Computer Science Review 45 (2022) 100471

Contents lists available at ScienceDirect

Computer Science Review

journal homepage: www.elsevier.com/locate/cosrev

Review article

A search for good pseudo-random number generators: Survey and empirical studies

Kamalika Bhattacharjee [a,*], Sukanta Das [b]

[a] Department of Computer Science and Engineering, National Institute of Technology, Tiruchirappalli, Tamilnadu, 620015, India
[b] Department of Information Technology, Indian Institute of Engineering Science and Technology, Shibpur, West Bengal, 711103, India

ARTICLE INFO

Article history:
Received 14 July 2021
Received in revised form 26 November 2021
Accepted 23 April 2022
Available online 24 May 2022

Keywords:
Pseudo-random number generator (PRNG)
Diehard
TestU01
NIST
Lattice test
Space–time diagram
Empirical facts
Ranking

ABSTRACT

This paper targets to search so-called *good* generators by doing a brief survey over the generators developed in the history of pseudo-random number generators (PRNGs), verify their claims and rank them based on strong empirical tests in same platforms. To do this, the genre of PRNGs developed so far are explored and classified into three groups — linear congruential generator based, linear feedback shift register based and cellular automata based. From each group, the well-known widely used generators which claimed themselves to be '*good*' are chosen. Overall 30 PRNGs are selected in this way on which two types of empirical testing are done — blind statistical tests with Diehard battery of tests, battery *rabbit* of TestU01 library and NIST statistical test-suite as well as graphical tests (lattice test and space–time diagram test). Finally, the selected PRNGs are divided into 24 groups and are ranked according to their overall performance in all empirical tests.

Contents

* Corresponding author.
E-mail addresses: kamalika.it@gmail.com (K. Bhattacharjee), sukanta@it.iiests.ac.in (S. Das).

https://doi.org/10.1016/j.cosrev.2022.100471
1574-0137/

1. Introduction

Society, arts, culture, science and even daily life is engulfed by the concept of randomness. One of the major usage of randomness is in generating numbers for diverse fields of practices. 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 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 19th century [1] to early 20th century [2–5] and evolving and getting more powerful ever since [6–17]. Initially, two dominating categories of PRNGs existed — Linear Congruential Generator (LCG) based and Linear Feedback Shift Register (LFSR) based. In 1985, due to inherent randomness quality of some cellular automata (CAs), CAs have also been introduced as a source of randomness [18]. Since then, it has bewitched many researchers to use it as PRNG, see as example [16,19–32].

Usually, latest PRNG claims to be superior to the previous ones. One of the reasons of this claim is based on the PRNG's performance in some statistical tests, like Diehard [33], TestU01 [34], NIST [35] etc. battery of tests, which empirically detect non-randomness in the generated numbers. However, many questions arise in this regard – *How much effective are these 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 than others really correct?* In this work, we target to address these questions. To do this, we have selected all uniform PRNGs developed in the history which have been widely used and are considered to be *good*. Some of these PRNGs are not used now, but in the history of development of PRNG, they have played a significant role making them part of our selection. Similarly, some of them do not have very good randomness quality but are still widely used as general purpose portable generators. So, we include them in our list.

Our target is to test all the PRNGs on the same framework to judge their randomness quality. Like already mentioned, one of the ways of a PRNG to claim its superiority is through statistical tests. Several battery of tests exists which offer to detect non-randomness in a PRNG. However, each of these batteries has some limitations. They (e.g., Diehard) are based on some approximations about distributions (and some of them are false), and the thresholds are taken quite arbitrarily. Therefore, they are *inherently incomplete* in nature. In fact, the PRNGs fools the testbed(s) by exploiting this inherent incompleteness such that the testbed cannot detect the non-randomness in it (which off course exists as it is *pseudo*-random). Nevertheless, as these are the only available metric to judge the PRNGs uniformly, in our work we build a common framework by using three of the well-known existing battery of tests, namely Diehard, TestU01 and NIST.

However, to address the inherent incompleteness of these *blind* test-beds, we also use graphical tests that allow human intervention in the form that an user can visualize if the numbers generated by a PRNG form some patterns. More specifically, we use lattice tests and introduce *space–time diagram* as a graphical randomness test in this work. In the next sections reader can find that, these graphical tests will help us to verify the blind test results. We will also see that using human intervention, space–time diagram will guide us to decide the *goodness* of the PRNG in those times when the blind test results and result from graphical tests do not tally.

Based on the overall performance of all these PRNGs on these selected testbeds and the graphical tests on the same setting, we rank the PRNGs. Obviously this ranking is not the last word, and we agree that linear ranking is not absolute. Because, sometimes the working principles of two PRNGs differ and are not comparable. Nevertheless, we do that by collecting data about existing tests and generators in a systematic way to classify the PRNGs into some groups according to our observation in this common platform.

This paper is organized as follows. In Section 2, the essential properties of the PRNGs are described. Section 3 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 30 currently used PRNGs are selected for empirical testing. The empirical tests and test-beds are described in Section 4. For each of these PRNGs, numbers are generated using the *C* programs available on the Internet. 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. Section 5 depicts the test results of these PRNGs. 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. A relative ranking of these PRNGs based on the empirical results is given in Section 5.3. For any intended application, an user may choose a PRNG according to its rank. Finally, Section 6 concludes the paper.

2. PRNGs and their properties

PRNGs are simple deterministic algorithms which produce deterministic sequence of numbers that appear random. 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 [36]:

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 $u_0 = g(s_0)$.

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

Fig. 1. Cycle structure of PRNGs.

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 outputs of the PRNG are the pseudo-random numbers, and the sequence $(u_i)_{i\geq 0}$ is the pseudo-random sequence.

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 an old state and the sequence will be repeated. This property is common to all sequences where the function f transforms a finite set into itself. This repeating cycle is known as *period*. The length of a period is the smallest positive integer ρ, such that, $\forall n \geq k, s_{\rho+n} = s_n$, here $k \geq 0$ is an integer. If $k = 0$, the sequence is *purely periodic*. Preferably, $\rho \approx |\mathscr{S}|$, or, $\rho \approx 2^b$, if b bits represent each state, that is, the period covers (almost) the whole state space.

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 from distinct cycles may be generated. This situation is shown in Fig. 1.

Nevertheless, only LCGs (described in Section 3.1) can attain the maximum possible period. For LFSRs based generators (described in Section 3.2), the largest achievable period is one less than the maximum period – such a generator is called a *maximal-period* or *maximal-length* generator. Similarly, CAs are also non-maximum period generators.

Every PRNG is classified by the functions f and g, and the seed s_0. Therefore, when a PRNG is observed for its randomness quality, it is considered that the algorithm is not known to the adversary. Following are the desirable properties, which are to be observed in a good PRNG.

1. Uniformity: This property implies that, if we divide the set of possible numbers generated by the PRNG (that is, the range of 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, $e_i = \frac{N}{K}$, where N is the range of the numbers. This ensures that, the generated numbers are equally probable in every part of the number space. Let F_i be the number of samples generated in subinterval i, $1 \leq i \leq K$, then the random variable F_i follows uniform distribution. Obviously, this property is not applicable for PRNGs that does not generate numbers in uniform distribution.

2. Independence: The generated numbers are to be independent of each other; that is, there should not be any correlation between numbers generated in succession. Let $u_t, u_{t'}$ be any two random numbers generated by the PRNG. Then, correlation between them, $Cor(u_t, u_{t'}) = 0$.[1] In that case, any subsequence of numbers have no correlation with any other subsequences. This means, given any length of previous numbers, one cannot 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, that is, $\rho \approx |\mathscr{S}|$. 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 a 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. Mathematically, the random sequence is $u_i = g(s_i)$ where $s_i = f(s_{i-1})$ for all $i \geq 0$, so, when same seed s_0 is provided, same u_is are generated.

