A Search for Good Pseudo-random Number Generators : Survey and Empirical Studies

Kamalika Bhattacharjee[a], Krishnendu Maity[a], Sukanta Das[a,*]

[a] *Department of Information Technology, Indian Institute of Engineering Science and Technology, Shibpur, West Bengal, India 711103*

Abstract

In today's world, several applications demand numbers which appear random but are generated by a background algorithm; that is, pseudo-random numbers. Since late 19^{th} century, researchers have been working on pseudo-random number generators (PRNGs). Several PRNGs continue to develop, each one demanding to be better than the previous ones. In this scenario, this paper targets to verify the claim of so-called good generators and rank the existing generators based on strong empirical tests in same platforms. To do this, the genre of PRNGs developed so far has been explored and classified into three groups – linear congruential generator based, linear feedback shift register based and cellular automata based. From each group, well-known generators have been chosen for empirical testing. Two types of empirical testing has been done on each PRNG – blind statistical tests with Diehard battery of tests, TestU01 library and NIST statistical test-suite and graphical tests (lattice test and space-time diagram test). Finally, the selected 29 PRNGs are divided into 24 groups and are ranked according to their overall performance in all empirical tests.

Keywords: Pseudo-random number generator (PRNG), Diehard, TestU01, NIST, Lattice Test, Space-time Diagram

I. Introduction

History of human race gives evidence that, since the ancient times, people has generated random numbers for various purposes. As an example, for them, the output of rolling a dice was a sermon of God! However, in the modern times, researchers and scientists have discovered diverse applications and fields, like probability theory, game theory, information theory, statistics, gambling, computer simulation, cryptography, pattern recognition, VLSI testing etc., which require random numbers. Most of these applications entail numbers, which

*Corresponding author

Email addresses: kamalika.it@gmail.com (Kamalika Bhattacharjee), krishnendu58@gmail.com (Krishnendu Maity), sukanta@it.iiests.ac.in (Sukanta Das)

appear to be random, but which can be reproduced on demand. Such numbers, which are generated by a background algorithm, are called pseudo-random numbers and the implementation of the algorithms as pseudo-random number generators (PRNGs). In this work, however, by random number, we will mean pseudo-random numbers only.

Even, PRNGs have a long history of development – its modern journey starting in late 19^{th} century [1] to early 20^{th} century [2–5] and evolving and getting more powerful ever since [6–17]. There are several papers which target to survey this development of random number generators, see for example [18–23]. Usually, latest PRNG claims to be superior to the previous ones. This claim is based on the PRNG's performance in some statistical tests, like Diehard [24], TestU01 [25], NIST [26] etc. battery of tests, which empirically detect non-randomness in the generated numbers. However, many questions arise in this regard – What is the necessary criterion of a good PRNG? What should be the measurement unit of its randomness quality? Should randomness of a number generated by a PRNG be relative to its intended application? Should a PRNG need to pass all the tests of batteries? How much effective are those statistical tests? Will numbers of a PRNG, which performs well in all the statistical tests, really appear random or noisy to the human eye? Is the claim of a PRNG to be superior really correct? How to verify the ranking of these PRNGs?

In this work, we target to address some of these questions. Here, we have selected the uniform PRNGs that are considered to be *good*. Numbers are generated using the *C* programs available on the Internet for these PRNGs. These numbers are tested uniformly using all existing statistical testbeds. Then, some visual tests are applied on these numbers. If a PRNG is really good, then the result of these statistical tests and visual tests should correlate and the numbers is to appear noisy to the human eye. The result of testing for all these PRNGs are further interpreted. We have observed that, for many PRNGs, the claim and actual independent result do not tally. Finally, a ranking for the existing renowned PRNGs is given based on the result we have got. For an intended application, any user may choose a PRNG according to its rank.

This paper is organized as follows. In Section II, the essential properties of the PRNGs are described. Section III classifies the journey of the PRNGs through three technologies – Linear Congruential Generators (LCGs), Linear Feedback Shift Registers (LFSRs) based and Cellular Automata (CAs) based. Total 29 currently used PRNGs are selected for empirical testing. The empirical tests and test-beds are described in Section IV. In Section V, the test results of the PRNGs, which are selected in Section III, are depicted. A relative ranking of these PRNGs based on the empirical results is given in Section V.3. Finally, Section VI concludes the paper.

II. PRNGs and their Properties

Pseudo-random number generators are simple deterministic algorithms which produce deterministic sequence of numbers that appear random. For this reason, such numbers are called pseudo-random numbers. In general, a PRNG

produces uniformly distributed, independent and uncorrelated real numbers in the interval $[0, 1)$. However, generation of numbers in other probability distribution is also possible. Mathematically, a PRNG is defined as the following [27]:

Definition 1. *A pseudo-random number generator G is a structure $(\mathscr{S}, \mu, f, \mathscr{U}, g)$, where $\mathscr{S}$ is a finite set of states, μ is the probability distribution on $\mathscr{S}$ for the initial state called seed, $f : \mathscr{S} \to \mathscr{S}$ is the transition function, $\mathscr{U}$ is the output space and $g : \mathscr{S} \to \mathscr{U}$ is the output function. The generator G generates the numbers in the following way.*

1. *Select the seed $s_0 \in \mathscr{S}$ based on μ. The first number is $u_0 = g(s_0)$.*
2. *At each step $i \geq 1$, the state of the PRNG is $s_i = f(s_{i-1})$ and output is $u_i = g(s_i)$. These output u_is of the PRNG are the (pseudo-)random numbers.*

Since a PRNG is a finite state machine with a finite number of states, after a finite number of steps, eventually it will come back to the same state and the sequence will be repeated. This property is common to all sequences where a function f transforms a finite set into itself, that is, $x_n = f(x_{n-1})$. This repeating cycle is known as the *period*. The period of a PRNG is the smallest positive integer ρ, such that, $\forall n \geq k, s_{\rho+n} = s_n$, here $k \geq 0$ is an integer. The smallest k which satisfies this equation is called transient. If $k = 0$, the sequence is purely periodic. Preferably, $\rho \approx |\mathscr{S}|$, or, $\rho \approx 2^b$, if b bits represent each state. A PRNG with maximum possible period is called *maximum-period generator*.

Ideally, a PRNG has only one period, that is, all unique numbers of the output space are part of the same cycle. In that case, the PRNG is *maximum-period generator*. However, many PRNGs exist, which have more than one cycle. So, depending on the seeds, completely different sequence of numbers of distinct cycles may be generated. This situation is shown in Figure 1.

(a) Maximum-period PRNG (b) PRNG with non-maximum periods

Figure 1: Cycle structure of PRNGs

However, only LCGs (described in Section III.1) can attain the maximum possible period. For LFSRs based generators (described in Section III.2), the largest achievable period is one less than the maximum period. Hence, they are non-maximum period generators. Similarly, CAs are also non-maximum period generators.

Every PRNG is classified by the functions f, g, the seed s_0 and the number of iterations i. Therefore, when a PRNG is observed for its randomness quality, it is considered that, the algorithm is not known to the adversary. In that

case, the properties which a PRNG needs to possess to be a good PRNG, are described next.

• **Properties of PRNG :** A PRNG is called *good*, if it satisfies the following properties –

1. **Uniformity :** This property implies that, if we divide the set of numbers generated by the PRNG into K equal subintervals, then expected number of samples (e_i) in each subinterval $i, (1 \leq i \leq K)$ is equal; that is, $\forall i, e_i = \frac{N}{K}$, where N is the range of the numbers. This ensures that, the generated numbers are equally probable in every part of the number space.

2. **Independence :** The generated numbers are to be independent of each other; that is, there should not be any serial correlation between numbers generated in succession. So, any subsequence of numbers have no correlation with any other subsequences. This means, given any length of previous numbers, one can not predict the next number in the sequence by observing the given numbers.

3. **Large Period :** Every PRNG has a period after which the sequence is repeated. A PRNG is considered good if it has a very large period. Otherwise, if one can exhaust the period of a PRNG, the sequence of numbers become completely predictable.

4. **Reproducibility :** One of the prominent reason of developing PRNG is its property of reproducibility. This ensures that given the same seed s_0, the same sequence of numbers is to be generated. This is very useful in simulation, debugging and testing purposes.

5. **Consistency :** The above properties of the PRNG are to be independent of the seed. That is, all these properties are to be maintained for every seed value.

6. **Disjoint subsequences :** There is to be little or no correlation between subsequences generated by different seeds.

7. **Permutations :** Every permutation of a number generated by a PRNG is expected to be equally likely. Otherwise, the numbers can be biased and may help to predict successive numbers.

8. **Portability :** A PRNG is to be portable; that is, the same algorithm can work on every system. Given the same seed, different machines with varied configuration are to give the same output sequence.

9. **Efficiency :** The PRNG is to be very fast; which means, generation of a random number takes insignificant time. Moreover, a PRNG should not use much storage or computational overhead. This is to make certain that, the use of PRNG in an application is not a hindrance to its efficiency.

10. **Coverage :** This implies whether the PRNG covers the output space for any seed. Many PRNG has less coverage. In case, the PRNG has more than one cycle, then it may happen that, although it covers the whole output space, but only a part of it is covered by a particular seed.

11. **Spectral Characteristics :** A good PRNG does not generate numbers of one frequency higher than any other. If we plot the consecutive numbers, there are not to be any pattern visible for any length of the sequence.

12. **Cryptographically Secure :** To be used in cryptographic applications, the generated numbers should be cryptographically secure. This is desirable property often missing in most of the algorithmic PRNGs.

Many of these properties are inter-related. For example, if the numbers are not uniform, they are correlated and have identifiable patterns. Ideally, the numbers of a good PRNG are to satisfy all these properties. However, practically, most of the PRNGs do not possess all these properties, for example, the properties $6, 7$ and 12 are often missing in the existing PRNGs. Still, in terms of usage in the applications for which they are intended, many PRNGs are considered good in today's standard. In the next section, we tour to the existing PRNGs to classify them with respect to their underlying architecture and verify their randomness quality.

III. Classification of the PRNGs

Earliest PRNGs which satisfied the properties of uniformity and independences with a relatively large period were based on linear recurrences modulo a prime number, popularly called *linear congruential sequence*. Introduced by Lehmer [28], such a PRNG is named *linear congruential generator* (LCG). Most of the existing PRNGs are variants of it. However, another type of linear recurrences, where the modulo operator is 2, soon became popular due to their ease of implementation and efficiency in computer's binary arithmetic. These types of recurrences work mostly based on a *linear feedback shift register* (LFSR). Introduced by Tausworthe [13], this scheme has instigated many researchers to implement their PRNGs based on its variants. For example, the celebrated PRNG *Marsenne Twister* [8] is implemented using a variation of this technology.

Another type of research on random number generators also exists, where the target is to exploit the intricate chaotic behavior originated by simple functions with local interaction to develop the random numbers. This research was initiated by Wolfram [29], where he used a cellular automaton (CA) as the source of psudo-randomness. Therefore, we can classify the PRNGs in three main categories – 1) LCG based, 2) LFSR based and 3) CA based.

III.1. LCG based PRNGs

One of the most popular random number generation technique is based on linear recursions on modular arithmetic. These generators are specialization on the

linear congruential sequences, represented by

$$x_{n+1} = (ax_n + c) \pmod{m}, \quad n \geq 0 \tag{1}$$

Here $m > 0$ is the modulus, a is the multiplier, c is the increment and x_0 is the starting value or seed; $0 \leq a < m$, $0 \leq c < m$, $0 \leq x_0 < m$. The sequence $(x_i)_{i \geq 0}$ is considered as the desired sequence, and the output is $u_i = \frac{x_i}{m}$, if anybody wants to see the numbers from $[0, 1)$. However, not all choices of m, a, c, x_0 generate a random sequence. For example, if $a = c = 1$, the sequence is not random. Therefore, selection of these magic numbers is crucial for getting a random sequence of numbers.

We can observe that, maximum period possible for an LCG is m. However, to get a maximum-period LCG, the following conditions need to be satisfied [30] :

1. c is relatively prime to m;
2. if m is multiple of 4, $a - 1$ is also multiple of 4;
3. for every prime divisor p of m, $a - 1$ is multiple of p.

Some well known LCGs are reported in [18, 30, 31]. The PRNGs used in computer programming are mainly LCGs, e.g. UNIX rand() and drand48(), Random() in java.util.Random class etc.

Many variations of LCGs were proposed. For example, if we take the increment $c = 0$, then the generator is called *multiplicative (*or, *mixed) congruential generator* (MCG):

$$x_{n+1} = ax_n \pmod{m}, \quad n \geq 0 \tag{2}$$

Although generation of numbers is slightly faster in this case, but the maximum period length of m is not achievable. Because, here $x_n = 0$ can never appear unless the sequence deteriorates to zero. When $c = 0$ and x_n is relatively prime to m for all n, the length of the period is limited to $\varphi(m)$, that is, the number of integers between 0 and m that are relatively prime to m [30]. Now, if $m = p^e$, where p is a prime number and $e \in \mathbb{N}$, Equation 2 reduces to:

$$x_n = a^n x_0 \pmod{p^e}$$

Taking a as relatively prime to p, the period of the MCG is the smallest integer λ such that,

$$x_0 = a^\lambda x_0 \pmod{p^e}$$

Let p^f be the gcd of x_0 and $m = p^e$, then this condition reduces to

$$a^\lambda = 1 \pmod{p^{e-f}}$$

When a is relatively prime to m, the smallest integer λ for which $a^\lambda = 1 \pmod{p^{e-f}}$ is called the *order of a modulo m*. Any value of a with maximum possible order modulo m is called a *primitive element modulo m*. Therefore, the maximum achievable period for MCGs is the order of a primitive element, or maximum possible order, modulo m, equal to $m - 1$ [30], when

1. m is prime;

2. a is a primitive element modulo m;

3. x_0 is relatively prime to m.

Some MCGs with large period are reported in [32–34]. However, these generators perform unsatisfactorily in spectral tests [30]. Therefore, higher order linear recurrences are proposed of the form

$$x_n = a_1 x_{n-1} + \cdots + a_k x_{n-k} \pmod{m} \tag{3}$$

where $k \geq 1$ is the order. When $k = 1$, the generator of Equation 3 is an MCG. Here, $x_0, \cdots, x_{k-1}$ are arbitrary but not all zero. For these recurrences, the best result can be derived when $m = p$ where p is a large prime. In this case, according to the theory of finite fields, multipliers $a_1, \cdots, a_k$ exist, such that, the sequence of Equation 3 has period of length $p^k - 1$, if and only if the polynomial

$$P(z) = z^k - a_1 z^{k-1} - \cdots - a_k \tag{4}$$

is a *primitive polynomial modulo* p [30]. That is, if and only if, the root of the polynomial is a primitive element of the Galois field with p^k elements[1]. A generator with such recurrence is called a *multiple recursive generator* (MRG) [27]. There are exactly $\varphi(p^k - 1)/k$ suitable choices of $a_1, \cdots, a_k$. To test primitivity modulo p of Equation 4 for any choice of $(a_1, \cdots, a_k)$, the following criteria can be used [30]: Let $r = (p^k - 1)/(p - 1)$, the conditions are

1. $(-1)^{k-1} a_k$ is a primitive root modulo p;

2. the polynomial z^r is congruent to $(-1)^{k-1} a_k$, modulo $P(z)$ and p;

3. for each prime divisor q of r, the degree of $z^{r/q} \pmod{P(z)}$ is positive.

However, the limiting factor in testing primitivity modulo p comes from prime factorization of $r = (p^k - 1)/(p - 1)$. For $k \geq 4$ and a large p, this prime factorization is difficult to handle. Nevertheless, finding the constants $a_1, \cdots, a_k$ for $p = 2$ that defines primitive polynomials modulo 2 is interesting for generating random sequence of bits with large period. Later, many generators (e.g. LFSRs, maximal-length CAs etc., described in the following subsections) have been developed based on this concept.

[1] A nonzero polynomial $P(z)$ is said to be *irreducible* if it cannot be factored into two non-constant polynomials $G(z)$ and $H(z)$ over the same field, that is, $P(z) \neq G(z) \times H(z)$. The straightforward criterion for a polynomial $P(z)$ of degree k over Galois Field $\mathbb{F}(m)$ to be irreducible is – (1) it divides the polynomial $z^{m^k} - z$ and (2) for all divisors d of k, $P(z)$ and $z^{m^d} - z$ are relatively prime. The polynomial $P(z)$ is *primitive*, if it is irreducible and $\min_{n \in \mathbb{N}} \{n | P(z) \text{ divides } z^n - 1\} = m^k - 1$. In this case, $P(z)$ has a root α in $\mathbb{F}(m^k)$ such that, $\{0, 1, \alpha, \alpha^2, \cdots, \alpha^{m^k - 2}\}$ is the entire field $\mathbb{F}(m^k)$.

A variant of MRG is the additive *lagged-Fibonacci* generators [35], which take the following form :

$$x_n = (\pm x_{n-r} \pm x_{n-s}) \pmod{2^w}$$

general form of which is a linear recurrence

$$q_0 x_n + q_1 x_{n+1} + \cdots + q_r x_{n+r} = 0 \pmod{2^w}$$

defined by a polynomial

$$Q(t) = q_0 + q_1 t + \cdots + q_r t^r$$

with integer coefficients and degree $r > 0$. Here, w is an exponent, which may be chosen according to the word length of computer. The desired random sequence is $(x_i)_{i\geq 0}$ where $x_0, \cdots, x_{r-1}$ are initially given and not all even. However, if $Q(t) = q_0 + q_s t^s + q_r t^r$ is a primitive trinomial with $r > 2$, and if q_0 and q_r are chosen as odd, the sequence $(x_i)_{i\geq 0}$ attains the maximal period of $2^{w-1}(2^r - 1)$. The PRNG, proposed in [35], uses this type of trinomials. As extension of lagged-Fibonacci generators, the PRNGs named *add-with-carry* (AWC) and *subtract-with-borrow* (SWB) generators [36], *multiply-with-carry* (MWC) generators [37, 38] were proposed.

Another type of generators named as *inversive congruential generators* (ICGs) were proposed in [39–41]. These generators are defined by the recursion

$$x_{n+1} = a {x_n}^{-1} + c \pmod{p}, \quad n \geq 0$$

where p is a large prime, x_n ranges over the set $\{0, 1, \cdots, p-1, \infty\}$ and the x_n^{-1} is the inverse of x_n, defined as: $0^{-1} = \infty$, $\infty^{-1} = 0$, otherwise $x^{-1}x \equiv 1$ (modulo p). For the purpose of implementation, one can consider $0^{-1} = 0$, as 0 is always followed by ∞ and then by c in the sequence. However, for many choices of a and c, maximum period length $p + 1$ is attainable [30].

To improve the randomness of an LCG, several techniques have been proposed. One important class of PRNGs exists which deals it by combining more than one LCG, see for example [42–45]. Several combining techniques have been suggested in the literature, like addition using integer arithmetic [42, 43], shuffling [46], bitwise addition modulo 2 [47] etc. In [44], it is shown that, we can get an MRG equivalent (or approximately equivalent) to the combined generator of two or more component MRGs, where the equivalent MRG has modulus equal to the product of the individual moduli of the component MRGs. For example, consider $J \geq 2$ component MRGs with m_js as pairwise relatively prime with period $\rho_j = m_j^{k_j} - 1$, where the j^{th} recurrence has order k_j and is shown as:

$$x_{j,n} = a_{j,1} x_{j,n-1} + \cdots a_{j,k} x_{j,n-k} \pmod{m_j}, \qquad 1 \leq j \leq J \tag{5}$$

Two combined generators can be defined, where δ_js are arbitrary integers such that each δ_j is relatively prime to m_j [44]:

$$w_n = \left(\sum_{j=1}^{J} \frac{\delta_j x_{j,n}}{m_j}\right) \pmod{1} \tag{6}$$

$$z_n = (\sum_{j=1}^{J} \delta_j x_{j,n}) \pmod{m_1}; \qquad \tilde{u}_n = \frac{z_n}{m_1}, \tag{7}$$

It is shown that, the MRGs of Equations 5 and 6 are equivalent to an MRG of Equation 3 with modulus $m = \prod_{j=1}^{J} m_j$ and period length $= lcm(\rho_1, \cdots, \rho_J)$. Similarly, the MRG of Equation 7 is approximately equivalent to the MRG of Equation 6.

