A STUDY ON RANDOM NUMBER GENERATION AND BUILDING STREAM CIPHERS

A.Mahendar[1], **Dr. E. Srinivas Reddy**[2],

[1]*Research scholar, Department of Computer Science & Engineering, Acharya Nagarjuna University, AP, India,* ***mahi.adapa@gmail.com***

[2] *Professors, Department of Computer Science & Engineering, Acharya Nagarjuna University. AP, India,* ***edara_67@yahoo.com.***

Abstract

Random numbers are a fundamental tool in many cryptographic applications like key generation, encryption, masking protocols, or for internet gambling. Every random number generator has its advantages and deficiencies. There are no ''safe'' generators. The practitioner's problem is how to decide which random number generator will suit his needs best. In this paper, we will discuss criteria for good random number generators: theoretical support, empirical evidence and practical aspects. We will study several recent algorithms that perform better than most generators in actual use. We require generators which are able to produce large amounts of secure random numbers. Simple mathematical generators, like linear feedback shift registers (LFSRs), or hardware generators, like those based on radioactive decay, are not sufficient for these applications. In this paper three types of random number generators in depth. It also includes mathematical techniques for transforming the output of generators to arbitrary distributions.

Index Terms: *Stream Cipher, Block cipher, Stream cipher, Random Number Generators, TRNG, PRNG, CSPRNG,*

--- *** ---

1. INTRODUCTION

Random simulation has long been a very popular and well studied field of mathematics. There exists a wide range of applications in biology, finance, insurance, physics and many others. So simulations of random numbers are crucial. In this note, we describe the most random number algorithms. Let us recall the only things, that are truly random, are the measurement of physical phenomena such as thermal noises of semiconductor chips or radioactive sources [2]. The security of stream ciphers hinges entirely on a "suitable" key stream s_0, s_1, s_2. Since randomness plays a major role, we will first learn about the three types of random number generators (RNG) that are important for us. A random number generator is an algorithm that, based on an initial seed or by means of continuous input, produces a sequence of numbers or respectively bits. We demand that this sequence appears "random" to any observer.[2]
In addition to the conditions above RNGs for cryptographic applications must be resistant against attacks, a scenario which is not relevant in stochastic simulation. [3] This means that an adversary should not be able to guess any current, future, or previous output of the generator, even if he or she has some information about the input, the inner state, or the current or previous output of the RNG[4].

1.1 Stream Ciphers vs. Block Ciphers

Symmetric cryptography is split into block ciphers and stream ciphers, which are easy to distinguish. Figure 1.1 depicts the operational differences between stream (Fig. 1.1 a) and block ciphers when we want to encrypt *b* bits at a time, where *b* is the width of the block cipher.[3]

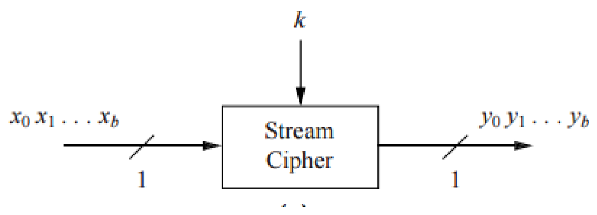

Fig.1.1. Principles of encrypting b bits with a stream (a) and a block (b) cipher.

A description of the principles of the two types of symmetric ciphers follows.

Stream ciphers encrypt bits individually. This is achieved by adding a bit from a Keystream to a plaintext bit[2]. There are synchronous stream ciphers where the key stream depends only on the key, and asynchronous ones where the key stream also depends on the ciphertext[5]. If the dotted line in Fig. 1.2 is present, the stream cipher is an asynchronous one. Most practical stream ciphers are synchronous ones and Sect. 2 of this chapter will deal with them.

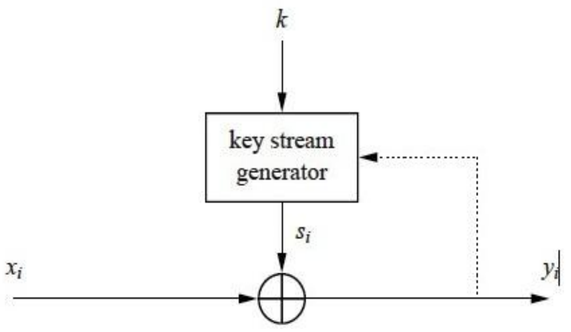

Fig. 1.2. Synchronous and asynchronous stream ciphers.