5. Consistency: The above properties of the PRNGs are to be independent of the seed. That is, for all s_0, these properties are to be maintained.

6. Disjoint subsequences: There is to be little or no correlation between subsequences generated by different seeds. Let $u = \{\forall i\ u_i$ such that $s_0 = x\}$ and $v = \{\forall j\ u_j$ such that $s_0 = y\}$. Then for any $X \in u$ and $Y \in v$, $Cor(X, Y) \approx 0$. However, this criterion is difficult to achieve in an algorithmic PRNG.

7. 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.

8. 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 PRNGs in an application is not a hindrance to its efficiency.

9. Coverage: The PRNG has to cover the complete output space for a seed. This property is linked with large period. Many PRNGs have less coverage. In case the PRNG has more than one cycle, then it may happen that, only a part of its output space is covered by its seed indicating less period length.

10. Spectral Characteristics: In any long run of a good PRNG, the expected frequency of generation of each number should be same.

11. Cryptographically Secure: To be used in cryptographic applications, the generated numbers have to be cryptographically secure, that is, the next output number cannot be predicted by a polynomial time algorithm (see [37] for more details). This is a desirable property often missing in most of the algorithmic PRNGs.

[1] Correlation of two random variables X and Y is related with variance of them. If $Var(X)Var(Y)$ is positive then it is defined by

$$Cor(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var(X)Var(Y)}}$$

If $Cor(X, Y) = 0$, then X and Y are said to be *uncorrelated*.

Many of these properties are inter-related. For example, if the numbers are not uniform, they are correlated and have identifiable patterns (poor spectral characteristics). 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 and 11 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 test their randomness quality.

3. 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 [38], 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 *Mersenne 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 [18], where he used a cellular automaton (CA) as the source of pseudo-randomness. Therefore, we can classify the PRNGs in three main categories – (1) LCG based, (2) LFSR based and (3) CAs based.

3.1. 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

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

Here $m > 0$ is the modulus, a is the multiplier, c is the increment and x_0 is the starting value or seed; $0 \leq a < m$, $0 \leq c < m$, $0 \leq x_0 < m$, that is, $a, c, x_0 \in \mathbb{Z}_m$ [39]. 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 becomes $x_0 + 1, x_0 + 2, x_0 + 3, \ldots$, which, off course, is not random. Similarly, if $a = 0$, the case is even worse. 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 [39]:

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.

A good maximum-period LCG is Knuth's LCG `MMIX` [39] where $a = 6364136223846793005$, $m = 2^{64}$ and $c = 1442695040888963407$. The PRNGs used in computer programming are mainly LCGs having maximum period. Some popular examples are `rand` of GNU C Library [40] where $a = 1103515245$, $c = 12345$ and $m = 2^{31}$, and `lrand48` of same library where $a = 25214903917$, $c = 11$ and $m = 2^{48}$. Another well-known LCG is `drand48` of GNU C Library which is similar to `lrand48`, except it produces normalized numbers. Borland LCG is also a well-liked PRNG having $a = 22695477$, $c = 1$ and $m = 2^{32}$.

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):

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

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 x_n is relatively prime to m for all n, the length of the period is limited to the number of integers between 0 and m that are relatively prime to m [39]. Now, if $m = p^e$, where p is a prime number and $e \in \mathbb{N}$, Eq. (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 p^e, then this condition turns down 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. It can be at maximum $m - 1$ [39], 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 [41–43]. C++11's `minstd_rand` [44,45], a good PRNG, is an MCG where $a = 48271$ and $m = 2^{31} - 1$. However, the MCGs perform unsatisfactorily in spectral tests [39]. So, 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. Here, $x_0, \ldots, x_{k-1}$ are arbitrary but not all zero. For these recurrences, the best result can be derived when m is a large prime. In this case, according to the theory of finite fields, multipliers $a_1, \ldots, a_k$ exist, such that, the sequence of Eq. (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* [39]. That is, if and only if, the root of $P(z)$ is a primitive element of the Galois field with p^k elements.[2] A generator with such recurrence is called *multiple recursive generator* (MRG) [36].

[2] A nonzero polynomial $P(z)$ is *irreducible* if it cannot be factored into two non-constant polynomials over the same field. 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 $\min_{n\in\mathbb{N}}\{n \mid P(z) \text{ divides } z^n - 1\} = m^k - 1$. In this case, $P(z)$ has a root α in $\mathbb{F}(m^k)$ such that, $\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{m^k-2}\}$ is the entire field $\mathbb{F}(m^k)$.

A variant of MRG is the additive *lagged-Fibonacci* generators [46], 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 $x_0, \ldots, x_{r-1}$ are initially given and not all are 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 length $2^{w-1}(2^r - 1)$. The PRNG, proposed in [46], uses this type of trinomials. Some extensions of lagged-Fibonacci generators are the PRNGs named *add-with-carry* (AWC) and *subtract-with-borrow* (SWB) generators [47], *Recurring-with-carry* generators [48], *multiply-with-carry* (MWC) generators [49] etc.

Another type of generators, named as *inversive congruential generators* (ICGs) are proposed in [50–52]. 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 $\{0, 1, \ldots, p-1, \infty\}$ and the x_n^{-1} is the inverse of x_n, defined as: $0^{-1} = \infty$, $\infty^{-1} = 0$, otherwise $x^{-1}x \equiv 1 \pmod{p}$. For the purpose of implementation, one can consider $0^{-1} = 0$, as 0 is always followed by ∞ and then by c in the sequence. Here, for many choices of a and c, maximum period length $p + 1$ is attainable [39].

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 [53–56]. Several combining techniques are suggested in the literature, like addition using integer arithmetic [53,54], shuffling [57], bitwise addition modulo 2 [58] etc.

In [55], 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. Let us consider $J \geq 2$ component MRGs where each MRG satisfies the recurrence

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

having order k_j and coefficients $a_{j,i}$ $(1 \leq i \leq k)$. Here, the moduli m_js are pairwise relatively prime and each MRG has period $\rho_j = m_j^{k_j} - 1$. Now, two combined generators can be defined [55]:

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

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

Here, $\delta_1, \delta_2, \ldots$ are arbitrary integers such that each δ_j is relatively prime to m_j. The MRGs of Eqs. (6) and (7) are approximately equivalent to each other. Further, it can be shown that, the MRGs of Eqs. (5) and (6) are equivalent to an MRG of Eq. (3) with $m = \prod_{j=1}^{J} m_j$ and period length $= lcm(\rho_1, \ldots, \rho_J)$. A well-known example of combined MRG is `MRG31k3p` [59] where two component MRGs of order 3 are used having the following parameters:

$$m_1 = 2^{31} - 1, \ a_{1,1} = 0, \ a_{1,2} = 2^{22}, \ a_{1,3} = 2^7 + 1$$

$$m_2 = 2^{31} - 21069, \ a_{2,1} = 2^{15}, \ a_{2,2} = 0, \ a_{2,3} = 2^{15} + 1$$

Here, for ease of implementation, each component MRG has two non-zero coefficients of the form 2^q and $2^q + 1$. The combined MRG follows Eq. (7) and its period length is approximately 2^{185}.

Another technique of improving randomness quality of a generator is to use a randomized algorithm over the outputs of a single LCG. This randomized algorithm is an efficient permutation function/hash function in [60], which introduces the family of generators as *permuted congruential generator* or *PCG*. Here, several operations are performed on the outputs of a fast 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. A good implementation is `PCG-32`, which produces 32-bit output and has a period length 2^{64}. Here, the multiplier is 6364136223846793005 and increment is taken as 1.