Another technique of improving randomness quality of a generator is to use a randomised algorithm over the outputs of a single LCG. This randomized algorithm is an efficient permutation function or hash function in [48], which introduces the family of generators as *permuted congruential generator* or *PCG*. Here, several operations are performed on the outputs of an LCG, like random shifts to drop bits, random rotation of bits, bitwise exclusive-or(xor)-shift and modular multiplication to perturb the lattice structure inherent to LCGs and improve its randomness quality.

Sometimes, LCG can be written in a matrix form as

$$\mathbf{X_n} = \mathbf{A}\mathbf{X_{n-1}} + \mathbf{C} \pmod{m} \tag{8}$$

Here, $S = \{\mathbf{X} = (x_1, \cdots, x_k)^T | 0 \leq x_0, \cdots, x_k < m\}$ is the set of k-dimensional vectors with elements in $F = \{0, 1, \cdots, m-1\}$, $\mathbf{A} = (a_{ij})$ is a $k \times k$ matrix with elements in F, $\mathbf{C} \in S$ is a constant vector and $\mathbf{X_0}$ is the seed [27]. If $k = 1$, the recurrence of Equation 8 reduces to 1. When $\mathbf{C} = 0$, the generator is an MCG:

$$\mathbf{X_n} = \mathbf{A}\mathbf{X_{n-1}} \pmod{m} \tag{9}$$

This form is useful because of its jumping-ahead property. Even for a large v, $\mathbf{X}_{i+v}$ can be reached from $\mathbf{X}_i$, by first computing $\mathbf{A}^v \pmod{m}$ in $\mathscr{O}(\log v)$ time and applying a matrix-vector multiplication $\mathbf{X}_{i+v} = (\mathbf{A}^v \pmod{m})\mathbf{X}_i \pmod{m}$ [27]. Moreover, using this matrix, any LCG of order k can be expressed by an MCG of oder $k+1$: modify $\mathbf{A}$ to add $\mathbf{C}$ as its $(k+1)^{th}$ column and a $(k+1)^{th}$ line containing all 0s except 1 in $(k+1)^{th}$ position; modify $\mathbf{X_n}$ to add 1 as its $(k+1)^{th}$ component. When m is prime and $\mathbf{C} = 0$, F and S are equivalent to $\mathbb{F}(m)$ and $\mathbb{F}(m^k)$, where $\mathbb{F}(m^k)$ is the Galois field with m^k elements. In this case, $\mathbf{X_n}$'s have maximal possible period $= m^k - 1$ if and only if the characteristic polynomial of $\mathbf{A}$,

$$f(x) = |xI - \mathbf{A}| \pmod{m} = (x^k - \sum_{i=1}^{k} a_i x^{k-i}) \pmod{m} \tag{10}$$

with coefficients a_i in $\mathbb{F}(m)$ is a primitive modulo m. For attaining this period, $\mathbf{A}$ must be nonsingular in arithmetic modulo m. Nevertheless, a polynomial of Equation 10 has a companion matrix $\mathbf{A}$:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{bmatrix} \tag{11}$$

In this case, by taking $\mathbf{X_n} = (x_n, \cdots, x_{n-k+1})^T$, MCG of Equation 9 is converted to recurrence of MRG (Equation 3), where $\mathbf{X_n}$ obeys the recursion:

$$\mathbf{X}_n = a_1\mathbf{X}_{n-1} + \cdots a_k\mathbf{X}_{n-k} \pmod m$$

In this work, however, we have selected 9 LCG based PRNGs. Following are these generators along with their parameters.

- **Knuth's LCG MMIX [30] :** For this LCG (Equation 1), the values of the modulus, multiplier and increment are as defined as:

  $$a = 6364136223846793005,\ m = 2^{64},\ c = 1442695040888963407$$

  The period of the PRNG is 2^{64} and output numbers are 64-bit normalized numbers.

- **rand() in GNU C Library [49] :** This is the most common PRNG used in programming. Here, the multiplier $a = 1103515245$, increment $c = 12345$ and the modulus $m = 2^{31}$. C program for this PRNG is part of ISO C standard library. The range of numbers generated by rand() is $[0, RAND\_MAX)$, where RAND_MAX is usually defined to be at least 32767.

- **lrand48() in GNU C Library [49] :** It returns non-negative 32-bit integers, uniformly distributed over the interval $[0, 2^{31}]$. In case of drand48(), the returned numbers are double-precision floating-point values between 0.0 to 1.0. In both cases, the multiplier $a = 25214903917$, increment $c = 11$ and the modulus is 2^{48} following Equation 1. This PRNG is part of C library on SVID systems.

- **C++11's minstd_rand [19, 50] :** For this LCG, $a = 48271$, $m = 2^{31} - 1$ and $c = 0$. So, it is actually an MCG (see Equation 2).

- **Borland LCG :** Here, $a = 22695477$, $c = 1$ and $m = 2^{32}$ following Equation 1.

- **MRG31k3p [51] :** It is a combined MRG consisting of 2 component MRGs of order $k = 3$ (see Equation 5 for component MRGs and Equation 7 for the combined MRG). Its period length is approximately 2^{185}. The parameters for the component MRGS are

  $$m_1 = 2^{31} - 1,\ a_{11} = 0,\ a_{12} = 2^{22},\ a_{13} = 2^7 + 1$$

  $$m_2 = 2^{31} - 21069,\ a_{21} = 2^{15},\ a_{22} = 0,\ a_{23} = 2^{15} + 1$$

  where m_i is the individual modulus and a_{ij} are the coefficients. Here, each component has two non-zero coefficients of the form 2^q and $2^q + 1$ for ease of implementation.

- **PCG [48] :** Here the output of a fast LCG (Equation 1) is passed to a permutation function for enhancing the output quality. The permutation operations applied to the output of LCGs is based on xorshift and random rotation of some bits. The output of this generator can be 32-bit or 64-bit. Here, we have tested PCG-32 bit only, which has a period length 2^{64}. The multiplier is 6364136223846793005 and increment is taken as 1.

III.2. LFSR based PRNGs

If modulus of the linear recurrence (Equation 1) $m = 2$ and $c = 0$, the linear recurrence is based on the Galois field $\mathbb{F}(2)$. These recurrences can be implemented on a linear feedback shift register (LFSR). A LFSR is a shift register where the output of some bit positions are xor-ed and feed as input to the register. This feedback connection ensures that the register cycles endlessly through repetitive sequences of values. To implement an LFSR in hardware, k number of memory elements (flip-flops) are connected via XOR gates (see Figure 2). The positions of xor in LFSR determines the characteristic polynomial of the LFSR, whereas the number of flip-flops (k) determines the degree of the polynomial. If flip-flop (FF) i is associated with a feedback connection, coefficient of x^i is 1 for the characteristic polynomial $P(x)$. If this characteristic polynomial is primitive over $\mathbb{F}_2$, a k-bit LFSR can generate a maximal length sequence of period $2^k - 1$, where k is the degree of the polynomial. Likewise MCGs, seed of a LFSR should always be a non-zero value, otherwise, the sequence degrades to zeros.

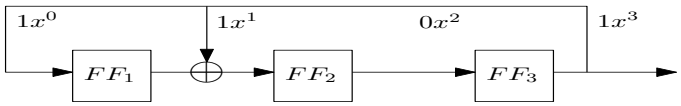


Figure 2: A schematic diagram of 3-bit LFSR with characteristic polynomial $P(x) = 1+x+x^3$

As the generated numbers are binary, elementary bit string operations like rotation, shift, mask, exclusive-or etc. can be applied on them efficiently on a computer. The advantage of using this scheme is, LFSRs can be implemented on hardware; therefore, the generated circuits can be fast, cost-effective and efficient in terms of computational overhead. For many applications demanding PRNG, like VLSI testing, pattern recognition, computer simulation etc., efficient hardware implementation of the PRNG with very low overhead is a basic requirement. For this reason, most of today's research on PRNG is directed towards these PRNGs. Many variations of this scheme are proposed, like Tausworthe generator, generalized linear feedback shift register (GFSR), twisted GFSR (TGFSR), Mersenne Twister, xorshift generators, WELL etc.

Tausworthe generator [13] is a linear recurrence of order $k > 1$ like Equation 3 where $m = 2$, defined by the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \pmod 2 \tag{12}$$

Here, $a_k = 1$ and $\forall i, a_i \in \mathbb{F}_2$. The random number is represented by

$$u_n = \sum_{l=1}^{L} x_{ns+l-1} 2^{-l} \tag{13}$$

where s, L are positive integers. The random number is represented by u_n, which is a number with L consecutive bit sequence of recurrence 12; with successive u_ns spaced s bits apart [13]. This PRNG can have a maximal period $\rho = 2^k - 1$, if and only if, the characteristic polynomial

$$P(z) = 1 + a_1 z + a_2 z^2 + \cdots + z^k \tag{14}$$

is primitive over $\mathbb{F}(2)$ and s is relatively prime to $2^k - 1$. Then the generated sequence is called *maximal-length linearly recurring sequence modulo* 2.

Initially LFSR-based Tausworthe generators used primitive trinomials [13, 52]. In [53], it is shown that, any Tausworthe generator that uses primitive trinomials of form

$$P(z) = z^p + z^q + 1 \quad (1 \leq q \leq (p-1)/2) \tag{15}$$

as the characteristic polynomial can be represented by a simple linear recurrence in $\mathbb{F}(2^p)$. Moreover, likewise combined MRGs, *combined* Tauseworthe generators have been proposed [6]. It consists of $J \geq 2$ Tausworthe generators with primitive characteristic polynomials $P_j(z)$ of degree k_j with $s = s_j$ as mutually prime to $2^{k_j} - 1$, $(1 \leq j \leq J)$. The sequence is denoted by $x_{j,n}$ (see Equation 5 with modulus 2) and random number by $u_{j,n} = \sum_{l=1}^{L} x_{j,ns_j+l-1} 2^{-l}$. The output of the combined generator is

$$u_n = (u_{1,n} \oplus u_{2,n} \oplus \cdots \oplus u_{J,n})$$

where $\oplus$ is the bitwise exclusive-or operation. As discussed in Section III.1, this generator has period $\rho = lcm(2^{k_1} - 1, 2^{k_2} - 1, \cdots, 2^{k_J} - 1)$, if the polynomial $P_j(z)$s are pairwise relatively prime, that is, every pair of polynomials have no common factor.

In [7], a new class of LFSR based PRNG, named generalized linear feedback shift register or GFSR was introduced. A GFSR sequence can be represented in binary as

$$X_n = x_{j_1+n-1} x_{j_2+n-1} \cdots x_{j_k+n-1} \tag{16}$$

where X_n is a sequence of k-bit integers and x_i is a LFSR sequence of Equation 12. These GFSR sequences can also be represented with the help of a companion matrix $\mathbf{D}$ of the characteristic polynomial of Equation 14:

$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & a_{k-1} & a_{k-2} & \cdots & a_1 \end{bmatrix} \tag{17}$$

Now, if $\alpha = (x_1, \cdots x_k)^T$, $\beta = (x_{j_1}, \cdots, x_{j-k})^T$, then a matrix $\mathbf{G}$ can be defined where $\beta = \mathbf{G}\alpha$. Therefore, a k-bit GFSR sequence is represented by the following [14]:

$$\mathbf{G}\alpha, \mathbf{GD}\alpha, \mathbf{GD^2}\alpha, ..., \mathbf{GD^n}\alpha, ...$$

If $P(z)$ is a primitive trinomial of Equation 15, the GFSR sequence is depicted by a recurrence $X_n = X_{n-p+q} \oplus X_{n-p}$, for $n = p, p+1, \cdots$, where $X_1, \cdots, X_k$ are a set of seeds and $p = k$ with period length $2^p - 1$.

However, this generator fails to reach its theoretical upper bound on period (equal to number of possible states) and has large memory requirement. So, another variation, named twisted GFSR (TGFSR), was proposed in [54, 55]. This generator is same as GFSR, however, its linear recurrence is

$$\mathbf{X}_{l+n} = \mathbf{X}_{l+m} \oplus \mathbf{X}_l\mathbf{A}, \quad (l = 0, 1, \cdots) \tag{18}$$

where $\mathbf{A}$ is a $w \times w$ matrix over $\mathbb{F}(2)$, n, m, w are positive integers with $n > m$ and $\mathbf{X}_i$s are vectors in $\mathbb{F}(2^w)$. The seed is the tuple $(\mathbf{X}_0, \mathbf{X}_1, \cdots, \mathbf{X}_{n-1})$ with at least one non-zero value. Usually, matrix $\mathbf{A}$ is chosen as 11. Therefore, a TGFSR sequence is denoted by $\mathcal{X}(n, m, \mathbf{A})$= $\mathbf{X}_0, \mathbf{X}_1, \cdots, \mathbf{X}_{n-1}$. This generator has a maximal period of $2^{nw} - 1$ if and only if $P(z^n + z^m)$ is a primitive polynomial of degree nw where $P(z)$ is the characteristic polynomial of matrix $\mathbf{A}$. However, this polynomial $P(z^n + z^m)$ is primitive if and only if the polynomial $P(z)$ is irreducible (that is, it has no divisor other than 1 and itself) and $(z^n + z^m + \eta)$ is primitive over $\mathbb{F}(2^w)$, where η is a root of $P(x)$ of degree w with coefficients in $\mathbb{F}(2)$. In that case, the generated sequence is called a *maximal TGFSR* (*m*-TFGR) sequence.

Likewise LCGs, all LFSR based linear recurrence modulo 2 generators can be represented in the following matrix form:

$$\mathbf{X_n} = \mathbf{A}\mathbf{X}_{n-1} \tag{19}$$

$$\mathbf{Y_n} = \mathbf{B}\mathbf{X}_n \tag{20}$$

$$u_n = \sum_{l=1}^{w} y_{n,l-1} 2^{-l} \tag{21}$$

Here, $k, w > 0$, $\mathbf{A}$ is a $k \times k$ matrix, called transition matrix, $\mathbf{B}$ is a $w \times k$ matrix, called output transformation matrix and elements of $\mathbf{A}, \mathbf{B}$ are in $\mathbb{F}_2$. The k-bit state vector at step n is $\mathbf{X}_n = (x_{n,0}, \cdots, x_{n,k-1})^T$, the w-bit output vector is $\mathbf{Y}_n = (y_{n,0}, \cdots, y_{n,k-1})^T$ and output at step i is $u_n \in [0, 1)$. All the operations in equations 19 and 20 are modulo 2 operations. The characteristic polynomial of matrix $\mathbf{A}$ is same as Equation 10 with modulus 2:

$$P(z) = det(z\mathbf{I} - \mathbf{A}) = (z^k - \sum_{i=1}^{k} a_i z^{k-i}) \tag{22}$$

where $a_j \in \mathbb{F}_2$ and $\mathbf{I}$ is the identity matrix. If $a_k = 1$, this recurrence is purely periodic with order k. The period of $\mathbf{X}_n$ is maximal, that is, $2^k - 1$, if and

only if, $P(z)$ is a primitive polynomial in $\mathbb{F}_2$. In this way, these PRNGs can be portrayed as LCGs in polynomials over $\mathbb{F}_2$.

Note that, matrix $\mathbf{B}$ is usually used for tempering [55], that is, to improve equidistribution property of the PRNG by elementary bitwise transformation operations, like exclusive-or, AND and shift. A TGFSR with tempering operations is called *tempered* TGFSR. The well-known PRNG *Mersenne Twister* (MT) is a variation of TGFSR where the linear recurrence is [8] :

$$\mathbf{X}_{k+n} = \mathbf{X}_{k+m} \oplus (\mathbf{X}_k^u | \mathbf{X}_{k+1}^l)\mathbf{A} \tag{23}$$

Here, n is the degree of recurrence, r, m, w are positive integers with $0 \leq r \leq w-1$, $1 \leq m \leq n$ where m is middle term and r is separation point of one word. $\mathbf{A}$ is a $w \times w$ matrix (like [11]) with entries in $\mathbb{F}(2)$, $|$ denotes bit vector wise concatenation operation, $\mathbf{X}_k^u$ is the upper $w-r$ bits of $\mathbf{X}_k$, and $\mathbf{X}_{k+1}^l$ is the lower r bits of $\mathbf{X}_{k+1}$. However, $\mathbf{X}_0, \mathbf{X}_1, \cdots, \mathbf{X}_{n-1}$ are taken as seeds. The generator generates $\mathbf{X}_n$ with $k = 0$. If $r = 0$, this recurrence reduces to TGFSR and if $r = 0$ and $\mathbf{A} = \mathbf{I}$, it reduces to GFSR [8]. Tampering is done by the following transformations in succession:

$$\mathbf{y} = \mathbf{x} \oplus (\mathbf{x} >> u)$$

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} << s) \text{ AND } \mathbf{b})$$

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} << t) \text{ AND } \mathbf{c})$$

$$\mathbf{z} = \mathbf{y} \oplus (\mathbf{y} >> l)$$

where l, s, t, u are integers called tempering parameters, $\mathbf{b}$ and $\mathbf{c}$ are suitable bitmasks of size w and $\mathbf{z}$ is the returned vector. The vector space for MT is an incomplete array of size $p = nw - r$ or an $(n \times w - r)$ array with r bits missing at the upper right corner. The state transition is directed by a linear transformation $\mathbf{B}$ on this incomplete array (see Figure 3), where $\mathbf{X}_n$ is defined

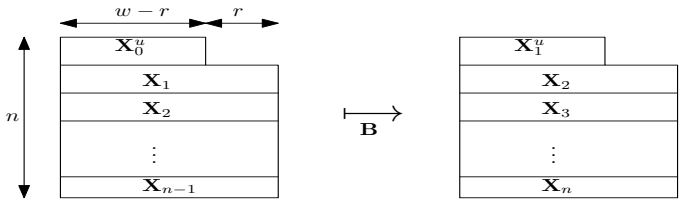


Figure 3: State transition of a Mersenne Twister

by the recursion 23 and

$$\mathbf{B} = \begin{bmatrix} 0 & \mathbf{I}_w & 0 & 0 & & & \\ 0 & 0 & \mathbf{I}_w & 0 & & & \\ \vdots & & \ddots & & & & \\ 0 & & & & & & \\ \mathbf{I}_w & & & & & & \\ 0 & & & & & & \\ \vdots & & & & \ddots & & \\ 0 & & & & 0 & \mathbf{I}_w & 0 \\ 0 & & & & 0 & 0 & \mathbf{I}_{w-r} \\ \mathbf{S} & & & & 0 & 0 & 0 \end{bmatrix}$$

Here, $\mathbf{I}_j$ is a $j \times j$ identity matrix, $\mathbf{0}$ is the zero matrix and

$$\mathbf{S} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_r \\ \mathbf{I}_{w-r} & \mathbf{0} \end{bmatrix} \mathbf{A} \tag{24}$$

The generated numbers are integers between 0 and $2^w - 1$ provided p is chosen as a Mersenne exponent such that, the characteristic polynomial of $\mathbf{B}$ is primitive and period is a Mersenne prime $2^p - 1 = 2^{nw-r} - 1$.

