Design & Implementation of Stream Cipher based on Linear Feedback Shift Registers

A Dissertation

Submitted in Partial Fulfillment For the Award of The Degree of

Master of Engineering in

Computer Technology and Application

By:

MANMOHAN
(27/CTA/03)

Under the guidance of

Prof. D. Roy Choudhury

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
DELHI UNIVERSITY, DELHI-110042
2003-2005

CERTIFICATE

This is to certify that the project entitled " **Design & Implementation of Stream Cipher based on Linear feedback shift registers** " submitted by **Manmohan,** class Roll No. 27/CTA/03 , **university Roll No. 3019** in the partial fulfillment of the requirement for the award of degree of Master of Engineering in Computer Technology and Application, Delhi College of Engineering is an account of his authentic work carried out under my guidance and supervision. He has not submitted this work for the award of any other degree.

Project Guide:

Dr. D. Roy Choudhury
Professor
Dept. of Computer Engg.
Delhi College of Engg., Delhi

Head of Department:

Dr. D. Roy Choudhury
Professor
Dept. of Computer Engg.
Delhi College of Engg., Delhi

ACKNOWLEDGEMENTS

I am delighted to express my heartily and sincere gratitude and indebtedness to Prof. D. Roy Choudhury, Head of Department, Computer Engineering, Delhi College of Engineering, Delhi for his invaluable guidance and wholehearted cooperation. His continuous inspiration only has made me complete this minor project.

I am thankful to many people whose feedback and suggestions proved to be invaluable. I would like to thanks Shri S.P.Mishra(Scientist 'E 'DRDO) Shri Prabho Singh(Statistical Analyst Indian Army) for their inspiring ideas and thoughts and Mr.Deepak Mittal for his technical assistance.

I am also thankful to Dr. SHRI KANT (Scientist 'F 'DRDO) for evaluating my work and providing me invaluable suggestions.

Last but not least, I would like to thank my family and my friend Ms. Kumkum Bagchi for their support and help throughout the years of study, without whom it would have been difficult to accomplish this task.

(MANMOHAN)

DCE 2003-2005

DECLARATION BY THE CANDIDATE

I, hereby declare that the dissertation work entitled **"Design & Implementation of Stream Cipher based on Linear feedback Shift Registers"** is an authentic work carried out by me under the guidance of **Prof. D.Roy Choudhury** for the partial fulfillment and award of the degree of M.E. in computer Technology and Application. I have not submitted this work anywhere else for the award of any other degree.

(MANMOHAN)
M.E. (CTA) 27/CTA/03
Department of Computer Engineering
DCE, Delhi-1100042

Abstract

In this thesis I propose a LFSRs based stream cipher. To avoid brute-force attack a crypto system should use at least 90-bit key. Our cryptosystem uses 128-bit key. Crack resistance of cryptosystem is more than billions of years.

To avoid other attacks I have conducted various randomness tests extensively. As far as encryption rate of cryptosystem is concerned it is comparable to modern cryptosystems like RC4. Reason behind using LFSRs is that they are well studied and they can be implemented efficiently in hardware as well as in software.

Proposed cryptosystem will be very useful with suitable combination of public key cryptosystem and hashing algorithm in securing any digital information. Cryptosystem can be easily extended for large key length also.

The software is written in C++/C for Windows and linux environment. For testing stream cipher NIST test suite on Linux Platform is being used.

Contents

CHAPTER 1

Introduction

Secret writing for transmission of message has been , according to David Kahn, the great historian of cryptography, practiced since forty centuries, the first example being an altered message on a tomb in Egypt in 1900 B.C. From that time onwards, secret messages have been used by people of different walks of life-diplomats ,military officers, bankers, scholars and citizens all over the world. The art and science of producing systems for secret writing –keeping the message secure – is cryptography. During the last twenty years, public and academic research in cryptography has explored new dimension. While classical cryptography could be used by ordinary citizens, computer cryptography has been the exclusive domain of the world's militaries since World War II. Today state of the art computer cryptography is being practiced outside the secured walls of the military agencies also.

Cryptography is the mathematics of making a system secure. The algorithm must be so strong that there is no better way to break it than with a brute force attack. This is not as easy as it might seem. Strong cryptosystems with a couple of minor changes can become weak. Good cryptosystems has nice property of making life harder for the attacker than the legitimate user. The designer of cryptosystem has to think of every possible means of attack and protect against all of them.

The need for information security in today's digital systems is growing. For this reason cryptography has become one of these systems' critical component. Cryptographic services are now required across a variety of electronic platforms such as secure access to private networks, data banks, electronic commerce, cellular and PCS phones, all kinds of data communications and smart card technology.

Cryptographic goals A well-defined and implemented cryptographic system should provide the following services:

1)Confidentiality: Keeps the data involved in an electronic transaction private. Meaning that the transmitted information is accessible only for reading by authorized parties. Encryption provides confidentiality.

2)Authentication: Ensures that the origin of a message or electronic document is correctly identified. In mutual authentication both the server and user or both parties

in general case authenticate each other. It is typically provided by verifying other parties digitally signed certificates.

3)Data Integrity: It basically means that the information exchanged in an electronic data transfer is not alterable without detection. Modification types include writing, changing, deleting, etc.

4)Nonrepudiation: This simply tells that the actions performed by the service user in an electronic transaction are no revocable so that they are legally binding. Therefore, neither the sender nor the receiver of a message should be able to deny the transaction.

A fundamental goal of cryptography is to adequately address these four areas in both theory and practice. Cryptography is about the prevention and detection of cheating and other malicious activities.

A good cryptosystem is one in which all the security is inherent in knowledge of the key and none is inherent in knowledge of algorithm. If a cryptographically weak process is used to generate keys then the whole system is weak. This is why Key management is so important in cryptography. The art and science of breaking cipher-text is cryptanalysis. The ranch of science encompassing both cryptography and cryptanalysis is cryptology.

1.1 A Short History of Cryptology from a Stream Ciphers Point of View

Modern cryptology deals with confidentiality, integrity, authenticity, random-number generation, zero-knowledge proofs and various other topics in a mathematical way.

However, most of these issues arose in the last decades. The propelling force in the centuries before was the human desire to transfer information secretly. Ancient approaches to transport information in a confidential way considered single letters and changed them according to various rules. Examples thereof are the Caesar cipher or simple substitution ciphers. The first approach which has much in common with today's stream ciphers is the cipher of Vignere [Kah67]. It can be formalized as:

$C_i = P_i + K_i \ mod \ 26$ (1.1)

Equation 1.1 can be interpreted as follows: each letter in the alphabet is under stood as a number ranging from 0 to 25. The keyletter K is added to the plaintext letter P in order to get the ciphertext letter C. The *mod* 26 operation makes sure that the resulting ciphertext is always inside the proper range. Keep in mind: in this scheme, the key is reused if the plaintext is longer than the key. The first cryptanalysis of this scheme

was published by Kasiski , which is based upon this fact. He came up with the following observation when encrypting two plaintexts P_a and P_b with the same secret key:

$C_a \oplus C_b = P_a \oplus P_b$ (1.2)

A ciphertext only attack reveals very much of the used plaintext, if the key K is reused. Even if the used plaintexts are not directly revealed, $P_a \oplus P_b$ contains much information about the plaintexts P_a and P_b.

Later, in the early 20th century, Vernam introduced a scheme which made the key as long as the plaintext. This was a big difference to the polyalphabetic substitution ciphers used at this time. Later on, this system has been used with a random key - hence the one-time pad was born. In 1949 Shannon proved in his landmark paper that this approach is indeed unconditionally secure; that is, even an attacker with unbounded computing resources cannot break such a system, and no future advance in mathematics can provide a shortcut attack because of its entropy conditions.

All of these approaches have two things in common: first, their security is based on a shared secret - they are all symmetrical ciphers. But even more important is that their working principle is based on single symbols, which is an important characteristic of stream ciphers.

1.2 The Structure of Stream Ciphers

Since it is very inconvenient and in many circumstances impractical to share and use keys which are the same size as the transmitted information (this effort is only made in exceptional circumstances) the goal was to reduce the key-size but maintain a reasonable level of security. The general approach to achieve this is to replace the key in the Vernam scheme by a pseudo-random sequence of bits or symbols which is the output of a generator that is initialized with a (shorter) key. Of course this scheme can no longer be unconditionally secure anymore because the entropy of the key is now smaller than the entropy of the message. In the context of generating random bits, several identifiers are used to refer to random number generators(RNG).

Informally, they can be structured as follows.

When talking about true random number generators (TRNG), some sort of physical activity is used as a source for randomness. Examples thereof are thermal noise, flipping a coin or radioactive decay. On the other hand, pseudo random number generators (PRNG) are based on algorithms and are thus deterministic. Usually, they are seeded by a TRNG. The randomness of a PRNG is verified by means of statistical

tests. If used for cryptographic purposes, PRNG need to have additional properties: it should be difficult to predict the future output from previous outputs and it should be difficult to determine the internal state of the generator by examining the output. If those properties are fulfilled, the PRNG is *cryptographically secure* (CSPRNG).

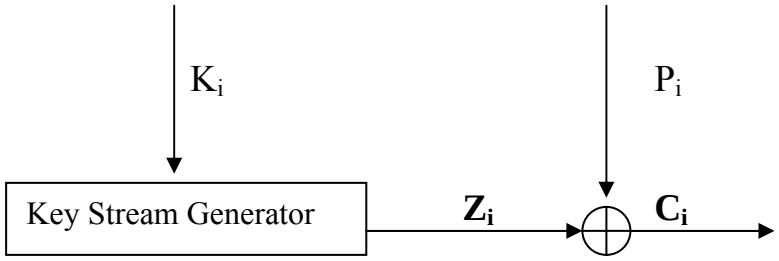
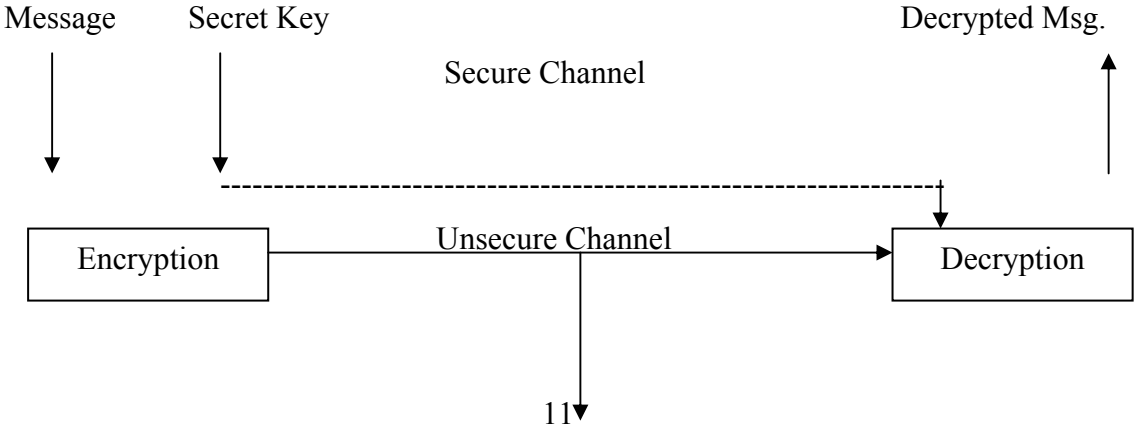


Figure 1.1 shows the general structure of a stream cipher. The short key denoted as *K* is used to seed the key-stream generator. The key-stream symbols Z_i are combined with the plaintext symbols P_i to produce a sequence of ciphertext symbols C_i. The generated key-stream is combined with the plaintext on a bit-per-bit basis. This combining step is most of the time an *XOR* function.

1.3 Secret Key vs. Public Key Cryptography

A major problem arises when a big number of devices/people want to exchange messages using a cipher and a secret key. Since nobody else than the two involved parties should be able to decrypt the exchanged message, pairwise secret keys need to be distributed among the participants. There are two straightforward solutions to this problem. One is the generation and pre-distribution of *n*(n-1)/2* keys for *n* parties. Another approach is to introduce a trusted party which generates needed keys on demand. In this case, the number of pre-distributed keys is in the order of *n*, but the trusted party is also a single point of failure.



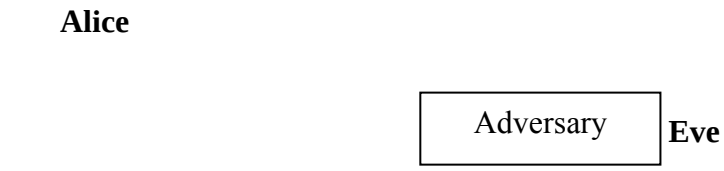


Fig 1.2:Two parties engaged in symmetric cipher conversation with an adversary listening in on unsecured channel

In the late 70s public key cryptography was introduced among others by Diffe and Hellman and by Rivest, Shamir and Adleman in the open literature. Their methods are based on hard number-theoretic problems and can be used to solve the key distribution problem. Instead of sharing a secret key, each party has a pair of keys, which consist of a private key only known to the owner and a public key. Despite this big structural advantage, public key cryptography did not replace secret key cryptography. Performance is one of the reasons, since public key cryptography is much slower than secret key cryptography. In practice, a two-stage approach is used. In a first step public key cryptography is used to derive a secret key. Subsequently, this key is used to encrypt the actual message. This thesis solely deals with secret key algorithms where stream ciphers are assigned to.

Asymmetric ciphers (also called **public-key algorithms** or generally **public-key cryptography**) permit the encryption key to be public (it can even be published in a newspaper), allowing anyone to encrypt with the key, whereas only the proper recipient (who knows the decryption key) can decrypt the message. The encryption key is also called the **public key** and the decryption key the **private key** or **secret key**.

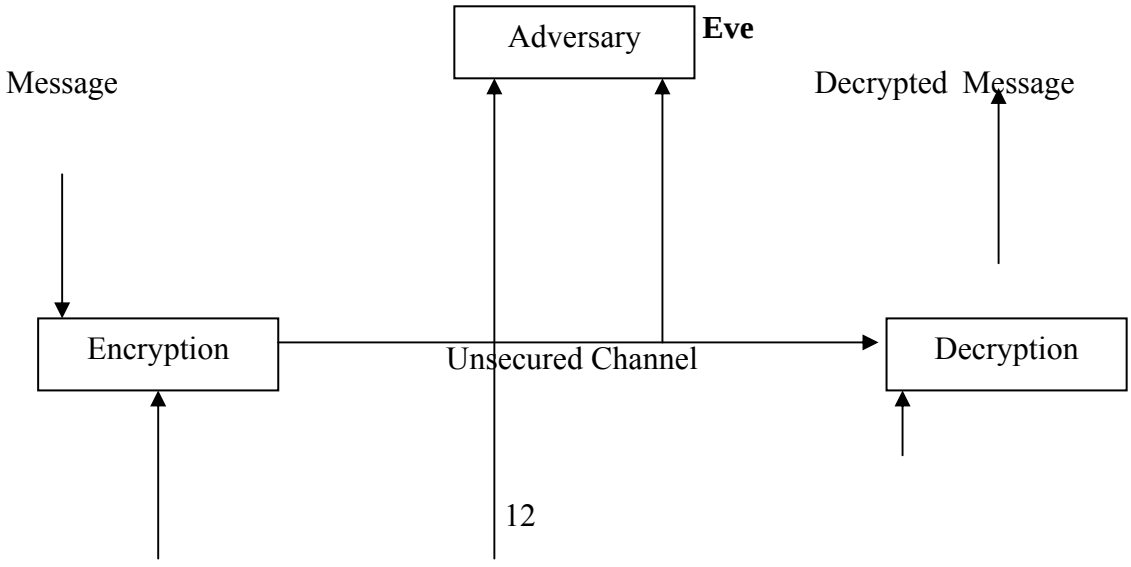


Fig 1.3:Two parties engaged in public key cipher conversation. The adversary can see the both cipher text and the public key

Encryption using public key k is denoted by:

$$E_k(P) = C$$

Even though the public key and private key are different , decryption with the corresponding private key is denoted by:

$$D_k(C) = P$$

Sometimes message will be encrypted with the private key and decrypted with the public key , this is used in digital signatures. These operations are also denoted by,

$$E_k(P) = C$$
$$D_k(C) = P$$

- **RSA** (Rivest-Shamir-Adelman) is the most commonly used public key algorithm. Can be used both for encryption and for digital signature. It is generally considered to be secure when sufficiently long keys are used (512 bits is insecure, 768 bits is moderately secure, and 1024 bits is good and 2048 bit keys are likely to remain secure for decades.). The security of RSA relies on the difficulty of factoring large integers. Dramatic advances in factoring large integers would make RSA vulnerable. RSA is currently the most important public key algorithm.

Let P and Q be the two large primes , each roughly of same size 10E100. Compute N = P*Q and $\phi(N) = (P-1)(Q-1)$. Select an integer E, $1 < E < \phi(N)$, such that **$(E, \phi(N)) = 1$** . The integer E is called the encryption key, used as public key, and the integer N is called the modulus of the system. Let M, 1<M<N

, be a numerical plane message , then the cryptogram C is obtained in the following way:-

$$C \equiv M^E \bmod N$$

Let D be a +ve integer such that $ED \equiv 1 \bmod \phi(N)$ then the plane message M is obtained back from the cryptogram C in the following way :

$$M \equiv C^D \bmod N$$

The integer D is called the decryption key of the system and used as the private key. These encryption and decryption keys E and D can be generated easily using the inverse generating algorithm **(IGA)** .

- **Diffie-Hellman, DSS** (Digital Signature Standard), **LUC** are others public key algorithms.

Modern cryptographic algorithms cannot really be executed by humans. Strong cryptographic algorithms are designed to be executed by computers or specialized hardware devices. In most applications, cryptography is done in computer software, and numerous cryptographic software packages are available.

Generally, **symmetric algorithms** are much faster to execute on a computer than asymmetric ones. In practice they are often used together, so that a public-key algorithm is used to encrypt a randomly generated encryption key, and the random key is used to encrypt the actual message using a symmetric algorithm.

Advantages and disadvantages Symmetric-key vs. public-key cryptography

Symmetric-key and public-key encryption schemes have various advantages and disadvantages, some of which are common to both. This section highlights a number of these and summarizes features pointed out in previous sections.

(i) Advantages of symmetric-key cryptography

1. Symmetric-key ciphers can be designed to have high rates of data throughput. Some hardware implementations achieve encrypt rates of hundreds of megabytes per second, while software implementations may attain throughput rates in the megabytes per second range.
2. Keys for symmetric-key ciphers are relatively short.

3 .Symmetric-key ciphers can be employed as primitives to construct various cryptographic mechanisms including pseudorandom number generators hash functions and computationally efficient digital signature schemes, to name just a few.

4. Symmetric-key ciphers can be composed to produce stronger ciphers. Simple transformations which are easy to analyze, but on their own weak, can be used to construct strong product ciphers.

5. Symmetric-key encryption is perceived to have an extensive history, although it must be acknowledged that, notwithstanding the invention of rotor machines earlier, much of the knowledge in this area has been acquired subsequent to the invention of the digital computer, and, in particular, the design of the Data Encryption Standard in the early 1970s.

(ii) Disadvantages of symmetric-key cryptography

1. In a two-party communication, the key must remain secret at both ends.

2. In a large network, there are many key pairs to be managed. Consequently, effective key management requires the use of an unconditionally trusted TTP .

3. In a two-party communication between entities A and B, sound cryptographic practice dictates that the key be changed frequently, and perhaps for each communication session.

4. Digital signature mechanisms arising from symmetric-key encryption typically require either large keys for the public verification function or the use of a TTP.

(iii) Advantages of public-key cryptography

1. Only the private key must be kept secret (authenticity of public keys must, however, be guaranteed).

2. The administration of keys on a network requires the presence of only a functionally trusted TTP (Definition 1.66) as opposed to an unconditionally trusted TTP. Depending on the mode of usage, the TTP might only be required in an "off-line" manner, as opposed to in real time.

3. Depending on the mode of usage, a private key/public key pair may remain unchanged for considerable periods of time, e.g., many sessions (even several years).

4. Many public-key schemes yield relatively efficient digital signature mechanisms. The key used to describe the public verification function is typically much smaller than for the symmetric-key counterpart.

5. In a large network, the number of keys necessary may be considerably smaller than in the symmetric-key scenario.

(iv) Disadvantages of public-key encryption

1. Throughput rates for the most popular public-key encryption methods are several orders of magnitude slower than the best known symmetric-key schemes.

2. Key sizes are typically much larger than those required for symmetric-key encryption, and the size of public-key signatures is larger than that of tags providing data origin authentication from symmetric-key techniques.

3. No public-key scheme has been proven to be secure (the same can be said for block ciphers). The most effective public-key encryption schemes found to date have their security based on the presumed difficulty of a small set of number-theoretic problems.

4. Public-key cryptography does not have as extensive a history as symmetric-key encryption, being discovered only in the mid 1970s.