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, $S = \{\mathbf{X} = (x_1, \ldots, x_k)^T \mid 0 \leq x_0, \ldots, x_k < m\}$ is the set of k-dimensional vectors with elements in $F = \{0, 1, \ldots, m-1\}$, $\mathbf{A} = (a_{ij})$ 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 [36]. If $k = 1$, the recurrence of Eq. (8) reduces to Eq. (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 $\mathcal{O}(\log v)$ time and applying a matrix–vector multiplication $\mathbf{X}_{i+v} = (\mathbf{A}^v \pmod{m})\mathbf{X}_i \pmod{m}$ [36]. Moreover, using this matrix, any LCG of order k can be expressed by an MCG of order $k + 1$: modify $\mathbf{A}$ to add $\mathbf{C}$ as its $(k + 1)$th column and a $(k + 1)$th row 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, the MCGs 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 modulo m arithmetic. Nevertheless, a polynomial of Eq. (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 $\mathbf{X_n} = (x_n, \ldots, x_{n-k+1})^T$, MCG of Eq. (9) is converted to recurrence of MRG (Eq. (3)), where $\mathbf{X_n}$ obeys the recursion:

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

3.2. LFSR based PRNGs

If modulus of the linear recurrence (Eq. (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

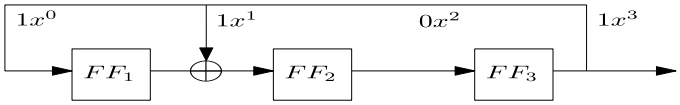


Fig. 2. A schematic diagram of 3-bit LFSR with characteristic polynomial $P(x) = 1 + x + x^3$.

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 Fig. 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. The initial states of all the flip-flops together is the seed of the LFSR. Likewise MCGs, seed of a LFSR, should always be a non-zero value, otherwise, the sequence degrades to zeros.

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 PRNGs, 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 genre of 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 Eq. (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 $a_1, a_2, \ldots, a_{k-1} \in \mathbb{F}_2$. This recurrence can be implemented on a linear feedback shift register (LFSR). However, the random number is represented by

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

which is a number with L consecutive bit sequence of Recurrence (12) with successive u_ns spaced s bits apart [13]. Here s and L are positive integers. 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. A popular example of this type of PRNG is `random` of GNU C Library, which returns numbers between 0 to 2147483647 having period $\rho \approx 16 \times (2^{31} - 1)$

Initially LFSR-based Tausworthe generators used primitive trinomials [13,61]. In [62], it is shown that, any Tausworthe generator that uses primitive trinomials of form

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

as the characteristic polynomial can be represented by a simple linear recurrence in $\mathbb{F}(2^p)$. Moreover, likewise combined MRGs, *combined* Tausworthe generators have also been proposed [6]. It consists of $J \ge 2$ Tausworthe generators with primitive characteristic polynomials $P_j(z)$ of degree k_j where s_j is mutually prime to $2^{k_j} - 1$, $1 \le j \le J$. The sequence is denoted by $x_{j,n}$ (see Eq. (5) with modulus 2) and random number by $u_{j,n} = \sum_{l=1}^{L} x_{j,ns_j+l-1} 2^{-l}$. L is usually the word size of the computer. 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 XOR operation. As discussed before in Section 3.2, this generator has period $\rho = lcm(2^{k_1} - 1, 2^{k_2} - 1, \ldots, 2^{k_J} - 1)$, if the polynomials $P_j(z)$ are pairwise relatively prime, that is, every pair of polynomials have no common factor. `Tauss88` [6] is such a generator where three component PRNGs are used with order $k_1 = 31$, $k_2 = 29$, and $k_3 = 28$ respectively. This PRNG has period length $\rho = (2^{31} - 1)(2^{29} - 1)(2^{28} - 1) \approx 2^{88}$ and returns either 32-bit unsigned integer or its normalized version. The C code for this PRNG is accessible from https://github.com/LuaDist/gsl/blob/master/rng/taus.c. There are two other good combined Tausworthe generators, named `LFSR113` and `LFSR258`. For `LFSR113`, number of component PRNGs $J = 4$ with period length $\rho \approx 2^{113}$. However, for `LFSR258`, $J = 5$ and period length $\rho \approx 2^{258}$. Both these PRNGs return 64 bit normalized numbers; for `LFSR113` the numbers are normalized by multiplying the unsigned long integer output of the LFSR with $2.3283064365387 \times 10^{-10}$ and for `LFSR258`, the same is done by multiplying the unsigned 64-bit output of the LFSR with $5.42101086242752217003726 4 \times 10^{-20}$. C codes for these two PRNGs are available at [63].

A generalization of LFSR (GFSR) has been proposed in [7] to improve the quality of the PRNGs. 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 Eq. (12). 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 [64,65]. This generator is same as GFSR, but, its linear recurrence is

$$\mathbf{X}_{l+n} = \mathbf{X}_{l+m} \oplus \mathbf{X}_l \mathbf{A}, \quad (l = 0, 1, \ldots) \tag{17}$$

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 $\mathbb{F}(2^w)$. Usually, matrix $\mathbf{A}$ is chosen as Eq. (11). The seed is the tuple $(\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_{n-1})$ with at least one non-zero value.

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

$$\mathbf{X_n} = \mathbf{A}\mathbf{X}_{n-1} \tag{18}$$

$$\mathbf{Y_n} = \mathbf{B}\mathbf{X}_n \tag{19}$$

$$u_n = \sum_{l=1}^{w} y_{n,l-1} 2^{-l} \tag{20}$$

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 $\mathbf{X}_n = (x_{n,0}, \ldots, x_{n,k-1})^T$, the w-bit output vector is $\mathbf{Y}_n = (y_{n,0}, \ldots, y_{n,k-1})^T$ and output at step i is $u_n \in [0, 1)$. All the operations in Eqs. (18) and (19) are modulo 2 operations. The characteristic polynomial of matrix $\mathbf{A}$ is same as Eq. (10) with modulus 2:

$$P(z) = det(z\mathbf{I} - \mathbf{A}) = (z^k - \sum_{i=1}^{k} a_i z^{k-i}) \tag{21}$$

where $a_j \in \mathbb{F}_2$ and $\mathbf{I}$ is the identity matrix. If $a_k = 1$, this recurrence is purely periodic (see Section 2) 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 $\mathbb{F}_2$.