Another PRNG, named *well-equidistributed long-period linear* generator or WELL is also based on tampered TGFSR [10]. For this PRNG, the characteristic polynomial of matrix $\mathbf{A}$ has degree $k = rw - j$, where $r > 0$ and $0 \leq j < w$, and it is primitive over $\mathbb{F}_2$. In [56], Marsagila proposed a very fast PRNG, named *xorshift* generator. The basic concept of such generators is – to get a random number, first shift a positions of a block of bits and then apply exclusive-or on the original block with this shifted block. In general, a xorshift generator has the following recurrence relation [9, 57]:

$$\mathbf{v}_n = \sum_{j=1}^{t} \tilde{\mathbf{A}}_\mathbf{j} \mathbf{v}_{n-m_j} \pmod 2 \tag{25}$$

where $t, m_j > 0$, for each n, $\mathbf{v}_n$ is a w-bit vector and $\tilde{\mathbf{A}}_j$ is either $\mathbf{I}$ or product of v_j xorshift matrices for $v_j \geq 0$. At step n, the state of the PRNG is $\mathbf{x}_n = (\mathbf{v}_{n-r+1}^T, \cdots, \mathbf{v}_n^T)^T$ where $\mathbf{v}_n = (v_{n,0}, \cdots, v_{n,w-1})^T$ and output is $u_n = \sum_{l=1}^{w} v_{n,l-1} 2^{-l}$. This generator converts into the general LFSR PRNG of Equations 19 and 20, if

$$\mathbf{A} = \begin{bmatrix} 0 & \mathbf{I} & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{I} \\ \mathbf{A}_r & \mathbf{A}_{r-1} & \cdots & \mathbf{A}_1 \end{bmatrix} \tag{26}$$

where $k = rw$, $\mathbf{y}_n = \mathbf{v}_n$ and $\mathbf{B}$ matrix has $\mathbf{I}$ matrix of size $w \times w$ in upper left

corner with zeros elsewhere. **A** matrix has characteristic polynomial of the form

$$P(z) = det(z^r \mathbf{I} + \sum_{j=1}^{r} z^{r-j} \mathbf{A}_j)$$

Therefore, the generator has maximal period length of $2^{rw} - 1$, if and only if, this polynomial $P(z)$ is primitive.

Although these generators are linear, many researchers have developed LFSR based PRNGs by combining these with some non-linear operations [17, 25, 58] to scramble the regularity of linear recurrence. For example, in [58], two component combined generators are proposed, where the major component is linear (LFSR or LCG), but the second component is distinct (nonlinear or linear). Whereas, in [17], to remove the flaws of xorshift generators, a non-linear operation is applied to scramble the results. As the non-linear operation, very simple operators are chosen. For instance, in *xorshift** PRNG, the nonlinear operation is multiplication by a constant, so, the PRNG have total 8 logical operators, 1 addition and 1 multiplication.

In this work, however, we have taken the following well-known LFSR-based PRNGs and studied them empirically.

- **random() in GNU C Library [49] :** This is LFSR based PRNG in GCC standard library derived from BSD (like Equation 12). It returns numbers between 0 to 2147483647 and its period is $\rho \approx 16 \times (2^{31} - 1)$.

- **Taus88 [6] :** This is a combined Tausworthe generator where number of component PRNGs (follow Equation 5 with $m = 2$) $J = 3$, with order $k_1 = 31$, $k_2 = 29$, and $k_3 = 28$ respectively. This PRNG has period length $\rho = (2^{31} - 1)(2^{29} - 1)(2^{28} - 1) \approx 2^{88}$. The C code for this PRNG is taken from *https : //github.com/LuaDist/gsl/blob/master/rng/taus.c* which returns either 32-bit unsigned integer or its normalized version.

- **LFSR113 [59] :** This is also a combined Tausworthe generator where number of component PRNGs (see Equation 5 with $m = 2$) $J = 4$, with period length $\rho \approx 2^{113}$. The C code is downloaded from [60] which returns a 64 bit normalized number by multiplying the unsigned long integer output of the LFSR with $2.3283064365387 \times 10^{-10}$.

- **LFSR258 [59] :** This is another combined Tausworthe generator (Equation 5 with $m = 2$) having $J = 5$ and period length $\rho \approx 2^{258}$. Here, the 64 bit normalized random number is generated by multiplying the unsigned 64-bit output of the LFSR with $5.42101086242752217003726 4 \times 10^{-20}$. C code for this PRNG is also downloaded from [60].

- **WELL [10] :** We have tested two WELL PRNGs, namely *WELL512a* and *WELL1024a* [60]. In case of WELL512 PRNG, the parameters are $k = 512$, $w = 32$, $n = 16$ and $r = 0$; so expected period is $\rho = 2^{512} - 1$. However, for WELL1024a, the parameters are $k = 1024, w = 32, n = 32$

and $r = 0$ with period length $\rho = 2^{1024} - 1$. The return values for both WELL512a and WELL1024a are 32-bit numbers normalized by multiplying with $2.32830643653869628906 \times 10^{-10}$. WELL generators follows the general equations of LFSR based PRNGs (Equation 19 and Equation 20).

- **XORSHIFT PRNGs [61] :** Four types of xorshift generators (see Equation 25) have been tested – Marsagila's xorshift32 generator, xorshift64* generator, xorshift1024*M_8 and xorshift128+ generator. In Marsagila's xorshift32 generator [56], 3 xorshift operations are performed, first XOR with left shift of 13 bits, then with right shift of 17 bits and finally again XOR with left shift of 15bits. Here the returned number is a 32 bit unsigned integer.

  In *xorshift64** generator, the returned number is current state perturbed by a non-linear operation, which is multiplication by 2685821657 736338717 [17]. Here also 3 xorshifts are performed – left with 12 bits, right with 25 bits and again left with 27 bits. However, in *xorshift1024**M_8 PRNG, the multiplier is 1181783497276652981 and shift parameters are $31, 11$, and 30. In both cases, the generated numbers are 64-bit unsigned integers.

  Further, in *xorshift+* generators, the returned number is sum of some previous consecutive xorshift outputs [62]. In *xorshift128+* generator, the outputs are 64-bits and two previous output states are added to get the result.

- **MT19937 [8]** This is a Mersenne Twister (Equation 23), based on tempering on a twisted GFSR and has a period $\rho = 2^{19937} - 1$. There are 32-bit and 64-bit word size variations of it. In case of MT19937 (32-bit), the associated parameters are $(w, n, m, r) = (32, 624, 397, 31)$, $\mathbf{a} = 9908B0DF$, $u = 11$, $s = 7$, $\mathbf{b} = 9D2C5680$, $t = 15$, $\mathbf{c} = EFC60000$, $l = 18$ and number of terms in the characteristics polynomial is 135. However, for MT19937 (64-bit), $(w, n, m, r) = (64, 312, 156, 31)$, $\mathbf{a} = B5026F5AA96619E916$, $u = 29$, $s = 17$, $\mathbf{b} = 71D67FFFEDA6000016$, $t = 37$, $\mathbf{c} = FFF7EEE000000000016$ and $l = 43$.

- **SMFT [11, 63]** SFMT stands for single instruction multiple data (SIMD)-oriented Fast Mersenne Twister. It is a LFSR PRNG that uses all features of MT along with multi-stage pipelines and Single Instruction Multiple Data (SIMD) (like 128-bit integer) operations of today's computer system. Its period length is same as MT19937. It can generate both 32-bit and 64-bit unsigned integer numbers, as well as double precision floating point numbers. However, in our empirical study, we have not used the floating point numbers for testing.

- **dSFMT[12]** dSFMT stands for double precision floating point SFMT. This is a variation of SFMT, specialized in producing double precision floating point numbers in IEEE 754 format. The output of this PRNG is a sequence of 52-bit pseudo-random patterns along with 12 MSBs (sign and

exponent) as constant. Here, instead of linear transition in $\mathbb{F}_2$, an affine transition function is adopted which keeps the constant part as $0x3FF$. However, in the C code [64], both 32-bit unsigned integer and double precision floating point output versions are available.

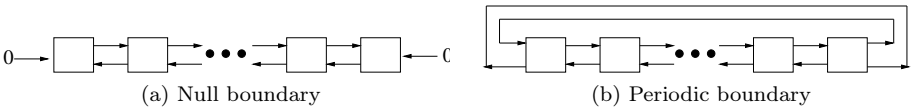
III.3. Cellular Automata based PRNGs

A cellular automaton (CA) is a discrete dynamical system comprising of a regular network of cells, where each cell is a finite state automaton. During evolution, a cell of a CA updates its state depending on the present states of its neighbors following a *next state function*, also known as *local rule*, or simply *rule*, whose arguments are the present states of the cell's neighbors. Therefore, a CA is identified by a quadruple $(\mathscr{L}, \mathcal{S}, \mathcal{N}, \mathcal{R})$, where $\mathscr{L} \subseteq \mathbb{Z}^D$ is the D-dimensional cellular space, $\mathcal{S}$ is the finite set of states which a cell can take, $\mathcal{N} = (\overrightarrow{v_1}, \overrightarrow{v_2}, \cdots, \overrightarrow{v_N})$ identifies N neighbors of each cell and $\mathcal{R} : \mathcal{S}^N \to \mathcal{S}$ is the rule of the automaton.

Collection of the states of all cells at a given time is called *configuration* of the CA. If $\mathcal{C}$ represents $\mathcal{S}^{\mathscr{L}}$, the set of all configurations, then, a CA is a function $G : \mathcal{C} \to \mathcal{C}$, which is called *global transition function*. Therefore, if a configuration $y = (y_{\overrightarrow{v}})_{\overrightarrow{v} \in \mathscr{L}}$ is successor of another configuration $x = (x_{\overrightarrow{v}})_{\overrightarrow{v} \in \mathscr{L}}$, that is, $y = G(x)$, then y is the result of following application: For each $\overrightarrow{v} \in \mathscr{L}$

$$y_{\overrightarrow{v}} = G(x)_{\overrightarrow{v}} = G(x_{\overrightarrow{v}}) = \mathcal{R}(x_{\overrightarrow{v}+\overrightarrow{v_1}}, x_{\overrightarrow{v}+\overrightarrow{v_2}}, \cdots, x_{\overrightarrow{v}+\overrightarrow{v_N}}) \tag{27}$$

Classically, a CA has infinite lattice where each cell follows the same local rule. However, for the purpose of simulation on a computer and real-life applications, this definition of CA has been abused – $\mathscr{L}$ is considered as finite (that is, with fixed number of cells n) having boundaries. Two boundary conditions are usually considered – (1) null boundary, where the boundary cells are connected to null or 0 state, (2) periodic boundary, where the boundary cells are neighbors of each other. Figure 4a and Figure 4b depict null boundary and periodic boundary conditions for 1-dimensional cellular automata (CAs) with 3-neighborhood dependency.



(a) Null boundary (b) Periodic boundary

Figure 4: Boundary conditions of 1-D finite CAs. Arrows pointing to a cell indicate the dependencies of the cell. Here all the CAs use 3-neighborhood dependency

The rules of the CAs can also be represented by a tabular form (see Table 1). The table has an entry for each combination of the neighborhoods. In case of 1-dimensional 3-neighborhood 2-state CAs (called, *elementary CAs* or *ECAs*), this rule is represented by the decimal equivalent of the binary string of the neighborhood combinations $\mathcal{R}(x, y, z)$, $x, y, z \in \{0, 1\}$, where $\mathcal{R}$ is a rule. Table 1 shows 3 local rules of ECAs, where the rule numbers of individual rules are shown

Table 1: Some rules of 1-dimensional 3-neighborhood 2-state CAs.

Neighborhood Combination	111	110	101	100	011	010	001	000	Rule Number
Next State	0	0	0	1	1	1	1	0	30
	0	0	1	0	1	1	0	1	45
	0	1	0	1	1	0	1	0	90
	1	0	0	1	0	1	1	0	150

in the last column. This notation of rules is very popular for 1-dimensional binary CAs with 3-neighborhood dependency.

The most exciting aspect of CAs is their complex global behavior, which is resulted from simple local interaction and computation and massive parallelism. Another important property of CAs is, like LFSR, CAs can be easily implemented in hardware – each cell consists of a memory element to store its state and a combinational logic circuit to find the next state of the cell. Figure 5 represents hardware implementation of an ECA with n cells under null boundary condition. These properties of CAs along with ease of scalability have made CAs, especially ECAs, an area of extensive research for applications like VLSI circuit testing [65–70], Monte-Carlo simulations [71], Field Programmable Gate Arrays (FPGAs) [72], cryptography [73, 74] etc.

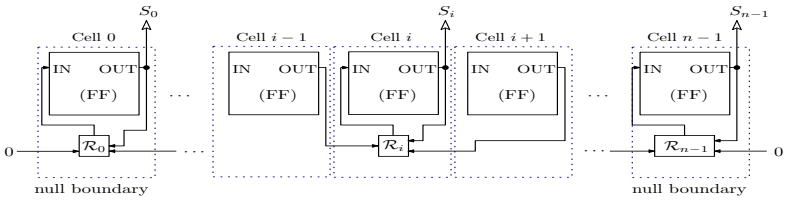


Figure 5: Hardware implementation of an n-cell non-uniform ECA under null boundary condition

In [29], CAs were introduced as a source of pseudo-randomness. Here, an ECA with infinite number of cells is considered, where each cell follows rule 30 of Table 1. In this case, a random sequence is generated using the next state values of the single cell with initial state 1 among all cells, initiated with state 0. Nevertheless, to use CA as a PRNG, generally an integer X_i is generated between zero and some number w (word size of the computer), where the fraction $U_i = \frac{X_i}{w}$ is the real number, uniformly distributed between 0 and 1. There have been many ways to generate these numbers. For instance, in [65], two ECAs with rule 30 and rule 45 are considered to generate random numbers from the whole configuration of the CA. Here, concept of site spacing (output number is collected from cells spaced by γ distance) and time spacing (output numbers are taken α clock pulses apart) are introduced (see Figure 6 and Figure 7). If $\gamma = 0$, the whole configuration of the CA is treated as a number. In a recent work, the numbers are generated by a small window of cells using a 3-state 3-neighborhood 1-D CA under periodic boundary condition [75]. The base-3 numbers, observed through the window, are considered the pseudo-random numbers (Figure 8).

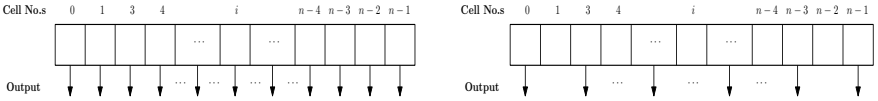


Figure 6: Site Spacing for even cell length n. Here, $\gamma = 0$ for Figure 6a and $\gamma = 1$ for Figure 6b. The random numbers are collected from the cells with arrows.

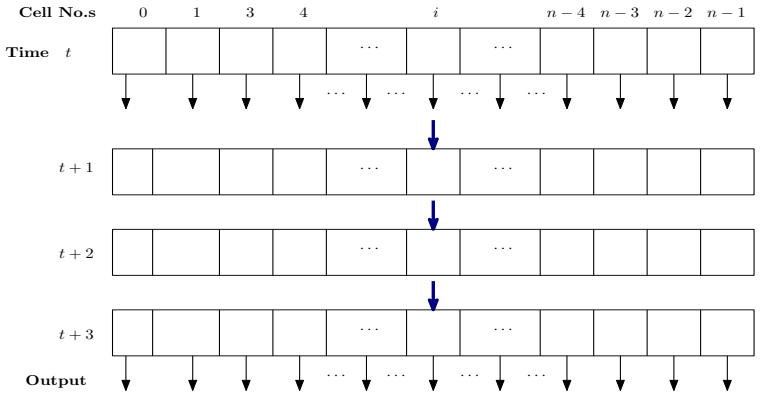


Figure 7: Time spacing with no site spacing. Here, α is taken as 2.

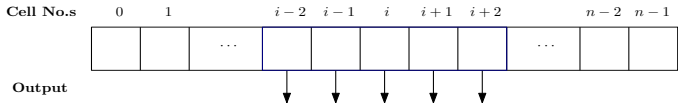


Figure 8: Window of size 5 taken from the middle cells.

Randomness of a CA-based PRNG is, in general, effected by its transition rule, cell size, seed and boundary condition. So, research on CA-based PRNGs have been to find the best possible result by varying these structures of the CAs. The rule of the CA is chosen as *autopletic*, that is, even simple initial conditions can derive pseudo-randomness. For instance, ECA rules 30, 45 are autopletic rules. However, none of these CAs can generate all numbers in a single cycle. Therefore, to improve the cycle length of the CA and introduce more complexity in the system, non-uniformity in the local rule is instigated [76]. In this case, the cells of the finite CA may take different rules. That means, instead of using a single next state function $\mathcal{R} : \mathcal{S}^N \rightarrow \mathcal{S}$, a vector $\mathscr{R} = \langle \mathcal{R}_0, \mathcal{R}_1, \cdots, \mathcal{R}_{n-1} \rangle$, called *rule vector*, is used, where n is the number of cells and the i^{th} cell uses rule $\mathcal{R}_i : \mathcal{S}^N \rightarrow \mathcal{S}$. The next state calculation in this case, is governed by $\mathcal{R}_i$ for each i. The situation for a 1-dimensional n-cell CA is shown in Figure 9, where each cell i depends on cell its previous cell, next cell and itself. We can imagine a sliding window which moves over the neighbors and sends arguments to each local rule $\mathcal{R}_i$ to calculate next state of the i^{th} cell. Such a CA with different rules for different cells is called a *hybrid*

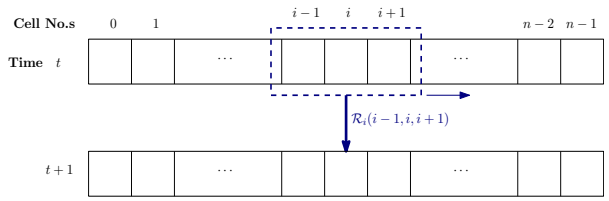


Figure 9: Next state calculation of each cell of an n-cell hybrid CA with 3 neighborhood dependency

or *non-uniform* CA. If $\mathcal{R}_0 = \mathcal{R}_1 = \cdots = \mathcal{R}_{n-1}$, the CA is uniform CA. In Figure 5, if combinational logic circuits for each cells are different, the ECA is a hybrid ECA.

However, if all rules of the CA can be expressed by a linear function, that is,

$$\forall i,\ \mathcal{R}_i(a_1, a_2, \cdots, a_N) = \sum_{j=1}^{N} c_j.a_j$$

where $c_j \in \mathcal{S}$ is a constant and a_j is the state of the j^{th} neighbor of cell i, then the CA is called a *linear* CA. In this case, the set $\mathcal{S}$ forms a commutative ring with identity. For example, the ECA 90 and 150, which are linear CAs, can be represented as:

$$90 : S_i(t+1) \ = \ S_{i-1}(t) \oplus S_{i+1}(t)$$

$$150 : S_i(t+1) \ = \ S_{i-1}(t) \oplus S_i(t) \oplus S_{i+1}(t)$$

where $S_i(t)$ is the state of i^{th} cell at time t. Moreover, a binary n-cell CA can be represented by an $n \times n$ characteristics matrix (T) operating on $GF(2)$ In this

matrix, the i^{th} row represents the dependency of the i^{th} cell to its neighbors. The characteristics matrix (T), in this case, is formed as:

$$T[i,j] = \begin{cases} 1 \text{ if the next state of the } i^{th} \text{ cell depends on the present state of the } j^{th} \text{ cell} \\ 0 \text{ otherwise} \end{cases} \tag{28}$$

For example, the characteristics matrix of a 4-cell hybrid CA with rule vector $\mathcal{R} = \langle 150, 150, 90, 150 \rangle$ under null boundary condition is:

$$T = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Here, as boundary condition is null, left neighbor of first cell and right neighbor of last cell are 0. For any one-dimensional linear CA using 3-neighborhood condition, this matrix is tridiagonal [77]. Now, if the characteristic polynomial of this T matrix is primitive over $\mathbb{F}_2$, the CA can generate maximal cycle length $2^n - 1$. Such CAs are called *maximal-length CAs*.