Summary of comparison Symmetric-key and public-key encryption have a number of complementary advantages. Current cryptographic systems exploit the strengths of each. An example will serve to illustrate.

Public-key encryption techniques may be used to establish a key for a symmetric-key system being used by communicating entities A and B. In this scenario A and B can take advantage of the long term nature of the public/private keys of the public-key scheme and

the performance efficiencies of the symmetric-key scheme. Since data encryption is frequently the most time consuming part of the encryption process, the public-key scheme for key establishment is a small fraction of the total encryption process between A and B. To date, the computational performance of public-key encryption is inferior to that of symmetric-key encryption. There is, however, no proof that this must be the case. The important points in practice are:

1. public-key cryptography facilitates efficient signatures (particularly non-repudiation) and key management ; and

2. symmetric-key cryptography is efficient for encryption and some data integrity applications.

Remark (key sizes: symmetric key vs. private key) Private keys in public-key systems must be larger (e.g., 1024 bits for RSA) than secret keys in symmetric-key systems (e.g.,64or 128 bits) because whereas (for secure algorithms) the most efficient attack on symmetric key systems is an exhaustive key search, all known public-key systems are subject to “shortcut” attacks (e.g., factoring) more efficient than exhaustive search. Consequently, for equivalent security, symmetric keys have

bit lengths considerably smaller than that of private keys in public-key systems, e.g., by a factor of 10 or more.

1.4 Stream Ciphers vs. Block Ciphers

Stream ciphers apply a simple, always changing transformation to one symbol at a time whereas block ciphers apply a more complex, but static transformation to a group of symbols at once. Stream ciphers can be faster than block ciphers, especially if they are based on LFSRs and implemented in hardware. However, the distinction between both types of ciphers is not clear. If a block cipher is used in cipher block chaining (CBC) mode, one can consider this as a stream cipher which operates on large symbols (*i. e.* symbols of the size of one block). This could lead to the conclusion that stream- and block ciphers work at different levels of abstraction. Whereas stream ciphers work in a particular mode of operation, block ciphers are just building blocks to construct a mode of operation.

Additionally, there are some modes of operation for block ciphers whose intention is to provide stream cipher like properties. These are the cipher feedback (CFB) mode, the output feedback (OFB) mode, the key feedback (KFB) mode and the counter(CTR) mode. There, the block cipher is used to generate a key-stream which is then *XOR*ed with the plaintext. On the other hand, turning a stream cipher into a block cipher is also possible, but less efficient though. Simplified, what is required is to put the stream cipher into the round function of a Feistel based block cipher. There is much theoretical knowledge and there are many tools available to analyze various properties of the building blocks of stream ciphers. Until recently nothing comparable was available for block ciphers. Till the mid 90s all used block ciphers were based on a Feistel structure [Fei73] which arose in the early 70s. The Data Encryption Standard (DES), which held a predominant position in the area of block ciphers for more than two decades, uses this structure as well. The design criteria for the DES, especially for its S-boxes, are kept secret and DES proved to be highly resistant against cryptanalytic shortcut attacks. Later Substitution Permutation Networks (SPNs) have been used to design new block ciphers. Together with the “Wide trail design strategy" , these efforts led to the ability of giving block ciphers valuable properties such as provable resistance against linear or differential cryptanalysis.

Stream ciphers on the other hand have never been standardized but tend to be proprietary and even classified. The AES is the current standard for general purpose

block ciphers, but there is no comparable standard for stream ciphers. This is reflected by the fact that the widely used RC4, perceived as the de facto standard for stream ciphers, is proprietary. Even though the number of new stream cipher proposals is growing and the state of affair for both seems to converge a little bit there is still much do be done: none of the stream ciphers submitted to the NESSIE project have been recommended because all suffered at least from slight weaknesses.

1.4.1Desirable Characteristics of stream and Block Ciphers

Stream Ciphers:

Large Period: For every key the sequence should have a very large period so that no part of enciphering sequence should have a very large period so that no part of enciphering sequence is used repeatedly within a reasonable time.

Complexity: Given a segment of sequence, it should not be possible to predict the preceding or following segment.

Sound Statistical Properties: In bit stream the ones and zeroes should be evenly distributed in the sequence and also poses good autocorrelation properties.

Variability: variability should be high to ensure that a brute force attack become infeasible.

Correlation Immunity: Functions used should be non linear and correlation immune.

Block Ciphers:

Avalanche Effect: A 1-bit change of the key or plain text should produce a radical change in the cipher text. If f(p,k) = C , where p stand for plain text , k for the key transformation and C for cipher text. One bit change either in p or k produces radical change in the cipher text C.

2) The algorithm should contain a non commutative combination of substitution and permutation.

3. The algorithm should include substitution and permutations under the control of both the input data and the key.

4. The length of cipher text should be same as length of the plaintext.

5.There should be no simple relationship between any possible keys and cipher text bits.

6.All possible keys should produce a strong cipher that does not have any statistical or language oriented weakness that can be exploited by the cryptanalyst.

7. The length of the key and the text should be adjusted to meet application requirements and security strength requirements.

8. Block length should be large.

9. The algorithms should be easily and efficiently implement able on main frames, minicomputers and microcomputers(IN fact the functions used in the algorithm are limited to Xor and bit Shifting).

1.5 Analysis of Stream Ciphers

Various stream ciphers are used in today's applications. They range from software based applications like web browsers to wireless communication hardware like mobile phones and WLAN equipment. The used stream ciphers range from the de facto standard RC4 to designs specially targeted for a particular use. Even those specially designed for a particular purpose rely on a number of well known structures. When designing stream ciphers, there are however more structures available than in the case of block ciphers.

The security of cryptographic algorithms can be evaluated by means of crypt-analysis. Mathematical relations between plaintext, ciphertext and key are used to find weaknesses.

Chapter 2

2.1 Introduction

Stream ciphers are an important class of encryption algorithms. They encrypt individual characters (usually binary digits) of a plaintext message one at a time, using an encryption transformation, which varies with time. By contrast, block ciphers tend to simultaneously encrypt groups of characters of a plaintext message using a fixed encryption transformation. Stream ciphers are generally faster than block ciphers in hardware, and have less complex hardware circuitry. They are also more appropriate, and in some cases mandatory (e.g., in some telecommunications applications), when buffering is limited or when characters must be individually processed as they are received. Because they have limited or no error propagation, stream ciphers may also be advantageous in situations where transmission errors are highly probable.

There is a vast body of theoretical knowledge on stream ciphers, and various design principles for stream ciphers have been proposed and extensively analyzed. However, there are relatively few fully specified stream cipher algorithms in the open literature. This unfortunate state of affairs can partially be explained by the fact that most stream ciphers used in practice tend to be proprietary and confidential. By contrast, numerous concrete block cipher proposals have been published, some of which have been standardized or placed in the public domain. Nevertheless, because of their significant advantages, stream ciphers are widely used today, and one can expect increasingly more concrete proposals in the coming years.

2.1.1 Classification of Stream Ciphers

As described in the introductory chapter, encryption schemes can be symmetric(using the same secret key for both encryption and decryption) as well as asymmetric(en- and decryption use different keys). Even for the special case of stream ciphers, there is this distinction: there are symmetric as well as asymmetric stream ciphers. No doubt, symmetric stream ciphers build the majority, but there is an example for the latter: the Blum-Goldwasser probabilistic encryption scheme which is based on the Blum-Blum-Shub pseudorandom number generator. This was a novel approach, since the BBS-generator has a strong security proof, which relates the difficulty of distinguishing output bits from random to the difficulty of integer factorization. However, this scheme has no relevance for practical (fast) stream ciphers and therefore, the remainder of this chapter will deal solely with symmetrical stream

ciphers. Stream ciphers can also be synchronous or asynchronous. In a synchronous stream cipher, the generated key-stream is independent from the processed plain text or ciphertext. Hence it can be pre-generated as well. A bit-error in the plaintext (*e. g.* a flipping bit) affects only the corresponding bit of the ciphertext. One drawback is that both the sender and the receiver have to be synchronized. Insertion or deletion of bits by an active attacker causes immediate loss of synchronization but is thus detected immediately as well.

Asynchronous stream ciphers on the other hand use parts of the generated cipher-text in the key-stream generation. Whereas a key-stream can be pre-generated with synchronous stream ciphers because it is independent of the plaintext, this is not the case for asynchronous stream ciphers. They can resynchronize after insertion- or deletion of bits and are therefore sometimes called self-synchronized. Even if this can be an important property, this type of stream cipher did not receive much interest; so the remainder of this chapter will deal with synchronous stream ciphers. Menezes *et al.* names the one-time pad as a separate type of stream cipher. In this case, the key-stream is truly random and not generated in a deterministic way as it is done by the other types.

Stream ciphers can be either symmetric-key or public-key. The focus of this chapter is symmetric-key stream ciphers; the Blum-Goldwasser probabilistic public-key encryption

2.2 Basic Building Blocks of Stream Ciphers

Block ciphers are round based; their basic building blocks are used in an iterative manner to produce a set of output symbols. Stream ciphers use different building blocks(though some are similar to block ciphers as well) and combine them in a certain way using an internal state to produce single output symbols. Some of these building blocks and combinations thereof are now considered.

2.3 Linear feedback shift registers

Linear feedback shift registers (LFSRs) are used in many of the keystream generators that have been proposed in the literature. There are several reasons for this:

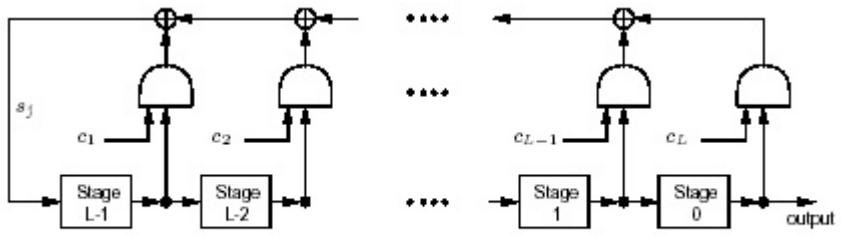
1. LFSRs are well suited to hardware implementation;
2. they can produce sequences of large period ;
3. they can produce sequences with good statistical properties , and
4. because of their structure, they can be readily analyzed using algebraic techniques.

2.3.1Definition A linear feedback shift register (LFSR) of length L consists of L stages (or delay elements) numbered 0; 1; : : : ; L - 1, each capable of storing one bit and having one input and one output; and a clock which controls the movement of data. During each unit of time the following operations are performed:

(i) the content of stage 0 is output and forms part of the *output sequence*;

(ii) the content of stage i is moved to stage i - 1 for each i, $1 \leq i \leq L - 1$; and

(iii) the new content of stage L - 1 is the *feedback bit* s_j which is calculated by adding together modulo 2 the previous contents of a fixed subset of stages 0; 1; : : : ; L - 1.



A linear feedback shift register (LFSR) of length L.

figure 2.1

Figure 2.1 above depicts an LFSR. Referring to the figure 2.1, each c_i is either 0 or 1; the closed semi-circles are AND gates; and the feedback bit s_j is the modulo 2 sum of the contents of those stages i, $0 \leq i \leq L - 1$, for which $c_{L-i} = 1$.

2.3.2 Definition The LFSR of Figure 2.1 is denoted [L;C(D)], where $C(D) = 1+c_1D +c_2D_2 + \_ \_ \_ + c_LD^L \in Z_2[D]$ is the *connection polynomial*. The LFSR is said to be *nonsingular* if the degree of C(D) is L (that is, $c_L = 1$). If the initial content of stage i is $s_i \in \{0,1\}$; for each i, $0 \leq i \leq L - 1$, then $[s_{L-1}; : : : ; s_1; s_0]$ is called the *initial state* of the LFSR.

2.3.3 Fact If the initial state of the LFSR in above figure is $[s_{L-1}; : : : ; s_1; s_0]$, then the output sequence $s = s_0; s_1; s_2; : : :$ is uniquely determined by the following recursion:

$sj = (c_1s_{j-1} + c_2s_{j-2} + \_ \_ \_ + c_Ls_{j-L}) \bmod 2$ for $j \geq L$:

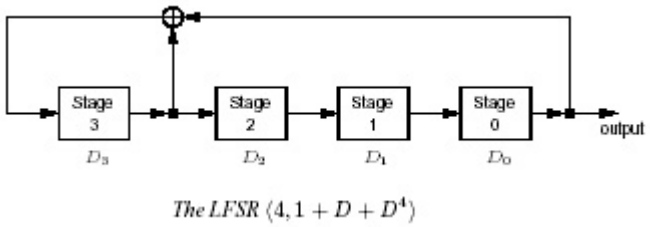
2.3.4 Example (output sequence of an LFSR) consider the LFSR $(4;1 + D + D^4)$ depicted in Figure 2.2. If the initial state of the LFSR is [0; 0; 0; 0], the output sequence is the zero sequence. The following tables show the contents of the stages D_3, D_2, D_1, D_0 at the end of each unit of time t when the initial state is [0; 1; 1; 0].

t	D_3	D_2	D_1	D_0
0	0	1	1	0
1	0	0	1	1
2	1	0	0	1
3	0	1	0	0
4	0	0	1	0
5	0	0	0	1
6	1	0	0	0
7	1	1	0	0

t	D_3	D_2	D_1	D_0
8	1	1	1	0
9	1	1	1	1
10	0	1	1	1
11	1	0	1	1
12	0	1	0	1
13	1	0	1	0
14	1	1	0	1
15	0	1	1	0

The output sequence is s = 0; 1; 1; 0; 0; 1; 0; 0; 0; 1; 1; 1; 1; 0; 1; : : :, and is periodic with period 15 .

Fact 2.3.5 explains the significance of an LFSR being non-singular.



The LFSR $\langle 4, 1 + D + D^4 \rangle$

Figure 2.2

2.3.5 Fact Every output sequence (i.e., for all possible initial states) of an LFSR (L,C(D)) is periodic if and only if the connection polynomial C(D) has degree L.

If an LFSR (L,C(D)) is *singular* (i.e., C(D) has degree less than L), then not all output sequences are periodic. However, the output sequences are *ultimately periodic*; that is, the sequences obtained by ignoring a certain finite number of terms at the beginning are periodic. For the remainder of this chapter, it will be assumed that all LFSRs are nonsingular.

Fact 2.3.6 determines the periods of the output sequences of some special types of non-singular LFSRs.

2.3.6 Fact (periods of LFSR output sequences) Let $C(D) \in Z_2[D]$ be a connection polynomial of degree L.

(i) If C(D) is irreducible over Z_2 then each of the 2^L - 1 nonzero initial states of the non-singular LFSR (L,C(D)) produces an output sequence with period equal to the

least positive integer N such that C(D) divides $1 + D^N$ in $Z_2[D]$. (Note: it is always the case that this N is a divisor of $2^L - 1$.

(ii) If C(D) is a primitive polynomial then each of the 2^L-1 nonzero initial states of the non-singular LFSR (L,C(D)) produces an output sequence with maximum possible period $2^L - 1$.

Fact 2.3.6(ii) motivates the following definition.

2.3.7 Definition If $C(D) \in Z_2[D]$ is a primitive polynomial of degree L, then (L,C(D)) is called a *maximum-length* LFSR. The output of a maximum-length LFSR with non-zero initial state is called an m-*sequence*.

Fact 2.3.8 demonstrates that the output sequences of maximum-length LFSRs have good statistical properties.

2.3.8 Fact (*statistical properties of* m-*sequences*) Let s be an m-sequence that is generated by a maximum-length LFSR of length L.

(i) Let k be an integer, $1 \leq k \leq L$, and let s be any subsequence of s of length $2^L + k - 2$. Then each non-zero sequence of length k appears exactly 2^{L-k} times as a subsequence of s. Furthermore, the zero sequence of length k appears exactly 2^{L-k}-1 times as a subsequence of s. In other words, the distribution of patterns having fixed length of at most L is almost uniform.

(ii) s satisfies Golomb's randomness postulates. That is, every m-sequence is also a pn-sequence .

2.3.9 Example (m-sequence) Since $C(D) = 1+D + D^4$ is a primitive polynomial over Z_2, the LFSR $(4,1+D+D_4)$ is a maximum-length LFSR. Hence, the output sequence of this LFSR is an m-sequence of maximum possible period $N = 2^4-1 = 15$ (Example 2.3.4).

2.4 Stream ciphers based on LFSRs

As mentioned in the beginning, linear feedback shift registers are widely used in keystream generators because they are well-suited for hardware implementation, produce sequences having large periods and good statistical properties, and are readily analyzed using algebraic techniques. Unfortunately, the output sequences of LFSRs are also easily predictable, as the following argument shows. Suppose that the output sequences of an LFSR has linear complexity L. The connection polynomial C(D) of an LFSR of length L which generates s can be efficiently determined using the

Berlekamp-Massey algorithm from any (short) subsequence t of s having length at least n = 2L. Having determined C(D), the LFSR (L,C(D)) can then be initialized with any substring of t having length L, and used to generate the remainder of the sequences.

An adversary may obtain the required subsequence t of s by mounting a known or chosen plaintext attack on the stream cipher: if the adversary knows the plaintext subsequence $m_1,m_2,\ldots.m_n$ corresponding to a ciphertext sequence c_1; c_2; : : : ; c_n, the corresponding keystream bits are obtained as $m_i \oplus c_i$, $1 \leq i \leq n$.

2.4.1Note (*use of LFSRs in keystream generators*) Since a well-designed system should be secure against known-plaintext attacks, an LFSR should never be used by itself as a keystream generator. Nevertheless, LFSRs are desirable because of their very low implementation costs. Three general methodologies for destroying the linearity properties of LFSRs are discussed in this section:

(i) using a nonlinear combining function on the outputs of several LFSRs ;

(ii) using a nonlinear filtering function on the contents of a single LFSR ; and

(iii) using the output of one (or more) LFSRs to control the clock of one (or more) other LFSRs .

Desirable properties of LFSR-based keystream generators

For essentially all possible secret keys, the output sequence of an LFSR-based keystream generator should have the following properties:

1. large period;

2. large linear complexity; and

3. good statistical properties .

It is emphasized that these properties are only *necessary* conditions for a keystream generator to be considered cryptographically secure. Since mathematical proofs of security of such generators are not known, such generators can only be deemed *computationally secure* after having withstood sufficient public scrutiny.

2.4.2 Note (*connection polynomial*) Since a desirable property of a keystream generator is that its output sequences have large periods, component LFSRs should always be chosen to be maximum-length LFSRs, i.e., the LFSRs should be of the form (L,C(D)) where $C(D) \in Z_2[D]$ is a primitive polynomial of degree L (see Definition 2.3.7 and Fact 2.3.6(ii)).

2.4.3 Note (*known vs. secret connection polynomial*) The LFSRs in an LFSR-based keystream generator may have *known* or *secret* connection polynomials. For known

connections, the secret key generally consists of the initial contents of the component LFSRs. For secret connections, the secret key for the keystream generator generally consists of both the initial contents and the connections.

For LFSRs of length L with secret connections, the connection polynomials should be selected uniformly at random from the set of all primitive polynomials of degree L over Z_2.

Secret connections are generally recommended over known connections as the former are more resistant to certain attacks, which use pre computation for analyzing the particular connection, and because the former are more amenable to statistical analysis. Secret connection LFSRs have the drawback of requiring extra circuitry to implement in hardware. However, because of the extra security possible with secret connections, choosing shorter LFSRs may sometimes compensate for this cost.

2.4.4 sparse vs. dense connection polynomial For implementation purposes, it is advantageous to choose an LFSR that is *sparse*; i.e., only a few of the coefficients of the connection polynomial are non-zero. Then only a small number of connections must be made between the stages of the LFSR in order to compute the feedback bit. For example, the connection polynomial might be chosen to be a primitive trinomial. However, in some LFSR-based keystream generators, special attacks can be mounted if sparse connection polynomials are used. Hence, it is generally recommended not to use sparse connection polynomials in LFSR-based keystream generators.

2.5 Linear complexity

This subsection summarizes selected results about the linear complexity of sequences. All sequences are assumed to be binary sequences. Notation: s denotes an infinite sequence whose terms are $s_0, s_1, s_2, \ldots$ s_n denotes a finite sequence of length n whose terms are $s_0; s_1; : : : ; s_{n-1}$.

2.5.1 Definition An LFSR is said to *generate* a sequence s if there is some initial state for which the output sequence of the LFSR is s. Similarly, an LFSR is said to *generate* a finite sequence s_n if there is some initial state for which the output sequence of the LFSR has s_n as its first n terms.

2.5.2 Definition The *linear complexity* of an infinite binary sequence s, denoted L(s), is defined as follows:

(i) if s is the zero sequence s = 0; 0; 0; : : : , then L(s) = 0;

(ii) if no LFSR generates s, then L(s) = 1;

(iii) otherwise, L(s) is the length of the shortest LFSR that generates s.