Block ciphers encrypt an entire block of plaintext bits at a time with the same key[4]. This means that the encryption of any plaintext bit in a given block depends on every other plaintext bit in the same block. In practice, the vast majority of block ciphers either have a block length of 128 bits (16 bytes) such as the ad-vanced encryption standard (AES)[6], or a block length of 64 bits (8 bytes) such as the data encryption standard (DES) or triple DES (3DES) algorithm[5]. All of these ciphers are introduced in later chapters.

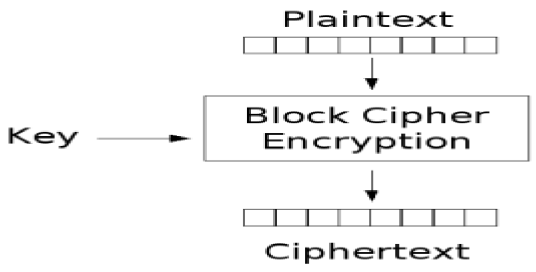

Fig. 1.3. Block ciphers.

2. Encryption and Decryption with Stream Ciphers

As mentioned above, stream ciphers encrypt plaintext bits individually. The question now is: [6] How does encryption of an individual bit work? The answer is surprisingly simple: Each bit xi is encrypted by adding a secret key stream bit s_i modulo 2.

Definition 2.1 Stream Cipher Encryption and Decryption

The plaintext, the ciphertext and the key stream consist of individual bits,
i.e., $x_i, y_i, s_i \in \{0,1\}$.

Encryption: $y_i = e_{si}(x_i) \equiv x_i + s_i \bmod 2$.
Decryption: $x_i = d_{si}(y_i) \equiv y_i + s_i \bmod 2$.

Since encryption and decryption functions are both simple additions modulo 2, we can depict the basic operation of a stream cipher as shown in Fig. 2.1. [6] Note that we use a circle with an addition sign as the symbol for modulo 2 additions.[3]
Just looking at the formulae, there are three points about the stream cipher encryption and decryption function which we should clarify:

1. Encryption and decryption are the same functions!
2. Why can we use a simple modulo 2 addition as encryption?
3. What is the nature of the key stream bits s_i?

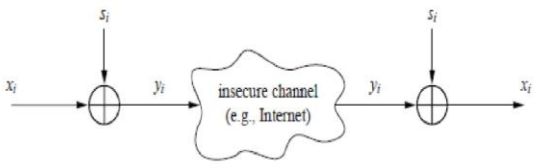

Fig. 2.1 Encryption and decryption with stream ciphers

2.1. Why Are Encryption and Decryption the Same Function?

The reason for the similarity of the encryption and decryption function can easily be shown. We must prove that the decryption function actually produces the plaintext bit *xi* again. We know that ciphertext bit *yi* was computed using the encryption function *yi* ≡ *xi* +*si* mod 2[8]. We insert this encryption expression in the decryption function.

$$\begin{aligned} d_{si}(y_i) &\equiv y_i + s_i \bmod 2 \\ &\equiv (x_i + s_i) + s_i \bmod 2 \\ &\equiv x_i + s_i + s_i \bmod 2 \\ &\equiv x_i + 2\, s_i \bmod 2 \\ &\equiv x_i + 0 \bmod 2 \\ &\equiv x_i \bmod 2 \quad \text{Q.E.D.} \end{aligned}$$

The trick here is that the expression (2 s_i mod 2) has always the value zero since 2 ≡ 0 mod 2. Another way of understanding this is as follows: If s_i has either the value 0, in which case $2s_i = 2 \cdot 0 \equiv 0 \bmod 2$. If $s_i = 1$, we have $2s_i = 2 \cdot 1 = 2 \equiv 0 \bmod 2$.

2.2. Why Is Modulo 2 Additions a Good Encryption Function?

A mathematical explanation for this is given in the context of the One-Time Pad in Sect. 2.2.2. However, it is worth having a closer look at addition modulo 2. If we do arithmetic modulo 2, the only possible values are 0 and 1 (because if you divide by 2, the only possible remainders are 0 and 1). Thus, we can treat arithmetic modulo 2 as Boolean functions such as AND gates, OR gates, NAND gates, etc[1]. Let's look at the truth table for modulo 2 addition:
This should look familiar to most readers: It is the truth table of the exclusive-OR, also called XOR, gate. This is in important fact: Modulo 2 addition is equivalent to the XOR operation. [2] The XOR operation plays a major role in modern cryptography and will be used many times in the remainder of this book.