The matrix $\mathbf{B}$ is usually used for tempering to improve *equidistribution*[3] property of the PRNG by elementary bitwise transformation operations, like XOR, AND and shift [65]. 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]:

$$\mathbf{X}_{k+n} = \mathbf{X}_{k+m} \oplus (\mathbf{X}_k^u | \mathbf{X}_{k+1}^l)\mathbf{A} \tag{22}$$

Here, r, m, w are positive integers with $0 \le r \le w-1$, m $(1 \le m \le n)$ is middle term and r is separation point of one word. $\mathbf{A}$ is a $w \times w$ matrix (like Eq. (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 $\mathbf{X}_{k+1}$. 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:

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

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

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

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

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 state transition is directed by a linear transformation

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

on an array of size $p = nw - r$ (or an $(n \times w - r)$ array with r bits missing at the upper right corner). Here, $\mathbf{I}_j$ is a $j \times j$ identity matrix, $\mathbf{0}$ is the zero matrix and

$$\mathbf{S} = \begin{bmatrix} \mathbf{0} & \mathbf{I}_r \\ \mathbf{I}_{w-r} & \mathbf{0} \end{bmatrix}\mathbf{A} \tag{23}$$

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 $2^p - 1 = 2^{nw-r} - 1$. A commonly used example of Mersenne Twister is MT19937 [8] having a period $\rho = 2^{19937} - 1$. There are 32-bit and 64-bit word size variations of it. The working principle of Mersenne Twister is represented in Fig. 3. Here, a 32-bit integer

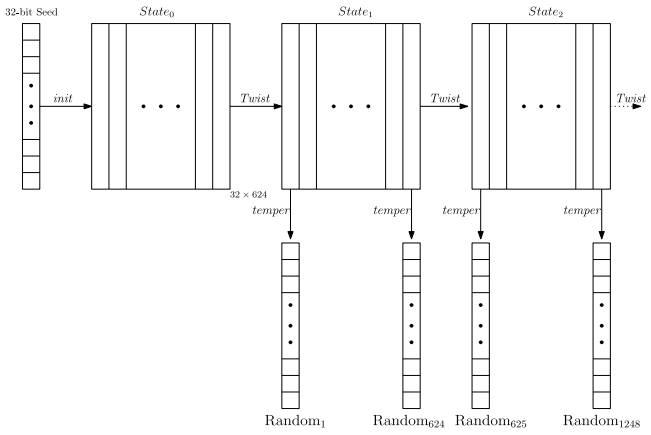


Fig. 3. Working principle of Mersenne Twister.

is taken as seed which is used to initialize the starting state of the 19937 length TGFSR. Every time *twist* function is used to generate the next state of the PRNG (Note that, $32 \times 624 = 19936$). But, instead of directly producing each state as output, each of the internal states of Mersenne Twister is divided into 624 32-bit words where each word (integer) is tempered using the *temper* function before getting produced as output. So, after each application of *twist*, the current state of the TGFSR is used to generate 624 32-bit integers consecutively. Once these 624 numbers are exhausted, then only again the state of the TGFSR is updated using the twist function.

In case of MT19937-32 (32-bit), the associated parameters are $(w, n, m, r) = (32, 624, 397, 31)$, $\mathbf{a} =$ 0x9908B0DF, $u = 11$, $s = 7$, $\mathbf{b} =$ 0x9D2C5680, $t = 15$, $\mathbf{c} =$ 0xEFC60000, $l = 18$ and number of terms in the characteristics polynomial is 135. However, for MT19937-64 (64-bit), $(w, n, m, r) = (64, 312, 156, 31)$, $\mathbf{a} =$ 0xB5026F5AA96619E916, $u = 29$, $s = 17$, $\mathbf{b} =$ 0x71D67FFFEDA6 000016, $t = 37$, $\mathbf{c} =$ 0xFFF7EEE00000000016 and $l = 43$. These parameters ensure that the period is the Mersenne prime $2^{19937} - 1$ which is longer than any other random number generator proposed before and is one of the reasons for the popularity of Mersenne Twister.

In [11], a variation of Mersenne Twister named single instruction multiple data (SIMD)-oriented Fast Mersenne Twister (SFMT) is proposed. It 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. A popular implementation is SFMT19937 which has same period as MT19937. It can generate both 32-bit and 64-bit unsigned integer numbers. Further, there is a variation of SFMT specialized in producing double precision floating point numbers in IEEE 754 format. This PRNG is, in fact, named as double precision floating point SFMT (dSFMT) [12]. Two versions of it are available - dSFMT-32 and dSFMT-52 [66]. For dSFMT-52, the output of the 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.

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, j are unique integers such that $r > 0$ and $0 \le j < w$, and it is primitive over $\mathbb{F}_2$. Two such good generators are WELL512a and WELL1024a [63]. In case of WELL512a, 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 $r =$

[3] A sequence of real numbers $\{x_n\}$ is equidistributed on an interval $[a, b]$ if the probability of finding x_n in any subinterval is proportional to the subinterval length. Equidistribution of t-dimensional unit hypercube into 2^{kl} cubic cells is defined as the set of all cubic cells in $[0, 1)^k$ with side length 2^{-l} having coordinates of the corners as multiples of 2^{-l} [13].

0 with period length $\rho = 2^{1024} - 1$. The return values for these PRNGs are 32-bit numbers normalized by multiplying with $2.32830643653869628906 \times 10^{-10}$. WELL generators follows the general Eqs. (18) and (19).

In [67], Marsaglia has 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 XOR on the original block with this shifted block. In general, a xorshift generator has the following recurrence relation [9,68]:

$$\mathbf{v}_n = \sum_{j=1}^{t} \tilde{\mathbf{A}}_j \mathbf{v}_{n-m_j} \pmod 2 \tag{24}$$

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 $\mathbf{x}_n = (\mathbf{v}_{n-r+1}^T, \ldots, \mathbf{v}_n^T)^T$ where $\mathbf{v}_n = (v_{n,0}, \ldots, v_{n,w-1})^T$ and output is $u_n = \sum_{l=1}^{w} v_{n,l-1} 2^{-l}$ where $r = \max \lim_{1 \leq j \leq t} m_j$. This generator converts into the general LFSR PRNG of Eqs. (18) and (19), if

$$\mathbf{A} = \begin{bmatrix} 0 & \mathbf{I} & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{I} \\ \mathbf{A}_r & \mathbf{A}_{r-1} & \cdots & \mathbf{A}_1 \end{bmatrix} \tag{25}$$

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. Matrix $\mathbf{A}$ 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.

Marsaglia's `xorshift32` generator [67] uses 3 xorshift operations — 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. There are three other good xorshift generators – `xorshift64*`, `xorshift1024*M_8` and `xorshift128+`. In `xorshift64*` generator, the returned number is the current state perturbed by a non-linear operation, which is multiplication by 2685821657736338717 [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`, the multiplier is 1181783497276652981 and shift parameters are 31,11, and 30. In both cases, the generated numbers are 64-bit unsigned integers. In `xorshift128+`, the outputs are 64-bits and two previous output states are added to get the result [69].

Although most of the above mentioned generators are linear, many researchers have developed LFSR based PRNGs by combining these with some non-linear operations [17,34,70] to scramble the regularity of linear recurrence. For example, in [70], 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. The *xorshift** PRNGs, discussed above, are a well-known example.

3.3. 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.[4] 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. Each of these combination of neighborhoods to R is also named as *Rule Min Term* or *RMT*, usually represented by its decimal equivalent. For example, Table 1 shows a rule for 2-state 3-neighborhood 1-D CAs, where each of these neighborhood combinations 000 to 111 is an RMT. Hence, any rule can be represented by the string corresponding to the next state values of the RMTs. Collection of the states of all cells at a given time is called *configuration* of the CA. So, during evolution a CA hops from one configuration to another.

In [18], CAs were introduced as a source of pseudo-randomness. Here, an infinite array of cells is considered, where each cell can take either of the states 0 or 1 depending on the present states of its left neighbor, itself and its right neighbor. Such a CA is an example of 1-dimensional 3-neighborhood 2-state CAs, popularly known as, *elementary CAs* or *ECAs*. For instance, the ECA rule chosen as source of randomness in [18] is rule 30[5] (decimal of the string 00011110) 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.

According to the original proposal, only a single bit is collected from each configuration of the CA where number of cells is infinite. However, to implement as PRNG, one needs to consider only a finite number of cells having boundaries. Two boundary conditions are usually considered – (1) null boundary, where the boundary cells are connected to null or 0 state (see Fig. 4(a)), (2) periodic boundary, where the boundary cells are neighbors of each other (see Fig. 4(b)).