2.5.3 Definition The *linear complexity* of a finite binary sequence s^n, denoted $L(s^n)$, is the length of the shortest LFSR that generates a sequence having s^n as its first n terms.

Facts 2.5.4 – 2.5.7 summarize some basic results about linear complexity.

2.5.4 properties of linear complexity Let s and t be binary sequences.

(i) For any $n \geq 1$, the linear complexity of the subsequence s^n satisfies $0 \leq L(sn) \leq n$.

(ii) $L(s^n) = 0$ if and only if s^n is the zero sequence of length n.

(iii) $L(s^n) = n$ if and only if $s^n = 0; 0; 0; : : : ; 0; 1$.

(iv) If s is periodic with period N, then $L(s) \leq N$.

(v) $L(s \oplus t) \leq L(s) + L(t)$, where $s \oplus t$ denotes the bitwise XOR of s and t.

2.5.5 Fact If the polynomial $C(D) \in Z_2[D]$ is irreducible over Z_2 and has degree L, then each of the 2^L-1 non-zero initial states of the non-singular LFSR (L;C(D)) produces an output sequence with linear complexity L.

2.5.6 Expectation and variance of the linear complexity of a random sequence

Let s^n be chosen uniformly at random from the set of all binary sequences of length n, and let $L(s^n)$ be the linear complexity of s^n. Let B(n) denote the parity function: B(n) = 0 if n is even;

B(n) = 1 if n is odd.

(i) The expected linear complexity of s^n is

$E(L(s^n)) = n/2+(4 + B(n))/18-1/2^n(n/3+2/9)$ Hence, for moderately large n, $E(L(s^n)) \cong n/2+2/9$ if n is even, and $E(L(s^n)) \cong n/2 + 5/18$ if n is odd.

(ii) The variance of the linear complexity of s^n is $Var(L(s^n)) =$

$86/81-1/2^n[(((14 - B(n))/27)n +(82 - 2B(n))/81] –1/2^{2n}(1/9n^2 +4/27n +4/81)$

Hence, $Var(L(s^n)) \cong 86/81$ for moderately large n.

2.5.7 Expectation of the linear complexity of a random periodic sequence Let s^n be chosen uniformly at random from the set of all binary sequences of length n, where $n = 2^t$ for some fixed $t \geq 1$, and let s be the n-periodic infinite sequence obtained by repeating the sequence s^n. Then the expected linear complexity of s is $E(L(s^n)) = n - 1 + 2^{-n}$.

The linear complexity profile of a binary sequence is introduced next.

2.5.8 Definition Let $s = s_0; s_1; : : :$ be a binary sequence, and let L_N denote the linear complexity of the subsequence $s^N = s_0; s_1; : : : ; s_{N-1}$, $N \geq 0$. The sequence $L_1; L_2;\ldots$is called the *linear complexity profile of* s. Similarly, if $s^n = s_0; s_1; : : : ; s_{n-1}$ is a finite

binary sequence, the sequence $L_1; L_2; : : : ; L_n$ is called the *linear complexity profile of* s^n. The linear complexity profile of a sequence can be computed using the Berlekamp-Massey algorithm. The following properties of the linear complexity profile can be deduced .

2.5.9 Properties of linear complexity profile

Let $L_1; L_2; : : :$ be the linear complexity profile of a sequence $s = s_0; s_1; : : : .$

(i) If $j > i$, then $L_j \geq L_i$.

(ii) $L_{N+1} > L_N$ is possible only if $L_N \leq N/2$.

(iii) If $L_{N+1} > L_N$, then $L_{N+1} + L_N = N + 1$.

The linear complexity profile of a sequence s can be graphed by plotting the points $(N;L_N)$, $N \geq 1$, in the $N \times L$ plane and joining successive points by a horizontal line followed by a vertical line, if necessary (see Figure 2.3). Fact 2.5.9 can then be interpreted as saying that the graph of a linear complexity profile is non-decreasing. Moreover, a (vertical) jump in the graph can only occur from below the line L = N=2; if a jump occurs, then it is symmetric about this line. Fact 2.5.10 shows that the expected linear complexity of a random sequence should closely follow the line L = N=2.

2.5.10 Expected linear complexity profile of a random sequence Let $s = s_0; s_1;$ be a random sequence, and let L_N be the linear complexity of the subsequence $s^N = s_0; s_1; .. ;s_{N-1}$ for each $N \geq 1$. For any fixed index $N \geq 1$, the expected smallest j for which $L_{N+j} > L_N$ is 2 if $L_N \leq N/2$, or $2 + 2L_N - N$ if $L_N > N/2$. Moreover, the expected increase in linear complexity is 2 if $L_N \geq N/2$, or $N - 2L_N + 2$ if $L_N < N/2$.

2.5.11 Example (linear complexity profile) Consider the 20-periodic sequence s with cycle

$s^{20} = 1; 0; 0; 1; 0; 0; 1; 1; 1; 1; 0; 0; 0; 1; 0; 0; 1; 1; 1; 0:$

The linear complexity profile of s is 1; 1; 1; 3; 3; 3; 3; 5; 5; 5; 6; 6; 6; 8; 8; 8; 9; 9; 10; 10; 11,11; 11; 11; 14; 14; 14; 14; 15; 15; 15; 17; 17; 17; 18; 18; 19; 19; 19; 19; : : :.
Figure 2.3 shows the graph of the linear complexity profile of s.

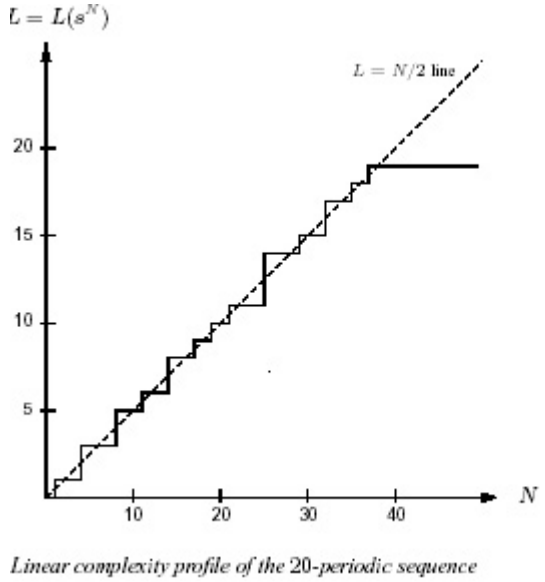


Figure 2.3: Linear complexity profile of the 20-periodic sequence of Example 2.5.11.

As is the case with all statistical tests for randomness, the condition that a sequence has a linear complexity profile that closely resembles that of a random sequence is *necessary* but not *sufficient* for s to be considered random. This point is illustrated in the following example.

2.5.12 Example (limitations of the linear complexity profile) The linear complexity profile of the sequence s defined as

$s_i = 1$; if $i = 2^j - 1$ for some $j \geq 0$,

0; otherwise; follows the line L = N/2 as closely as possible. That is, $L(s^N) = [(N + 1)]/2$ for all $N \geq 1$. However, the sequence s is clearly non-random.

2.6 Golomb's randomness postulates

Golomb's randomness postulates are presented here for historical reasons – they were one of the first attempts to establish some *necessary* conditions for a periodic pseudorandom sequence to look random. It is emphasized that these conditions are far from being *sufficient* for such sequences to be considered random. Unless otherwise stated, all sequences are binary sequences.

2.6.1 Definition Let $s = s_0, s_1, s_2, \ldots$ be an infinite sequence. The subsequence consisting of the first n terms of s is denoted by $s_n = s_0, s_1, \ldots, s_{n-1}$.

2.6.2 Definition The sequence $s = s_0, s_1, s_2, \ldots$ is said to be N-*periodic* if $s_i = s_{i+N}$ for all $i \geq 0$. The sequence s is *periodic* if it is N-periodic for some positive integer N. The *period* of a periodic sequence s is the smallest positive integer N for which s is N-periodic.

If s is a periodic sequence of period N, then the *cycle* of s is the subsequence s_N.

2.6.3 Definition Let s be a sequence. A *run* of s is a subsequence of s consisting of consecutive 0's or consecutive 1's which is neither preceded nor succeeded by the same symbol. A run of 0's is called a *gap*, while a run of 1's is called a *block*.

2.6.4 Definition Let $s = s_0, s_1, s_2, \ldots$ be a periodic sequence of period N. The *autocorrelation function* of s is the integer-valued function C(t) defined as

$$C(t) = 1/N \left(\sum_{i=0}^{N-1} (2s_i - 1) \cdot (2s_{i+t} - 1) \right) \text{ for } 1 \le t \le N-1$$

The autocorrelation function C(t) measures the amount of similarity between the sequence s and a shift of s by t positions. If s is a random periodic sequence of period N, then $|N \cdot C(t)|$ can be expected to be quite small for all values of t, $0 < t < N$.

2.6.5 Definition Let s be a periodic sequence of period N. *Golomb's randomness postulates* are the following.

R1: In the cycle s^N of s, the number of 1's differs from the number of 0's by at most 1.

R2: In the cycle s^N, at least half the runs have length 1, at least one-fourth have length 2, at least one-eighth have length 3, etc., as long as the number of runs so indicated exceeds 1. Moreover, for each of these lengths, there are (almost) equally many gaps and blocks.6

R3: The autocorrelation function C(t) is two-valued. That is for some integer K,

$$N \cdot C(t) = \sum_{i=0}^{N-1} (2s_i - 1) \cdot (2s_{i+t} - 1) = N, \text{ if } t = 0,$$

K, if $1 \le t \le N-1$

Note: Postulate R2 implies postulate R1.

2.6.6 Definition A binary sequence which satisfies Golomb's randomness postulates is called a *pseudo-noise sequence* or a *pn-sequence*.

Pseudo-noise sequences arise in practice as output sequences of maximum-length linear feedback shift registers (Fact 2.38).

2.6.7 Example (pn-sequence) Consider the periodic sequence s of period N = 15 with cycle

$s^{15} = 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1.$

The following shows that the sequence s satisfies Golomb's randomness postulates.

R1: The number of 0's in s_{15} is 7, while the number of 1's is 8.

R2: s_{15} has 8 runs. There are 4 runs of length 1 (2 gaps and 2 blocks), 2 runs of length 2 (1 gap and 1 block), 1 run of length 3 (1 gap), and 1 run of length 4 (1 block).

R3: The autocorrelation function C(t) takes on two values: C(0) = 1 and C(t) = -1/15 for $1 \leq t \leq 14$. Hence, s is a pn-sequence.

2.7 Attacks on Stream Ciphers

The purpose of a good key-stream generator is to produce an output which can ideally not be distinguished from a truly random source. This is however impossible, but the goal is to make it as close to a truly random symbol stream as possible. The cryptanalyst can refer to a multitude of statistical tools and theoretical knowledge to attempt an attack. A way to classify attacks on stream ciphers is as follows:

Ciphertext-only attack: This is the hardest type of attack for the cryptanalyst since no other information than the output of the cipher can be used. The goal is to recover the used key.

Known-plaintext attack: Having a ciphertext, the corresponding plaintext is known. In the case of a stream cipher, knowing the plaintext means knowing the key-stream which has been applied to it. The goal is to determine the key which generated the key-stream.

Distinguishing from a truly random sequence: This new security criterion was proposed by Coppersmith *et al.* , but it doesn't lead to any knowledge about the plaintext or the used key. Particular attacks often rely on the existence of certain building blocks inside the stream cipher. However, some don't: *e. g.* there can be a time/memory tradeoff:

the time effort of a brute force attack is separated into a smaller time effort and an additional memory effort which could lead to a practical attack. An approach, which can be better than brute force, is described subsequently.

2.7.1 Correlation Attacks

Correlation attacks on stream ciphers were introduced by Siegenthaler and try to establish a link between the key-stream and one of the LFSRs inside the generator. This is done via exploiting weaknesses in the combining function, which has several LFSRs as input. Once one LFSR is analyzed, attention is given to another LFSR. This approach is sometimes referred to as divide-and-conquer attack.

Further improvements have been made on the basic idea, and these are generally called fast correlation attacks.

2.7.2 Other Attacks

A short description and further references on other attacks such as linear consistency attack; sub-key guessing attack, inversion attack etc. can be found. A fairly new type of attack are algebraic attacks (or higher order correlation attack).

2.8 Specific Example of a Stream Cipher: RC4

RC4 was designed by Ron Rivest in 1987 in an attempt to make a stream cipher, which is more suitable for software implementations. He did not use LFSRs at all, but used a dynamic permutation instead. The design was a trade secret of RSA Inc. but leaked 1994 when someone anonymously posted the source code to the Cypherpunks mailing list. The security of RC4 was of course not affected as it is solely based on the used key. Even though this alleged version was never officially confirmed to be equivalent to the original version by RSA Inc., there is strong evidence to assume this. Subsequently, the notation RC4 refers to the alleged version of the algorithm. RC4 is one of the most popular stream ciphers, it is heavily used in SSL/TLS or IEEE 802.11 and is integrated into many widely used open-source libraries or applications of Microsoft or Oracle. Hardware implementations have been considered as well.

2.8.1 The RC4 Algorithm

RC4 specifies a whole family of algorithms whose differences lie in the used word-size n; typically the word-size is 8. Furthermore, key-sizes between n and 2048 bits are possible, however practical values are between 40 and 256 bits.

The algorithm consists of two parts, which are executed sequentially:

An initialization phase or KSA (Key Scheduling Algorithm)

An output phase or PRGA (Pseudo Random Generation Algorithm)

Both parts access an internal table of size $n \times 2^n$ bits, which can be viewed as an array containing 2^n words of size n. Let's look at the two parts of the algorithm in more detail. Even if the internal table has a size of $n \times 2^n$ bits, the total number of states is not $2^{n \times 2^n}$ (which would be 2^{2048} in the case of $n = 8$). Only permutations of 2^n words are possible, thus leading to a total number of $(2^n)!$ different table-states. Using a word-size of 8, the number of different table-states is $(2^8)! \approx 2^{1684}$.

2.9 Practical Considerations

Practical stream ciphers employ more than one of the above mentioned strategies to remove linearity from the output of LFSRs. Examples thereof are SNOW or SOBER. However, stream ciphers don't have to be based on LFSRs. Especially if software

performance is important, LFSRs are not the best choice. Stream ciphers, which take a complete different approach, are RC4, SEAL or SCREAM. The above description of LFSRs is suitable for hardware implementations, but its implementation in software is very inefficient. However, LFSRs do not have to use a binary representation, in fact any finite field is possible. A finite field more suitable for software implementation might be $GF((2^n)^m)$ where n is the word size of the underlying processor and m is the degree of the polynomial. One important property of synchronous stream ciphers is that the key-stream is independent of the plaintext and can be pre-generated. This strictly prohibits the reuse of the same key for different plaintexts. There are two solutions to this problem.

To seed the key-stream generator, use initialization vectors and subsequently derive a session key for it (*e. g.* using the output of a hash function with the concatenation of IV and key as input).

Reuse of key-stream; in this case, for practical reasons the stream ciphers should have the property to reach every point of the key-stream within sub linear amount of time.

Another feature has received considerable interest: the ability to add message integrity protection or message authentication to the ciphertext of the stream ciphers. This is especially important for the ciphertext of (synchronous) stream ciphers since bit manipulations in the ciphertext can lead to predictable modifications of the corresponding plaintext. Proposals like Helix , Multi-S01 from Hitachi, which has been submitted to CRYPTREC - a Japanese e-Government initiative, or SOBER 128 from Qualcomm, which is the successor of the NESSIE submission SOBER-t32, already include such a feature.

Chapter 3

Introduction to Random Number Testing

The need for random and pseudorandom numbers arises in many cryptographic applications. For example, common cryptosystems employ keys that must be generated in a random fashion.

Many cryptographic protocols also require random or pseudorandom inputs at various points, e.g., for auxiliary quantities used in generating digital signatures, or for generating challenges in authentication protocols.

This chapter discusses the randomness testing of random number and pseudorandom number generators that may be used for many purposes including cryptographic, modeling and simulation applications. The focus of this document is on those applications where randomness is required for cryptographic purposes. A set of statistical tests for randomness is described in this document. The National Institute of Standards and Technology (NIST) believes that these procedures are useful in detecting deviations of a *binary sequence* from randomness. However, a tester should note that apparent deviations from randomness may be due to either a poorly designed generator or to anomalies that appear in the binary sequence that is tested (i.e., a certain number of failures is expected in random sequences produced by a particular generator).

3.1 General Discussion

There are two basic types of generators used to produce random sequences: ***random number generators*** (RNGs) and ***pseudorandom number generators*** (PRNGs). For cryptographic applications, both of these generator types produce a stream of zeros and ones that may be divided into substreams or blocks of random numbers.

3.1.1 Randomness

A ***random*** bit sequence could be interpreted as the result of the flips of an unbiased "fair" coin with sides that are labeled "0" and "1," with each flip having a probability of exactly ½ of producing a "0" or "1." Furthermore, the flips are ***independent*** of each other: the result of any previous coin flip does *not* affect future coin flips. The unbiased "fair" coin is thus the perfect random bit stream generator, since the "0" and "1" values will be randomly distributed (and[0,1] uniformly distributed). All elements of the sequence are generated independently of each other, and the value of the next element in the sequence cannot be predicted, regardless of how many elements have already been produced.

Obviously, the use of unbiased coins for cryptographic purposes is impractical. Nonetheless, the hypothetical output of such an idealized generator of a true random sequence serves as a benchmark for the evaluation of random and pseudorandom number generators.

3.1.2 Unpredictability

Random and pseudorandom numbers generated for cryptographic applications should be unpredictable. In the case of PRNGs, if the seed is unknown, the next output number in the sequence should be unpredictable in spite of any knowledge of previous random numbers in the sequence. This property is known as forward unpredictability. It should also not be feasible to determine the seed from knowledge of any generated values (i.e., backward unpredictability is also required). No correlation between a seed and any value generated from that seed should be evident; each element of the sequence should appear to be the outcome of an independent random event whose probability is 1/2.

To ensure forward unpredictability, care must be exercised in obtaining seeds. The values produced by a PRNG are completely predictable if the seed and generation algorithm are known. Since in many cases the generation algorithm is publicly available, the seed must be kept secret and should not be derivable from the pseudorandom sequence that it produces. In addition, the seed itself must be unpredictable.

3.1.3 Random Number Generators (RNGs)

The first type of sequence generator is a random number generator (RNG). An RNG uses a nondeterministic source (i.e., the entropy source), along with some processing function (i.e., the entropy distillation process) to produce randomness. The use of a distillation process is needed to overcome any weakness in the entropy source that results in the production of non-random numbers (e.g., the occurrence of long strings of zeros or ones). The entropy source typically consists of some physical quantity, such as the noise in an electrical circuit, the timing of user processes (e.g., key strokes or mouse movements), or the quantum effects in a semiconductor. Various combinations of these inputs may be used.

The outputs of an RNG may be used directly as a random number or may be fed into a pseudorandom number generator (PRNG). To be used directly (i.e., without further processing), the output of any RNG needs to satisfy strict randomness criteria as

measured by statistical tests in order to determine that the physical sources of the RNG inputs appear random.

For example,

a physical source such as electronic noise may contain a superposition of regular structures, such as waves or other periodic phenomena, which may appear to be random, yet are determined to be non-random using statistical tests.

For cryptographic purposes, the output of RNGs needs to be unpredictable. However, some physical sources (e.g., date/time vectors) are quite predictable. These problems may be mitigated by combining outputs from different types of sources to use as the inputs for an RNG.

However, the resulting outputs from the RNG may still be deficient when evaluated by statistical tests. In addition, the production of high-quality random numbers may be too time consuming, making such production undesirable when a large quantity of random numbers is needed. To produce large quantities of random numbers, pseudorandom number generators may be preferable.

3.1.4 Pseudorandom Number Generators (PRNGs)

The second generator type is a pseudorandom number generator (PRNG). A PRNG uses one or more inputs and generates multiple “pseudorandom” numbers. Inputs to PRNGs are called ***seeds***. In contexts in which unpredictability is needed, the seed itself must be random and unpredictable. Hence, by default, a PRNG should obtain its seeds from the outputs of an RNG,i.e., a PRNG requires a RNG as a companion.

The outputs of a PRNG are typically deterministic functions of the seed; i.e., all true randomness is confined to seed generation. The deterministic nature of the process leads to the term “pseudorandom.” Since each element of a pseudorandom sequence is reproducible from its seed, only the seed needs to be saved if reproduction or validation of the pseudorandom sequence is required.

Ironically, pseudorandom numbers often appear to be more random than random numbers obtained from physical sources. If a pseudorandom sequence is properly constructed, each value in the sequence is produced from the previous value via transformations which appear to introduce additional randomness. A series of such transformations can eliminate statistical autocorrelations between input and output. Thus, the outputs of a PRNG may have better statistical properties and be produced faster than an RNG.