Xi	Si	$Y_i \equiv X_i + S_i$ mod 2
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1 Truth table of the XOR operation

The question now is, why is the XOR operation so useful, as opposed to, for instance, the AND operation? [6] Let's assume we want to encrypt the plaintext bit xi = 0. If we look at the truth table we find that we are on either the 1st or 2nd line of the truth table:

Xi	Si	Y_i
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.2 Truth table of the XOR operation

Depending on the key bit, the ciphertext yi is either a zero (si = 0) or one (si = 1). If the key bit si behaves perfectly randomly, i.e., it is unpredictable and has exactly a 50% chance to have the value 0 or 1, then both possible ciphertexts also occur with a 50% likelihood. Likewise, if we encrypt the plaintext bit xi = 1, we are on line 3 or 4 of the truth table. Again, depending on the value of the key stream bit s_i, there is a 50% chance that the ciphertext is either a 1 or a 0.[3]
We just observed that the XOR function is perfectly balanced, i.e., by observing an output value, there is exactly a 50% chance for any value of the input bits. This distinguishes the XOR gate from other Boolean functions such as the OR, AND or NAND gate. Moreover, AND and NAND gates are not invertible. Let's look at a very simple example for stream cipher encryption. [9]

Example 2.1. Alice wants to encrypt the letter A, where the letter is given in ASCII code. [3] The ASCII value for A is 6510 = 10000012. Let's furthermore assume that the first key stream bits are (s0,..., s6)= 0101100.

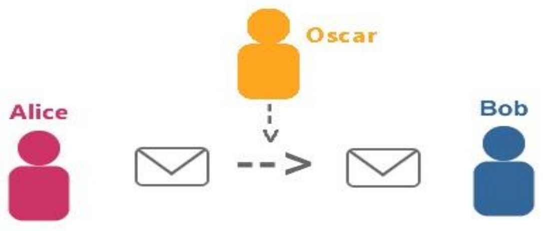

Fig: 2.2: **Encryption and decryption Alice and Bob**

Note that the encryption by Alice turns the uppercase A into the lower case letter m. Oscar,[7] the attacker who eavesdrops on the channel, only sees the cipher text letter m. Decryption by Bob with the same key stream reproduces the plaintext A again.

So far, stream ciphers look unbelievably easy: One simply takes the plaintext, performs an XOR [9] operation with the key and obtains the cipher text. On the receiving side, Bob does the same. The "only" thing left to discuss is the last question from above.

2.3. What Exactly Is the Nature of the Key Stream?

It turns out that the generation of the values si, which are called the key stream, is the central issue for the security of stream ciphers. In fact, the security of a stream cipher completely depends on the key stream. The key stream bits si are not the key bits themselves. So, how do we get the key stream? Generating the key stream is pretty much what stream ciphers are about .[11] This is a major topic and is discussed later in this chapter. However, we can already guess that a central requirement for the key stream bits should be that they appear like a random sequence to an attacker. [15] Otherwise, an attacker Oscar could guess the bits and do the decryption by himself. Hence, we first need to learn more about random numbers.

Historical Remark Stream ciphers were invented in 1917 by Gilbert Vernam, even though they were not called stream ciphers back at that time.[12] He built an elec- tromechanical machine which automatically encrypted teletypewriter communica- tion. The plaintext was fed into the machine as one paper tape, and the key stream as a second tape. This was the first time that encryption and transmission was au- tomated in one machine.[19] Vernam studied electrical engineering at Worcester Poly- technic Institute (WPI) in Massachusetts where, by coincidence, one of the authors of this book was a professor in the 1990s. Stream ciphers are sometimes referred to as Vernam ciphers. Occasionally, one-time pads are also called Vernam ciphers. For further reading on Vernam's machine, the book by Kahn [97] is recommended.

3. Random Numbers and an Unbreakable Stream Cipher

3.1. Random Number Generators

As we saw in the previous section, the actual encryption and decryption of stream ciphers is extremely simple. The security of stream ciphers hinges entirely on a "suitable" key stream $s_0, s_1, s_2, \ldots$.[16] Since randomness plays a major role, we will first learn about the three types of random number generators (RNG) that are important for us.