So, to use as a PRNG, one may abuse [18] slightly by running the CA under periodic boundary condition, and collecting 32 bits from 32 consecutive configurations to report a single 32-bit integer.[6]

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. Fig. 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 [19,20,26,27,74–76], Monte-Carlo simulations [77], Field Programmable Gate Arrays (FPGAs) [78,79], cryptography [80–82] etc. However, most of these works considers 1-dimensional CAs where the cells follow 3-neighborhood dependency. So, in this paper we have taken only such CAs based PRNGs for comparison with the other class of generators.

Randomness of a CA-based PRNG is, in general, effected by its transition rule, cell size, initial configuration (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.

[4] 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, $\mathcal{N} = (\vec{v_1}, \vec{v_2}, \ldots, \vec{v_m})$ identifies m neighbors of each cell and $\mathcal{R} : \mathcal{S}^m \to \mathcal{S}$ is the rule of the automaton. For more details on CAs, interested reader may see [71,72].

[5] Historically, the notion of representing these rules by the decimal equivalent of the binary string corresponding to the neighborhood combinations $\mathcal{R}(x, y, z)$, $x, y, z \in \{0, 1\}$ was introduced in [73], where $\mathcal{R} : \{0, 1\}^3 \to \{0, 1\}$ is a rule.

[6] We have taken the number of cells as 101 and next states of the middle cells are collected to generate 32-bit numbers.

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

Neighborhood combination (RMT)	111 (7)	110 (6)	101 (5)	100 (4)	011 (3)	010 (2)	001 (1)	000 (0)	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

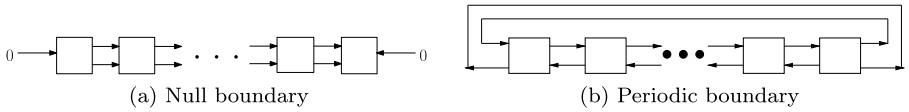


Fig. 4. Boundary conditions for 1-D finite CAs. Arrows pointing to a cell indicate the dependencies of the cell. Here all the CAs use 3-neighborhood dependency.

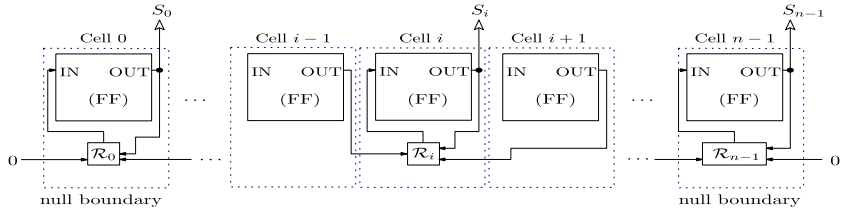


Fig. 5. Hardware implementation of an n-cell ECA under null boundary condition.

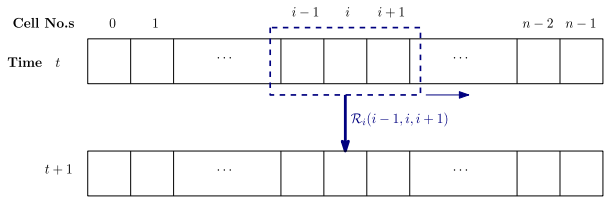


Fig. 6. Next state calculation for an n-cell hybrid CA with 3 neighborhood dependency.

The rule of the CA is chosen as *autoplectic*[7] such that feature extraction require more sophisticated computation than the simple run of the CA. For instance, ECA rule 30 used in [18] is a autoplectic rule. However, to improve the cycle length of the CA and introduce more complexity in the system, non-uniformity in the local rule is instigated [84]. In this case, instead of using a single rule $\mathcal{R}$, a vector $\mathscr{R} = \langle \mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{n-1} \rangle$, called *rule vector*, is used, where n is the number of cells and the ith cell uses rule $\mathcal{R}_i$. The next state calculation in this case, is governed by $\mathcal{R}_i$ for each i (see, for example, Fig. 6). Such a CA where different cells may take different rules is called a *hybrid* or *non-uniform* CA. For instance, in [19], a rule vector $\mathscr{R} = \{30, 45\}^{16}$ is used under periodic boundary condition to generate 32-bit numbers from the whole configuration of the CA. Here, if $\mathcal{R}_0 = \mathcal{R}_1 = \cdots = \mathcal{R}_{n-1}$, the CA is uniform CA. In Fig. 5, if combinational logic circuits for each cells are different, the ECA is a hybrid ECA.

Many other ways are also proposed to generate random numbers from the configuration of a CA. In [19], concept of site spacing (output number is collected from cells spaced by γ distance) and time spacing (output numbers are taken α time steps apart) are introduced (see Figs. 7 and 8). If $\gamma = 0$, the whole configuration of the CA is treated as a number.

However, to further improve the period of the PRNGs, a special type of non-uniform CAs are introduced where the rules are *linear*.[8] For example, the ECAs 90 and 150 are linear CAs as they 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 ith cell at time t. Linear CAs can be easily characterized by using standard algebraic tools [75]. For instance, a binary n-cell linear CA can be represented by an $n \times n$ characteristics matrix (T) operating on $\mathbb{F}(2)$. In this matrix, the ith row represents the dependency of the ith cell to its neighbors. The characteristics matrix (T), in this case, is formed as:

$$T[i,j] = \begin{cases} 1 \text{ if the next state of the } i\text{th cell depends on the} \\ \quad \text{present state of the } j\text{th cell} \\ 0 \text{ otherwise} \end{cases} \quad (26)$$

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 [85]. 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* which has to be defined over null-boundary condition.

[7] This term, first coined by Stephen Wolfram indicates intrinsic randomness where without any outside input, randomness can be generated inside the system by the system itself [83]. In this case, even simple initial conditions can derive randomness.

[8] If all rules of the CA can be expressed by a linear function, that is,

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

where $c_j \in \mathcal{S}$ is a constant and a_j is the state of the jth 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.

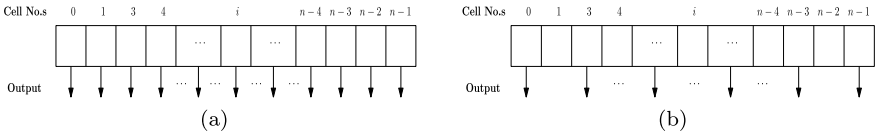


Fig. 7. Site Spacing for even cell length n. Here, $\gamma = 0$ for Fig. 7(a) and $\gamma = 1$ for Fig. 7(b). The random numbers are collected from the cells with arrows.

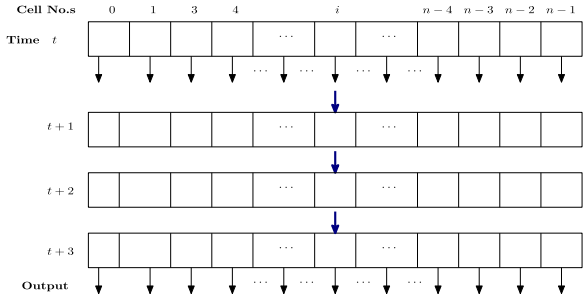


Fig. 8. Time spacing with no site spacing. Here, α is taken as 2.

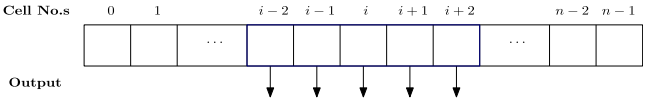


Fig. 9. Window of size 5 taken from the middle cells.