3.1.5 Testing

Various statistical tests can be applied to a sequence to attempt to compare and evaluate the sequence to a truly random sequence. Randomness is a probabilistic property; that is, the properties of a random sequence can be characterized and described in terms of probability. The likely outcome of statistical tests, when applied to a truly random sequence, is known a priori and can be described in probabilistic terms. There are an infinite number of possible statistical tests, each assessing the presence or absence of a "pattern" which, if detected, would indicate that the sequence is nonrandom. Because there are so many tests for judging whether a sequence is random or not, no specific finite set of tests is deemed "complete." In addition, the results of statistical testing must be interpreted with some care and caution to avoid incorrect conclusions about a specific generator.

A statistical test is formulated to test a specific ***null hypothesis*** (*H0*). For the purpose of this document, the null hypothesis under test is that the sequence being tested is *random*. Associated with this null hypothesis is the alternative hypothesis (Ha) which, for this document, is that the sequence is *not* random. For each applied test, a decision or conclusion is derived that accepts or rejects the null hypothesis, i.e., whether the generator is (or is not) producing random values, based on the sequence that was produced.

For each test, a relevant randomness statistic must be chosen and used to determine the acceptance or rejection of the null hypothesis. Under an assumption of randomness, such a statistic has a distribution of possible values. A theoretical reference distribution of this statistic under the null hypothesis is determined by mathematical methods. From this reference distribution, a ***critical value*** is determined (typically, this value is "far out" in the tails of the distribution, say out at the 99 % point). During a test, a test statistic value is computed on the data (the sequence being tested). This test statistic value is compared to the critical value. If the test statistic value exceeds the critical value, the null hypothesis for randomness is rejected.

Otherwise, the null hypothesis (the randomness hypothesis) is *not* rejected (i.e., the hypothesis is accepted).

3.1.6 Considerations for Randomness, Unpredictability and Testing

The following assumptions are made with respect to random binary sequences to be tested:

1. Uniformity: At any point in the generation of a sequence of random or pseudorandom bits, the occurrence of a zero or one is equally likely, i.e., the

probability of each is exactly 1/2. The expected number of zeros (or ones) is *n/2*, where *n* = the sequence length.

2. Scalability: Any test applicable to a sequence can also be applied to subsequences extracted at random. If a sequence is random, then any such extracted subsequence should also be random. Hence, any extracted subsequence should pass any test for randomness.

3. Consistency: The behavior of a generator must be consistent across starting values (seeds). It is inadequate to test a PRNG based on the output from a single seed, or an RNG on the basis of an output produced from a single physical output.

CHAPTER 4
RANDOM NUMBER GENERATION TESTS

The NIST Test Suite is a statistical package consisting of 16 tests that were developed to test the randomness of (arbitrarily long) binary sequences produced by either hardware or software based cryptographic random or pseudorandom number generators. These tests focus on a variety of different types of non-randomness that could exist in a sequence. Some tests are decomposable into a variety of subtests. The 16 tests are:

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

The order of the application of the tests in the test suite is arbitrary. However, it is recommended that the **Frequency test** be run first, since this supplies the most basic evidence for the existence of non-randomness in a sequence, specifically, non-uniformity. If this test fails, the likelihood of other tests failing is high. (Note: The most time-consuming statistical test is the Linear Complexity test).

A number of tests in the test suite have the ***standard normal*** and the ***chi-square (χ^2)*** as reference distributions. If the sequence under test is in fact non-random, the calculated test statistic will fall in extreme regions of the reference distribution. The

standard normal distribution (i.e., the bell-shaped curve) is used to compare the value of the test statistic obtained from the RNG with the expected value of the statistic under the assumption of randomness. The test statistic for the standard normal distribution is of the form $z = (x - \mu)/\sigma$, where x is the sample test statistic value, and μ and σ^2 are the expected value and the variance of the test statistic. The χ^2 distribution (i.e., a left skewed curve) is used to compare the goodness-of-fit of the observed frequencies of a sample measure to the corresponding expected frequencies of the hypothesized distribution. The test statistic is of the form $\chi^2 = \sum((o_i-e_i)^2/e_i)$, where o_i and e_i are the observed and expected frequencies of occurrence of the measure, respectively.

For many of the tests in this test suite, the assumption has been made that the size of the sequence length, n, is large (of the order 10^3 to 10^7). For such large sample sizes of n, asymptotic reference distributions have been derived and applied to carry out the tests. Most of the tests are applicable for smaller values of n. However, if used for smaller values of n, the asymptotic reference distributions would be inappropriate and would need to be replaced by exact distributions that would commonly be difficult to compute.

4.1 Frequency (Monobit) Test

4.1.1 Test Purpose

The focus of the test is the proportion of zeroes and ones for the entire sequence. The purpose of this test is to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence. The test assesses the closeness of the fraction of ones to ½, that is, the number of ones and zeroes in a sequence should be about the same. All subsequent tests depend on the passing of this test; there is no evidence to indicate that the tested sequence is non-random.

4.1.2 Test Description

(1) Conversion to ±1: The zeros and ones of the input sequence ($\in$) are converted to values of –1 and +1 and are added together to produce $S_n = X_1 + X_2 + \ldots X_3$, where $X_i = 2\in_i - 1$.

For example, if $\in = 1011010101$, then n=10 and $S_n = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 = 2$.

(2) Compute the test statistic $s_{obs} = |S_n| / n^{1/2}$

For the example in this section, $s_{obs} = |2| /10^{1/2} = .632455532$.

(3) Compute *P-value* = ***erfc*** **(**$s_{obs}/\ 2^{1/2}$**)** **,** where ***erfc*** is the complementary error function .

For the example in this section, *P-value* = ***erfc*** (0.*632455532/*$2^{1/2}$ =0.527089.

4.1.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.1.4 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 100 bits (i.e., $n \geq 100$).

4.2 Frequency Test within a Block

4.2.1 Test Purpose

The focus of the test is the proportion of ones within M-bit blocks. The purpose of this test is to determine whether the frequency of ones in an M-bit block is approximately M/2, as would be expected under an assumption of randomness. For block size *M*=1, this test degenerates to test1, the Frequency (Monobit) test.

4.2.2 Test Description

(1) Partition the input sequence into *N* = |n/M| non overlapping blocks. Discard any unused bits.

For example, if *n = 10, M = 3* and ∈ *= 0110011010*, 3 blocks (*N = 3*) would be created, consisting of *011, 001 and 101*. The final *0* would be discarded.

(2) Determine the proportion π_i of ones in each *M*-bit block using the equation

$$\prod_i = \frac{\sum_{j=1}^{M} \in_{(i-1)M+j}}{M} \quad \text{For } 1 \leq i \leq N.$$

For the example in this section, $\prod_1 = 2/3$, $\prod_2 = 1/3$, and $\prod_3 = 2/3$.

(3) Compute the χ^2 statistic: $\chi^2\ (obs) = 4\ M \sum_{i=1}^{N} (\prod_i - ½)^2$

(4) Compute *P-value* = **igamc** (*N/2,* χ^2(obs)/2) , where **igamc** is the incomplete gamma function for Q(*a,x*) .

Note: that *Q(a,x) = 1-P(a,x).*

4.2.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.2.4 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 100 bits (i.e., $n \geq 100$). Note that $n \geq MN$. The block size M should be selected such that $M \geq 20$, $M > .01n$ and $N < 100$.

4.3 Runs Test

4.3.1 Test Purpose

The focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits. A run of length k consists of exactly k identical bits and is bounded before and after with a bit of the opposite value. The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence . In particular, this test determines whether the oscillation between such zeros and ones is too fast or too slow.

4.3.2 Test Description

Note: The Runs test carries out a Frequency test as a prerequisite.

(1)Compute the pre-test proportion $\prod$ of ones in the input sequence: $\prod = \sum_j \in_j /n$

For example, if $\in$ = *1001101011*, then n=10 and $\prod$ = *6/10 = 3/5*.

(2) Determine if the prerequisite Frequency test is passed: If it can be shown that $|\prod - 1/2| \geq \tau$, then the Runs test need not be performed (i.e., the test should not have been run because of a failure to pass test 1, the Frequency (Monobit) test). If the test is not applicable, then the *P-value* is set to 0.0000. Note that for this test, $\tau = 2/n^{1/2}$ has been pre-defined in the test code.

(3) Compute the test statistic $Vn(obs) = \sum_{k=i}^{n-i} r(k)+1$, where $r(k)=0$ if $\in_k = \in_k +1$, and r$(k)=1$ otherwise.

(4) Compute *P-value* = ***erfc*** $[(\frac{|Vn(obs) - 2n\prod(1-\prod)}{2\sqrt{2n}\prod(1-\prod)}]$

4.3.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.3.4 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 100 bits (i.e., $n \geq 100$).

4.4 Test for the Longest Run of Ones in a Block

4.4.1 Test Purpose

The focus of the test is the longest run of ones within M-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence.

Note that an irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeroes. Therefore, only a test for ones is necessary.

4.4.2 Test Description

(1) Divide the sequence into *M*-bit blocks.

(2) Tabulate the frequencies v_i of the longest runs of ones in each block into categories, where each cell contains the number of runs of ones of a given length.

For the values of *M* supported by the test code, the v_i cells will hold the following counts:

V_i	M=8	M=128	M=104
V_0	≤1	≤4	≤10
V_1	2	5	11
V_2	3	6	12
V_3	≥4	7	13
V_4		8	14
V_5		≥9	15
V_6			≥16

(3) Compute χ^2 *(obs)* = $\sum_{i=0}^{k} \frac{(v_i - N\prod_i)^2}{N\prod_i} = 4.882605$

(4) Compute *P-value* = **igamc** (k/2**,** χ^2 *(obs)/2)*

4.4.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.4.4 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of bits as specified in the table .

Minimum n	M
128	8
6272	128
750,000	10^4

4.5 Binary Matrix Rank Test

4.5.1 Test Purpose

The focus of the test is the rank of disjoint sub-matrices of the entire sequence. The purpose of this test is to check for linear dependence among fixed length substrings of the original sequence.

4.5.2 Test Description

(1)Sequentially divide the sequence into $M \cdot Q$-bit disjoint blocks; there will exist N=[n/MQ] such blocks. Discarded bits will be reported as not being used in the computation within each block. Collect the $M \cdot Q$ bit segments into M by Q matrices. Each row of the matrix is filled with successive Q-bit blocks of the original sequence $\in$.

For example, if $n = 20$, $M = Q = 3$, and $\in = 01011001001010101101$, then partition the stream into $N = [n/3.3]=2$ matrices of cardinality $M \cdot Q$ ($3 \cdot 3 = 9$). Note that the last two bits (0 and 1) will be discarded. The two matrices are

$$\begin{bmatrix} 010 \\ 110 \\ 010 \end{bmatrix} \text{ and } \begin{bmatrix} 010 \\ 101 \\ 011 \end{bmatrix}$$

Note that the first matrix consists of the first three bits in row 1, the second set of three bits in row 2, and the third set of three bits in row 3. The second matrix is similarly constructed using the next nine bits in the sequence.

(2) Determine the binary rank (R_l) of each matrix, where l = l….,N . The rank of the first matrix is 2 ($R_1 = 2$), and the rank of the second matrix is 3 ($R_2 = 3$).

(3) Let F_M = the number of matrices with $R_l = M$ (full rank),

F_{M-1} = the number of matrices with $R_l = M-1$ (full rank - 1),

$N - F_M - F_{M-1}$ = the number of matrices remaining. For the example in this section, $F_M = F_3 = 1$ (R_2 has the full rank of 3), $F_{M-1} = F_2 = 1$ (R_1 has rank 2), and no matrix has any lower rank.

(4) Compute χ^2(obs) .

(5) Compute *P -value* = $e^{-\chi 2(obs)/2}$. = 0.741948(for this section)

4.5.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.5.4 Input Size Recommendations

The probabilities for $M = Q = 32$ have been calculated and inserted into the test code. Other choices of *M* and *Q* may be selected, but the probabilities would need to be calculated. The minimum number of bits to be tested must be such that $n \geq 38MQ$ (i.e., at least 38 matrices are created). For $M = Q = 32$, each sequence to be tested should consist of a minimum of 38,912.

4.6 Discrete Fourier Transform (Spectral) Test

4.6.1 Test Purpose

The focus of this test is the peak heights in the Discrete Fourier Transform of the sequence. The purpose of this test is to detect periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness. The intention is to detect whether the number of peaks exceeding the 95 % threshold is significantly different than 5 %.

4.6.2 Test Description

(1) The zeros and ones of the input sequence ($\in$) are converted to values of –1 and +1 to create the sequence $X = x_1, x_2, \ldots, x_n$, where $x_i = 2\in_i - 1$.

For example, if $n = 10$ and $e = 1001010011$, then $X = 1, -1, -1, 1, -1, 1, -1, -1, 1, 1$.

(2) Apply a Discrete Fourier transform (DFT) on X to produce: $S = DFT(X)$. A sequence of complex variables is produced which represents periodic components of the sequence of bits at different frequencies

(3) Calculate $M = modulus(S') \equiv |S'|$, where S' is the substring consisting of the first $n/2$ elements in *S*, and the modulus function produces a sequence of peak heights.

(4) Compute $T = n^{1/3}$ = the 95 % peak height threshold value. Under an assumption of randomness, 95 % of the values obtained from the test should not exceed *T*.

(5) Compute $N_0 = .95n/2$. N_0 is the expected theoretical (95 %) number of peaks (under the assumption of randomness) that are less than *T*.

(6) Compute N_1 = the actual observed number of peaks in M that are less than T. For the example in this section, $N_1 = 4$.

(7) Compute $d = \dfrac{(N_1 - N_0)}{\sqrt{n(0.95)(0.05)/2}}$

(8) Compute $P\text{-}value = erfc\left(|d|/\sqrt{2}\right)$.

4.6.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.6.4 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 1000 bits (i.e., $n \geq 1000$).

4.7 Non-overlapping Template Matching Test

4.7.1 Test Purpose

The focus of this test is the number of occurrences of pre-specified target strings. The purpose of this test is to detect generators that produce too many occurrences of a given non-periodic (aperiodic) pattern. For this test and for the Overlapping Template Matching test of Section 2.8, an *m*-bit window is used to search for a specific m-bit pattern. If the pattern is *not* found, the window slides one bit position. If the pattern is found, the window is reset to the bit after the found pattern, and the search resumes.

4.7.2 Test Description

(1) Partition the sequence into N independent blocks of length M.

For example, if $e = 10100100101110010110$, then $n = 20$. If $N = 2$ and $M = 10$, then the two blocks would be 1010010010 and 1110010110.

(2) Let Wj ($j = 1, \ldots, N$) be the number of times that B (the template) occurs within the block j. Note that $j = 1,\ldots,N$. The search for matches proceeds by creating an m-bit window on the sequence, comparing the bits within that window against the template. If there is no match, the window slides over one bit , e.g., if $m = 3$ and the current window contains bits 3 to 5, then the next window will contain bits 4 to 6. If there is a match, the window slides over m bits, e.g., if the current (successful) window contains bits 3 to 5, then the next window will contain bits 6 to 8.

(3) Under an assumption of randomness, compute the theoretical mean μ and variance σ^2

$\mu = (M-m+1)/2^m$, $\sigma^2 = M(1/2^m - 2m-1/2^{2m})$

(4) Compute χ^2(obs.) = $\sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2}$

(5) Compute *P-value* = **igamc** (N/2, χ^2(obs.)/2). Note that multiple *P-values* will be computed, i.e., one *P-value* will be computed for each template. For $m = 9$, up to 148 *P-values* may be computed; for $m = 10$, up to 284 *P-values* may be computed.

4.7.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.7.4 Input Size Recommendations

The test code has been written to provide templates for $m = 2, 3,\ldots,10$. It is recommended that $m = 9$ or $m = 10$ be specified to obtain meaningful results. Although $N = 8$ has been specified in the test code, the code may be altered to other sizes. However, N should be chosen such that $N \leq 100$ to be assured that the *P-values* are valid. The test code has been written to assume a sequence length of $n = 10^6$ (entered via a calling parameter) and $M = 131072$ (hard coded). If values other than these are desired, be sure that $M > 0.01 \cdot n$ and $N = \lfloor n / M \rfloor$.

4.8 Overlapping Template Matching Test

4.8.1 Test Purpose

The focus of the Overlapping Template Matching test is the number of occurrences of pre specified target strings. Both this test and the Non-overlapping Template Matching test of Previous test use an *m*-bit window to search for a specific *m*-bit pattern. As with the test in Previous test, if the pattern is *not* found, the window slides one bit position. The difference between this test and the test in Previous test is that when the pattern *is* found, the window slides only one bit before resuming the search.

4.8.2 Test Description

(1) Partition the sequence into N independent blocks of length M.

For example, if $\in$ = *10111011110010110100*1110010111011111000*0101101001*, then $n = 50$. If $K = 2$, $M = 10$ and $N = 5$, then the five blocks are *1011101111*, *0010110100*, *0111001011*, *1011111000*, and *0101101001* .

(2) Calculate the number of occurrences of B in each of the N blocks. The search for matches proceeds by creating an *m*-bit window on the sequence, comparing the bits within that window against B and incrementing a counter when there is a match. The

window slides over one bit after each examination, e.g., if $m = 4$ and the first window contains bits 42 to 45, the next window consists of bits 43 to 46. Record the number of occurrences of B in each block by incrementing an array v_i (where $i = 0,\ldots 5$), such that v_0 is incremented when there are no occurrences of B in a substring, v_1 is incremented for one occurrence of B,…and v_5 is incremented for 5 or more occurrences of B.

(3) Compute values for l and h that will be used to compute the theoretical probabilities π_i corresponding to the classes of v_0:

$\mu = (M-m+1)/2^m$ $\eta = \lambda/2$

(4) Compute $\chi^2(\text{obs}) = \sum (v_i - N\pi_i)^2 / N\pi_i$, where $\pi_0 = 0.367879$, $\pi_1 = 0.183940$, $\pi_2 = 0.137955$, $\pi_3 = 0.099634$, $\pi_4 = 0.069935$ and $\pi_5 = 0.140657$ after computation.

(5) Compute *P-value* = **igamc**$(5/2, \chi^2(\text{obs})/2)$

4.8.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.8.4 Input Size Recommendations

The values of K, M and N have been chosen such that each sequence to be tested consists of a minimum of 106 bits (i.e., $n \geq 106$). Various values of m may be selected, but for the time being, NIST recommends $m = 9$ or $m = 10$. If other values are desired, please choose these values as follows:

- $n \geq MN$.
- N should be chosen so that $N \cdot (\min \pi_i) > 5$.
- $l = (M-m+1)/2^m \approx 2$
- m should be chosen so that $m \approx \log_2 M$
- Choose K so that $K \approx 2l$. Note that the π_i values would need to be
- recalculated for values of K other than 5.

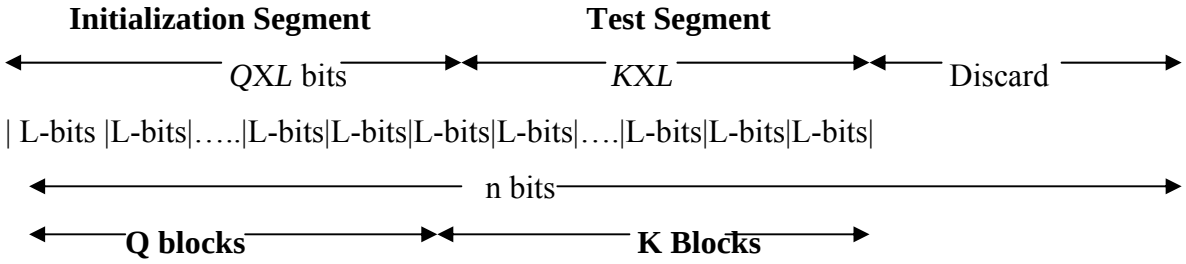
4.9 Maurer's "Universal Statistical" Test

4.9.1 Test Purpose

The focus of this test is the number of bits between matching patterns (a measure that is related to the length of a compressed sequence). The purpose of the test is to detect whether or not the sequence can be significantly compressed without loss of information. A significantly compressible sequence is considered to be non-random.

4.9.2 Test Description

(1)The *n*-bit sequence (*e*) is partitioned into two segments: an initialization segment consisting of *Q L*-bit non-overlapping blocks, and a test segment consisting of *K L*-bit non-overlapping blocks. Bits remaining at the end of the sequence that do not form a complete *L*-bit block are discarded.



The first *Q* blocks are used to initialize the test. The remaining *K* blocks are the test blocks ($K = \lfloor n/L \rfloor - Q$).

(2) Using the initialization segment, a table is created for each possible *L*-bit value (i.e., the *L*-bit value is used as an index into the table). The block number of the last occurrence of each *L*-bit block is noted in the table (i.e., For *i* from *1* to *Q*, *Tj*= *i*, where *j* is the decimal representation of the contents of the i^{th} *L*-bit block).

