), Z(1000), p, q, u, x As Double


    p = Val(Text1) 'enter the first prime

    q = Val(Text2) ' enter the second prime

    N = p * q

    x = Val(Text3) ' enter x which should be relatively prime  to N


    u = x * x

  S(0) = u Mod N ' initial value is computed
```

```
For i = 0 To 1000 ' the loop performs BBS iteration and

'returns the random numbers

    S(i + 1) = (S(i) * S(i)) Mod N

   List1.AddItem CStr(S(i + 1)), i

  Z(i) = S(i) Mod 2 ' this returns the least significant bit of S(i) as output

List2.AddItem CStr(Z(i)), i

  Next i

End Sub

Private Sub Command2_Click()
        Text1 = ""

        Text2 = ""

        Text3 = ""

        Text1.SetFocus

        List1.Clear

        List2.Clear

End Sub

Private Sub Command3_Click()
         Me.Hide

          Form5.Show
End Sub
```