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 [21,85]. 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 [86]. In [87], a list of maximal-length CAs for each degree from 1 to 500 is synthesized for the corresponding primitive polynomials given in [88,89]. 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 [19–22,24,90]. For example, in [19], the maximal length CA with rule vector $\mathcal{R}$ = (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) is used. To implement this PRNG, we have considered two types of site spacing – 1 site spacing (that is, $\gamma = 1$) and no site spacing ($\gamma = 0$). For 1 site spacing, two consecutive output sequences are concatenated to get one 32-bit number, whereas, for no site spacing, the whole configuration of the CA at each time instant is treated as a 32-bit number.

However, due to difficulty in finding the primitive polynomial required for a maximal-length CA, non-linear CAs were introduced as PRNGs [26,27]. For example, in [26], an algorithm is given to select a (2-state 3-neighborhood) non-uniform nonlinear CA as the random number generator. 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}. Each of its 45-bit configuration is considered as a number. Some other works of using hybrid CAs are [19,20,25]. In [25], 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.

In some recent works, the numbers are generated by a small window ($w < n$) out of the n cells using 3-neighborhood 1-D CAs having more than 2-states per cell. For example, in [91], a 3-state CA with local rule $\mathcal{R}$ = 120021120021021120021021210[9] under periodic boundary condition is used. The base-3 numbers, observed through the window, are considered as pseudo-random numbers (Fig. 9). To generate ternary strings of length 20 and 40, the CA sizes can be taken as 51 and 101 with window length w = 20 and 40 respectively. When converted, these ternary numbers are equivalent to 32-bit and 64-bit numbers respectively.

In [92], 1-dimensional 3-neighborhood decimal CAs are exploited as source of randomness. Here, it is argued that the CAs which are *chaotic*[10] while defined over infinite lattice, but have no *self-similar* and *self-organizing* patterns can be used as good source of randomness. Further, this paper also identifies a list of desirable properties for these finite CAs to be good candidates as PRNGs, such as, *balancedness*, *information flow*, *asymmetric configuration space*, large cycle length etc. (see Ref. [92] for more details). A heuristic synthesis algorithm is reported to generate candidate CAs following these properties. One can observe that, a rule of decimal CAs has 1000 RMTs which may be cumbersome to represent using, say, format like Table 1. To simplify the representation, the RMTs of a rule is divided into 100 non-overlapping strings where each string contains 10 consecutive RMTs. These strings are named as $Sibl_0$ to $Sibl_{99}$. For example, the RMTs 0(000), 1(001), 2(002), 3(003), 4(004), 5(005), 6(006), 7(007), 8(008), 9(009) in the order 9, 8, . . . , 0 forms $Sibl_0$. Similarly $Sibl_i$ contains the RMTs $\langle 10j+9, 10i+8, \ldots, 10j\rangle$. However, here the values of $Sibl_i$ are used as a string without comma for ease of implementation. Therefore, a rule is a concatenation of the strings $Sibl_{99}\ Sibl_{98} \cdots Sibl_0$. A simple synthesis algorithm is also provided in [92] which generates the complete rule from a small fragment of 10 RMTs given as a permutation of '0123456789'; for the sake of completeness, the algorithm is reproduced as Algorithm 1.

ALGORITHM 1: *GenerateRuleFromPermutation*

Input : A permutation of '0123456789' (*initPerm*)
Output: A CA rule

```
Step 1  Set Sibl_0 ← initPerm;
Step 2  for i = 1 to 9 do
            Set Sibl_i ← Sibl_0 >> i ; // Set Sibl_i with i times circular right
              shift of Sibl_0
        end
Step 3  for j = 10 to 99 do
            Set Sibl_j ← Sibl_{j (mod 10)} >> (j/10) ; // Set Sibl_j with j/10 times
              circular right shift of Sibl_k where j (mod 10)
        end
Step 4  for i = 0 to 8 do
            for j = i + 1 to 9 do
                Set Sibl_{j*10+i} ← Sibl_{i*10+j} ; // The RMTs of Sibl_{j*10+i} and Sibl_{i*10+j}
                  have same values
            end
        end
```

[9] Here, the rule signifies the string corresponding to all possible RMTs starting from 222 up to 000.

[10] Chaos is defined over infinite lattice. However, while constructing a PRNG, we need a finite system.

For example, when 1632405789 is given as input to Algorithm 1, first RMTs of $Sibl_0$ are assigned, that is, RMT 9 gets 1, RMT 8 gets 6, and so on. Similarly, by cyclic permutation of this string, the other strings are generated. Finally, we can get the actual rule of the decimal CA by concatenating those strings. It is shown that, rules synthesized in this way also follow the desirable properties mentioned earlier to be a good source of randomness. Further, to extract random numbers from the configurations of the CAs, two window-based schemes are proposed. A specialty of this generator is it can create random decimal numbers of any arbitrary length as well as random binary numbers of any length as multiple of 32. For our purpose, we have taken the binary output generator version of this PRNG with rule 1632405789 and generated 32 as well as 64 bits depending on the size of the seed.

Sometimes, optimization techniques are applied to the CAs to improve their randomness qualities. In [16,23], genetic algorithms are applied to co-evolve hybrid CAs for generating random numbers. For example, in [23], cellular programming is used over 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 to develop a PRNG. In [28], numbers are generated using evolutionary multi-objective optimization techniques on controllable CA, whereas, in [29], 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. Another example of self-programmable CAs used for implementation of PRNG on FPGA is [93]. In [94], quantum-dot cellular automata is explored in developing a pseudo random number generator based cryptographic architecture. In [22,30–32,95–100], 2-dimensional CAs are used as the PRNGs. For example, in [32], 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 [101].

3.4. Special purpose generators

Recently, a new class of generators are proposed where chaotic systems are utilized to introduce non-linearity in developing pseudo-random number generators. Mostly, these generators are developed for some special applications, like cryptographic purposes etc. For example, in [102] high dimensional Hamiltonian conservative chaotic systems are exploited to design pseudo-random number generator using FPGA technique. Another PRNG for video encryption is proposed in [103], which uses the output of a 16-cell LFSR perturbed by two chaotic maps, namely a bit-reorganizer, and a nonlinear function. Its output is 32 bit word. In [104], the Lorenz and Lü chaotic systems based reconfigurable PRNG is designed on FPGA. Some other similar generators are [105–115]. In [116], SRAM Physical Unclonable Functions (PUFs) are taken as the entropy source to provide seed to a hash based Deterministic Random Bit Generator (DRBG). Sometimes, chaotic source are added with the existing PRNGs to produce new hybrid PRNGs; see, for example [117–119]. In [120] a family of matrix random number generators called MIXMAX random number generator is defined that uses linear matrix recursions modulo a prime number p. Its output is 53-bit double precision number with the default $p = 2^{61} - 1$. This class of generators is a generalization of LCGs that uses the ergodicity of chaotic dynamical systems. A recent survey on FPGA implemented PRNGs is reported in [121]. With the advancement of quantum computing, some researchers are using a quantum computation model, called quantum random walks (QRWs) for constructing PRNGs [122,123]. On the other hand, with the introduction of GPUs, a new research direction is opened for generating uniform random numbers for parallel computing environment [124]. However, in this work, we have not considered these special class of generators for testing and comparison.

Table 2
List of 30 good PRNGs.