(3) Examine each of the *K* blocks in the test segment and determine the number of blocks since the last occurrence of the same *L*-bit block (i.e., $i - T_j$). Replace the value in the table with the location of the current block (i.e., $T_j = i$). Add the calculated distance between re-occurrences of the same *L*-bit block to an accumulating log_2 sum of all the differences detected in the *K* blocks (i.e., $sum = sum + log_2(i - T_j)$).

(4) Compute the test statistic: $f_n = 1/k \sum_{i=Q+1}^{Q+K} \log_2(i - T_j)$, where T_j is the table entry corresponding to the decimal representation of the contents of the i^{th} *L*-bit block.

(5) Compute *P-value* = ***erfc*** $\left(\left| \frac{fn - epectedValue(L)}{\sqrt{2}\sigma} \right| \right)$**,**where ***erfc*** is error function.

and *expectedValue(L)* and σ are taken from a table of precomputed values. Under an assumption of randomness, the sample mean, *expectedValue(L)*, is the theoretical expected value of the computed statistic for the given *L*-bit length. The theoretical standard deviation is given by $\sigma = c\sqrt{\frac{\text{var}\,iance(L)}{K}}$,

where $c = 0.7 - 0.8/L + (4 + 32/L)\, K^{-3/L}/15$

4.9.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.9.4 Input Size Recommendations

This test requires a long sequence of bits $[n \geq (Q + K)L]$ which are divided into two segments of L-bit blocks, where L should be chosen so that $6 \leq L \leq 16$. The first segment consists of Q initialization blocks, where Q should be chosen so that $Q = 10 \cdot 2^L$. The second segment consists of K test blocks, where $K = \lceil n/L \rceil - Q \approx 1000 \cdot 2^L$. The values of L, Q and n should be chosen from the table.

4.10 Lempel-Ziv Compression Test

4.10.1 Test Purpose

The focus of this test is the number of cumulatively distinct patterns (words) in the sequence. The purpose of the test is to determine how far the tested sequence can be compressed. The sequence is considered to be non-random if it can be significantly compressed. A random sequence will have a characteristic number of distinct patterns.

4.10.3 Test Description

(1) Parse the sequence into consecutive, disjoint and distinct words that will form a "dictionary" of words in the sequence. This is accomplished by creating substrings from consecutive bits of the sequence until a substring is created that has not been found previously in the sequence. The resulting substring is a new word in the dictionary.

Let *Wobs* = the number of cumulatively distinct words.

(2) Compute *P-value* = ½ ***erfc*** $\left(\frac{\mu - Wobs}{\sqrt{2\sigma^2}} \right)$ where μ = *69586.25* and σ $=(70.448718)^{1/2}$ when $n = 10^6$. For other values of n, the values of m and s would need to be calculated. Note that since no known theory is available to determine the exact values of μ and σ, these values were computed (under an assumption of randomness) using SHA-1. The Blum-Blum-Shub generator will give similar results for μ and σ^2.

4.10.4 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.10.5 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 1,000,000 bits (i.e., $n \geq 10^6$).

4.11 Linear Complexity Test

4.11.1 Test Purpose

The focus of this test is the length of a linear feedback shift register (LFSR). The purpose of this test is to determine whether or not the sequence is complex enough to be considered random. Random sequences are characterized by longer LFSRs. An LFSR that is too short implies non-randomness.

4.11.2 Test Description

(1) Partition the *n*-bit sequence into *N* independent blocks of *M* bits, where $n = MN$.

(2) Using the Berlekamp-Massey algorithm5, determine the linear complexity *Li* of each of the *N* blocks ($i = 1,\ldots,N$). *Li* is the length of the shortest linear feedback shift register sequence that generates all bits in the block *i*. Within any L_i-bit sequence, some combination of the bits, when added together modulo 2, produces the next bit in the sequence (bit $L_i + 1$).

For example, if $M = 13$ and the block to be tested is *1101011110001*, then $Li = 4$, and the sequence is produced by adding the 1st and 2nd bits within a 4-bit subsequence to produce the next bit (the 5th bit). The examination proceeded as follows:

The first 4 bits and the resulting 5th bit:

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5
1	1	0	1	0
1	0	1	0	1
0	1	0	1	1
1	0	1	1	1
0	1	1	1	1
1	1	1	1	0
1	1	1	0	0
1	1	0	0	0
1	0	0	0	1

Bits 2-5 and the resulting 6th bit:

Bits 3-6 and the resulting 7th bit:

…….

Bits 9-12 and the resulting 13th bit:

For this block, the trial feedback algorithm works. If this were not the case, other feedback algorithms would be attempted for the block (e.g., adding bits 1 and 3 to produce bit 5, or adding bits 1, 2 and 3 to produce bit 6, etc.).

(3) Under an assumption of randomness, calculate the theoretical mean μ:

$$\mu = M/2 + \frac{(9+(-1)M+1)}{36} - \left(\frac{M/3+2/9}{2M}\right)$$

here μ = 6.777222.

(4) For each substring, calculate a value of *Ti*, where $T_i = (-1)^M \bullet (L_i - \mu) + 2/9$.

Here $T_I = 2.999444$.

(5) Record the T_i values in $v_0, \ldots, v_6$ as follows:

If: $Ti \leq -2.5$	Increment v_o by one
$-2.5 < Ti \leq -1.5$	Increment v_1 by one
$-1.5 < Ti \leq -0.5$	Increment v_2 by one
$-0.5 < Ti \leq 0.5$	Increment v_3 by one
$0.5 < Ti \leq 1.5$	Increment v_4 by one
$1.5 < Ti \leq 2.5$	Increment v_5 by one
$Ti > 2.5$	Increment v_6 by one

(6) Compute $\chi^2(\text{obs}) = \sum_{i=0}^{k} \frac{(v_i - n\Pi_i)^2}{N\Pi_i}$, where $\pi_0 = 0.01047$, $\pi_1 = 0.03125$, $\pi_2 = 0.125$, $\pi_3 = 0.5$, $\pi_4 = 0.25$, $\pi_5 = 0.0625$, $\pi_6 = 0.02078$ are the probabilities computed by the equations in Section 3.11.

(7) Compute *P-value* = **igamc** (K/2, χ^2(obs)/2)

4.11.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.11.4 Input Size recommendations

Choose $n \geq 10^6$. The value of *M* must be in the range $500 \leq M \leq 5000$, and $N \geq 200$ for the χ^2 result to be valid .

4.11.5 Example

(input)	$\in$ = "the first 1,000,000 binary digits in the expansion of *e*"
(input)	$n = 1000000 = 10^6$, $M = 1000$
(processing)	$v_0 = 11$; $v_1 = 31$; $v_2 = 116$; $v_3 = 501$; $v_4 = 258$; $v_5 = 57$; $v_6 = 26$
(processing)	χ^2 *(obs)* = 2.700348
(output)	*P-value* = 0.845406

(conclusion) Since the *P-value* $\geq$ 0.01, accept the sequence as random.

4.12 Serial Test

4.12.1 Test Purpose

The focus of this test is the frequency of all possible overlapping *m*-bit patterns across the entire sequence. The purpose of this test is to determine whether the number of occurrences of the 2*m* *m*-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity; that is, every *m*-bit pattern has the same chance of appearing as every other *m*-bit pattern. Note that for $m = 1$, the Serial test is equivalent to the Frequency test .

4.12.2 Test Description

(1) Form an augmented sequence $\in$‘: Extend the sequence by appending the first *m-1* bits to the end of the sequence for distinct values of *n*.

For example, given $n = 10$ and $\in = 0011011101$. If $m = 3$, then $\in' = 001101110100$. If $m = 2$, then $\in' = 00110111010$. If $m = 1$, then $\in'$ = the original sequence *0011011101*.

(2) Determine the frequency of all possible overlapping *m*-bit blocks, all possible overlapping (*m-1)*-bit blocks and all possible overlapping *(m-2)*-bit blocks. Let $v_i \ldots i_m$ denote the frequency of the *m* bit pattern $i_1 \ldots i_m$; let $v_i \ldots v_{m-1}$ denote the frequency of the (*m-1)*-bit pattern $i_1 \ldots i_{m-1}$; and let $v_i \ldots i_{m-2}$ denote the frequency of the (*m-2)*-bit pattern $i_1 \ldots i_{m-2}$.

(3) Compute: $\Psi_m^2 = 2^m/n \sum_{i1 \ldots im} (v_{i1 \ldots im} - n/2^m)^2 = 2^m/n \sum_{i1 \ldots im} v_{i1 \ldots im-1}^2 - n$

similarly for Ψ_{m-1}^2 and Ψ_{m-2}^2.

(4) Compute $\nabla^2\Psi_m^2 = \Psi_m^2 - \Psi_{m-1}^2$

$\nabla^2\Psi_m^2 = \Psi_m^2 - 2\Psi_{m-1}^2 + \Psi_{m-2}^2$

(5) Compute: *P-value1* = **igamc** $(2^{m-2} - \nabla\Psi_m^2)$

P-value2 = **igamc** $(2^{m-3} - \nabla^2\Psi_m^2)$

4.12.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.12.4 Input Size Recommendations

Choose *m* and *n* such that $m < \lfloor \log_2 n \rfloor - 2$.

4.13 Approximate Entropy Test

4.13.1 Test Purpose

As with the Serial test of Section 2.12, the focus of this test is the frequency of all possible overlapping m-bit patterns across the entire sequence. The purpose of the test

is to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (*m* and *m+1*) against the expected result for a random sequence.

4.13.2 Test Description

(1) Augment the *n*-bit sequence to create *n* overlapping *m*-bit sequences by appending *m-1* bits from the beginning of the sequence to the end of the sequence.

For example, if $\in$ = *0100110101* and *m = 3*, then *n = 10*. Append the 0 and 1 at the beginning of the sequence to the end of the sequence. The sequence to be tested becomes *010011010101*. (Note: This is done for each value of *m*.)

(2) A frequency count is made of the *n* overlapping blocks (e.g., if a block containing $\in_j$ to $\in_{j+m-1}$ is examined at time *j*, then the block containing $\in_{j+1}$ to $\in_{j+m}$ is examined at time *j+1*). Let the count of the possible *m*-bit ((*m*+1)-bit) values be represented as C_i^m ,where *i* is the *m*-bit value.

(3) Compute C_i^m =# *i/n* for each value of i.

(4) Compute $\varphi_m = \sum_{i=0}^{2^m-1} \Pi_i \log \Pi_i$ where $\pi = C^3_j$, and $j=log_2\ i$.

(5) Repeat steps 1-4, replacing *m* by *m+1*.

(6) Compute the test statistic: $\chi^2 = 2n[log\ 2 - ApEn(m)]$, where $ApEn(m) = \phi^{(m)} - \phi^{(m+1)}$

(7) Compute *P-value* = **igamc**$(2^{m-1},\chi^2/2)$.

4.13.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.13.4 Input Size Recommendations

Choose *m* and *n* such that $m < \lfloor \log_2 n \rfloor - 2$.

4.14 Cumulative Sums (Cusum) Test

4.14.1 Test Purpose

The focus of this test is the maximal excursion (from zero) of the random walk defined by the cumulative sum of adjusted (-1, +1) digits in the sequence. The purpose of the test is to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. This cumulative sum may be considered as a random walk. For a random sequence, the excursions of

the random walk should be near zero. For certain types of non-random sequences, the excursions of this random walk from zero will be large.

4.14.2 Test Description

(1) Form a normalized sequence: The zeros and ones of the input sequence ($\in$) are converted to values X_i of –1 and +1 using $X_i = 2\in_i - 1$.

For example, if $\in = 1011010111$, then $X = 1, (-1), 1, 1, (-1), 1, (-1), 1, 1, 1$.

(2) Compute partial sums S_i of successively larger subsequences, each starting with X_1 (if *mode = 0*) or X_n (if *mode = 1*).

(3) Compute the test statistic $z = max_{1\le k\le n}|S_k|$, where $max_{1\le k\le n}|Sk|$ is the largest of the absolute values of the partial sums S_k.

(4) Compute *P-value* = $1 - \sum_{(-n/z+1)}^{(n/z-1)/4}\left[\Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k-1)z}{\sqrt{n}}\right)\right] +$

$$\sum_{(-n/z-3)}^{(n/z-1)/4}\left[\Phi\left(\frac{(4k+3)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right)\right]$$

where ϕ is the Standard Normal Cumulative Probability Distribution Function .

4.14.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.14.4 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 100 bits (i.e., $n \ge 100$).

4.15 Random Excursions Test

4.15.1 Test Purpose

The focus of this test is the number of cycles having exactly *K* visits in a cumulative sum random walk. The cumulative sum random walk is derived from partial sums after the (0,1) sequence is transferred to the appropriate (-1, +1) sequence. A cycle of a random walk consists of a sequence of steps of unit length taken at random that begin at and return to the origin. The purpose of this test is to determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence. This test is actually a series of eight tests (and conclusions), one test and conclusion for each of the states: -4, -3, -2, -1 and +1, +2, +3, +4.

4.15.2 Test Description

(1) Form a normalized (-1, +1) sequence X: The zeros and ones of the input sequence (ϵ) are changed to values of –1 and +1 via $X_i = 2\epsilon_i - 1$.

For example, if ϵ = 0110110101, then $n = 10$ and $X = -1, 1, 1, -1, 1, 1, -1, 1, -1, 1$.

(2) Compute the partial sums S_i of successively larger subsequences, each starting with X_1.

Form the set $S = \{S_i\}$.

$S_1 = X_1$

$S_2 = X_1 + X_2$

$S3 = X_1 + X_2 + X_3$

$S_k = X_1 + X_2 + X_3 + \ldots + X_k$

.

.

$Sn = X_1 + X_2 + X_3 + \ldots . + X_k + \ldots + X_n$

(3) Form a new sequence S' by attaching zeros before and after the set S. That is, $S' = 0, s_1, s_2, \ldots, s_n, 0$.

(4) Let J = the total number of zero crossings in S', where a zero crossing is a value of zero in that occurs in S' after the starting zero. J is also the number of cycles in S', where a cycle of S'is a subsequence of S' consisting of an occurrence of zero, followed by no zero values, and ending with another zero. The ending zero in one cycle may be the beginning zero in another cycle. The number of cycles in S ' is the number of zero crossings. If $J < 500$, discontinue the test.

(5) For each cycle and for each non-zero state value x having values $-4 \le x \le -1$ and $1 \le x \le 4$, compute the frequency of each x within each cycle.

(6) For each of the eight states of x, compute $v_k(x)$ = the total number of cycles in which state x occurs exactly k times among all cycles, for $k = 0, 1, \ldots, 5$ (for $k = 5$, all frequencies ≥ 5 are stored in $v_5(x)$). Note that $\sum_{k=0}^{5} v_k(x) = j$.

(7) For each of the eight states of x, compute the test statistic $\chi^2 = \sum_{k=0}^{5} \frac{(v_k(x) - j\Pi_k(x))^2}{j\Pi_k(x)}$, where $\pi_k(x)$ is the probability that the state x occurs k times in a random distribution .

(8) For each state of x, compute *P-value* = **igamc**$(5/2, \chi^2(obs)/2$. Eight *P-values* will be produced.

4.15.3 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.15.4 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 1,000,000 bits (i.e., $n \geq 10^6$).

4.16 Random Excursions Variant Test

4.16.1 Test Purpose

The focus of this test is the total number of times that a particular state is visited (i.e., occurs) in a cumulative sum random walk. The purpose of this test is to detect deviations from the expected number of visits to various states in the random walk. This test is actually a series of eighteen tests (and conclusions), one test and conclusion for each of the states: -9, -8, …, -1 and +1, +2, …, +9.

(1) Form a normalized (-1, +1) sequence *X*: The zeros and ones of the input sequence ($\in$) are changed to values of –1 and +1 via $X_i = 2\in_i - 1$.

For example, if $\in$ = 0110110101, then $n = 10$ and $X = -1, 1, 1, -1, 1, 1, -1, 1, -1, 1$.

(2) Compute the partial sums S_i of successively larger subsequences, each starting with X_1.

Form the set $S = \{S_i\}$.

$S_1 = X_1$

$S_2 = X_1 + X_2$

$S3 = X_1 + X_2 + X_3$

$S_k = X_1 + X_2 + X_3 + \ldots + X_k$

.

.

$Sn = X_1 + X_2 + X_3 + \ldots . + X_k + \ldots + X_n$

(3) Form a new sequence *S*' by attaching zeros before and after the set *S*. That is, $S' = 0, s_1, s_2, \ldots, s_n, 0$.

(4) For each of the eighteen non-zero states of *x*, compute $\xi(x)$ = the total number of times that state *x* occurred across all *J* cycles.

(5) For each $\xi(x)$, compute *P-value* = $\boldsymbol{erfc}\left(\frac{|\xi(x) - J|}{\sqrt{2J(4|x|-2)}}\right)$. Eighteen *P-values* are computed.

4.16.2 Decision Rule (at the 1 % Level)

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

4.16.3 Input Size Recommendations

It is recommended that each sequence to be tested consist of a minimum of 1,000,000 bits (i.e., $n \geq 10^6$).

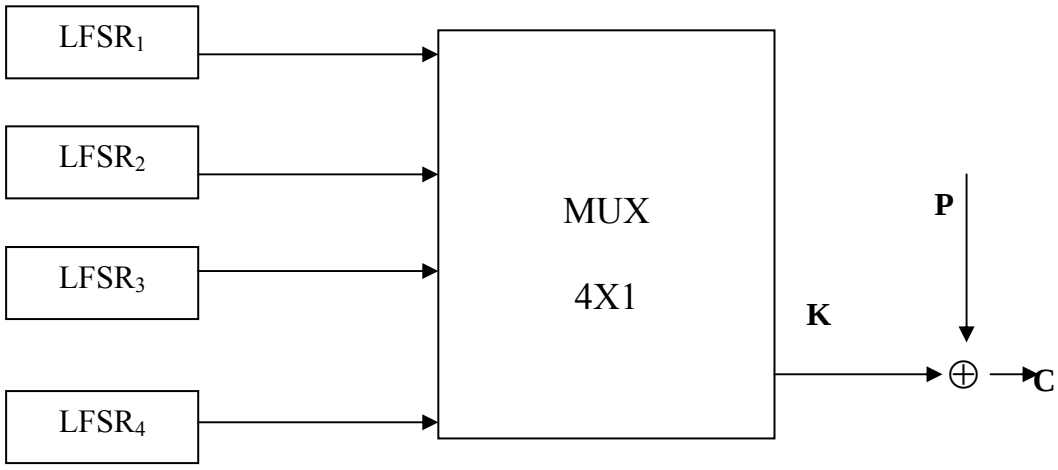
CHAPTER 5

Algorithm

Suppose in a system there are 4 Linear feedback shift registers and a 4X1 MUX. Each shift register will generate a PN sequence. The length of sequence will depend on the size of shift register and the total period of the system will be the LCM of the periods of the 4 shift registers. The output stream bits produced by LFSRs is passed in the 4X1 MUX . We convert these bits into decimal which will vary from 0 to15.

In MUX modulo 4 operation is performed, in which if the mod 4 of the sum is 1 then first stream is taken as final keystream, if mod4 of sum is 2 then second stream is taken as keystream, if mod4 of sum is 3 then third stream is taken as keystream, if mod4 of sum is 4 then fourth stream is taken as keystream.

This keystream is mixed with the input data bits(plain text) using the XOR operation. This will be the output sequence bit (ciphertext).



Mathematically if X1,X2,X3,X4 are the output sequence bit of the 4 shift registers R1, R2, R3 and R4 and P is the plain text(in ASCII) form . Then we can say

P ⊕ K = C

Where value of Keystream(K) is calculated by using the function

Sum modulo 4 = K

Assuming 4 Linear Feed back shift registers of length 31,29,27 and 41 respectively . The primitive polynomials for each shift registers will be:

1) $x^{31}+x^3+1 = 0$
2) $x^{29}+x^2+1 = 0$
3) $x^{27}+x^5+x^2+x+1 = 0$
4) $x^{41}+x^3+1 = 0$

These sequences will be used as the tapping point ie for 31 stage shift register the tapping point will be 31 and 3rd . For 29 bit shift register tapping point will be 29 and 2nd and so on.

These potions are XOR ie for 31 stage shift register 31 and 3rd bit is XORed . In this way these are XOR and right shifted.

Same operation is applied for each shift register.

Suppose we get output bit X1 from LFSR R1, X2 from shift Register R2, X3 from shift Register R3 and X4 from Shift register R4 .

These output stream bits are passed through the 4X1 MUX to decide the control bit. For deciding that we have applied **Sum modulo 4 = K** operation as described above. The output generated from 4X1 MUX will be the final keystream **K**

This keystream **K** is XOR with the input text P(this is converted into ASCII code) to get the cipher text.

At the receiving end the system is again activated so that the 4 shift registers in the system will gain generate the sequence and the crypt bit will be XOR to get back the initial input.