3.2. True Random Number Generators (TRNG)

True random number generators (TRNGs) are characterized by the fact that their output cannot be reproduced.[8] For instance, if we flip a coin 100 times and record the resulting sequence of 100 bits, it will be virtually impossible for anyone on Earth to generate the same 100 bit sequence[8]. The chance of success is $1/2^{100}$, which is an extremely small probability. TRNGs are based on physical processes. Examples include coin flipping, rolling of dice, semiconductor noise, clock jitter in digital circuits and radioactive decay. In cryptography, TRNGs are often needed for gener- ating session keys, which are then distributed between Alice and Bob, and for other purposes.

3.3. (General) Pseudorandom Number Generators (PRNG)

Pseudorandom number generators (PRNGs) generate sequences which are com- puted from an initial seed value. Often they are computed recursively in the follow- ing way:

$$s0 = seed$$
$$s_i+1 = f(s_i),\quad i = 0, 1, \ldots$$

A generalization of this are generators of the form si+1 = f (si, si 1,..., si t), where t is a fixed integer. A popular example is the linear congruential generator:

$$s_0 = seed$$
$$s_i+1 \equiv as_i + b \bmod m,\ i = 0, 1, \ldots$$

where a, b, m are integer constants. Note that PRNGs are not random in a true sense because they can be computed and are thus completely deterministic. A widely used example is the rand() function used in ANSI C. It has the parameters:

$$s_0 = 12345$$
$$s_{i+1} \equiv 1103515245\ s_i + 12345 \bmod 2^{31},\ i = 0, 1,...$$

A common requirement of PRNGs is that they possess good statistical proper- ties, meaning their output approximates a sequence of true random numbers. [3] There are many mathematical tests, e.g., the chi-square test, which can verify the statistical behavior of PRNG sequences. Note that there are many, many applications for pseu- dorandom numbers outside cryptography. For instance, many types of simulations or testing, e.g., of software or of VLSI chips, need random data as input. That is the reason why a PRNG is included in the ANSI C specification.[18]

3.4. Cryptographically Secure Pseudorandom Number Generators (CSPRNG)

Cryptographically secure pseudorandom number generators (CSPRNGs) are a special type of PRNG which possess the following additional property: A CSPRNG is PRNG which is unpredictable. Informally[9], this means that given n output bits of the key stream s_i, s_i+1,..., $s_{i+n-1,}$ where n is some integer, it is computationally infeasible to compute the subsequent bits s_{i+n}, s_{i+n+1},.. .. A more exact definition is that given n consecutive bits of the key stream, there is no polynomial time algorithm that can predict the next bit s_{n+1} with better than 50% chance of success.[12] Another property of CSPRNG is that given the above sequence, it should be computationally infeasible to compute any preceding bits si_{-1}, s_{i-2},…

Note that the need for unpredictability of CSPRNGs is unique to cryptography [13]. In virtually all other situations where pseudorandom numbers are needed in computer science or engineering, unpredictability is not needed..

4. Basic Building Blocks of stream cipher

In this section we describe some commonly used building blocks for the construction of stream ciphers. We start with a description of a linear feedback shift register [18], which is a very popular building block in stream cipher primitives and then we shift our focus to Non-linear feedback shift registers, which gives us an opportunity to avoid the inherent linearity faced in LFSRs. Following this, we present a discussion on various methods of introducing nonlinearity [2].

4.1. Linear feedback shift register

Linear Feedback Shift Registers (LFSRs) [16] have always received considerable attention in cryptography. Owing to the good statistical properties, large period and low implementation costs, LFSR have achieved wide acceptance in developing stream ciphers [15]. An LFSR is a shift register that, using feedback, elevates the bits through the register from current location to the next most-significant location, on each rising edge of the clock. Selected outputs (taps) are combined in an exclusive-OR (or exclusive-NOR) fashion to form a feedback mechanism, which causes the value in the shift register iterate endlessly through a sequence of unique values. An LFSR of any given size n (number of registers) is capable of producing every possible state during the period N=2^{n-1} excluding the all-zero state, such a sequence is called maximal sequence (abbreviated as m-sequence).Linear feedback shift registers as maximal length sequence generators are widely used in stream ciphers for key stream generation due to their good statistical properties [17], large period, low implementation costs and are readily analysed using algebraic techniques. Maximal length sequences are generated when the LFSR passes through every non-zero state once and only once and are obtained when the feedback polynomial to which LFSR corresponds is primitive [JJD09], a feedback polynomial of degree n is primitive if it is irreducible (cannot be factored) and has a period equivalent to $2^{n-1.}$