Several researches have been conducted to establish the isomorphism of a 1-dimensional linear hybrid CA with its corresponding LFSR, where both have the same primitive characteristic polynomial [77, 78]. It is shown that, for every irreducible polynomial, there are exactly two hybrid CAs with rules $90, 150$ under null boundary condition; the construction process of such a CA is shown in [79]. In [80], a list of maximal-length CAs for each degree from 1 to 500 is synthesized for the corresponding primitive polynomials given in [81, 82]. Needless to say, only specific combinations of the local rules 90 and 150 over null boundary condition, can generate a maximal-length CA. Due to this maximal cycle length property, many researchers have used these CAs as their generators [65, 70, 78, 83–85].

However, due to difficulty in finding the primitive polynomial required for a maximal-length CA, non-linear CAs were introduced as PRNGs [68, 69]. For example, in [68], an algorithm is given to select a non-uniform non-linear CA as the random number generator. Some other works of using hybrid CAs are [65, 70, 86]. In [86], cells of the CAs were allowed to hold memory of their last two state values. Here, numbers were taken from overlapping window of size 50 and the CAs are with rules 30, 90 or 150.

Sometimes, optimization techniques are applied to the CAs, to improve their randomness qualities. In [16, 87], genetic algorithms are applied to co-evolve hybrid CAs for generating random numbers. For example, in [87], a CA of size 50, where the first 22 cells have rule 165, next 22 cells have rule 90 and last 6 cells have rule 150 is used as PRNG. In [88], numbers are generated using evolutionary multiobjective optimization techniques on controllable CA, whereas, in [89], self-programmable CA is used. In a controllable CA, the update of some cells is controlled via some control signals, while in programmable CA, spatial and temporal variations are allowed in the CA rules using some external control scheme. In [83, 90–92], 2-dimensional CAs are used as the PRNGs. For example, in [92],

2-state periodic boundary CA with the rules $165, 105, 90, 150, 153, 101, 30, 86$ is combined with Langton's ants to generate the numbers. Here, Langton's ant is a simple 2-dimensional Turing machine with complex behavior [93].

In this work, however, we have studied the following 1-dimensional CAs-based PRNGs.

- **Rule** 30 **CA [15] :** In [15], an infinite cell elementary CA rule 30 is used as a PRNG. However, for our testbed, that is practically infeasible, so, we have to compromise by taking smaller cell length with periodic boundary condition. Here, the cell length is taken as 101 and next states of the middle cells are collected to generate 32-bit numbers.

- **Hybrid** $30 - 45$ **CA [65] :** This PRNG uses a rule vector $\mathscr{R} = \{30, 45\}^{16}$ with periodic boundary condition, to generate 32-bit numbers.

- **Maximal-length CA [65] :** Here, a maximal length CA with rule vector $\mathscr{R} = \{90, 150, 90, 90, 90, 150, 150, 90, 90, 90, 90, 90, 150, 90, 90, 150, 150, 90, 150, 150, 150, 90, 150, 150, 150, 150, 90, 150, 90, 150, 90, 150\}$ for null boundary condition is used. We have considered both the cases – with 1 site spacing and with no site spacing. So, for site spacing $\gamma = 1$, two consecutive output sequences are concatenated as one 32-bit number, however, for no site spacing, the whole to generate configuration at each time instant is treated as a 32-bit number.

- **Non-linear 2-state CA [68] :** Here, using the given algorithm, a 45-cell null boundary 2-state 3-neighborhood non-uniform CA is generated. For example, one such CA has rule vector $\mathscr{R} = \{5, 105, 90, 90, 165, 150, 90, 105, 150, 105, 90, 165, 150, 150, 165, 90, 165, 90, 165, 150, 150, 90, 165, 105, 90, 165, 150, 90, 105, 150, 165, 90, 105, 105, 90, 150, 90, 90, 165, 150, 150, 105, 90, 165, 20\}$. Its output is a 45-bit number.

- **3-state CA [75] :** Here, a 3-state 3-neighborhood periodic boundary CA with local rule $\mathcal{R} = 120021120021021120021021210$ is used as a PRNG. We have taken cell size $n = 51$ and window length $w = 20$ to generate ternary strings of length 20 and cell size $n = 101$ and window length $w = 40$ to generate ternary strings of length 40. These ternary numbers can be converted to equivalent 32-bit and 64-bit numbers respectively.

Remark :

• In most of the PRNGs, the underlying backbone is existence of a primitive polynomial of large degree. This polynomial ensures that the PRNG has a large period. All celebrated PRNGs today depend on this theory. A primitive polynomial belongs to the class of irreducible polynomials. There exists algorithms to determine whether a polynomial P is reducible or not [94]. However, testing primitivity of an irreducible polynomial requires prime factorization which is difficult to handle. To avoid this, researchers use known tricks (like, use

of Mersenne primes) that guaranty that the characteristic polynomial of the PRNG is primitive and the period is maximal. Therefore, the main problem is synthesizing a primitive polynomial. If there was efficient ways to synthesize a primitive polynomial, it would have been possible to develop PRNGs with any desirable period.

• The reason of development of LFSR and CA based PRNGs is mainly ease of cost effective hardware implementation. However, for PRNGs like Mersenne Twister, which takes a primitive polynomial of large degree, this hardware implementation is so costly that, it is infeasible. Moreover, for applications like VLSI circuit testing, efficiency and portability (see Section II) of a PRNG is more essential than intricate randomness. Nevertheless, for CA-based PRNGs, as feedback connections are from neighboring memory elements (cells), cost of interconnection on hardware implementation is lesser than LFSR. Due to this reason, and lack of parallelism, CA-based, more specifically ECA-based PRNGs are attractive as VLSI test pattern generators, compared to LFSR or LCG based PRNGs.

IV. Empirical Tests

Empirical tests target to find some pattern in the generated numbers of a PRNG to prove its non-randomness. These tests aim to check the local randomness property, that is, randomness of the numbers are approximated over a minimum sequence length, rather than the whole period [30]. Note that, for empirical tests, numbers of a complete period are not necessary. Innumerable such tests can be developed which aim to find any violation of the desirable properties (described in Section II), if exists, in a PRNG. If a PRNG passes all *relevant* empirical tests, it is declared as a good PRNG. However, usually, there is no known method to find which tests are pertinent for a PRNG to certify its randomness quality. Therefore, the common practice is to use empirical test-beds to identify non-randomness in the generator.

In general, empirical tests can be classified into two groups – blind tests and graphical tests. In case of blind tests, the tests are based on statistics and computation, so, no human intervention is required in taking a decision. On the other hand, in case of graphical tests, the performance is measured by finding visible patterns in the generated image; so here decision is taken by the coordinating person(s). In the next subsections, the tests used for our purpose are described in more details.

IV.1. Blind (Statistical) tests

In blind tests or statistical tests, the properties of a random sequence are considered to be probabilistic. So, when applied on a random sequence, the likely outcome of these tests is believed to be known a priori and measured in terms of probability. As arbitrary number of statistical tests are possible, there is no complete set of tests to test a PRNG. However, all these tests may not be relevant, if we consider the overall application area of the PRNG. Therefore, the

Table 2: Conclusions and Errors in statistical test

Real situation	Conclusion	
	$\mathscr{H}_0$ is rejected	$\mathscr{H}_0$ is not rejected
Data is random ($\mathscr{H}_0$ is true)	Type I error	Correct decision
Data is not random ($\mathscr{H}_0$ is not true)	Correct decision	Type II error

requirement of being a good PRNG is, to pass all the simple tests, along with, all relevant difficult-to-pass tests as well.

The target of statistical tests is to find evidence against a specific null hypothesis ($\mathscr{H}_0$). Usually, this $\mathscr{H}_0$ is, "the sequence to be tested is random", that is, the PRNG satisfies all essential properties of Section II. For each test, based on the random sequence produced by the PRNG, a decision is taken either to reject or not to reject the null hypothesis $\mathscr{H}_0$. To do this, a pertinent randomness statistic, having a distribution of possible values, has to choose which determines the rejection of $\mathscr{H}_0$. Under $\mathscr{H}_0$, the reference theoretical distribution (usually standard normal or chi-square distribution) of this statistic is calculated mathematically. A *critical value* (t) is computed for this reference distribution. During a statistical test, the relevant statistic is calculated on the generated random sequence and compared to the critical value. If the test statistic value is greater than the critical value, $\mathscr{H}_0$ is rejected, otherwise it is not rejected. The probable conclusions for any situation are shown in Table 2.

When a conclusion is made to reject the null hypothesis, while in truth, the data is random, is called a *Type I* error. However, if the data is not random, but in conclusion, $\mathscr{H}_0$ is not rejected, it results in generating *Type II* error. In other cases, the conclusion is correct. The *level of significance* (α) of a test is defined as the probability of generating a Type I error. Usually, it is set prior to the test as a number between 0.0001 and 0.01. Probability of generating a Type II error is denoted by β which indicates that a bad generator has produced a sequence which can fool the test.

If X denotes the test statistics and t the critical value, then the probability of Type I error is $P(X > t|\mathscr{H}_0$ is true) and probability of Type II error is $P(S \leq t|\mathscr{H}_0$ is false). The p-value of a test measures the strength of evidence against the null hypothesis. If p-value is very close to 0 or 1, which indicates that the sequence generated by the PRNG is not random. Normally, if p-value $\geq \alpha$, then the sequence tested is considered random, and $\mathscr{H}_0$ is not rejected, otherwise it is rejected. However, if the test statistic has a discrete distribution, the p-value is redefined as :

$$p = \begin{cases} p_R, & \text{if } p_R > p_L \\ 1 - p_L, & \text{if } p_R \geq p_L \text{ and } p_L < 0.5 \\ 0.5, & \text{otherwise} \end{cases} \tag{29}$$

where $p_R = P(X \geq t|\mathscr{H}_0$ is true) and $p_L = P(X \leq t|\mathscr{H}_0$ is true).

If $\mathscr{H}_0$ is not rejected by a set of tests, it may still be rejected by the next test or by other tests. There are many statistical battery of tests available which

target to find non-randomness based on a collection of statistical tests. The first known statistical battery of tests was offered by Donald Knuth in 1969 in his book "The Art of Computer Programming, Vol. 2"[30]. In our work, we have selected tests from three well-known test suites, namely *Diehard*, *TestU01* and *NIST*. The PRNGs selected earlier are tested in these tests from each of the testbeds. A PRNG is a good source of randomness, if for none of the tests of any battery of tests, the null hypothesis is rejected. However, practically, there is want of such a PRNG. Nevertheless, in this work, we take the count of the number of tests for which the null hypothesis is not rejected for a PRNG as its merit for randomness.

To maintain uniformity in testing, we have tested the stream of binary numbers generated in sequence by the PRNGs. For each PRNG, binary (*.bin*) files are produced which contain sequence of numbers (in binary form) without any gap between two consecutive numbers in the sequence. That means, if output of a PRNG is equivalent to x-bit numbers, then successive non-overlapping x bits in the binary file corresponds to each number generated by the PRNG. Therefore, if a particular test on a test-bed uses y consecutive bits, where $y \neq x$, it takes it from the binary sequence. However, often, the generated numbers from PRNG(s) are not binary, rather normalized. In that case, first we convert fractional part of each number into its binary equivalent and then add these bits to the .bin file ignoring the binary point. Size of this file depends on the testbed which uses this file. So, the tests used in each testbed along with the setup required to perform these tests, are stated next.

IV.1.1. Diehard battery of Tests
George Marsaglia in 1996 provided this battery of tests [24], which is the basic testbed for PRNGs. It consists of 15 different tests -

Diehard Battery of Tests

1. Birthday spacings, 2. Overlapping permutations, 3. Ranks of 31×31 and 32×32 matrices, 4. Ranks of 6×8 matrices, 5. Monkey tests on 20-bit Words, 6. Monkey tests : OPSO (Overlapping-Pairs-Sparse-Occupancy), OQSO (Overlapping-Quadruples-Sparse-Occupancy) and DNA tests, 7. Count the 1's in a stream of bytes, 8. Count the 1's in specific bytes, 9. Parking lot test, 10. Minimum distance test, 11. Random spheres test, 12. The squeeze test, 13. Overlapping sums test, 14. Runs up and runs down test, and 15. The craps test (number of wins and throws/game).

To test a PRNG on Diehard for a particular seed, a binary file of size $10-12$MB is created using the generated numbers of the PRNG with that seed. In our case, we have taken the file size as 11.5MB for all PRNGs. For each test, one or multiple p-values are derived. A test is called *passed*, if every p-value of the test is within 0.025 to 0.975 [24]. A PRNG is supposed to be a good PRNG, if it passes all tests of every test-bed for any seed. However, in general, it is rare to find a PRNG which can fool all tests of Diehard. For example, it

is very difficult to pass overlapping permutations test and parking lot test for every seed of a PRNG.

IV.1.2. TestU01 library of Tests

This library offers implementations of many stringent tests – the classical ones as well as many recent ones. It was developed by Pierre L'Ecuyer and Richard Simard [25] to remove the limitations of existing testbeds – like inability to modify the test parameters (such as, the input file type, p-values etc) as well as include new updated tests. It includes several battery of tests, including most of the tests in Diehard and many more with more flexibility to select the test parameters than in Diehard. However, we have selected the battery *rabbit* (bbattery_RabbitFile()) to test the PRNGs. The reason for choosing this test-suite having only a particular selection of tests is – this battery is specifically designed to test a sequence of random bits produced by a generator. As already mentioned, our target is to test the binary sequences generated by the PRNGs, hence, this test-suite accomplish our requirement. It contains the following 26 tests from different modules (mentioned in parenthesis for each test) which are said to be sufficient to test a PRNG for general non-cryptographic purposes :

Battery Rabbit of TestU01

1. MultinomialBitsOver test (smultin), 2. ClosePairsBitMatch in $t = 2$ dimensions (snpair) and 3. ClosePairsBitMatch in $t = 4$ dimensions (snpair), 4. AppearanceSpacings test (svaria), 5. LinearComplexity test (scomp), 6. LempelZiv test (scomp), 7. spectral test of Fourier1 (sspectral), and 8. spectral test of Fourier3 (sspectral), 9. LongestHeadRun test (sstring), 10. PeriodsInStrings test (sstring), 11. HammingWeight with blocks of $L = 32$ bits test (sstring), 12. HammingCorrelation test with blocks of $L = 32$ bits (sstring), 13. HammingCorrelation test with blocks of $L = 64$ bits (sstring) and 14. HammingCorrelation test with blocks of $L = 128$ bits (sstring), 15. HammingIndependence with blocks of $L = 16$ bits (sstring), 16. HammingIndependence with blocks of $L = 32$ bits (sstring) and 17. HammingIndependence with blocks of $L = 64$ bits (sstring), 18.AutoCorrelation test with a lag $d = 1$ (sstring)and 19. AutoCorrelation test with a lag $d = 2$ (sstring), 20. Run test (sstring), 21. MatrixRank test with 32×32 matrices (smarsa) and 22. MatrixRank test with 320×320 matrices (smarsa), 23. RandomWalk1 test with walks of length $L = 128$ (swalk), 24. RandomWalk1 test with walks of length $L = 1024$ (swalk), and 25. RandomWalk1 test with walks of length $L = 10016$ (swalk).

Here *smultin* is a module of tests based on the multinomial distribution [95] which tests uniformity in the t-dimensional unit hypercube. The module *snpair* implements tests based on the distances between the closest points in a sample of n uniformly distributed points in the unit torus in t-dimensions [96]. *svaria* is a module that implements different uniformity tests, mainly based on some simple statistics. The module *scomp* contains tests based on linear complexity of bit sequence as well as on the compressibility of it, measured by the Lempel-Ziv complexity [97]. The statistical tests developed by George Marsaglia and his collaborators in [98] are implemented in *smarsa* module. In

case these tests are spacial cases of the tests of module *smultin*, the function smultin_MultinomialOver is called. The module *sspectral* contains tests based on spectral methods, which computes the discrete Fourier transform of a bit string of size n and looks for deviations in the spectrum inconsistent with $\mathscr{H}_0$. *sstring* module implements tests on strings of random bits made by concatenating blocks of s bits from each. In module *swalk*, statistical tests based on discrete random walks over $\mathbb{Z}$ is implemented [99]. Among these tests, spectral tests are the most difficult ones to pass.

The battery *rabbit* takes two arguments – a filename and number of bits (nb). The first nb bits of the binary file, filled by the random numbers generated by the PRNG, is tested. For each test, the parameters are a function of nb, to make it dynamic. For the PRNGs, selected for ranking, we have set $nb = 10^7$ and the file size as 10.4MB. Here, a test is declared to be passed for a seed, if each of the p-values of the test is within 0.001 to 0.999 [25].

Remark : Between Diehard and TestU01's rabbit battery of tests, we have observed that, for the selected PRNGs, some tests of Diehard are more stringent to pass than that of battery rabbit of TestU01. For example, for a specific seed, even if the PRNG passes all tests of rabbit, but it may fail to pass overlapping permutations test of Diehard.

IV.1.3. NIST Statistical Test suite

The NIST Statistical Test Suite is a test suite developed to test a PRNG for cryptographic properties [26]. This testbed consists of 15 tests –

NIST Test Suite

1. The Frequency (Monobit) Test, 2. Frequency Test within a Block, 3. The Runs Test, 4. Tests 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 Linear Complexity Test, 11. The Serial Test, 12. The Approximate Entropy Test, 13. The Cumulative Sums (Cusums) Test, 14. The Random Excursions Test, and 15. The Random Excursions Variant Test.

This test suite has mainly three tasks – (1) investigate the distribution of 0s and 1s, (2) using spectral methods, analyze the harmonics of bit stream and (3) detect patterns based on information theory or probability theory. For this test suite, the significance level $\alpha = 0.01$. So, for a sample size m generated by a PRNG with a particular seed, if proportion of sequences with p-values ≥ 0.01 is x, then for the PRNG to pass the test, x should lie between the acceptable proportions. This range of acceptable proportions is calculated as $(1-\alpha) \pm 3\sqrt{\frac{\alpha(1-\alpha)}{m}}$, for a sample size α. This x is the minimum pass rate. It is approximately 980 for sample size 1000 for each statistical test (except the random excursion (variant) test) and 615 for a sample size of 629 for the random excursion (variant) test.

To test a PRNG using NIST test suite, a binary or ASCII file containing the random numbers is given as input. We have taken sample size as 10^3 with sequence length = 10^6 and generated binary file of size 125MB as the input. The default parameters are not updated, that is, block length (M) for block frequency test is 128 and for linear complexity test is 500. Similarly, block length (m) for both non-overlapping template test and overlapping template test is 9, for approximate entropy test is 10 and for serial test is 16. The file *finalAnalysisReport.txt* contains summary of results of all the tests. In this file, the first 10 columns note the frequency of p-values in each of the 10 sub-intervals between 0 to 1 and column 11 is the p-value derived by applying chi-square test on these columns. For the corresponding statistical test noted in column 13, column 12 records the passed proportions of samples. This file also indicates the tests (or parts of a test) which are not passed, by marking it with '$*$'. If all parts of a test are passed, the PRNG is said to have passed that test.