For Encryption of plain text the function used is P ⊕ K=C

Where P is plain input text, C is the Cipher text and K is the Key stream given by

Keystream **K = Sum modulo 4**

i) if sum is 0 then consider first stream as keystream.

ii) if sum is 1 then take second stream as keystream.

iii) If sum is 2 then take third stream as keystream.

iv) If sum is 3 then take fourth stream as keystream.

Similarly for Decryption the function used is C ⊕ K=P

Using NIST test suite most randomness tests were passed by the keystream **K.** Report of which is shown further.

While designing the stream cipher many more options are also consider and tested few of them are listed below :

i) **K = (X1&X2) ⊕ X3 ⊕ X4**

ii) **K = (X1&X2) ⊕ (X2&X3) ⊕ (X3&X4) ⊕ (X4&X1)**

iii) **K = (X1&X2&X3) ⊕ (X2&X3&X4) ⊕ (X4&X1&X2)**

iv) **K= X1 ⊕ X2 ⊕ X3 ⊕ X4**

Where X1,X2,X3,X4 are the output bits generated from the LFSRs without MUX. These ciphers fails many tests of NIST so have been ruled out.

Period of Key Sequence: For the $LFSR_1$, Time period(T1) will be 2^{31} -1

For $LFSR_2$, time period (T_2) = $2^{29} - 1$

For $LFSR_3$, time period (T3) = $2^{27} - 1$

For $LFSR_4$, time period (T4) = $2^{41} - 1$

Using Knuth algorithm

$GCD(2^p - 1 , 2^q - 1) = 2^{GCD(p,q)} - 1$

The GCD of above LFSRs is

$GCD(T_1,T_2,T_3,T_4) = 2^{GCD(31,29,27,41)} - 1$

$= 2^1 - 1$

$= 2 - 1 = 1$

Feeding of shift register(Key generation) : User is asked to enter 16 character password so that 128 bits can be generated. Out of this each shift register is feeded with bits sequentially i,e first 31st bits for first shift register, next 29 bits for second shift register , next 27 bits for 3rd shift register, and next 41 bits for 4th shift register.

Same operation is applied at the receiving end too for decryption of bits to get the original plain text.

CHAPTER 6.

Conclusion and Comparison

Machine Configuration:

Number of clients	**Machine/CPU**	**# of CPUs**	**Memory**	**Disk**	**Software**
1	Intel Pentium IV 2.4GHz	1	256MB	40 GB	Red Hat Linux 9.

Performance Test Results: The method that was tested encrypts the data first and then decrypts the encrypted bytes.

We performed the tests with a data size of 20MB, 40MB, 60MB, 80MB, and 100MB to see how the size of data impacts performance. As we increases the size of the test data, the rate of encryption become constant. So we conclude that :

- For our algorithm the rate of encryption is 400 Mbits per sec.
- For RC4 Algorithm the rate of encryption is 300 Mbits per sec.

CHAPTER7

7.1Test Results using NIST test Suite

RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
8	8	9	9	13	14	14	9	8	8	0.739918	1.0000	Frequency
9	3	10	11	14	10	12	15	7	9	0.304126	0.9800	Block-Frequency
11	8	10	13	8	9	13	12	8	8	0.911413	0.9900	Cusum
11	9	7	8	10	10	10	12	15	8	0.851383	1.0000	Cusum
13	9	4	10	16	12	11	10	8	7	0.350485	1.0000	Runs
12	11	8	14	10	8	10	12	6	9	0.834308	0.9900	Long-Run
9	9	17	10	8	9	5	13	13	7	0.289667	1.0000	Rank
6	9	10	14	10	10	11	10	8	12	0.897763	1.0000	FFT
16	13	13	6	7	11	4	8	13	9	0.162606	0.9900	Aperiodic-Template
18	13	14	12	5	6	8	5	12	7	0.040108	1.0000	Aperiodic-Template
10	8	13	12	4	13	12	6	12	10	0.474986	1.0000	Aperiodic-Template
15	9	10	9	10	9	13	7	6	12	0.678686	0.9800	Aperiodic-Template
10	11	11	6	14	12	13	6	10	7	0.616305	0.9700	Aperiodic-Template
10	13	13	12	10	11	5	6	10	10	0.699313	0.9900	Aperiodic-Template
11	6	13	7	7	11	16	11	4	14	0.145326	0.9900	Aperiodic-Template
5	6	10	10	14	9	13	12	10	11	0.616305	0.9900	Aperiodic-Template
11	9	8	11	9	11	5	11	13	12	0.851383	1.0000	Aperiodic-Template
12	11	9	8	14	7	16	10	7	6	0.383827	0.9800	Aperiodic-Template
8	7	7	10	12	12	12	8	8	16	0.554420	0.9800	Aperiodic-Template
12	16	12	13	6	6	13	11	5	6	0.137282	0.9900	Aperiodic-Template
8	8	10	9	11	11	14	7	11	11	0.924076	0.9800	Aperiodic-Template
9	5	8	14	15	10	11	13	6	9	0.366918	1.0000	Aperiodic-Template
10	11	14	10	10	10	4	11	9	11	0.779188	0.9900	Aperiodic-Template
15	8	9	7	11	7	13	9	7	14	0.494392	1.0000	Aperiodic-Template
10	9	8	5	9	10	15	14	8	12	0.534146	0.9800	Aperiodic-Template
12	16	16	10	5	6	9	6	9	11	0.137282	0.9800	Aperiodic-Template
10	7	13	13	11	7	7	8	14	10	0.678686	0.9900	Aperiodic-Template

14	7	10	14	6	13	8	10	8	10	0.595549	1.0000	Aperiodic-Template
6	12	10	8	14	11	11	7	11	10	0.816537	1.0000	Aperiodic-Template
9	11	10	10	13	7	8	13	12	7	0.867692	1.0000	Aperiodic-Template
9	14	10	8	10	5	12	11	14	7	0.574903	1.0000	Aperiodic-Template
8	14	6	12	6	7	11	10	14	12	0.474986	0.9800	Aperiodic-Template
12	10	12	12	9	12	7	10	6	10	0.897763	0.9900	Aperiodic-Template
12	14	7	10	5	11	10	8	10	13	0.657933	1.0000	Aperiodic-Template
3	6	7	11	9	14	13	13	18	6	0.025193	1.0000	Aperiodic-Template
10	6	11	15	11	5	15	9	9	9	0.383827	0.9800	Aperiodic-Template
5	10	12	7	15	13	8	13	6	11	0.334538	0.9800	Aperiodic-Template
12	6	15	8	9	7	11	8	12	12	0.616305	0.9900	Aperiodic-Template
11	5	13	12	13	9	11	6	8	12	0.595549	0.9600	Aperiodic-Template
11	7	11	10	8	9	14	13	7	10	0.834308	0.9900	Aperiodic-Template
6	6	8	13	15	13	13	11	9	6	0.304126	1.0000	Aperiodic-Template
11	10	8	15	7	13	13	5	10	8	0.474986	1.0000	Aperiodic-Template
10	6	7	13	9	12	12	12	9	10	0.851383	0.9800	Aperiodic-Template
10	12	13	16	5	9	10	7	10	8	0.455937	1.0000	Aperiodic-Template
9	8	8	11	7	14	7	13	13	10	0.719747	1.0000	Aperiodic-Template
10	8	12	9	6	11	11	13	9	11	0.924076	1.0000	Aperiodic-Template
14	8	8	4	13	11	8	9	12	13	0.455937	0.9900	Aperiodic-Template
6	19	6	7	12	8	14	7	6	15	0.020548	0.9900	Aperiodic-Template
5	11	8	12	11	9	12	8	14	10	0.739918	1.0000	Aperiodic-Template
8	11	14	10	9	9	12	9	6	12	0.851383	1.0000	Aperiodic-Template
10	12	11	12	8	6	7	5	10	19	0.108791	0.9900	Aperiodic-Template
9	8	14	10	14	14	6	3	9	13	0.171867	0.9900	Aperiodic-Template
14	8	14	14	10	7	7	7	5	14	0.213309	0.9800	Aperiodic-Template
11	10	12	15	8	11	7	10	5	11	0.637119	0.9900	Aperiodic-Template
12	16	13	7	7	8	14	10	7	6	0.262249	0.9900	Aperiodic-Template
10	11	12	4	16	5	11	11	9	11	0.304126	1.0000	Aperiodic-Template
11	9	15	10	10	7	9	10	10	9	0.924076	0.9800	Aperiodic-Template
14	11	12	13	11	7	10	9	5	8	0.637119	0.9600	Aperiodic-Template
11	9	15	11	12	6	11	7	5	13	0.419021	1.0000	Aperiodic-Template
6	13	14	13	14	7	7	9	5	12	0.249284	1.0000	Aperiodic-Template
9	11	9	11	8	9	6	10	15	12	0.798139	0.9900	Aperiodic-Template
10	15	12	8	12	8	5	11	9	10	0.657933	1.0000	Aperiodic-Template
6	9	12	9	17	8	13	8	8	10	0.419021	1.0000	Aperiodic-Template
8	14	8	11	11	13	9	5	14	7	0.474986	0.9900	Aperiodic-Template

14	10	4	10	9	12	8	8	13	12	0.554420	1.0000	Aperiodic-Template
12	10	15	3	7	16	7	12	11	7	0.102526	1.0000	Aperiodic-Template
13	14	12	7	8	10	5	9	11	11	0.637119	0.9900	Aperiodic-Template
7	10	12	11	9	9	14	9	11	8	0.924076	0.9800	Aperiodic-Template
7	13	12	14	9	15	7	8	8	7	0.437274	0.9800	Aperiodic-Template
13	12	9	10	10	9	10	12	11	4	0.779188	0.9800	Aperiodic-Template
8	7	9	7	11	11	12	15	13	7	0.616305	1.0000	Aperiodic-Template
10	10	10	7	10	8	11	10	6	18	0.401199	0.9900	Aperiodic-Template
8	12	9	11	10	10	7	14	10	9	0.935716	0.9800	Aperiodic-Template
11	10	11	9	11	10	11	6	10	11	0.987896	1.0000	Aperiodic-Template
10	17	7	9	6	14	12	11	4	10	0.153763	0.9800	Aperiodic-Template
10	6	8	8	12	14	13	11	10	8	0.759756	0.9900	Aperiodic-Template
11	11	8	7	8	13	10	7	17	8	0.437274	0.9900	Aperiodic-Template
7	9	13	13	4	13	7	10	12	12	0.437274	0.9900	Aperiodic-Template
10	10	11	10	12	8	6	9	14	10	0.897763	0.9900	Aperiodic-Template
10	9	11	6	12	4	8	12	12	16	0.304126	1.0000	Aperiodic-Template
13	9	9	14	7	12	9	7	11	9	0.816537	0.9900	Aperiodic-Template
13	11	13	10	6	10	10	14	8	5	0.534146	0.9900	Aperiodic-Template
16	13	13	6	7	11	4	9	12	9	0.202268	0.9900	Aperiodic-Template
12	9	11	6	9	12	12	11	8	10	0.935716	1.0000	Aperiodic-Template
11	10	13	9	10	5	6	11	15	10	0.554420	0.9900	Aperiodic-Template
9	12	11	10	4	12	11	11	11	9	0.834308	0.9900	Aperiodic-Template
5	9	7	13	17	12	11	8	10	8	0.304126	0.9700	Aperiodic-Template
8	14	10	8	11	9	17	10	8	5	0.319084	0.9900	Aperiodic-Template
5	10	14	9	9	11	11	12	9	10	0.834308	0.9900	Aperiodic-Template
11	9	5	17	7	9	10	9	13	10	0.383827	1.0000	Aperiodic-Template
9	8	7	7	10	10	9	9	18	13	0.366918	0.9700	Aperiodic-Template
8	12	8	14	8	11	10	12	10	7	0.867692	0.9800	Aperiodic-Template
10	9	10	12	12	13	8	8	9	9	0.971699	0.9800	Aperiodic-Template
12	10	8	9	10	13	5	8	12	13	0.739918	0.9900	Aperiodic-Template
13	6	12	11	12	6	8	12	8	12	0.678686	0.9900	Aperiodic-Template
8	11	16	9	6	15	10	7	8	10	0.383827	0.9900	Aperiodic-Template
10	9	12	8	8	11	8	12	9	13	0.955835	0.9900	Aperiodic-Template
14	6	14	10	7	9	11	13	10	6	0.494392	0.9600	Aperiodic-Template
7	8	9	15	10	17	8	7	8	11	0.304126	1.0000	Aperiodic-Template
8	10	11	10	13	11	11	10	7	9	0.978072	1.0000	Aperiodic-Template
13	11	9	10	9	6	13	10	14	5	0.554420	0.9900	Aperiodic-Template

13	7	10	9	6	7	11	18	8	11	0.249284	0.9800	Aperiodic-Template
12	8	12	9	10	8	7	11	11	12	0.955835	0.9700	Aperiodic-Template
10	9	9	8	12	12	6	14	11	9	0.851383	1.0000	Aperiodic-Template
15	6	9	9	9	10	11	8	13	10	0.759756	1.0000	Aperiodic-Template
7	11	11	14	9	7	7	10	14	10	0.719747	0.9900	Aperiodic-Template
7	9	13	14	9	11	16	8	7	6	0.334538	1.0000	Aperiodic-Template
7	9	8	10	9	10	10	12	9	16	0.779188	1.0000	Aperiodic-Template
12	11	9	6	6	7	12	18	10	9	0.236810	0.9900	Aperiodic-Template
8	11	9	17	10	8	17	5	5	10	0.071177	0.9800	Aperiodic-Template
10	10	12	7	12	10	10	8	13	8	0.946308	0.9800	Aperiodic-Template
7	12	13	10	11	9	7	11	8	12	0.897763	1.0000	Aperiodic-Template
10	12	12	13	15	6	5	5	12	10	0.262249	0.9900	Aperiodic-Template
13	8	9	14	11	10	5	9	10	11	0.759756	0.9700	Aperiodic-Template
11	9	4	9	12	17	10	9	9	10	0.401199	1.0000	Aperiodic-Template
10	12	14	7	12	12	9	7	8	9	0.816537	1.0000	Aperiodic-Template
9	10	9	11	12	10	9	7	15	8	0.867692	0.9900	Aperiodic-Template
13	7	6	17	9	10	7	8	13	10	0.304126	0.9700	Aperiodic-Template
12	4	14	13	12	5	6	13	15	6	0.066882	0.9900	Aperiodic-Template
6	14	11	10	14	9	9	7	9	11	0.719747	1.0000	Aperiodic-Template
13	12	10	11	8	7	12	9	7	11	0.897763	0.9700	Aperiodic-Template
5	11	8	11	9	11	11	12	9	13	0.851383	1.0000	Aperiodic-Template
8	16	3	6	14	12	10	10	7	14	0.090936	0.9900	Aperiodic-Template
12	9	7	12	12	7	12	11	9	9	0.924076	0.9800	Aperiodic-Template
9	15	5	10	10	8	10	14	10	9	0.616305	1.0000	Aperiodic-Template
13	11	11	9	15	9	9	7	9	7	0.759756	1.0000	Aperiodic-Template
13	13	5	9	9	10	6	8	13	14	0.437274	0.9900	Aperiodic-Template
4	13	10	13	7	12	10	8	14	9	0.455937	1.0000	Aperiodic-Template
10	3	12	4	12	12	9	16	10	12	0.129620	0.9800	Aperiodic-Template
9	9	8	18	7	9	13	10	10	7	0.366918	0.9900	Aperiodic-Template
12	8	13	9	9	7	11	11	6	14	0.719747	0.9900	Aperiodic-Template
10	13	9	8	8	14	10	10	8	10	0.924076	1.0000	Aperiodic-Template
9	8	12	15	12	6	13	8	9	8	0.616305	0.9700	Aperiodic-Template
10	16	8	6	7	11	6	9	15	12	0.262249	1.0000	Aperiodic-Template
14	11	10	9	8	12	13	2	12	9	0.319084	1.0000	Aperiodic-Template
13	8	15	10	10	8	12	6	7	11	0.616305	0.9800	Aperiodic-Template
7	5	6	10	19	11	9	7	16	10	0.037566	0.9700	Aperiodic-Template
8	18	6	10	10	8	6	12	9	13	0.224821	0.9800	Aperiodic-Template

7	10	10	10	4	18	11	13	14	3	0.030806	0.9900	Aperiodic-Template
9	16	14	16	6	10	11	5	7	6	0.075719	0.9900	Aperiodic-Template
11	11	12	5	15	11	9	12	10	4	0.366918	0.9900	Aperiodic-Template
10	11	8	17	12	7	7	8	10	10	0.534146	0.9900	Aperiodic-Template
8	10	10	5	9	11	12	10	14	11	0.816537	0.9700	Aperiodic-Template
12	9	7	8	7	11	7	13	11	15	0.616305	0.9900	Aperiodic-Template
11	17	8	5	11	6	11	10	12	9	0.334538	1.0000	Aperiodic-Template
10	10	4	13	12	8	12	16	6	9	0.275709	1.0000	Aperiodic-Template
14	5	14	11	7	9	12	6	13	9	0.366918	0.9900	Aperiodic-Template
13	9	14	7	14	9	9	8	8	9	0.719747	1.0000	Aperiodic-Template
15	11	8	11	7	12	14	8	5	9	0.437274	0.9900	Aperiodic-Template
6	13	9	12	7	14	8	10	11	10	0.739918	1.0000	Aperiodic-Template
14	8	9	4	12	14	8	10	10	11	0.514124	0.9800	Aperiodic-Template
13	10	6	5	9	18	10	12	5	12	0.096578	0.9900	Aperiodic-Template
9	4	18	8	13	9	10	10	12	7	0.171867	0.9900	Aperiodic-Template
11	9	11	10	9	13	11	7	11	8	0.971699	0.9900	Aperiodic-Template
10	6	11	10	8	8	17	10	8	12	0.514124	1.0000	Aperiodic-Template
13	11	13	10	6	10	10	14	8	5	0.534146	0.9900	Aperiodic-Template
11	8	10	14	13	6	12	8	6	12	0.595549	0.9900	Periodic-Template
12	9	7	13	12	7	9	14	11	6	0.637119	0.9800	Universal
34	14	8	7	12	14	3	5	2	1	0.000000 *	0.9300 *	Apen
5	6	3	2	6	7	7	6	6	11	0.202268	0.9831	Random-Excursion
5	1	7	7	8	6	6	4	6	9	0.304126	0.9831	Random-Excursion
5	7	7	7	6	4	5	7	5	6	0.924076	0.9492 *	Random-Excursion
5	5	8	2	8	5	7	11	2	6	0.080519	1.0000	Random-Excursion
10	5	3	6	2	5	7	10	7	4	0.102526	0.9831	Random-Excursion
6	7	7	6	8	7	4	3	7	4	0.678686	1.0000	Random-Excursion
10	4	2	3	3	5	8	4	10	10	0.014550	1.0000	Random-Excursion
8	8	7	5	3	2	9	3	9	5	0.115387	1.0000	Random-Excursion
5	9	7	8	7	4	6	5	5	3	0.554420	1.0000	Random-Excursion-V
5	6	10	10	7	2	5	4	6	4	0.145326	1.0000	Random-Excursion-V
5	7	6	11	7	5	3	5	3	7	0.249284	0.9831	Random-Excursion-V
6	4	6	6	10	6	9	6	4	2	0.249284	0.9831	Random-Excursion-V
3	8	6	4	8	7	9	0	6	8	0.071177	1.0000	Random-Excursion-V
5	4	8	4	6	6	6	5	7	8	0.798139	0.9831	Random-Excursion-V
6	7	5	5	7	4	6	5	9	5	0.798139	0.9831	Random-Excursion-V
8	7	7	4	6	6	6	4	5	6	0.867692	0.9831	Random-Excursion-V

11	6	7	6	5	5	6	1	7	5	0.181557	0.9831	Random-Excursion-V
6	11	3	4	8	5	11	4	5	2	0.021999	0.9831	Random-Excursion-V
5	7	5	8	4	8	9	6	4	3	0.437274	0.9831	Random-Excursion-V
7	2	2	10	9	10	6	4	5	4	0.032923	0.9661	Random-Excursion-V
5	2	8	8	4	8	6	3	8	7	0.275709	0.9661	Random-Excursion-V
5	2	9	6	6	4	6	8	7	6	0.474986	0.9831	Random-Excursion-V
6	3	6	7	3	13	3	5	6	7	0.042808	0.9831	Random-Excursion-V
4	10	2	6	5	5	10	3	4	10	0.032923	0.9831	Random-Excursion-V
5	6	7	3	4	9	7	5	8	5	0.554420	0.9831	Random-Excursion-V
6	5	7	6	5	6	6	11	4	3	0.366918	0.9831	Random-Excursion-V
12	11	10	12	8	5	18	7	10	7	0.213309	1.0000	Serial
11	9	11	11	16	11	7	7	11	6	0.574903	1.0000	Serial
13	7	17	10	9	12	10	7	6	9	0.366918	0.9800	Lempel-Ziv
13	11	10	11	5	8	10	13	11	8	0.798139	0.9800	Linear-Complexity

- -- - - - - - - - - - - - -

The minimum pass rate for each statistical test with the exception of the random excursion (variant) test is approximately = 0.960150 for a sample size = 100 binary sequences.