Class of PRNGs	Name of the PRNGs
LCGs	`minstd_rand`, Borland LCG, Knuth's LCG `MMIX`, `rand`, `lrand48`, `MRG31k3p`, `PCG-32`
LFSRs	`random`, `Tauss88`, `LFSR113`, `LFSR258`, `WELL512a`, `WELL1024a`, `MT19937-32`, `MT19937-64`, `SFMT19937-32`, `SFMT19937-64`, `dSFMT19937-32`, `dSFMT19937-52`, `xorshift32`, `xorshift64*`, `xorshift1024*`, `xorshift128+`
CAs	Rule 30 with CA size 101, Hybrid CA with Rules 30 & 45, Maximal Length CA with $\gamma = 0$ and $\gamma = 1$, Non-linear 2-state CA, 3-state CA, Decimal CA

3.5. Remark and discussion

• 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 [125]. 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 2) 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 can be lesser than LFSR. Due to this reason, and option of parallelism, CA-based, more specifically ECA-based PRNGs are attractive as VLSI test pattern generators.

To conclude this short survey, we list out the good PRNGs in Table 2, which we have already discussed in this section. Our target is to measure the randomness quality of these PRNG uniformly on the same platform. To do this we use some well-known empirical testbeds which we briefly discuss next.

4. 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 [39]. 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 2), 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 testbeds to identify non-randomness in the generated numbers.

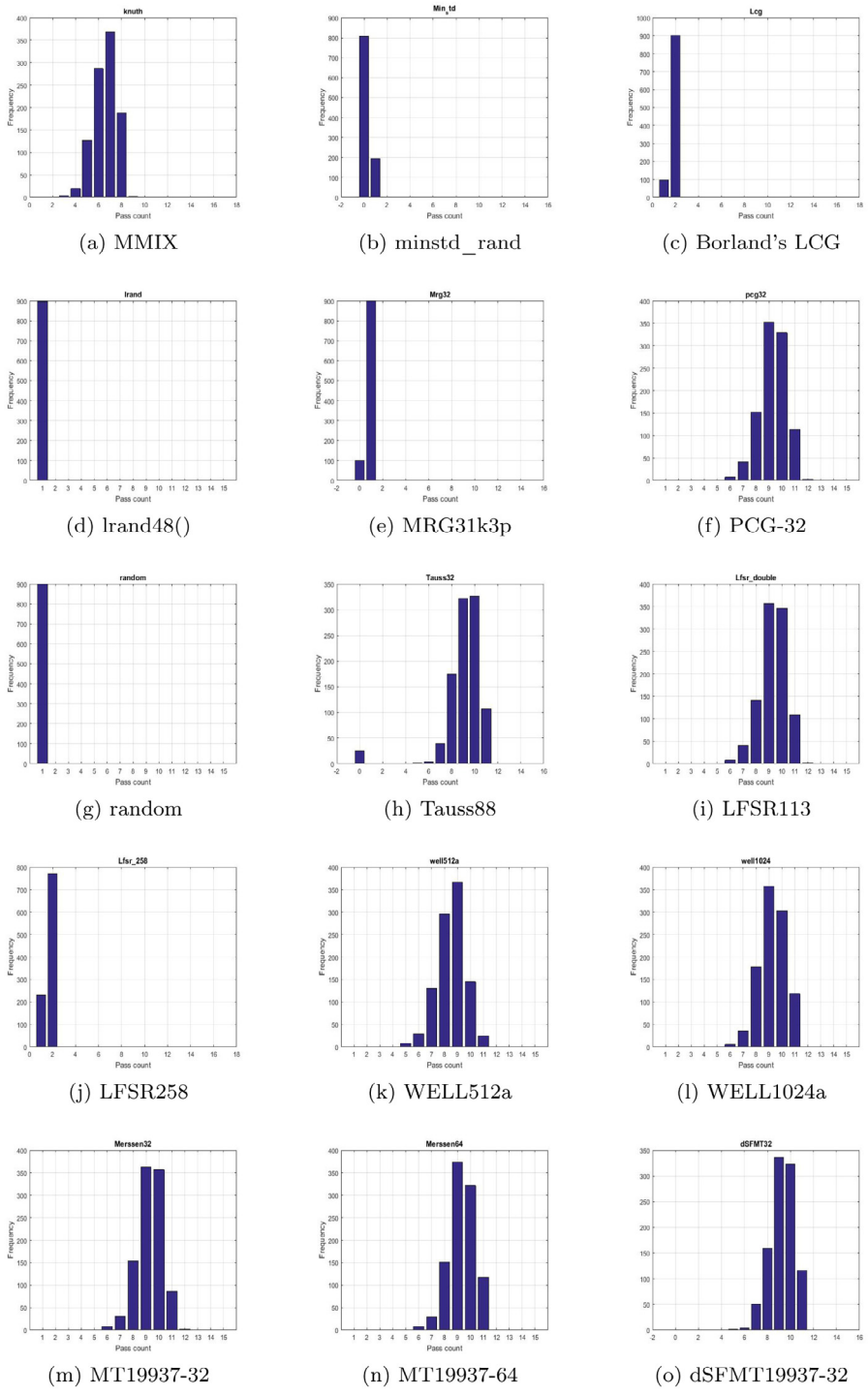
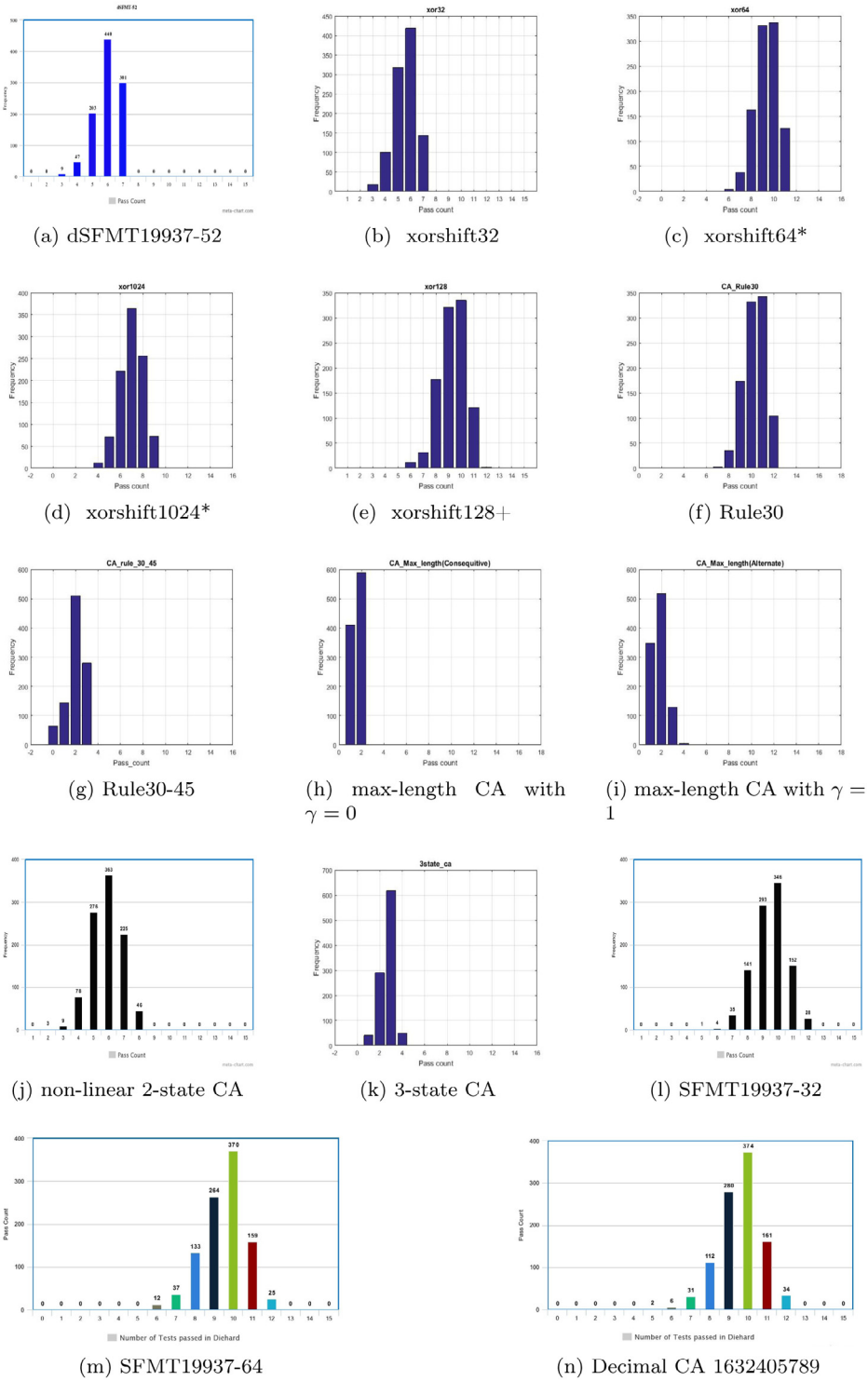