Remark : In general, simple non-cryptographic PRNGs fail to pass NIST tests. However, a good PRNG, which passes all or most of the tests of TestU01 and Diehard, also perform well in NIST test-suite. Therefore, a good non-cryptographically secure PRNG may pass all tests of NIST for some seeds.

IV.2. Graphical Test

As discussed already, goal of every empirical test is to detect a pattern in the numbers generated by a PRNG to prove its non-randomness. In statistical tests, this is done by generating the p-value. However, it may happen that, a Type I or Type II error has occurred, and the wrong conclusion is reached. Therefore, it is more useful to actually see how the numbers look like in a 2-dimensional or 3-dimensional plot.

In graphical tests, the numbers are plotted in a graph to see whether any visible pattern exists or not. As period length a PRNG is expected to be very large, so, for every graphical test also, all numbers of a period can not be used; rather a set of numbers need to be generated based on some seed. We have mainly used two graphical tests – (1) Lattice tests, (2) Space-time diagram. Let us now explain these two tests.

IV.2.1. Lattice Test

This test identifies whether the random numbers form some patterns. To test this, the consecutive random numbers (in normalized form), generated from a seed, are paired and plotted. Two types of lattice tests are executed on these normalized numbers, namely 2-D lattice test (takes two consecutive numbers as a point) and 3-D lattice tests (three consecutive numbers form a point). If the random numbers are correlated, the plots show patterns. Otherwise, the PRNG is considered to be good.

IV.2.2. Space-time Diagram

Space-time diagram is an important theoretical tool that has long been used to observe and predict the behavior and evolution of a CA [100]. For CAs, it is

a graphical representation of the configurations (on x-axis) at each time t (on y-axis). Each of the CA states are depicted by some color. So, the evolution of the CA can be visible from the patterns generated in the state-space diagram.

In this work, we propose this tool as an useful measure of randomness of a PRNG. The x-axis of a diagram represents a number generated at any time instant and y-axis depicts time. To test a PRNG with space-time diagram, the numbers need to be non-normalized. If numbers are in base b, then b different colors are required to represent a number where each color signifies a particular digit of that base. For example, if numbers are binary, then two colors, usually black for 1 and white for 0, are required to represent any number. So, each binary number is then a combination of black and white. For $b > 2$, more colors are required for each number.

Starting with a seed, a set of numbers are generated over time t and each number is plot against t. If there is a pattern among any consecutive numbers, or in any part of a number, then it can be seen from this diagram, as colors make this pattern more prominent. If there is no pattern, and the numbers appear noisy in color, the PRNG has good randomness quality. Therefore, using this diagram, clear idea about the randomness properties of a PRNG can be developed.

In the next section, result of these empirical tests applied on the PRNG are recorded. For both the graphical tests, 1000 numbers are generated for each seed. In case of space-time diagrams, these numbers are directly represented, whereas, for lattice tests, pairs (or, triplets) of consecutive numbers are plot as each point in the 2-D (or, 3-D) plane.

V. Empirical Facts

This section depicts the output of the empirical tests described in Section IV on the PRNGs selected in sections III.1, III.2 and III.3. To do these tests, first we have chosen the seeds. After getting results of all empirical tests for all PRNGs, these results are analyzed to compare the PRNGs and rank them accordingly.

V.1. Choice of Seeds

Although a good PRNG should be independent of seeds, but to run a PRNG we need to choose the seeds. This seed can be any number from the period of a PRNG. However, it is not possible to test every PRNG for all possible seeds. So, we have taken the following greedy approach:

1. As we have collected the C programs of the PRNGs from their respective websites, each of them has an available seed for normal usage. For example, for MT19937, the seed was 19650218. Nevertheless, in most of the cases, this seed is 1234 or 12345. We, therefore, have collected all the seeds hard-coded in the C programs of all PRNGs, and used these as the set of seeds for each PRNG. These seeds are 7, 1234, 12345, 19650218 and 123456789123456789. We have tested each PRNg with all these seeds.

2. Apart from studying the behavior of the PRNGs for the fixed seeds, we also want to observe the average case behavior of the PRNGs. For this reason, we have chosen a simple LCG, *rand*() to generate seeds for all other PRNGs. This rand() is initialized with *srand*(0). The next 1000 numbers of rand() are supplied as seeds to each PRNG to test it 1000 times with these random seeds.

All PRNGs are tested empirically for each of these seeds and the results are compared impartially. Whenever a PRNG requires more than one seed to initialize its components, we have supplied the same seed to all its components.

V.2. Results of Empirical Tests

The selected PRNGs (LCG-based, LFSR-based and CA-based) are tested with blind or statistical tests as well as with the graphical tests. Here, summary of the results of these tests are documented.

V.2.1. Results of Statistical Tests

Table 3 shows the summary of results of Diehard battery of tests, battery *rabbit* of TestU01 library and NIST statistical test suite for the fixed seeds. In this table, for each PRNG, result (in terms of numbers of tests passed) of the testbeds per each seed is recorded.

Table 3: Summary of Statistical test results for different fixed seeds

	Seeds ⟶	7			1234			12345			19650218			123456789123456789			**Ranking**
	Name of the PRNGs	Diehard	TestU01	NIST	Diehard	TestU01	NIST	Diehard	TestU01	NIST	Diehard	TestU01	NIST	Diehard	TestU01	NIST	(First Level)
LCGs	MMIX	6	19	7	5	18	7	6	17	8	4	16	8	5	18	8	8
	minstd_rand	0	1	1	0	1	1	0	1	2	0	1	1	0	2	1	12
	Borland LCG	1	3	5	0	3	5	1	3	5	1	3	4	1	3	5	11
	rand()	1	1	2	1	1	2	1	3	2	1	2	2	1	2	2	11
	lrand48()	1	3	2	1	2	2	1	3	2	1	3	2	1	2	2	11
	MRG31k3p	1	2	1	1	1	1	1	2	1	1	1	1	0	0	2	12
	PCG-32	9	25	15	9	25	14	11	25	14	10	24	15	9	25	15	2
LFSRs	random()	1	1	1	1	1	1	1	3	1	1	2	1	1	2	1	11
	Tauss88	11	21	15	9	23	15	11	23	15	11	23	14	10	23	15	4
	LFSR113	5	6	1	11	23	14	9	23	15	7	23	14	9	23	15	7
	LFSR258	0	0	1	0	5	2	1	5	2	1	5	2	1	5	0	12
	WELL512a	9	23	15	10	23	14	10	23	15	8	23	15	7	23	15	5
	WELL1024a	9	25	15	10	24	15	9	24	14	9	25	15	9	25	15	3
	MT19937-32	10	25	13	9	25	13	9	25	14	9	25	15	9	25	15	3
	MT19937-64	10	25	15	10	24	15	8	24	15	11	25	15	10	25	15	2
	SFMT-32	10	25	15	9	25	15	10	25	15	9	25	15	10	25	15	1
	SFMT-64	11	25	15	10	25	15	10	25	15	9	25	15	10	25	15	1
	dSFMT-32	7	25	15	8	25	15	11	24	13	11	25	15	10	24	15	5
	dSFMT-52	5	11	3	5	10	3	7	11	3	6	10	3	7	9	3	9
	XORShift32	4	17	4	4	17	4	4	17	2	0	17	13	4	17	13	9
	XORShift64*	10	25	15	10	25	15	8	25	15	7	25	15	8	25	14	5
	XORShift1024*	7	20	6	9	21	15	7	20	15	8	20	15	6	21	15	6
	XORShift128+	9	25	14	9	25	14	10	24	15	10	25	15	8	24	15	4
CAs	Rule 30	11	25	15	10	25	15	9	25	15	8	25	15	11	24	15	2
	Hybrid CA with Rules 30 & 45	3	8	3	0	1	0	2	8	1	1	7	2	1	8	2	11
	Maximal Length CA with $\gamma = 0$	2	12	10	0	12	11	1	12	11	1	12	11	2	12	10	10
	Maximal Length CA with $\gamma = 1$	4	17	14	3	16	14	3	17	14	4	15	14	3	16	14	8
	Non-linear 2-state CA	6	11	4	8	10	2	5	12	3	5	12	4	7	12	4	9
	3-state CA	3	12	6	3	12	6	3	11	5	2	11	4	3	11	4	10

Note that, none of the PRNGs can pass all tests of these blind test-beds. However, we can notice the following:

- Among all, the LCGs minstd_rand, Borland's LCG, rand, lrand48, MRG31k3p and the LFSRs LFSR258 and random perform very poorly. In case of

diehard tests, these PRNGs can pass at most the runs test only, except LFSR258, which passes only the rank test of 31×31 and 31×32 matrices and runs down test.

- The remaining two LCGs based PRNGs, namely Knuth's MMIX and PCG 32 bit behave well in comparison to the other LCGs as well as many of the LFSRs. For example, Knuth's MMIX is better than LFSR258 and Xorshift32, whereas PCG 32 bit is better than MMIX as well as LFSR113, Xorshift PRNGs, WELL PRNGs and dSFMTs. In fact, performance of PCG-32 bit is comparable to MTs and SFMTs.

- Performance of LFSR113 is dependent on seed; whereas, WELL512a and WELL1024a are invariant of seeds.

- Among the Mersenne Twister and its variants, performance of dSFMTs (especially, dSFMT-52), are unexpectedly poor in terms of the blind empirical tests.

- Among the CA-based PRNGs, rule 30 can compete with the elite group of PRNGs like Mersenne Twisters, WELL and PCG. However, other CA-based PRNGs are not so good. Because in those CAs, the complete (or, a block from a) configuration of the CA is taken as a number. Among these CA-based PRNGs, performance of the max-length CA ($\gamma = 1$) is better than the non-linear CA, which is better than the max-length CA ($\gamma = 0$) and hybrid CA rule 30-45.

Therefore, we can define first level ranking of the PRNGs from these test results. Last column of Table 3 shows this ranking. For this ranking, we have considered the overall tests passed in each test-suite. If two PRNGs gives similar results, they have the same rank. It can be observed that, among the LCG-based PRNGs, PCG (32-bit) gives the best result and among the LFSR-based PRNGs, performances of SFMTs (32 and 64 bits) are comparable. Moreover, among the CA-based PRNGs, Wolfram's rule 30 gives best result, which is comparable to the results of SFMTs and MTs. However, as multiple PRNGs have performed similarly, we have applied same rank to group of PRNGs, as follows –

- As SFMTs are best performers, these are ranked as 1. Similarly, performance of MT-64 bit, PCG-32 bit and rule 30 are comparable (ranked 2), performance of MT-32 bit and WELL1024a are comparable (ranked 3), whereas performance of Xorshift128+ and Tauss88 are comparable (ranked 4). These are the elite group of best performing PRNGs.

- Xorshift64* performs well, but it has more dependency on seed than the MTs and SFMTs. For this reason it has rank lower than MT and SFMT. Similarly, LFSR113 can perform better than WELLs and non-linear CA-based PRNG for some seeds, but as performance of WELL is more invariant of seed, it has higher rank than LFSR113.

- As performance of WELL512a, Xorshift64* and dSFMT-32 bit are comparable, they are ranked as 5. In terms of performance in NIST test-suite, Xorshift1024* (rank 6) is better than LFSR113 (rank 7), but worse than dSFMT-32 bit PRNG.

- Among the other PRNGs, maximal-length CA with $\gamma = 1$ and MMIX are ranked 8, the non-linear ECA based PRNG, dSFMT-52 bit PRNG and Xorshift32 are ranked 9 and maximal-length CA with $\gamma = 0$ and 3-state CA are ranked 10.

- Rest of the PRNGs form two groups – rand, lrand, Borland's LCG, random and rule $30 - 45$ as rank 11 and minstd_rand, MRG31k3p and LFSR258 as rank 12.

Note that, in this ranking, many of the PRNGs are in the same group. However, this situation may be improved by considering the average number of tests passed by the PRNGs and the range of the tests passed. To analyze this average case result, we have tested these PRNGs with Diehard battery of tests only. Figure 10 and Figure 11 show the plots for all the PRNGs (except rand(), as rand() is used to generate the seeds). For each of the figures, x axis represents the number of tests passed by a PRNG and y axis denotes the frequency of passing these tests. By using these plots, we can get a second level of ranking of the PRNGs, as shown in Table 4.

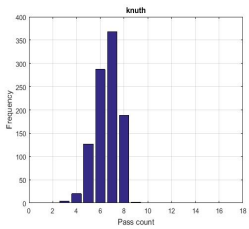
From this table, we can observe that, among all PRNGs, minstd_rand is the worst performer in Diehard Tests. Moreover, the LCGs rand, lrand, MRG31k3p and LFSRs random, LFSR258 are worse than the CA-based PRNGs. So, these PRNGs are among the lowest rank holders. However, although most of the CA based PRNGs can not compete with the elite group of PRNGs, but having non-linearity as a characteristics can provide advantage in many applications where linear system is undesirable. Further, using the average case results, some of the PRNGs holding same rank in 1^{st} level ranking can be differentiated, as discussed below–

- In terms of average tests passed and range, rule 30 beats all other PRNGs. However, considering results of NIST and TestU01, SFMTs still hold the first rank, while rule 30 is ranked 2. As average of MT-64 bit is better than PCG-32 bit, it is ranked 3.

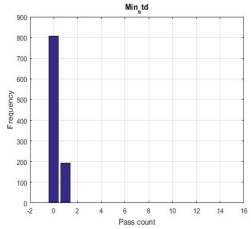
- Average of PCG-32 bit, MT-32 bit, Xorshift128+ and WELL1024a are similar and in terms of performance in NIST and TestU01 battery of tests, they are alike. So, all these PRNGs are ranked 4.

- The next rank holders are dSFMT-32 bit (rank 5), WELL512a and Xorshift64* (rank 6). As, Tauss88 sometimes fails to pass any tests of Diehard and average of LFSR113 is quite high, so they are put in the same rank 7.

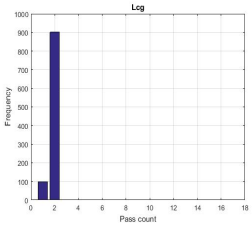
- Xorshift1024a has better rank (rank 8) than non-linear CA and MMIX (rank 9), because of performance in NIST and TestU01 library.



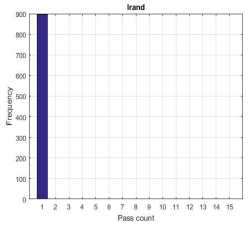
(a) MMIX



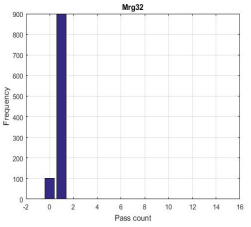
(b) minstd_rand



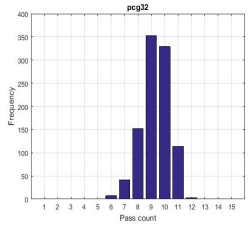
(c) Borland's LCG



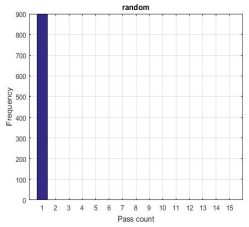
(d) lrand48()



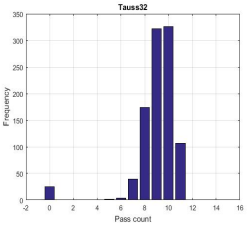
(e) MRG31k3p



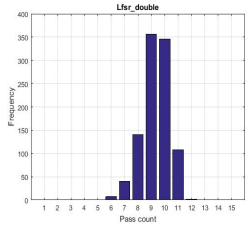
(f) PCG-32



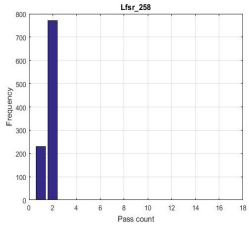
(g) random



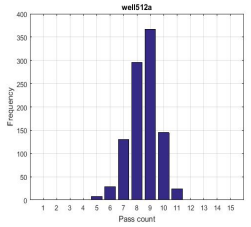
(h) Tauss88



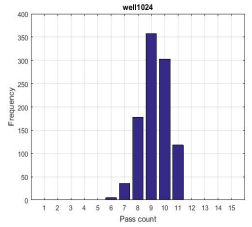
(i) LFSR113



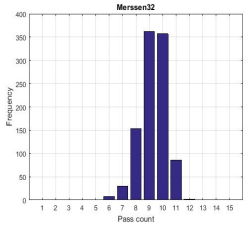
(j) LFSR258



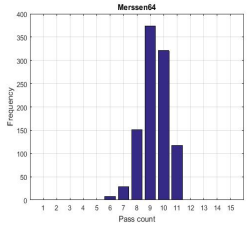
(k) WELL512a



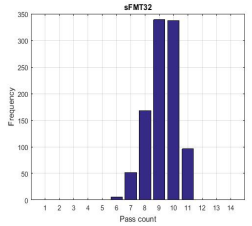
(l) WELL1024a



(m) MT19937-32

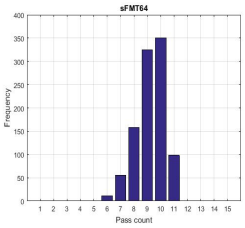


(n) MT19937-64

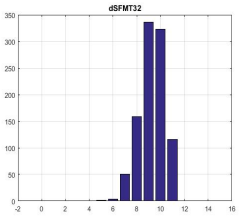


(o) SFMT19937-32

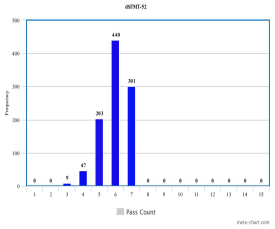
Figure 10: Average Test results of PRNGs for 1000 random seeds with Diehard battery of Tests



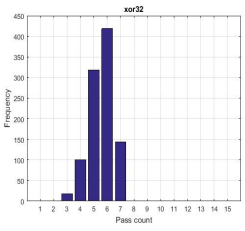
(a) SFMT19937-64



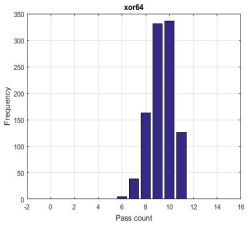
(b) dSFMT19937-32



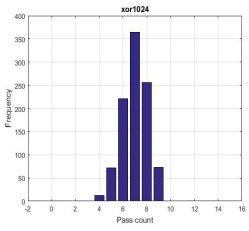
(c) dSFMT19937-64



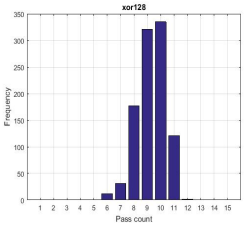
(d) XORShift32



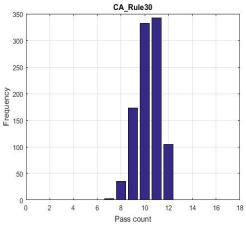
(e) XORShift64*



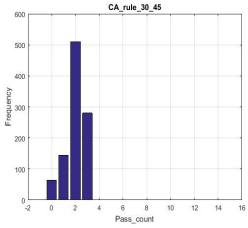
(f) XORShift1024*



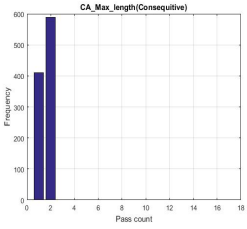
(g) XORShift128+



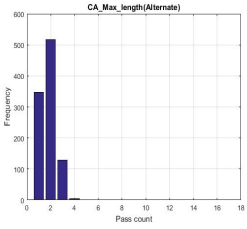
(h) Rule30



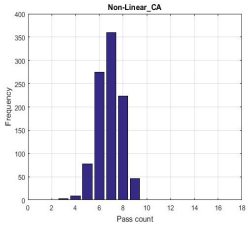
(i) Rule30-45



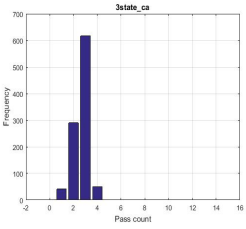
(j) max-length CA with $\gamma = 0$



(k) max-length CA with $\gamma = 1$



(l) Non-linear CA



(m) 3-state CA

Figure 11: Average Test results of PRNGs for 1000 random seeds with Diehard battery of Tests (continued)