The minimum pass rate for the random excursion (variant) test is approximately 0.951139 for a sample size = 59 binary sequences.

For further guidelines construct a probability table using the MAPLE program provided in the addendum section of the documentation.

- -- - - - - - - - - - - - -

7.2 RESULT of Randomness Testing for Keystream on Mulyankan Software of DRDO

RESULT FOR keystream

Total Length of Sequence=100000000

BLOCK size =10000000

Level of significance : 0.05

Name of Test	# Blocks	Passed	Failed

Frequency	10	10	0
Serial	10	9	1
Poker5	10	10	0
Poker7	10	10	0
Runs	10	9	1
Atcr-1	10	9	1
Atcr-n/4	10	10	0
K-S	10	10	0

Appendix –A Source Code :

Encryption

```
#include<stdio.h>
#include<string.h>
#include <curses.h>
#include<stdlib.h>
#define max1 100000000
//max1 is the number of characters in the file
struct abc
{
```

```
unsigned int bin:8;
};

struct b
{
unsigned int a:4;
unsigned int t:4;
};

int R1[31],R2[29],R3[27],R4[41];
int z[max1]={0};
unsigned char data[max1];
//The Register are of length 31,29,27 and  41  respectively
//THE PRIMITIVE POLYNOMIALS ARE STORED IN THESED REGISTERS
int key[16];
int cipher[6];
char infile[20];
char input[16];
//For the  XOR Operations I have assumed that if the total No. of 1 is odd then the
//result will be 1 ELSE 0.

void fillTheShiftRegisters(void);
void sshiftTheRegisterR1(void);
void sshiftTheRegisterR2(void);
void sshiftTheRegisterR3(void);
void sshiftTheRegisterR4(void);
void eencrypt(void);
void decrypt(void);
void fun(char num[])
{
//Feeding of shift registers
struct b b1;
int i=0,j,k;
struct abc x[16];
```

```
for(i=0;i<16;i++)
x[i].bin=0;

for(i=0;i<16;i++)
{
int s=num[i];
b1.a=s%16;
s=s/16;
b1.t=s;
x[i].bin=b1.t;
x[i].bin=x[i].bin<<4|b1.a;
}
j=30;
for(k=0;k<3;k++)
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R1[j--]=1;
else
R1[j--]=0;
}
}
for(i=7;i>0;i--)
{
if((x[3].bin)&(1<<i))
R1[j--]=1;
else
R1[j--]=0;
}

R2[28]=x[3].bin&1?1:0;
j=27;
for(k=4;k<=6;k++)
```

```
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R2[j--]=1;
else
R2[j--]=0;
}
}
for(i=7;i>=3;i--)
{
if((x[7].bin)&(1<<i))
R2[j--]=1;
else
R2[j--]=0;
}
j=26;
for(i=3;i>=0;i--)
{
if((x[7].bin)&(1<<i))
R3[j--]=1;
else
R3[j--]=0;
}
for(k=8;k<=9;k++)
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R3[j--]=1;
else
R3[j--]=0;
}
}
```

```
for(i=7;i>0;i--)
{
if((x[10].bin)&(1<<i))
R3[j--]=1;
else
R3[j--]=0;
}

R4[40]=x[10].bin& 0x01?1:0;

j=39;
for(k=11;k<=15;k++)
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R4[j--]=1;
else
R4[j--]=0;
}
}
}
int main(){

        int i=0;
        int count=0;
    //  fillTheShiftRegisters();
        FILE *fp;
        char ch;
        printf("PLEASE ENTER THE INPUT TEXT OF ONLY 16 CHARACTERS
):-\n");
        scanf("%s",input);

        fun(input);
```

```
        printf("\nEnter the input plain text file name:");
        scanf("%s",infile);
        fp=fopen(infile,"rb");
        i=0;
        printf("\nContent of the plain text file is: ");
   printf("\n-----------------------------------------\n");

         while(!feof(fp))
        {
        ch='\0';
        fread(&ch,sizeof(ch),sizeof(ch),fp);
        printf("%c",ch);
        data[i++]=ch;
        }
      printf("\nThe number of character in the file is%d",i-1);
      eencrypt();printf("\n");
        return 0;
}
void eencrypt()
{
int i,p[8]={0},j=0,k=0,n=0;
struct abc a,aa;
int sum =0;
int count=0;
unsigned char ch,che;
FILE *fp1,*fp3;
printf("\n--------you are going to encrypt the file now----------\n");
printf("\nEnter the encrypt file name:");
scanf("%s",infile);
fp1=fopen(infile,"wb");
a.bin=0;
aa.bin=0;
fp3=fopen("keystream","w");
for(i=0;i<100000000;i++)
```

```
{
// Calculation of Keystream
if(k==8)
k=0;
if (R1[30]==1)
sum+=1;
if (R2[28]==1)
sum+=2;
if (R3[26]==1)
sum+=4;
if (R4[40]==1)
sum+=8;
sum= sum/4;
if(sum==0)
z[i]=R4[40];
if(sum==1)
z[i]=R3[26];
if(sum==2)
z[i]=R2[28];
if(sum==3)
z[i]=R1[30];
sum=0;

fprintf(fp3,"%d",z[i]);
sshiftTheRegisterR1();
sshiftTheRegisterR2();
sshiftTheRegisterR3();
sshiftTheRegisterR4();
ch=data[i/8];
a.bin=ch;
n=((a.bin)&(0x01<<k))?1:0;
p[j]=(z[i]^n)?1:0;
k++;
j++;
```

```
if(k==8)
{
aa.bin=0;
for(n=7;n>=0;n--)
{
aa.bin=aa.bin|p[n];
if(n==0)
break;
aa.bin=aa.bin<<1;
}
che=aa.bin;
aa.bin=0;
j=0;
}

}
//end of the for loop

fclose(fp1);

fp1=fopen(infile,"rb");
printf("This is the content of the file outfile \n");
printf("----------------------------------------\n");
while(!feof(fp1))
        {
        ch='\0';
        fread(&ch,sizeof(ch),sizeof(ch),fp1);
        count++;
        printf("%c",ch);
        }
printf("\nThe number of charactre in the file is  %d",count-1);
fclose(fp1);
}
```

```
void decrypt()
{
   FILE *fp, *fp1;
   char str[max1];
   unsigned char ch,che;
   int j=0,k=0,n=0,i = 0;
   int p[8]={0};
   int length;
   struct abc a,aa;
   //time_t ltime1, ltime2;
   printf("\nYou are going to decrypt the file now\n");
   printf("\nEnter name of file to be decrypted: ");
   scanf("%s", infile);

   fp = fopen(infile,"rb");
   if(fp == NULL)
   {
         printf("\nInput file is not present in the current directory");
         exit(0);

   }
   printf("\nContent of the file to be decrypted is : ");
   printf("\n-----------------------------------------\n");

   while(!feof(fp))
   {
         ch = '\0';
         fread(&ch, sizeof(ch), sizeof(ch), fp);
         printf("%c", ch);
         str[i++]=ch;

   }
   printf("\n-----------------------------------------");
```

```
   length=i-1;
   fclose(fp);
   printf("\nNumber of characters in the cipher text is: %d\n", length);

   printf("\nEnter name of output plain text file: ");
   scanf("%s", infile);
   fp1 = fopen(infile,"wb");
   if (fp == NULL)
   {
         printf("\nOutput file can not open");
         exit(0);
            }
   printf("\n\nDecrypted text is: ");
   printf("\n-----------------------------------------\n");
         a.bin=0;
         aa.bin=0;

for(i=0;i<(8*strlen(str));i++)
{
if(k==8)
k=0;
ch=str[i/8];
a.bin=ch;
n=((a.bin)&(0x01<<k))?1:0;
p[j]=(z[i]^n)?1:0;
k++;
j++;

if(k==8)
{
aa.bin=0;
for(n=7;n>=0;n--)
{
```

```
aa.bin=aa.bin | p[n];
if(n==0)
break;
aa.bin=aa.bin<<1;
}
che=aa.bin;
fwrite(&che, sizeof(che), sizeof(che), fp1);
aa.bin=0;
j=0;
}//end of if

}//end of for loop

//    printf("\nNumber of characters in plain text: %d",length);
fclose(fp1);
  //  printf("\nTime taken for Decryption: %ld", ltime2 - ltime1);
  fp1=fopen(infile,"rb");
printf("This is the content of the new plain text file is \n");
printf("\n----------------------------------------\n");
while(!feof(fp1))
            {
            ch='\0';
            fread(&ch,sizeof(ch),sizeof(ch),fp1);
            printf("%c",ch);
//           printf("\n%c  %x\n",ch,ch);
            //data[i++]=ch;
            }
printf("\nnumber of characters in the file %d\n",strlen(str));

    fclose(fp1);
    //return 0;
} //end of decrypt
```

```
// THIS FUNCTION CALCULATES THE XOR OF THE TAP POINTS AND THEN
//SHIFTS THE REGISTER
void  sshiftTheRegisterR1(){
// I AM NOW CALCULATING THE DIGITS AT 31 13 7 6 3 AND 1 AND
//XORING THEM AND RETURNING THE RESULT AND SHIFTING THE
//DIGITS
        int x31,x13,x7,x6,x3;
        int x,i;
        x31=R1[30];
        x13=R1[12];
        x7=R1[6];
        x6=R1[5];
        x3=R1[2];
        x=(x31^x13^x7^x6^x3);

        //NOW RIGHT SHIFT THE REGISTER
        for(i=30;i>0;i--)
        R1[i]=R1[i-1];
        //STORE THE XOR IN MSB
        R1[0]=x;

}

void sshiftTheRegisterR2(){
        int x29,x2;
        int xor,i;
        x29=R2[28];
        x2=R2[1];
        xor=(x29^x2);
        //NOW RIGHT SHIFT THE REGISTER
        for(i=28;i>0;i--)
        R2[i]=R2[i-1];

        R2[0]=xor;
```

```
  }

void sshiftTheRegisterR3(){
        int x27,x5,x2,x1;
        int xor,i;
        x27=R3[26];
        x5=R3[4];
        x2=R3[1];
        x1=R3[0];
        xor=(x27^x5^x2^x1);
        //NOW RIGHT SHIFT THE REGISTER
        for(i=26;i>0;i--)
        R3[i]=R3[i-1];
        R3[0]=xor;

}

void sshiftTheRegisterR4(){
        int x41,x3;
        int xor,i;
        x41=R4[40];
        x3=R1[2];
        xor=(x41^x3);

        //for(i=40;i>=0;i--)
        //printf("%d",R4[i]);printf("\n");

        //NOW RIGHT SHIFT THE REGISTER
        for(i=40;i>0;i--)
        R4[i]=R4[i-1];

        //for(i=40;i>=0;i--)
        //printf("%d",R4[i]);printf("\n");
```

```
        R4[0]=xor;
        //for(i=40;i>=0;i--)
        //printf("%d",R4[i]);printf("\n");

}
```

Decryption

```
#include<stdio.h>
#include<string.h>
//#include<conio.h>
#include<stdlib.h>
#define max1 100000000
//max is the number of characters in the file
struct abc
{
unsigned int bin:8;
};
struct b
{
unsigned int a:4;
unsigned int b:4;
};

int R1[31],R2[29],R3[27],R4[41];

int z[8*max1]={0};

int c_num;

unsigned char data[max1];
//The register are of length 31,29,27 and 41 respectively.
//THE PRIMITIVE POLYNOMIALS ARE STORED IN THESED REGISTERS
char name[20];
//char input[16]="ABCDEFGHIJKLMNOP";
char input[16];
// FOR THE  XOR OPERATIONS I HAVE ASSUMED THAT IF THE TOTAL NO. OF 1 IS ODD
THEN THE //RESULT WILL BE 1 ELSE 0.

void sshiftTheRegisterR1(void);
void sshiftTheRegisterR2(void);
```

```
void sshiftTheRegisterR3(void);
void sshiftTheRegisterR4(void);
//void eencrypt(void);
void decrypt(void);
void fun(char num[])
{
//Feeding of shift registers
struct b b1;
int i=0,j,k;
struct abc x[16];
for(i=0;i<16;i++)
x[i].bin=0;

for(i=0;i<16;i++)
{
int s=num[i];
b1.a=s%16;
s=s/16;
b1.b=s;
x[i].bin=b1.b;
x[i].bin=x[i].bin<<4|b1.a;
}
j=30;
for(k=0;k<3;k++)
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R1[j--]=1;
else
R1[j--]=0;
}
}
for(i=7;i>0;i--)
```

```
{
if((x[3].bin)&(1<<i))
R1[j--]=1;
else
R1[j--]=0;
}

R2[28]=x[3].bin&1?1:0;
j=27;
for(k=4;k<=6;k++)
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R2[j--]=1;
else
R2[j--]=0;
}
}
for(i=7;i>=3;i--)
{
if((x[7].bin)&(1<<i))
R2[j--]=1;
else
R2[j--]=0;
}
j=26;
for(i=3;i>=0;i--)
{
if((x[7].bin)&(1<<i))
R3[j--]=1;
else
R3[j--]=0;
}
```

```
for(k=8;k<=9;k++)
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R3[j--]=1;
else
R3[j--]=0;
}
}
for(i=7;i>0;i--)
{
if((x[10].bin)&(1<<i))
R3[j--]=1;
else
R3[j--]=0;
}

R4[40]=x[10].bin& 0x01?1:0;

j=39;
for(k=11;k<=15;k++)
{
for(i=7;i>=0;i--)
{
if((x[k].bin)&(1<<i))
R4[j--]=1;
else
R4[j--]=0;
}
}
}
void main(){
```

```
        int i=0;
       //fillTheShiftRegisters();
        //printf("%s",input);
        FILE *fp;
        char ch;
        //clrscr();
        printf("PLEASE ENTER THE INPUT TEXT ONLY 16 CHARACTERS ):-
\n");
        scanf("%s",input);

        fun(input);
    /* for(i=30;i>=0;i--)
        printf("%d",R1[i]);printf("\n");
        for(i=28;i>=0;i--)
        printf("%d",R2[i]); printf("\n");
        for(i=26;i>=0;i--)
        printf("%d",R3[i]); printf("\n");
        for(i=40;i>=0;i--)
        printf("%d",R4[i]); printf("\n");
        */
        printf("\nEnter the cipher text file name:");
        scanf("%s",name);
        fp=fopen(name,"rb");
        i=0;
        printf("\n Content of the cipher text file is: ");
   printf("\n-----------------------------------------\n");
        //j=0;
        c_num=0;
         while(!feof(fp))
        {
        ch='\0';
        fread(&ch,sizeof(ch),sizeof(ch),fp);
        printf("%c",ch);
        data[c_num]=ch;
```

```
        c_num++;
        }
         printf("\nThe number of character in the file is: %d\n",c_num-1);
        fclose(fp);
        printf("\nThe content of the data array \n");
        //for(i=0;i<c_num;i++)
        //{
        //        printf("%c",data[i]);
        //}
        printf("\n\n");
    decrypt();
        getch();
        exit(0);
}
void decrypt()
{
int i,p[8]={0},j=0,k=0,n=0;
struct abc a,aa;
int count=0;
int sum=0;
unsigned char ch,che;

FILE *fp1;
printf("\nyou are going to decrypt the file now\n");
printf("\nEnter the decrypted file name:");
scanf("%s",name);
fp1=fopen(name,"wb");
a.bin=0;
aa.bin=0;
printf("\n Here there data file length %d",strlen(data));
for(i=0;i<(8*(c_num-1));i++)
{
if(k==8)
k=0;
```

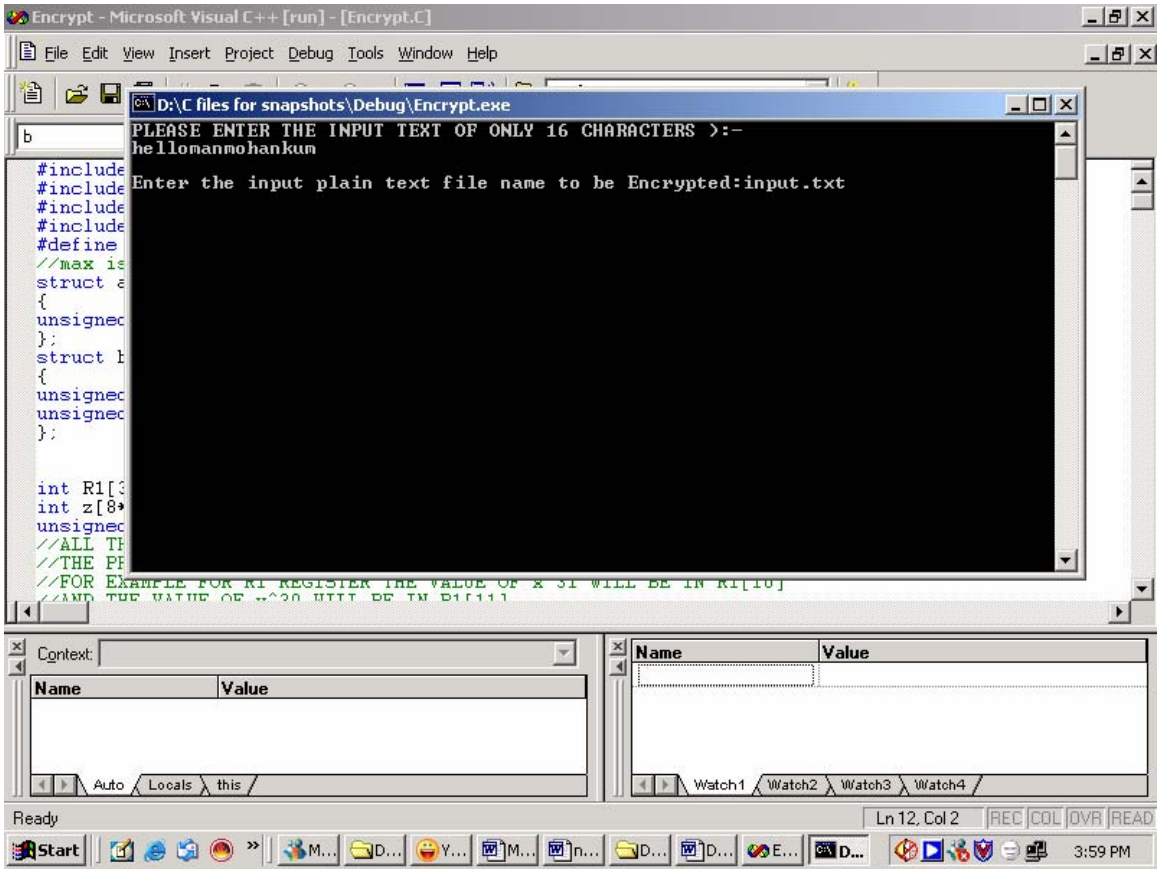
```
//z[i]=R1[30]^R2[28]^R3[26]^R4[40];
// Calculation of Keystream
if(R4[40]==1)
sum+=1;
if(R3[26]==1)
sum+=2;
if(R2[28]==1)
sum+=4;
if(R1[30]==1)
sum+=8;

sum=sum/4;

if(sum==0)
z[i]=R1[30];
if(sum==1)
z[i]=R2[28];
if(sum==2)
z[i]=R3[26];
if(sum==3)
z[i]=R4[40];
sum=0;

sshiftTheRegisterR1();
sshiftTheRegisterR2();
sshiftTheRegisterR3();
sshiftTheRegisterR4();
ch=data[i/8];
a.bin=ch;
n=((a.bin)&(0x01<<k))?1:0;
p[j]=(z[i]^n)?1:0;
//printf("n=%d z[%d]=%d p[%d]=%d \n",n,i,z[i],j,p[j]);
k++;
```

```
j++;

if(k==8)
{
aa.bin=0;
for(n=7;n>=0;n--)
{
aa.bin=aa.bin|p[n];
//printf("hello %x %d\n",aa.bin,p[n]);
if(n==0)
break;
aa.bin=aa.bin<<1;
}
che=aa.bin;
//printf(" HI  %x   %x   \n",che,aa.bin);
fwrite(&che, sizeof(che), sizeof(che), fp1);
aa.bin=0;
j=0;
}

}//end of the for loop

fclose(fp1);

fp1=fopen(name,"rb");
printf("This is the content of the file decrypted file \n");
printf("----------------------------------------\n");
while(!feof(fp1))
        {
        ch='\0';
        fread(&ch,sizeof(ch),sizeof(ch),fp1);
        count++;
        printf("%c",ch);
        }
```