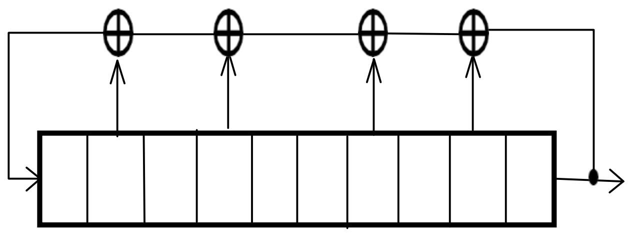

Fig 4.1: Linear feedback shift Register (LFSR)

The predominant characteristic like large linear complexities, large period, statistical properties and pseudo randomness of the key stream generated by LFSR make LFSR [14] a good choice for developing stream ciphers besides allowing a ready algebraic analysis of keystream generated by linear feedback shift registers.

4.2. Mechanism of an LFSR:

The implementation of Linear feedback shift register consists of n input shift registers, where the input bit is calculated as a linear function of the content of the register. An n stage LFSR consists of clocked storage elements in the form of a shift register S and a feedback path in the form of tap sequence T where shift register S=(s_n,s_{n-1},sn- 2,……..s_1) and a tap sequence T=(t_n,t_{n-1},t_{n-2},……..t_1), with each s_1 and t_1 being one binary digit. At each clock interval, all the bits are shifted right except bit s_1, which is appended to the key stream, and a new bit derived from S and T is fed back as input to the left end of the register. Whether a feedback is active or not is determined by feedback coefficients T= (t_n, tn_{-1}...t1)

- If ti = 1, feedback is active
- if ti = 0, feedback is passive

A Linear feedback function produces a sequence S, satisfying the linear recurrence function. Assuming that the LFSR is initially loaded with some seed value s0, s1, s2…sn-1. [3]The next output bit is computed by the XOR-sum of products operation of storage elements and corresponding taps:

$$S_n \equiv s_{n-1}t_{n-1} + \dots\dots\dots + s_1t_1 + s_0t_0 \text{ mod } 2$$

Similarly next output is:

$$S_{n+1} \equiv s_n t_{n-1} + \cdots + s_2t_1 + s_1t_0 \text{ mod } 2$$

Finally the general output can be shown as:

$$S_{n+1} \sum_{j!0}^{n!1} tj \cdot Si + jmod2$$

Since the number of recurring states is finite, the generated sequence produced by LFSR must repeat itself after a finite period and also the length of the sequence is completely determined by the feedback coefficients and seed value.
Since an n-bit vector can assume only 2n-1 states excluding an all-zero state, An n-bit LFSR [12]can deterministically assume its next state based on its previous state, as a result of which, as soon as an LFSR encounters a previous state, It starts to repeat itself. Therefore the maximum sequence length without repetition is 2n-1. An all-zero state is discarded because if an LFSR assumes this state, it will get “stuck” and will never be able to leave this state [19].
Modification of the feedback scheme allows us to implement LFSR in two different variations. Though the variations are cryptographically no better, but it can still affect the periodicity and software implementation [SCH96]. [16] Based on the configuration of gates and registers,[18] LFSR can be divided in two categories.

- The Fibonacci LFSR, also known as External-XOR LFSR or just LFSR.
- The Galois LFSR, also known as Internal-XOR or canonical LFSR.

In the Fibonacci implementation, the taps are XORed sequentially with the output bit and the fed back into the leftmost bit. The shift register is initially loaded with bits a0, a1… ar-1 called the seed value [15] (any value except all zeroes) and then clocked while as in the Galois implementation, with each clock cycle, bits that are not taps are shifted one position to the right unchanged. The taps on the other hand, are XORed with the output bit before they are stored in the next bit. The shift register is initially loaded with bits a0, a1… ar-1 called the seed value (any value except all zeroes) and then clocked.