Table 4: Summary of Statistical test results for different seeds

	Name of the PRNGs	Fixed Seeds			Random Seeds		Previous Rank	2^{nd} level Rank
		Diehard	TestU01	NIST	Average	Range		
LCGs	MMIX	4-6	16-19	7-8	6.5	2-9	8	9
	minstd_rand	0	1	1-2	0.38	0-1	12	14
	Borland LCG	1	3	4-5	1.9	1-2	11	12
	rand()	1	1-3	2-3			11	13
	lrand48()	1	2-3	2	1	1	11	13
	MRG31k3p	0-1	1-2	1-2	0.9	0-1	12	14
	PCG-32	9-11	24-25	14-15	9.3	6-12	2	4
LFSRs	random()	1	1-3	1	1	1	11	13
	Tauss88	9-11	21-23	14-15	9.0	0-12	4	7
	LFSR113	5-11	6-23	1-15	9.3	6-12	7	7
	LFSR258	0-1	0-5	0-2	1.8	1-2	12	14
	WELL512a	7-10	23	14-15	8.5	5-11	5	6
	WELL1024a	9-10	24-25	14-15	9.2	6-11	3	4
	MT19937-32	9-10	25	13-15	9.3	6-12	3	4
	MT19937-64	8-11	24-25	15	9.4	6-11	2	3
	SFMT-32	9-10	25	15	9.3	6-11	1	1
	SFMT-64	9-11	25	15	9.3	6-11	1	1
	dSFMT-32	7-11	24-25	13-15	9.3	5-11	5	5
	dSFMT-52	5-7	9-11	3	5.98	3-7	9	10
	XORShift32	2-4	17	2-13	5.5	3-7	9	10
	XORShift64*	7-10	25	14-15	8.0	6-11	5	6
	XORShift1024*	6-9	20-21	6-15	7.0	4-9	6	8
	XORShift128+	8-10	24-25	14-15	9.4	6-12	4	4
CAs	Rule 30	8-11	24-25	15	10.2	7-12	2	2
	Hybrid CA with Rules 30 & 45	0-3	1-8	0-3	2.0	0-3	11	12
	Maximal Length CA with $\gamma = 0$	0-2	12	10-11	1.6	1-2	10	11
	Maximal Length CA with $\gamma = 1$	3-4	15-17	14	1.8	1-4	8	11
	Non-linear 2-state CA	5-8	10-12	3-4	7.3	3-9	9	9
	3-state CA	2-3	11-12	4-6	2.7	1-4	10	11

- The next rank holder is Xorshift32 and dSFMT-52 bit PRNG (rank 10).
- As maximal-length CA with $\gamma = 1$ has lower average, its rank is degraded. It is put in the same group as 3-state CA and maximal-length CA with $\gamma = 0$ (rank 11).
- Rule $30 - 45$ and Borland's LCG are ranked 12, whereas other PRNGs previously on the same group, like rand, lrand and random are ranked 13.
- Like previous ranking, minstd_rand, MRG31k3p and LFSR258 are the last rank holders based on their overall performance and the fact that for many seeds, these PRNGs fails to pass any test.

In the next section, graphical tests are further incorporated on these PRNGs to verify this ranking as well as to check whether any second level ranking of the intra group PRNGs is possible or not.

V.2.2. Results of Graphical Tests

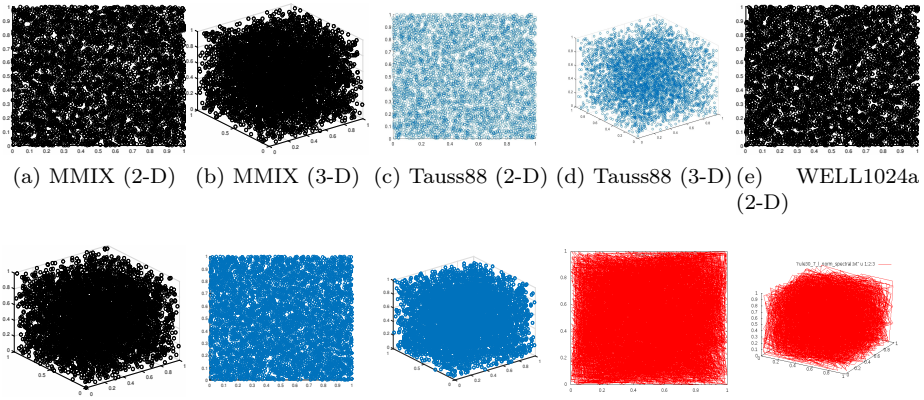
As mentioned, two types of graphical tests are performed on the PRNGs – lattice tests (2-D and 3-D) and space-time diagram test. Here, each of the PRNG is tested using only the five fixed seeds. Note that, for the output images of these graphical tests, we have used naming of seeds; the five seeds are named as follows–
seed 7 as s_1, 1234 as s_2, 12345 as s_3, 19650218 as s_4 and seed 123456789123456789 as s_5.
In all the figures of this section, this naming convention is used instead of using the seed value. The motivation behind these graphical tests are – (a) to understand why some PRNGs perform very poorly, (b) to differentiate the behavior of the PRNGs which perform similarly in the blind empirical tests and (c) to visualize the randomness of the PRNGs. The results of these tests are shown in the following.

1. **Result of Lattice Tests :** For each of the seeds, the 2-dimensional and 3-dimensional lattice tests are performed on every PRNG. As expected, for rand, lrand48, minstd_rand, Borland's LCG, and random, the points are either scattered or concentrated on a specific part of the 2-D and 3-D planes. However, for the good PRNGs like MTs, SFMTs, WELL, the plots are relatively filled. For example, see Figure 12 for output of MMIX, Tauss88, WELL1024a, 3-state CA and rule 30.

   However, this test fails to further enhance or modify the ranking shown in Table 4. Therefore, we have avoided supplying all the images of lattice test results and further tested these PRNGs by space-time diagram.

2. **Result of Space-time Diagram Test :** For space-time diagram, a set of 1000 numbers are generated from each seed and printed on $X - Y$ plane. In this paper, for each PRNG, space-time diagram of 4 seeds s_1, s_2, s_3, s_4 are shown in figures 13, 14, 15, 16, 17, 18, and 19.

   From these figures, we can observe the following:



(a) MMIX (2-D) (b) MMIX (3-D) (c) Tauss88 (2-D) (d) Tauss88 (3-D) (e) WELL1024a (2-D)

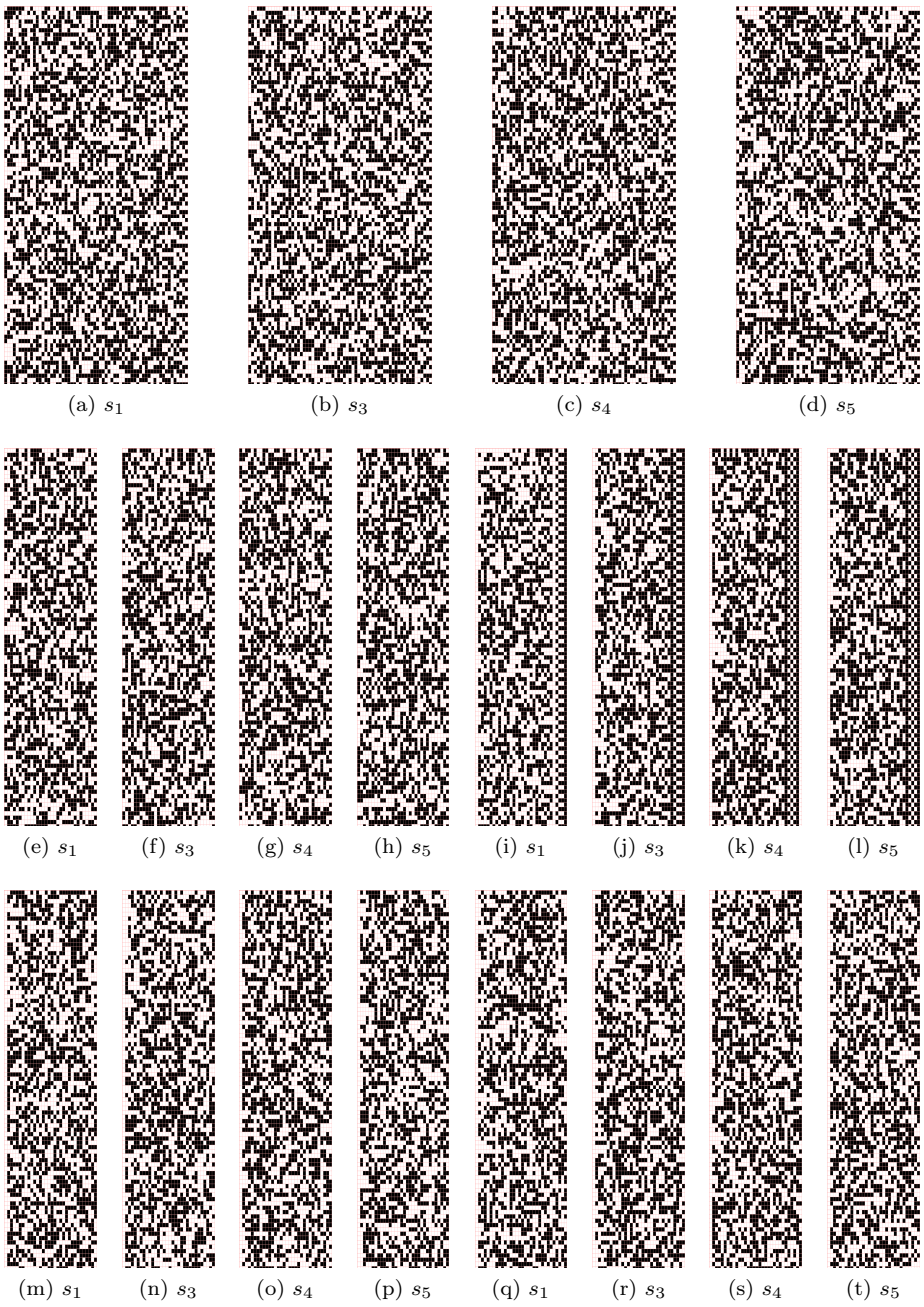
(f) WELL1024a (3-D) (g) 3-state CA (2-D) (h) 3-state CA (3-D) (i) Rule 30 (2-D) (j) Rule 30 (3-D)

Figure 12: Lattice Test results for rand, MMIX, Tauss88, WELL1024a, 3-state CA and Rule 30 with seed 7

- For minstd_rand, the last 6 bits of the generated numbers are fixed and for Knuth's MMIX, last 2 bits of four consecutive numbers form a pattern.
- For rand, lrand, Borland's LCG, MRG31k3p and random, the percentage of black and white boxes representing 1s and 0s are not same. Even for PCG 32-bit generator, there is pattern visible in the diagrams.
- LFSR113 forms pattern for some seeds. For WELL and Xorshift generators, dependency on seed is visible for the initial numbers.
- For MTs and SFMTs, the dependency on seed is visible for very few levels. For dSFMTs, there is pattern visible in the diagrams.
- Among the CA-based generators, rule $30 - 45$ has visible patterns and max-length CAs have dependency on seed up to some initial configurations. However, the figures for rule 30 CA, non-linear CA and 3-state CA appear relatively random.
- For the LCGs, the dependency on seed is less visible than the LFSRs.
- If observed minutely, every PRNG has some kind of clubbing of white boxes and black boxes, that is, none of the figures is actually free of pattern. That is why, none of these can pass all blind tests. However, for the good PRNGs, these patterns are non-repeating. Hence, these can serve most of the purposes.

V.3. Final Ranking

By using the space-time diagrams along with the statistical tests, we can further enhance the rankings of the PRNGs –

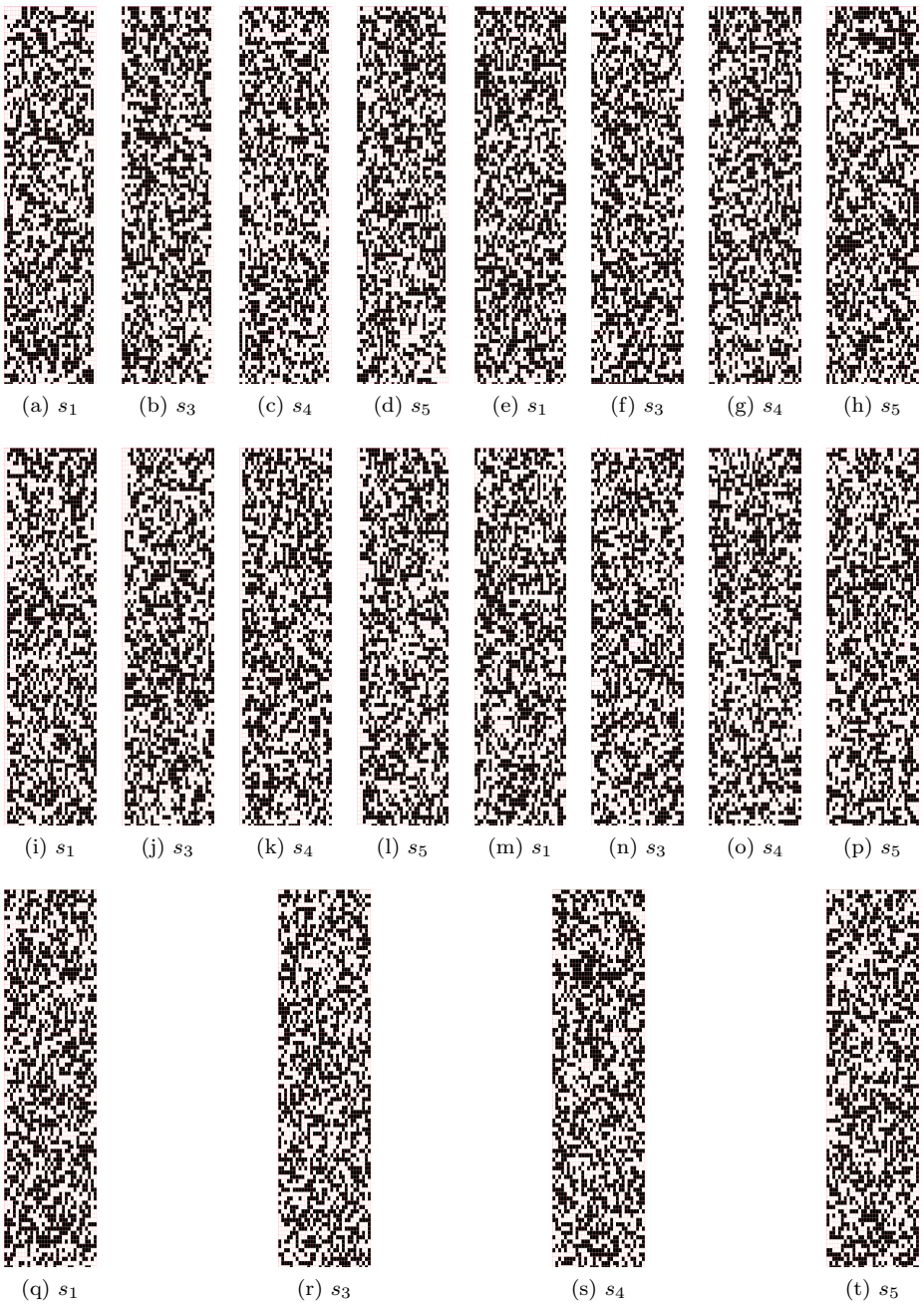


(a) s_1 (b) s_3 (c) s_4 (d) s_5

(e) s_1 (f) s_3 (g) s_4 (h) s_5 (i) s_1 (j) s_3 (k) s_4 (l) s_5

(m) s_1 (n) s_3 (o) s_4 (p) s_5 (q) s_1 (r) s_3 (s) s_4 (t) s_5

Figure 13: Space-time diagram for Knuth's MMIX (13a to 13d), Borland's LCG (13e to 13h) and minstd_rand (13i to 13l), rand (13m to 13p) and lrand (13q to 13t) of UNIX