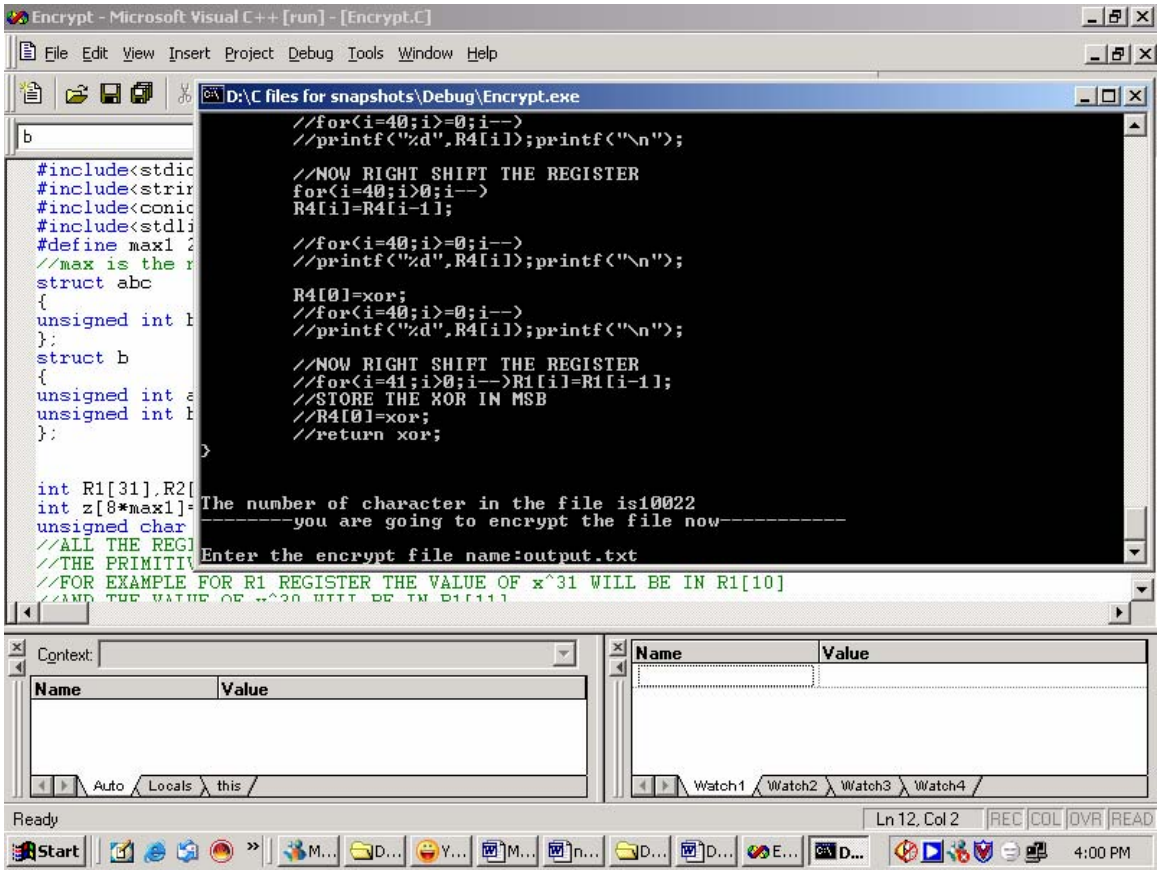
```
printf("\n The number of character in the file is %d",count-1);
fclose(fp1);
}

// THIS FUNCTION CALCULATES THE XOR OF THE TAP POINTS AND THEN
SHIFTS THE REGISTER
void  sshiftTheRegisterR1()
{
// I AM NOW CALCULATING THE DIGITS AT 31 13 7 6 3 AND 1 AND
//XORING THEM AND RETURNING THE RESULT AND SHIFTING THE
//DIGITS
        int x31,x13,x7,x6,x3;
        int x,i;
        x31=R1[30];
        x13=R1[12];
        x7=R1[6];
        x6=R1[5];
        x3=R1[2];
        x=(x31^x13^x7^x6^x3);
        //NOW RIGHT SHIFT THE REGISTER
        for(i=30;i>0;i--)
        R1[i]=R1[i-1];
        R1[0]=x;
}

void sshiftTheRegisterR2(){
        int x29,x2;
        int xor,i;
        x29=R2[28];
        x2=R2[1];
        xor=(x29^x2);
        //NOW RIGHT SHIFT THE REGISTER
        for(i=28;i>0;i--)
        R2[i]=R2[i-1];
```

```
        R2[0]=xor;

}

void sshiftTheRegisterR3(){
        int x27,x5,x2,x1;
        int xor,i;
        x27=R3[26];
        x5=R3[4];
        x2=R3[1];
        x1=R3[0];
        xor=(x27^x5^x2^x1);
        //NOW RIGHT SHIFT THE REGISTER
        for(i=26;i>0;i--)
        R3[i]=R3[i-1];
        R3[0]=xor;

}

void sshiftTheRegisterR4()
{
        int x41,x3;
        int xor,i;
        x41=R4[40];
        x3=R1[2];
        xor=(x41^x3);

        //NOW RIGHT SHIFT THE REGISTER
        for(i=40;i>0;i--)
        R4[i]=R4[i-1];
        R4[0]=xor;
}
```

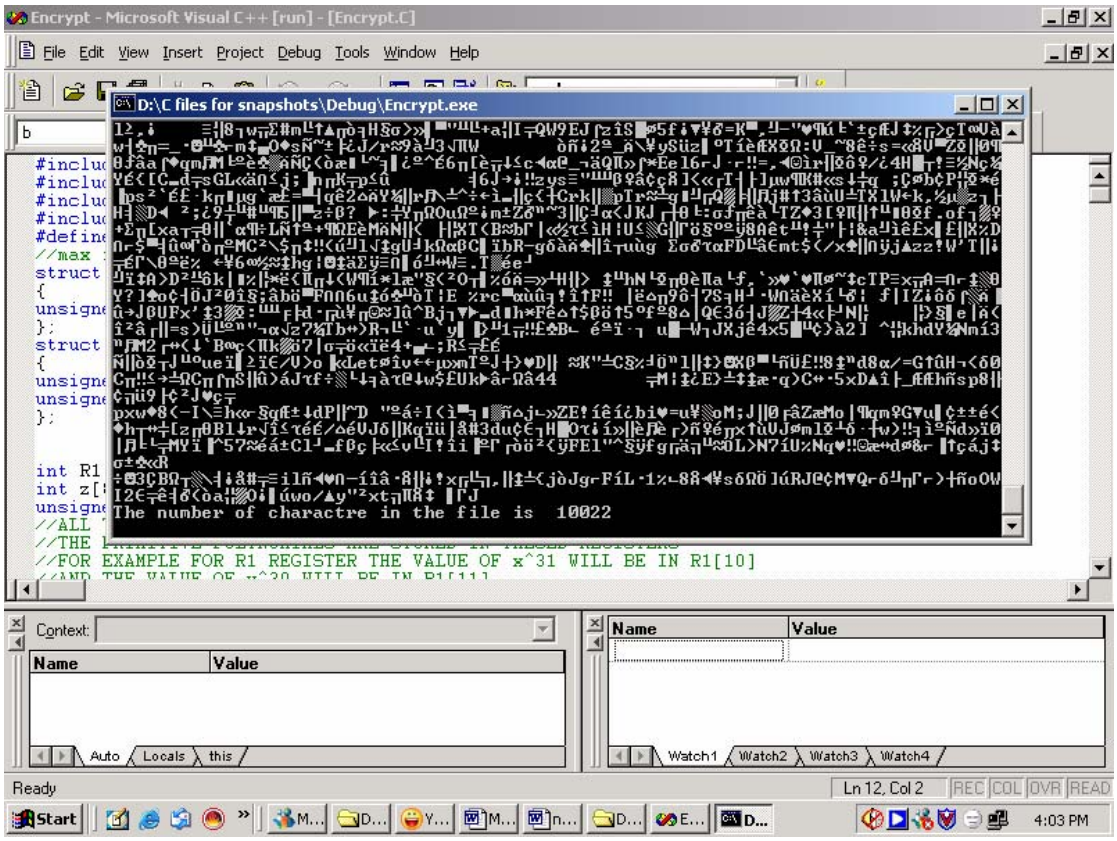
Appendix –B Snapshots



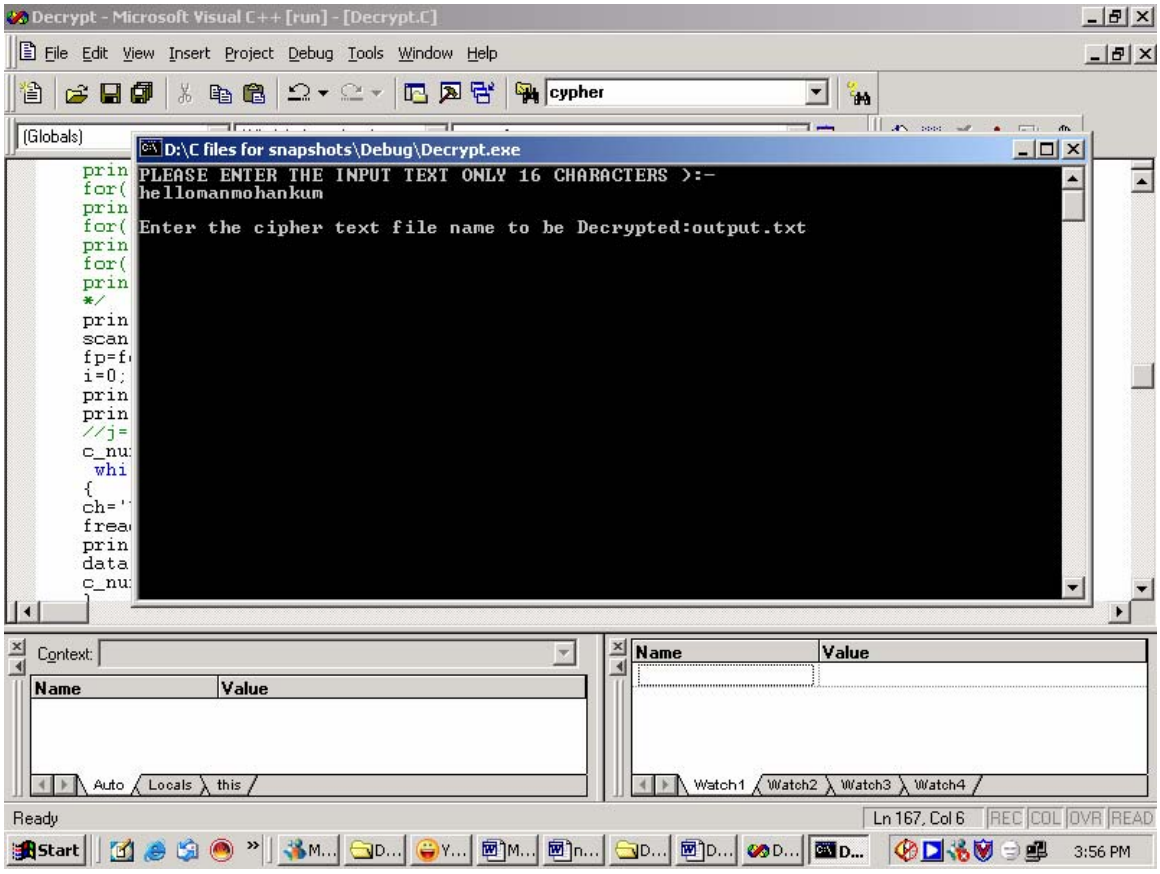
Screen for generating keys and Reading the input file to be Encrypted



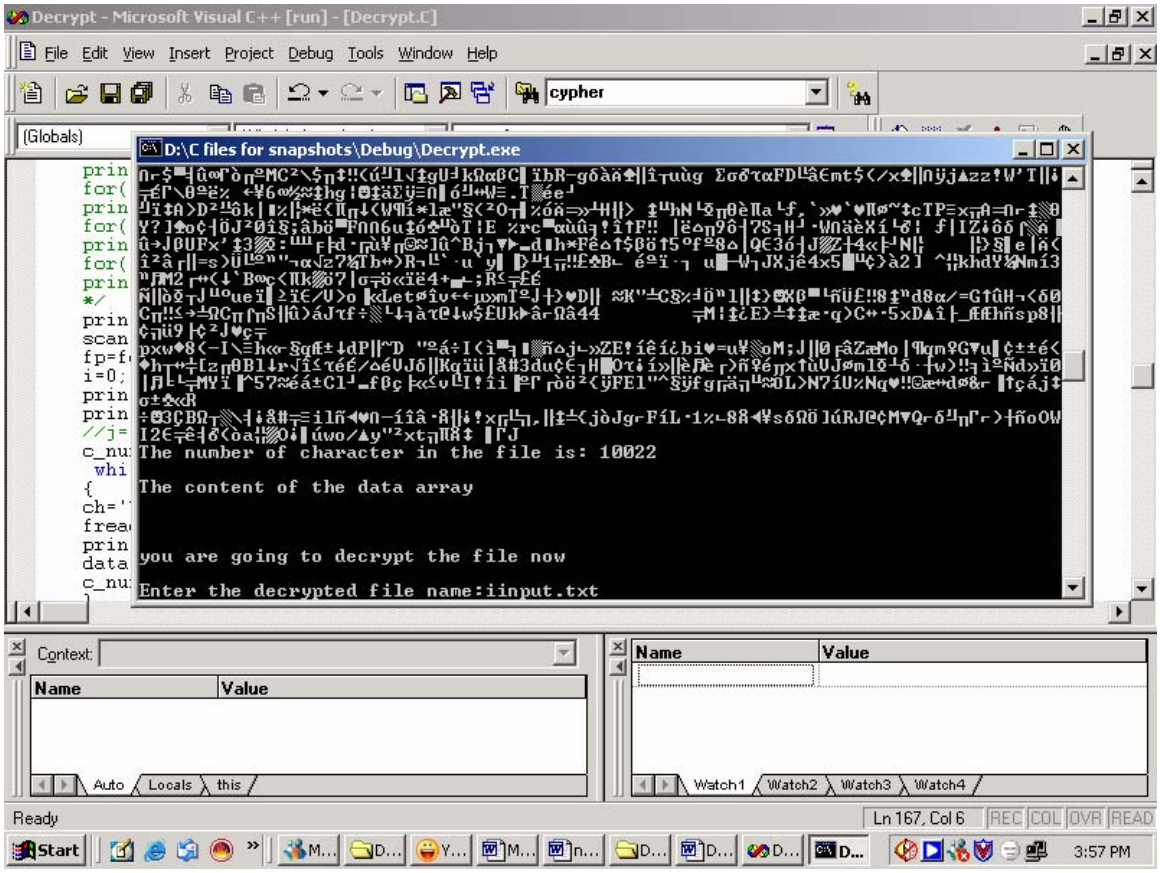
Content of Text file to be Encrypted



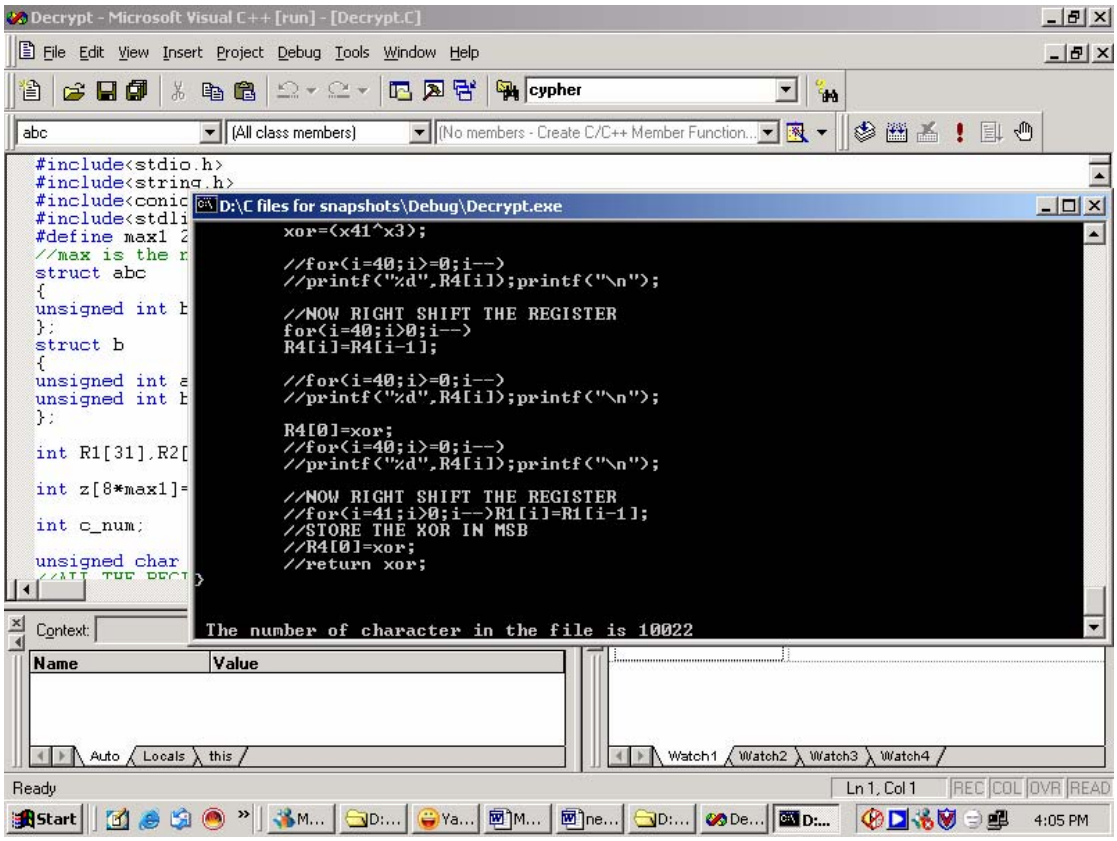
Encrypted Message



Screen for generating keys and Reading the input file to be Decrypted



Content of Encrypted file to be Decrypted



Decrypted Message(original Plain text)

REFERENCES AND BIBLIOGRAPHY

(1) Bruce Schneier , Applied Cryptography, Second Edition ,Protocols, Algorithms, and Source Code in C, John Wiley & Sons, Inc. New York, 1996.

(2) Soloman W. Golomb, Shift Register Sequences, Holden-Day, Inc, California,1967.

(3) Evangelos Kranakis, Primality and Cryptography John Wiley & Sons, Inc. New York, 1986.

(4) A. Menezes, P. Van Oorschot and S. Vanstone, "Handbook of Applied Cryptography" , CRC Press, 1996.

(5) "Advanced encrypton standard", http://csrc.nist.gov.

(6) "A statistical Test Suite For Random and Pseudo Random Number Generators for Cryptographic Applications", NIST special Publication 800-22(with revision dated May 15,2001).

(7) Bruce Schneier, Niels Ferguson, Practical Cryptography, Wiley Publishing Inc. 2003.

(8) F.C. Piper, Stream Ciphers, Lecture Notes in Computer Science, Proceedings of workshop on cryptography from 29th March to Aprila 2nd 1982, pp.179-216.

(9) Knuth, D.E, The Art of Computer Programming, Vol.2 : Semi Numerical Algorithms, 3rd edition, Addision –Wesley ,2000.

(10) S. Lakshmivarahan , "Algorithms for Public key Cryptosystem: Theory and application", Advances in Computer Science Vol.22 pp.45-107.

(11) Mohamad Peyravian, Stephen M. Matyas, Allen Rokingsky and Nev Zunic, "Generating User- Based Cryptographic keys and Random Numbers" Elsevier Limited, Great Britain 1999,pp.619-626.

(12) T. D. Mitra, "An Algorithm for Generation of Keys For the RSA Cryptosystem", Proceedings of the National Seminar On Cryptology, July 9-10, 1998,pp.G-1 SAG, DRDO, New Delhi.

(13) Horbert S.Bright, "Modern Computational Cryptography", Advance in Computer Security Management, Vol.2, 1983, pp173-201.

(14) George I. Davida, YVO Deshmedt, "Cryptography based data security", Advances in Computer Science, Vol. 30, pp.171-222.

(15) William Stallings, Cryptography and Network Security, Principles and Practice, Third Edition, Pearson Education Inc.

(16) Cees J.A. Jansen, Stream cipher Design : Make your LFSRs Jump!, SASC Workshop October 14-15, 2004.

(17) Patric Ekdahl, On LFSR based Stream Cipher, Anlysis and Design, LUND university.

(18) J.L. Massey, Shift Register Synthesis and BCH decoding. IEEE transaction on Information Theory, IT-15(1):122-127, January 1969.

(19) C.E. Shannon, Communication theory of secrecy systems, Bell System Technical Journal, 27:651-715,1949.

(20) Greg Rose, Phil Hawkes, Turing: a fast software stream cipher, Fast Software Encryption 2003, Lecture Notes in Computer Science. Springer-Verlag 2003.