5. Nonlinear feedback shift register

Non-Linear Feedback Shift Registers (NLFSRs) have been proposed as an alternative to Linear Feedback Shift Registers (LFSRs) for generating pseudo-random sequences for stream ciphers. A Non-Linear Feedback Shift Register (NLFSR) consists of n binary storage elements, called bits[16]. Each bit i has an associated state variable xi which represents the current value of the bit i and a feedback function fi which determines how the value of i is updated.[18] A state of an NLFSR is an ordered set of values of its state variables (x0, x1, ... xn-1). At every clock cycle, the next state is determined from the current state by updating the values of all bits simultaneously to the values of the corresponding fi. NLFSRs have been shown to be more resistant to cryptanalytic attacks than LFSRs. However, construction of large NLFSRs with guaranteed long periods remains an open problem.
Cryptographically strong pseudo-random sequences are produced by combining more than one LFSR[18] with some method to introduce non linearity. Three general methods of combining LFSRs employed to overcome the problem of linearity in LFSR based stream ciphers are:

- Non linear combination generator
- Non linear filter generator
- Clock-controlled generator

5.1. Non Linear Combination Generator

The key stream is generated by manipulating the outputs of several parallel LFSRs using a non linear Boolean function f. The function f is called the combining function and maps one or more binary input variables to a binary output variable[14].

The Boolean function must have a high algebraic degree, high nonlinearity and preferably a high order of correlation immunity. The keystream generated z is given by z = f(x1,x2 ,… xn), where x1,x2….xn [15] are the outputs of n-sub generators. General model for a nonlinear combination generator is shown in figure below.

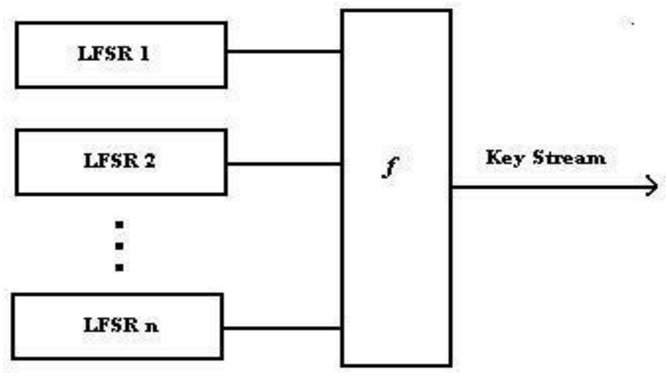

Fig 5.1: Non linear combination generator

If n maximum-length LFSRs with lengths$l_1,l_2,\ldots$ln are used together with the Boolean function f, the linear complexity of the keystream is

$f (l_{1,}l_2,\ldots l_n)= a_0+ a_1l_1+\ldots a_nl_n+\ldots a_{12\ldots n}l_1l_2\ldots.. l_n$

Where $a_0,a_1\ldots a_n$ are the coefficients of in the algebraic normal form of f.

5.2. Non Linear Filter Generator

A nonlinear filter generator uses a single maximum-length LFSR, and the keystream [16] is generated as a nonlinear function f of the state of the LFSR.

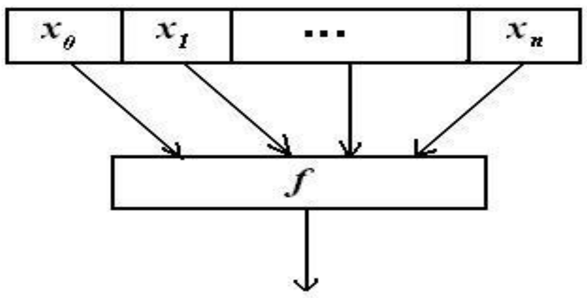

Fig 5.2: Non linear filter generator

The function *f* is called the filtering function. In this generator a single maximum length LFSR is used in contrast to the non linear combination generator where several LFSRs were used and different stages of single LFSR[13]are used as input t the filtering function *f*. If a nonlinear filter generator is constructed by using a maximum-length LFSR of length L and a filtering function f of nonlinear order m then the linear complexity of the keystream is at most:

$$L_m = \sum_{i=1}^{m} \binom{n}{i}$$

In particular, here are some problems with nonlinear-feedback shift register sequences.