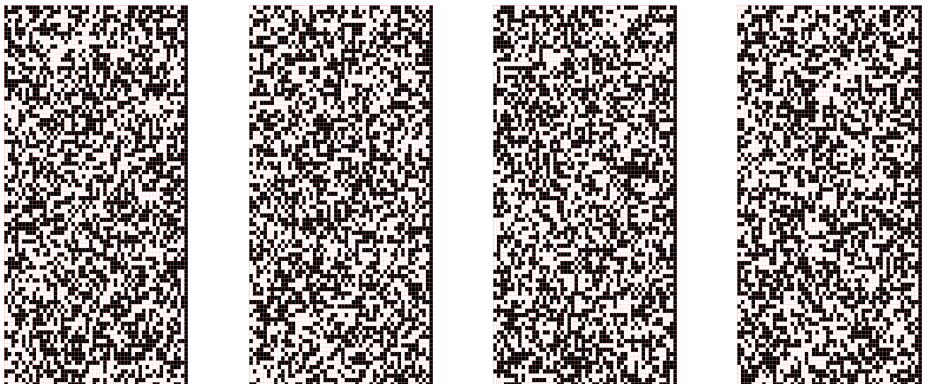


(a) s_1 (b) s_3 (c) s_4 (d) s_5 (e) s_1 (f) s_3 (g) s_4 (h) s_5

(i) s_1 (j) s_3 (k) s_4 (l) s_5 (m) s_1 (n) s_3 (o) s_4 (p) s_5

(q) s_1 (r) s_3 (s) s_4 (t) s_5

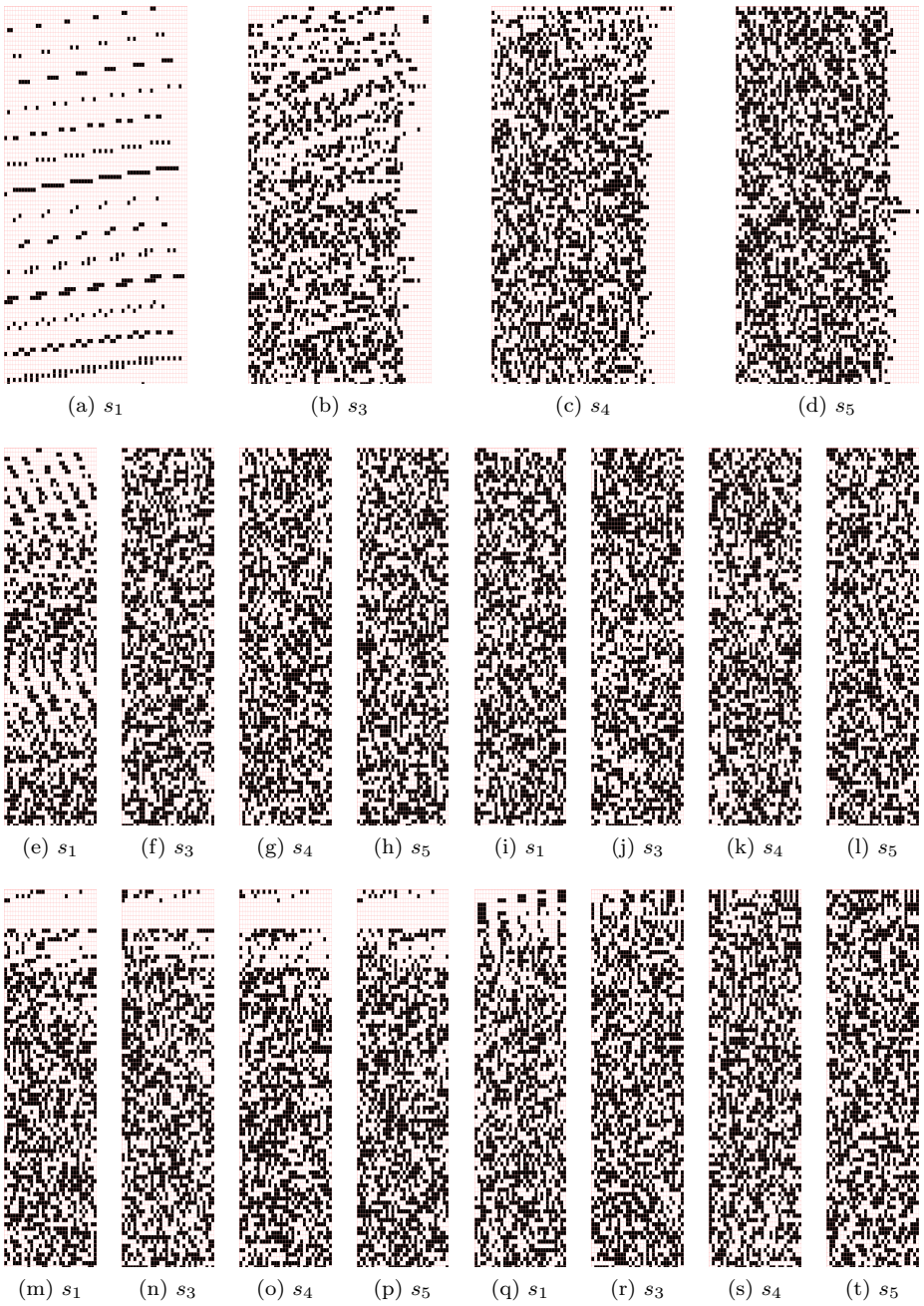
Figure 14: Space-time diagram for MRG31k3p (14a to 14d) and PCG 32-bit (14e to 14h), random (14i to 14l), Tauss88 (14m to 14p) and dSFMT19937 32 bit (14q to 14t)



(a) s_1 (b) s_3 (c) s_4 (d) s_5

Figure 15: Space-time diagram for dSFMT19937 64 bit (15a to 15d)

- SFMT-64 bit PRNG holds the first position as it appears more random than SFMT-32 bit PRNG.
- Rule 30 holds the 3^{rd} rank, whereas MT-64 bit PRNG holds rank 4. PCG 32 bit PRNG is better than MT-32 bit and dSFMT-32 bit PRNGs. So, it holds rank 5. The next rank holder is MT-32 bit PRNG.
- dSFMT 32-bit has less dependency on seed than WELL1024a and Xorshift128+. So, it is ranked 7^{th} position.
- XorShift64* has no dependency on seed, so it is ranked higher than WELL1024a and Xorshift128+.
- WELL512a is ranked lower than WELL1024a and Xorshift128+, as it has more dependency on seed. As Tauss88 (rank 11) sometimes cannot pass any tests, so it is ranked lower than WELL512a (rank 10).
- dSFMT-52 bit PRNG has less dependency on seeds than non-linear CA based PRNG and max-length CA with $\gamma = 1$. So, it holds rank 12.
- Non-linear CA based PRNG, max-length CA with $\gamma = 1$ and 3-state CA based PRNG form the group of 13 rank holders.
- Although LFSR113 and Xorshift1024* can perform well for some seeds, but because of its dependency on seeds and visible patterns in the space-time diagram, these are ranked lower than max-length CA with $\gamma = 1$. Therefore, LFSR113 and Xorshift1024* downgrade to rank 14.
- Knuth's MMIX and Xorshift32 generator are in the same group (rank 15).
- Max-length CA with $\gamma = 0$ has better rank (rank 16) than rule $30-45$ CA (rank 17).
- Among rand, lrand, minstd_rand, Borland's LCG, MRGk13p and random, the ranking is minstd_rand (rank 23) $>$ MRGk13p (rank 22) $>$ Borland's

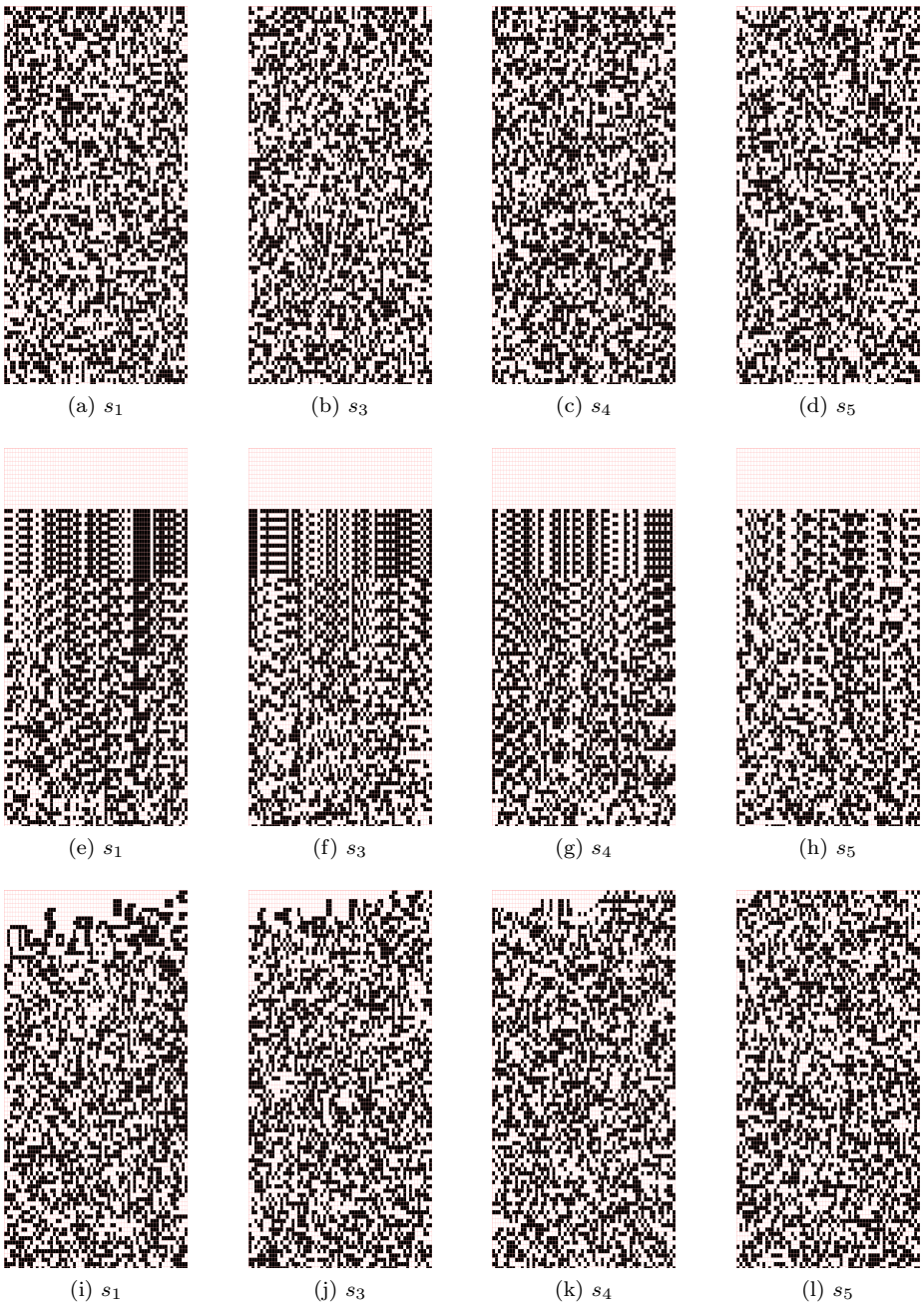


(a) s_1 (b) s_3 (c) s_4 (d) s_5

(e) s_1 (f) s_3 (g) s_4 (h) s_5 (i) s_1 (j) s_3 (k) s_4 (l) s_5

(m) s_1 (n) s_3 (o) s_4 (p) s_5 (q) s_1 (r) s_3 (s) s_4 (t) s_5

Figure 16: Space-time diagram for LFSR258 (16a to 16d), LFSR113 (16e to 16h) and xorshift (16i to 16l), WELL512 (16m to 16p) and WELL1024a (16q to 16t)

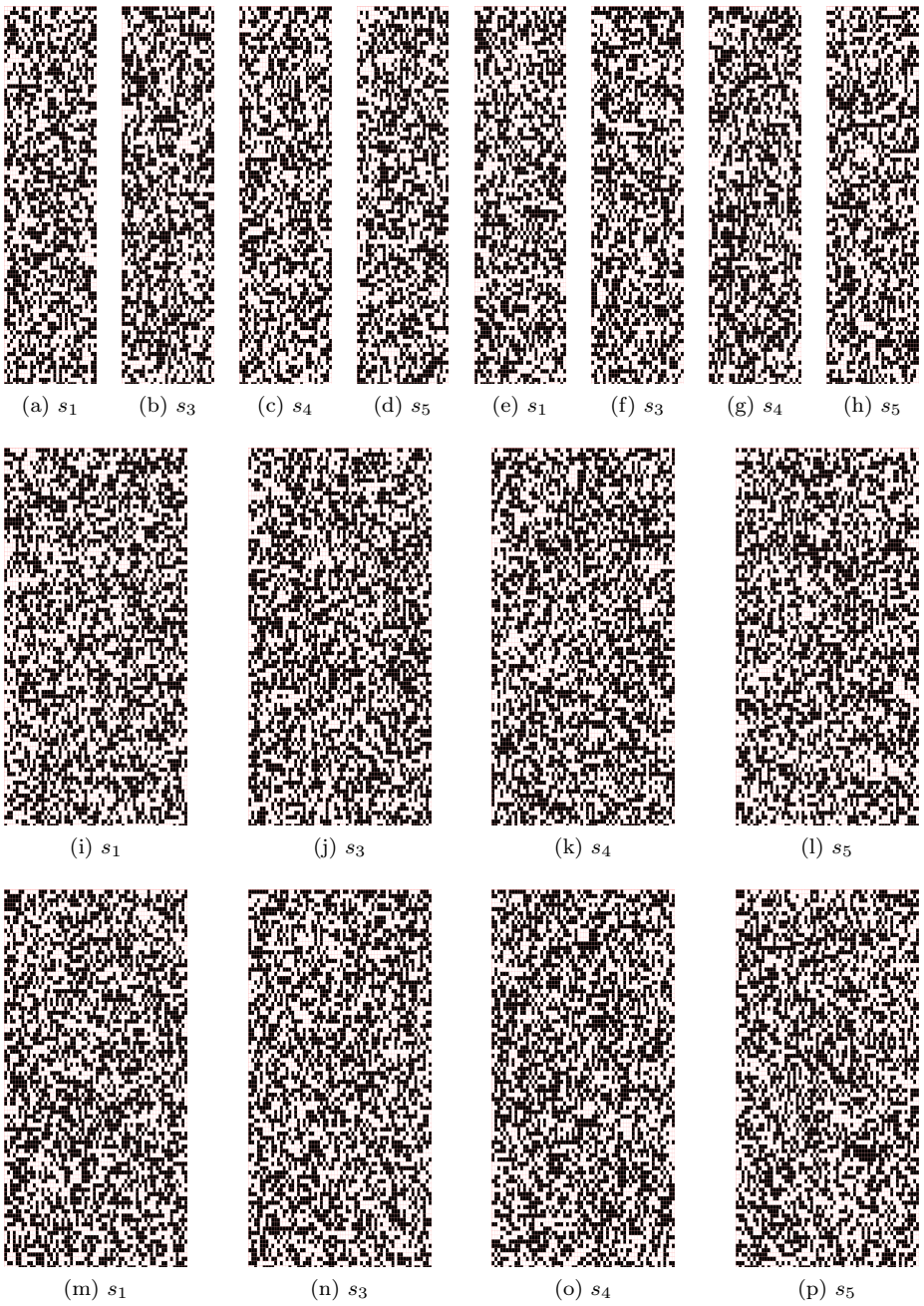


(a) s_1 (b) s_3 (c) s_4 (d) s_5

(e) s_1 (f) s_3 (g) s_4 (h) s_5

(i) s_1 (j) s_3 (k) s_4 (l) s_5

Figure 17: Space-time diagram for xorshift64* (17a to 17d), xorshift1024* (17e to 17h) and xorshift128+ (17i to 17l)

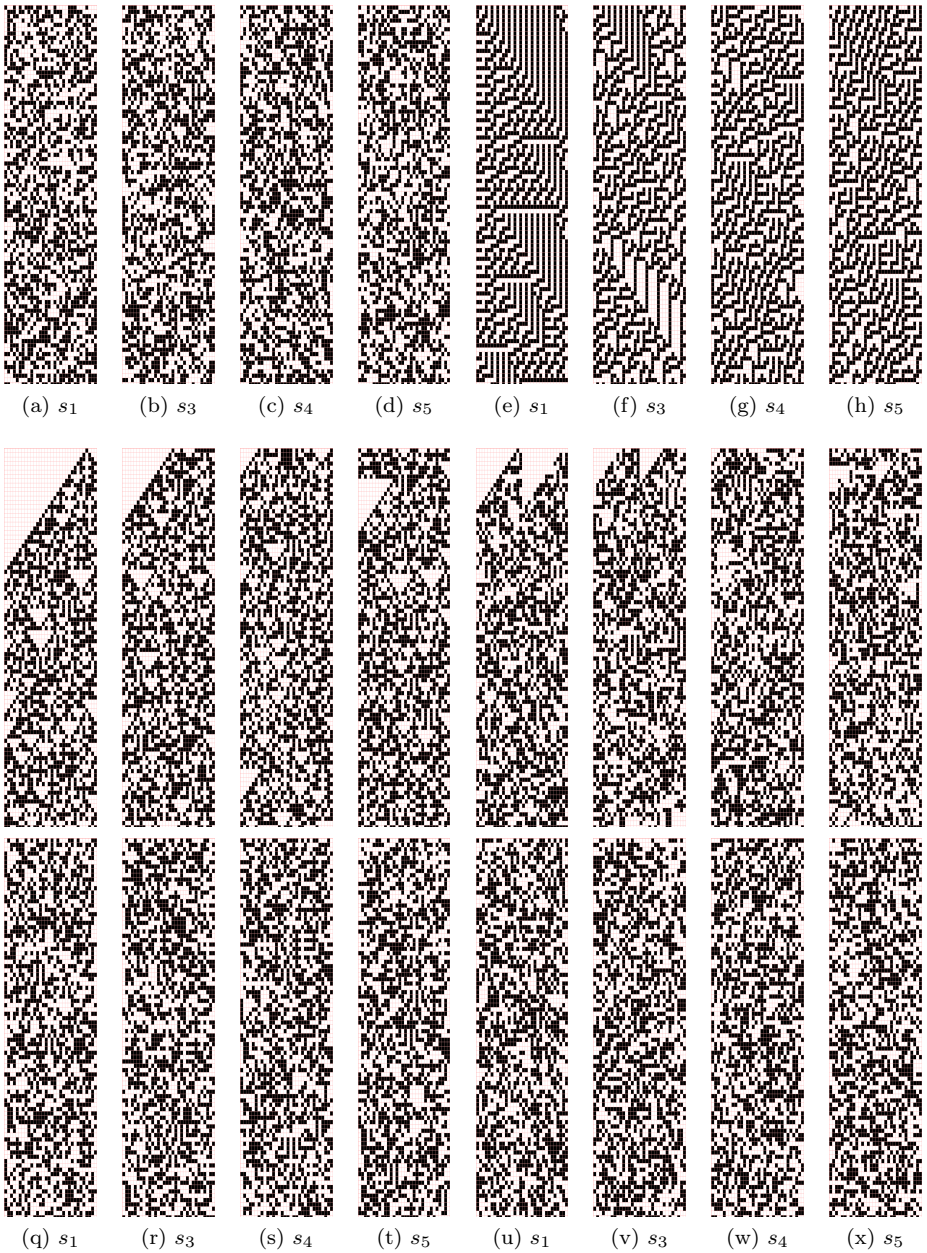


(a) s_1 (b) s_3 (c) s_4 (d) s_5 (e) s_1 (f) s_3 (g) s_4 (h) s_5

(i) s_1 (j) s_3 (k) s_4 (l) s_5

(m) s_1 (n) s_3 (o) s_4 (p) s_5

Figure 18: Space-time diagram for MT19937 32 bit (18a to 18d), SFMT19937 32 bit (18e to 18h), MT19937 64 bit (18i to 18l) and SFMT19937 64 bit (18m to 18p)



(a) s_1 (b) s_3 (c) s_4 (d) s_5 (e) s_1 (f) s_3 (g) s_4 (h) s_5

(q) s_1 (r) s_3 (s) s_4 (t) s_5 (u) s_1 (v) s_3 (w) s_4 (x) s_5

Figure 19: Space-time diagram for rule 30 (19a to 19d) and rule 30-45 (19e to 19h), max-length CA with $\gamma = 0$ (19i to 19l), max-length CA with $\gamma = 1$ (19m to 19p), non-linear CA (19q to 19t) and 3-state CA (19u) to 19x)

LCG (rank 21)> random (rank 20) > rand (rank 19) > lrand (rank 18), where '>' indicates left PRNG has poorer performance than the right one.

- LFSR258 is the worst generator among the selected PRNGs.

Hence, based on the empirical tests, we can finally rank the selected PRNGs into 24 groups, where any group may or may not contain more than one PRNGs. This final ranking is shown in Table 5.

Table 5: Summary of all empirical test results and final ranking

	Name of the PRNGs	Fixed Seeds			Random Seeds		Lattice Test	Space-time Diagram	Ranking		
		Diehard	TestU01	NIST	Average	Range			1^{st} Level	2^{nd} Level	Final Rank
LCGs	MMIX	4-6	16-19	7-8	6.5	2-9	Not Filled	Last 2 bits fixed	8	9	15
	minstd_rand	0	1	1-2	0.38	0-1	Not Filled	last 6 bits fixed	12	14	23
	Borland LCG	1	3	4-5	1.9	1-2	Not Filled	Last 2 bits fixed	11	12	21
	rand()	1	1-3	2-3			Not Filled	More 0s than 1s	11	13	19
	lrand48()	1	2-3	2	1	1	Not Filled	More 0s than 1s	11	13	18
	MRG31k3p	0-1	1-2	1-2	0.9	0-1	Scattered	LSB is 0, block of 0s, dependency on seed	12	14	22
	PCG-32	9-11	24-25	14-15	9.3	6-12	Relatively Filled	Independent of seed	2	4	5
LFSRs	random()	1	1-3	1	1	1	Not Filled	MSB is 0, blocks of 0s	11	13	20
	Tauss88	9-11	21-23	14-15	9.0	0-12	Relatively Filled	Independent of seed, block of 0s	4	7	11
	LFSR113	5-11	6-23	1-15	9.3	6-12	Relatively Filled	Dependency on seed, Block of 0s	7	7	14
	LFSR258	0-1	0-5	0-2	1.8	1-2	Scattered	Pattern	12	14	24
	WELL512a	7-10	23	14-15	8.5	5-11	Relatively filled	First few numbers are fixed with seed dependency	5	6	10
	WELL1024a	9-10	24-25	14-15	9.2	6-11	Relatively Filled	Dependency on seed	3	4	9
	MT19937-32	9-10	25	13-15	9.3	6-12	Relatively Filled	Independent of seed	3	4	6
	MT19937-64	8-11	24-25	15	9.4	6-11	Relatively Filled	Independent of seed	2	3	4
	SFMT-32	9-10	25	15	9.3	6-11	Relatively Filled	Independent of seed	1	1	2
	SFMT-64	9-11	25	15	9.3	6-11	Relatively Filled	Independent of seed	1	1	1
	dSFMT-32	7-11	24-25	13-15	9.3	5-11	Relatively Filled	Independent of seed	5	5	7
	dSFMT-52	5-7	9-11	3	5.97	3-7	Relatively Filled	Less dependency on seed	9	10	12
	XORShift32	2-4	17	2-13	5.5	3-7	Not Filled	Blocks of 0s	9	10	15
	XORShift64*	7-10	25	14-15	8.0	6-11	Relatively Filled	Independent of seed	5	6	8
	XORShift1024*	6-9	20-21	6-15	7.0	4-9	Not Filled	Dependency on seed, Pattern	6	8	14
	XORShift128+	8-10	24-25	14-15	9.4	6-12	Relatively Filled	Dependency on seed for first few numbers	4	4	9
CAs	Rule 30	8-11	24-25	15	10.2	7-12	Relatively Filled	Independent of seed	2	2	3
	Hybrid CA with Rules 30 & 45	0-3	1-8	0-3	2.0	0-3	Not Filled	Pattern	11	12	17
	Maximal Length CA with $\gamma = 0$	0-2	12	10-11	1.6	1-2	Not Filled	Pattern	10	11	16
	Maximal Length CA with $\gamma = 1$	3-4	15-17	14	1.8	1-4	Relatively Filled	Dependency on seed for first few numbers	8	11	13
	Non-linear 2-state CA	5-8	10-12	3-4	7.3	3-9	Relatively Filled	Less dependency on seed	9	9	13
	3-state CA	2-3	11-12	4-6	2.7	1-4	Relatively Filled	Less dependency on seed	10	11	13

VI. Conclusion

In this paper, we have surveyed the evolution of PRNGs over several technologies – LCGS, LFSRs and CA-based. Our target has been to test the well-known PRNGs which are currently in use and check how they actually perform in the similar platform with same seeds. We have used three empirical test-beds – Diehard, TestU01 and NIST for blind statistical tests with some fixed seeds. Using these results, a first level ranking of the PRNGs is done. Then, to enhance this ranking, we have used average case results of the PRNGs on Diehard battery of tests for 1000 seeds. Further, two graphical tests – lattice tests and space-time diagram test have been used to verify this ranking and understand why some PRNGs behave badly. Finally, using the space-time diagrams, a final ranking

has been done in Table 5. According to our tests, SFMT-64 bit generator is the best pseudo-random number generator among all our selected PRNGs.

VII. Acknowledgments

This research is partially supported by Innovation in Science Pursuit for Inspired Research (INSPIRE) under Dept. of Science and Technology, Govt. of India.

References

[1] F. Galton, Dice for statistical experiments, Nature 42 (1070) (1890) 13–14.

[2] L. Tippett, Random sampling numbers, Tracts for computers, Cambridge University Press, 1927.
URL https://books.google.co.in/books?id=DfbuAAAAMAAJ

[3] M. G. Kendall, B. Babington-Smith, Randomness and random sampling numbers, Journal of the Royal Statistical Society 101 (1) (1938) 147–166.

[4] M. G. Kendall, B. Babington-Smith, Second paper on random sampling numbers, Supplement to the Journal of the Royal Statistical Society 6 (1) (1939) 51–61.

[5] J. Von Neumann, 13. various techniques used in connection with random digits, Appl. Math Ser 12 (1951) 36–38.

[6] P. L'ecuyer, Maximally equidistributed combined tausworthe generators, Mathematics of Computation 65 (213) (1996) 203–213.

[7] T. G. Lewis, W. H. Payne, Generalized feedback shift register pseudorandom number algorithm, J. ACM 20 (3) (1973) 456–468.

[8] M. Matsumoto, T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 3–30.

[9] F. Panneton, P. L'Ecuyer, On the xorshift random number generators, ACM Trans. Model. Comput. Simul. 15 (4) (2005) 346–361.

[10] F. Panneton, P. L'Ecuyer, M. Matsumoto, Improved long-period generators based on linear recurrences modulo 2, ACM Trans. Math. Softw. 32 (1) (2006) 1–16.

[11] M. Saito, M. Matsumoto, SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 607–622.

[12] M. Saito, M. Matsumoto, A PRNG Specialized in Double Precision Floating Point Numbers Using an Affine Transition, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 589–602.

[13] R. C. Tausworthe, Random numbers generated by linear recurrence modulo two, Mathematics of Computation 19 (90) (1965) 201–209.

[14] S. Tezuka, On the discrepancy of gfsr pseudorandom numbers, J. ACM 34 (4) (1987) 939–949.

[15] S. Wolfram, Random sequence generation by cellular automata, Advances in applied mathematics 7 (2) (1986) 123–169.

[16] M. Tomassini, M. Sipper, M. Zolla, M. Perrenoud, Generating high-quality random numbers in parallel by cellular automata, Future Gen. Compt. Syst. 16 (1999) 291–305.

[17] S. Vigna, An experimental exploration of marsaglia's xorshift generators, scrambled, ACM Trans. Math. Softw. 42 (4) (2016) 30:1–30:23.

[18] T. E. Hull, A. R. Dobell, Random number generators, SIAM Review 4 (3) (1962) 230–254.

[19] S. K. Park, K. W. Miller, Random number generators: Good ones are hard to find, Commun. ACM 31 (10) (1988) 1192–1201.

[20] F. James, A review of pseudorandom number generators, Computer Physics Communications 60 (3) (1990) 329 – 344.

[21] P. L'Ecuyer, Uniform random number generators: A review, in: Proceedings of the 29th Conference on Winter Simulation, WSC '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 127–134.

[22] P. L'Ecuyer, F. Panneton, Fast random number generators based on linear recurrences modulo 2: Overview and comparison, in: Proceedings of the 37th Conference on Winter Simulation, WSC '05, Winter Simulation Conference, 2005, pp. 110–119.

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

[24] G. Marsaglia, DIEHARD: A battery of tests of randomness, in: http://stat.fsu.edu/~ geo/diehard.html, 1996.

[25] P. L'Ecuyer, R. Simard, Testu01: A c library for empirical testing of random number generators, ACM Trans. Math. Softw. 33 (4) (2007) 22:1–22:40. doi:10.1145/1268776.1268777.
URL http://doi.acm.org/10.1145/1268776.1268777

[26] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, A statistical test suite for random and pseudorandom number generators for cryptographic applications, Tech. rep., DTIC Document (2001).

[27] P. L'Ecuyer, Random numbers for simulation, Commun. ACM 33 (10) (1990) 85–97.

[28] H. U. C. Laboratory, Annals of the Computation Laboratory of Harvard University, Vol. 26, Harvard University Press, 1951.

[29] S. Wolfram, Cryptography with cellular automata, Proc. of Crypto '85 (1985) 429–432.

[30] D. E. Knuth, The Art of Computer Programming – Seminumerical Algorithms, 3rd Edition, Vol. 2, Pearson Education, 2000.

[31] A. Rotenberg, A new pseudo-random number generator, J. ACM 7 (1) (1960) 75–77.

[32] P. A. W. Lewis, A. S. Goodman, J. M. Miller, A pseudo-random number generator for the system/360, IBM Systems Journal 8 (2) (1969) 136–146.

[33] G. S. Fishman, Multiplicative congruential random number generators with modulus 2^{β}: an exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$, Mathematics of Computation 54 (189) (1990) 331–344.

[34] G. S. Fishman, I. Louis R. Moore, An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$, SIAM Journal on Scientific and Statistical Computing 7 (1) (1986) 24–45.

[35] R. P. Brent, On the periods of generalized fibonacci recurrences, Mathematics of Computation 63 (207) (1994) 389–401.

[36] G. Marsaglia, A. Zaman, A new class of random number generators, The Annals of Applied Probability 1 (3) (1991) 462–480.

[37] C. Koç, Recurring-with-carry sequences, Journal of Applied Probability 32 (4) (1995) 966–971.

[38] R. Couture, P. L'ecuyer, Distribution properties of multiply-with-carry random number generators, Mathematics of Computation of the American Mathematical Society 66 (218) (1997) 591–607.

[39] J. Eichenauer, J. Lehn, A non-linear congruential pseudo random number generator, Statistische Hefte 27 (1) (1986) 315–326.

[40] J. Eichenauer-Herrmann, Construction of inversive congruential pseudorandom number generators with maximal period length, Journal of Computational and Applied Mathematics 40 (3) (1992) 345 – 349.

[41] J. Eichenauer-Herrmann, Statistical independence of a new class of inversive congruential pseudorandom numbers, Mathematics of Computation 60 (201) (1993) 375–384.

[42] P. L'Ecuyer, Efficient and portable combined random number generators, Commun. ACM 31 (6) (1988) 742–751.

[43] B. A. Wichmann, I. D. Hill, Algorithm as 183: An efficient and portable pseudo-random number generator, Journal of the Royal Statistical Society. Series C (Applied Statistics) 31 (2) (1982) 188–190.

[44] P. L'Ecuyer, Combined multiple recursive random number generators, Operations Research 44 (5) (1996) 816–822.

[45] P. L'Ecuyer, Good parameters and implementations for combined multiple recursive random number generators, Operations Research 47 (1) (1999) 159–164.

[46] R. E. Nance, C. Overstreet Jr, Some experimental observations on the behavior of composite random number generators, Operations Research 26 (5) (1978) 915–935.

[47] P. Bratley, B. L. Fox, L. E. Schrage, A guide to simulation, Springer Science & Business Media, 1987.

[48] M. E. O'Neill, Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation (HMC-CS-2014-0905).

[49] R. M. et. al., Gnu c library, https://www.gnu.org/software/libc/manual/html_node/Pseudo_002dRandom-Numbers.html#index-pseudo_002drandom-numbers.

[50] S. K. Park, K. W. Miller, P. K. Stockmeyer, Technical correspondence : Response, Commun. ACM 36 (7) (1993) 105–110.

[51] P. L'Ecuyer, R. Touzin, Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$, in: Proceedings of the 32Nd Conference on Winter Simulation, WSC '00, Society for Computer Simulation International, San Diego, CA, USA, 2000, pp. 683–689.

[52] J. P. R. Tootill, W. D. Robinson, D. J. Eagle, An asymptotically random tausworthe sequence, J. ACM 20 (3) (1973) 469–481.

[53] J. P. R. Tootill, W. D. Robinson, A. G. Adams, The runs up-and-down performance of tausworthe pseudo-random number generators, J. ACM 18 (3) (1971) 381–399.

[54] M. Matsumoto, Y. Kurita, Twisted gfsr generators, ACM Transactions on Modeling and Computer Simulation (TOMACS) 2 (3) (1992) 179–194.

[55] M. Matsumoto, Y. Kurita, Twisted gfsr generators ii, ACM Trans. Model. Comput. Simul. 4 (3) (1994) 254–266.

[56] G. Marsaglia, et al., Xorshift rngs, Journal of Statistical Software 8 (14) (2003) 1–6.

[57] R. P. Brent, et al., Note on marsaglia's xorshift random number generators, Journal of Statistical Software 11 (5) (2004) 1–4.

[58] P. L'Ecuyer, J. Granger-Piché, Combined generators with components from different families, Mathematics and Computers in Simulation 62 (3) (2003) 395–404.

[59] P. L'Ecuyer, Tables of maximally equidistributed combined lfsr generators, Mathematics of Computation of the American Mathematical Society 68 (225) (1999) 261–269.

[60] P. L'Ecuyer, Random number generators, http://www-labs.iro.umontreal.ca/~simul/rng/ (2017).

[61] S. Vigna, xoroshiro+ / xorshift* / xorshift+ generators and the prng shootout, http://xoroshiro.di.unimi.it/ (2017).

[62] S. Vigna, Further scramblings of marsaglia's xorshift generators, Journal of Computational and Applied Mathematics 315 (Supplement C) (2017) 175 – 181.

[63] M. Saito, M. Matsumoto, A uniform real random number generator obeying the ieee 754 format using an affine transition, in: 8th International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing (MCQMC '08), 2008, p. 151.

[64] M. Saito, M. Matsumoto, Simd-oriented fast mersenne twister (sfmt): twice faster than mersenne twister, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/#dSFMT (2017).

[65] P. D. Hortensius, R. D. McLeod, W. Pries, D. M. Miller, H. C. Card, Cellular automata-based pseudorandom number generators for built-in self-test, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 8 (8) (1989) 842–859.

[66] M. Matsumoto, Simple cellular automata as pseudorandom m-sequence generators for built-in self-test, ACM Trans. Model. Comput. Simul. (TOMACS) 8 (1) (1998) 31–42.

[67] P. P. Chaudhuri, D. R. Chowdhury, S. Nandi, S. Chattopadhyay, Additive Cellular Automata – Theory and Applications, Vol. 1, IEEE Computer Society Press, USA, ISBN 0-8186-7717-1, 1997.

[68] S. Das, B. K. Sikdar, A scalable test structure for multicore chip, IEEE Trans. on CAD of Integrated Circuits and Systems 29 (1) (2010) 127–137.

[69] S. Das, Theory and Applications of Nonlinear Cellular Automata In VLSI Design, Ph.D. thesis, Bengal Engineering and Science University, Shibpur, India (2007).

[70] P. D. Hortensius, R. D. McLeod, H. C. Card, Parallel random number generation for VLSI systems using cellular automata, IEEE Trans. on Computers C-38 (10) (1989) 1466–1473.

[71] M. Saraniti, S. M. Goodnick, Hybrid fullband cellular automaton/monte carlo approach for fast simulation of charge transport in semiconductors, IEEE Trans. on Electron Devices 47 (10) (2000) 1909–1916.

[72] J. M. Comer, J. C. Cerda, C. D. Martinez, D. H. Hoe, Random number generators using cellular automata implemented on fpgas, in: Proc. of 44th Southeastern Symposium on System Theory (SSST), 2012, 2012, pp. 67–72.

[73] S. Wolfram, Cryptography with Cellular Automata, Advances in Cryptology - Crypto'85, Springer-Verlag 218 (1986) 429–432.

[74] J. Machicao, A. G. Marco, O. M. Bruno, Chaotic encryption method based on life-like cellular automata, Expert Systems with Applications 39 (16) (2012) 12626–12635.

[75] K. Bhattacharjee, D. Paul, S. Das, Pseudo-random number generation using a 3-state cellular automaton, Intl J. Mod. Physics C 28 (06) (2017) 1750078.

[76] W. Pries, A. Thanailakis, H. C. Card, Group properties of cellular automata and VLSI applications, IEEE Trans. on Computers C-35 (12) (1986) 1013–1024.

[77] M. Serra, T. Slater, J. C. Muzio, D. M. Miller, The analysis of one-dimensional linear cellular automata and their aliasing properties, IEEE Trans. on CAD 9 (7) (1990) 767–778.

[78] P. H. Bardell, Analysis of cellular automata used as pseudorandom pattern generators, in: Proc. Intl. Test Conf. 1990, 1990, pp. 762–768.

[79] K. Cattell, J. C. Muzio, Synthesis of one-dimensional linear hybrid cellular automata, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 15 (3) (1996) 325–335.

[80] K. Cattell, S. Zhang, Minimal cost one-dimensional linear hybrid cellular automata of degree through 500, J. Electron. Test.: Theory and Applications 6 (2) (1995) 255–258.

[81] P. H. Bardell, W. H. McAnney, J. Savir, Built-in Test for VLSI: Pseudorandom Techniques, Wiley-Interscience, New York, NY, USA, 1987.

[82] P. H. Bardell, Primitive polynomials of degree 301 through 500, J. of Electron. Test. 3 (2) (1992) 175–176.

[83] A. Compagner, A. Hoogland, Maximum-length sequences, cellular automata, and random numbers, Journal of Computational Physics 71 (2) (1987) 391 – 428.

[84] D. Bhattacharya, D. Mukhopadhyay, D. Roy Chowdhury, A cellular automata based approach for generation of large primitive polynomial and its application to rs-coded mpsk modulation, in: Proc. of Intl. Conf. on Cellular Automata, Research and Industry, ACRI 2006, France, 2006, pp. 204–214.

[85] K. Cattell, M. Serra, The analysis of one dimensional multiple-valued linear cellular automata, in: Proc. of the Twentieth Intl. Symp. on Multiple-Valued Logic, 1990, pp. 402–409.

[86] R. Alonso-Sanz, L. Bull, Elementary cellular automata with minimal memory and random number generation, Complex Systems 18 (2) (2009) 195 – 213.

[87] M. Sipper, M. Tomassini, Generating parallel random number generators by cellular programming, Intl. J. Modern Phys. 7 (2) (1996) 180–190.

[88] S. Guan, S. Zhang, An Evolutionary Approach to the Design of Controllable Cellular Automata Structure for Random Number Generation, IEEE Trans. on CAD 7 (1) (2003) 23–36.

[89] S. Guan, S. K. Tan, Pseudorandom Number Generation With Self-Programmable Cellular Automata, IEEE Trans. on CAD 23 (7) (2004) 1095–1101.

[90] M. Tomassini, M. Sipper, M. Perrenoud, On the generation of high-quality random numbers by two-dimensional cellular automata, IEEE Trans. on Computers 49 (10) (2000) 1146–1151.

[91] S. Guan, S. Zhang, T. Quieta, 2-d CA Variation With Asymetric Neighborship for Psedorandom Number Generation, IEEE Trans. on CAD 23 (3) (2004) 378–388.

[92] S. M. Hosseini, H. Karimi, M. V. Jahan, Generating pseudo-random numbers by combining two systems with complex behaviors, Journal of Information Security and Applications 19 (2) (2014) 149 – 162.

[93] C. G. Langton, Studying artificial life with cellular automata, Physica D 22 (1986) 120–149.

[94] R. J. McEliece, Finite Field for Scientists and Engineers, Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[95] P. L'Ecuyer, R. Simard, S. Wegenkittl, Sparse serial tests of uniformity for random number generators, SIAM Journal on Scientific Computing 24 (2) (2002) 652–668.

[96] P. L'Ecuyer, J.-F. Cordeau, R. Simard, Close-point spatial tests and their application to random number generators, Operations Research 48 (2) (2000) 308–317.

[97] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Info. Theory IT-24 (5) (1978) 530–536.

[98] G. Marsaglia, A current view of random number generators, in: Computer Science and Statistics, Sixteenth Symposium on the Interface. Elsevier Science Publishers, North-Holland, Amsterdam, 1985, pp. 3–10.

[99] L. N. Shchur, J. R. Heringa, H. W. J. Blöte, Simulation of a directed random-walk model the effect of pseudo-random-number correlations, Physica A: Statistical Mechanics and its Applications 241 (3) (1997) 579 – 592.

[100] S. Wolfram, A New kind of Science, Wolfram-Media, 2002.