- There may be biases, such as more ones than zeros or fewer runs than expected, in the output sequence.
- The maximum period of the sequence may be much lower than expected.
- The period of the sequence might be different for different starting values
- The sequence may appear random for a while, but then “dead end” into a single value.

(This can easily be solved by XORing the nonlinear function with the rightmost bit.) [7] On the plus side, if there is no theory to analyze nonlinear-feedback shift registers for security, there are few tools for cryptanalysis of stream ciphers based on them. We can use nonlinear-feedback shift registers in stream-cipher design, but we have to becareful.

6. CONCLUSION

As an overview of the state of the art of stream ciphers build from block ciphers, this paper presented the standard modes of operation along with some uncommon modes. Apart from pointing out methods of exploiting and avoiding the most common design errors. The categorization into synchronous and asynchronous stream ciphers proved helpful as a first estimate for the properties of a stream cipher and hence can be used as a starting point for the choice of algorithms during the development of a cryptosystem.

Having a right source of random numbers is a complex assumption of many protocol systems. There have been several high profile failures in random numbers generators that led to severe practical problems. For this wide application of random numbers we just were trying to find a robust & efficient algorithm to make it more effective in the relevant domain. In this connection, we also enjoy to work with such kind of challenging research. We think it is truly successful.

REFERENCES

[1]. M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete securitytreatment of symmetric encryption. pages 394–403, 1997.

[2]. D. Chaum and J.-H. Evertse. Cryptanalysis of DES with a reduced number of rounds. 1985.

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

[4]. P. L'Ecuyer. Random Number Generation, chapter 2, pages 35{70. Springer- Verlag, Berlin, Germany, 2004. Handbook of Computational Statistics.

[5]. P. Hellekalek. Good random number generators are (not so) easy to _nd. Mathematics and Computers in Simulation, 46:485{505, 1998.

[6]. C. Ellison. Cryptographic Random Numbers. IEEE P1363 Appendix E, November 1995. Draft version 1.0, http://world.std.com/~cme/P1363/ranno. html.

[7]. H. Niederreiter. Random number generation and quasi-Monte Carlo methods. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1992.

[8]. J. Eichenauer, J. Lehn, A non-linear congruential pseudo random number generator, Statist. Papers 27 (1986) 315±326.

[9]. J. Eichenauer-Herrmann, Explicit inversive congruential pseudorandom numbers: the compound approach, Computing 51 (1993) 175±182.

[10]. A. Compagner, Operational conditions for random-number generation, Phys. Review E 52 (1995) 5634±5645.

[11]. P. Hellekalek, Inversive pseudorandom number generators: concepts, results, and links, in: C. Alexopoulos, K. Kang, W.R. Lilegdon, D. Goldsman (Eds.), Proceedings of the 1995 Winter Simulation Conference, 1995, pp. 255±262.

[12]. M. Hennecke, Random number generators homepage, http://www.uni-karlsruhe.de/ RNG/.

[13]. J. Eichenauer-Herrmann, Explicit inversive congruential pseudorandom numbers: the compound approach, Computing 51 (1993) 175±182.

[14]. P. L'Ecuyer, Random number generation, In Jerry Banks (Ed.), Handbook on Simulation, Wiley, New York, 1997.

[15]. Luby, Michael George., Pseudorandomness and Cryptographic Applications,(Princeton, NJ: Princeton University Press, 1996).

[16]. "Cryptographic Random Numbers", IEEE P1363 Working Draft, Appendix G, 6 February 1997.

[17]. O. Goldreich. Modern Cryptography, Probabilistic Proofs and Pseudorandomness. Algorithms and Combinatorics. Springer-Verlag, Berlin, Germany, 1999.

[18]. Stinson, Douglas R., Cryptography: Theory and Practice, (Boca Raton:Chapman & Hall, 2002).

[19]. P. L'Ecuyer, Uniform random number generation, Ann. Oper. Res. 53 (1994) 77±120.

[20]. "Using and Creating Cryptographic-Quality Random Numbers", John Callas, `http://www.merrymeet.com/jon/usingrandom.html`, 3 June 1996.

BIOGRAPHIES

1. **Mr. A.Mahendar** is research scholar, department of computer science & engineering, Acharya Nagarjuna University. mahi.adapa@gmail.com.

2. **Dr. E. Srinivasa Reddy** is working as a professor in the department of computer science & engineering, at Acharya Nagarjuna University. edara_67@yahoo.com.