Fig. 10. Test results of PRNGs for 1000 seeds with Diehard battery of tests to get a rough estimate

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.

4.1. Blind (statistical) tests

The target of these 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". For each test, based on the random sequence produced by a PRNG, a decision is taken either to reject or not to reject the null hypothesis $\mathscr{H}_0$. To do this, a suitable 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



(a) dSFMT19937-52 (b) xorshift32 (c) xorshift64*

(d) xorshift1024* (e) xorshift128+ (f) Rule30

(g) Rule30-45 (h) max-length CA with $\gamma = 0$ (i) max-length CA with $\gamma = 1$

(j) non-linear 2-state CA (k) 3-state CA (l) SFMT19937-32

(m) SFMT19937-64 (n) Decimal CA 1632405789

Fig. 11. Test results of PRNGs for 1000 seeds with Diehard battery of tests to get a rough estimate (continued).

theoretical distribution (usually standard normal or chi-square distribution) of this statistic is calculated. 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 3.

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. Whereas, the probability of generating a Type II error is denoted by β.

The p-value of a test measures the strength of evidence against the null hypothesis. It is defined as

$$p = \Pr[X \geq t \mid \mathscr{H}_0 \text{ is true}]$$

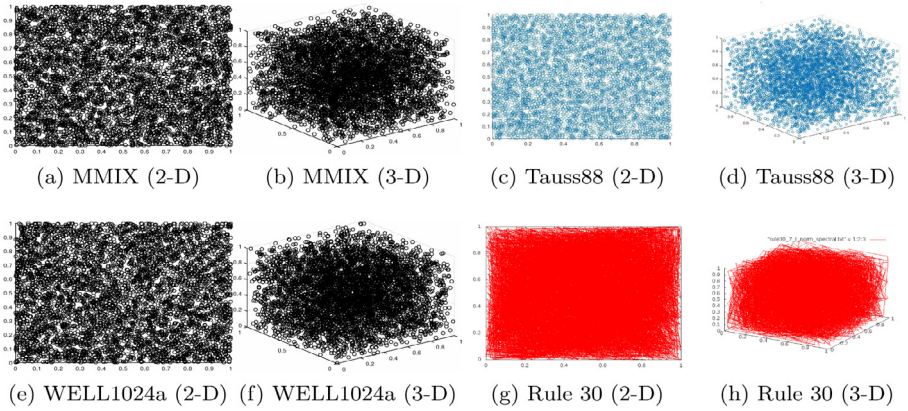


Fig. 12. Lattice test results for rand, MMIX, Tauss88, WELL1024a and Rule 30 with s_1.

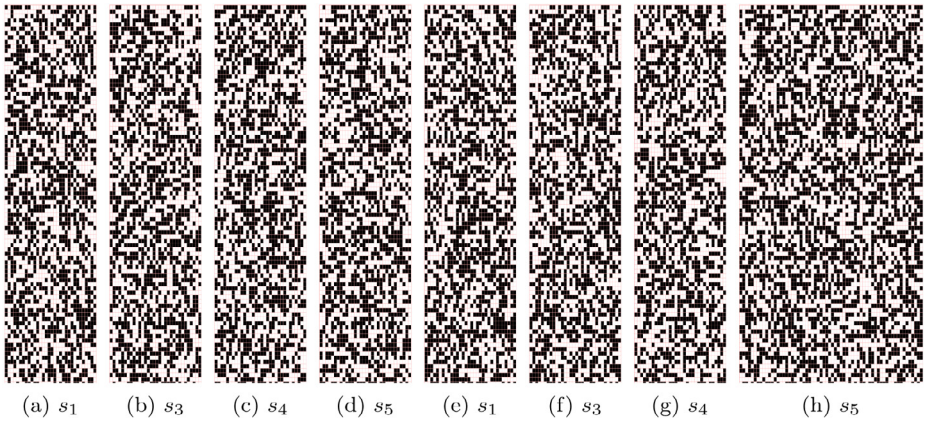


Fig. 13. Space–time diagram for 3-state CA 13(a) to 13(d) and Decimal CA 1632405789 13(e) to 13(h). For seed s_5, the output of Decimal CA is taken as 64 bits.

where X denotes the test statistics and t the critical value [34]. If p-value is very close to 0 or 1, it indicates that the sequence generated by the PRNG is not random. Normally, if $p \geq \alpha$, then the tested sequence 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:

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

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

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″ [39]. Later, other batteries have been developed improving the testing procedures of Knuth, such as Diehard, NIST, TestU01, PractRand, Dieharder, etc. In this paper, we have selected three well-known test suites, namely *Diehard*, *TestU01* and *NIST*. A PRNG is supposed to be a good PRNG, if it passes all tests of every testbed for any seed.

4.1.1. Diehard battery of tests

George Marsaglia in 1996 provided this battery of tests [33], 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 11.5 MB. 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 [33].

4.1.2. 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 [34] 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 to encompass new updated tests. It comprises of several battery of tests, including most of the tests in Diehard and many more with more flexibility to select the test parameters than in

Table 3
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

Diehard. However, we have selected the battery *rabbit* to test the PRNGs. The reason for choosing this test-suite is — this battery is specifically designed to test a sequence of random bits produced by a generator. It contains the following 26 tests from different modules (mentioned in parenthesis for each test):

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 [126] 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 [127]. *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 [128]. The statistical tests developed by George Marsaglia and his collaborators in [129] 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 [130]. 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. Here, to test a PRNG using *rabbit*, we have set $nb = 10^7$ and the file size is taken as 10.4 MB. 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 [34].

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.

4.1.3. NIST statistical test suite

The NIST Statistical Test Suite is developed to test a PRNG for cryptographic properties [35]. It 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 and probability theory. This test suite 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.

For this test suite, the level of significance $\alpha = 0.01$. So, for a sample size m generated by a PRNG with a particular seed, if x is the minimum pass rate, then for the PRNG to pass the test, minimum number of sequences with p-values ≥ 0.01 has to be x. This minimum pass rate is approximately 615 for a sample size of 629 for the random excursion (variant) test and 980 for sample size 1000 for each of the other tests. In general, the range of acceptable proportions for x is calculated as $(1-\alpha) \pm 3\sqrt{\frac{\alpha(1-\alpha)}{m}}$, for a sample size α.

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 125 MB 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. For each run, a file named *finalAnalysisReport.txt* is generated that summarizes the results of all the tests for that run. In this file, the first ten columns note the frequency of p-values in each of the 10 equal 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.

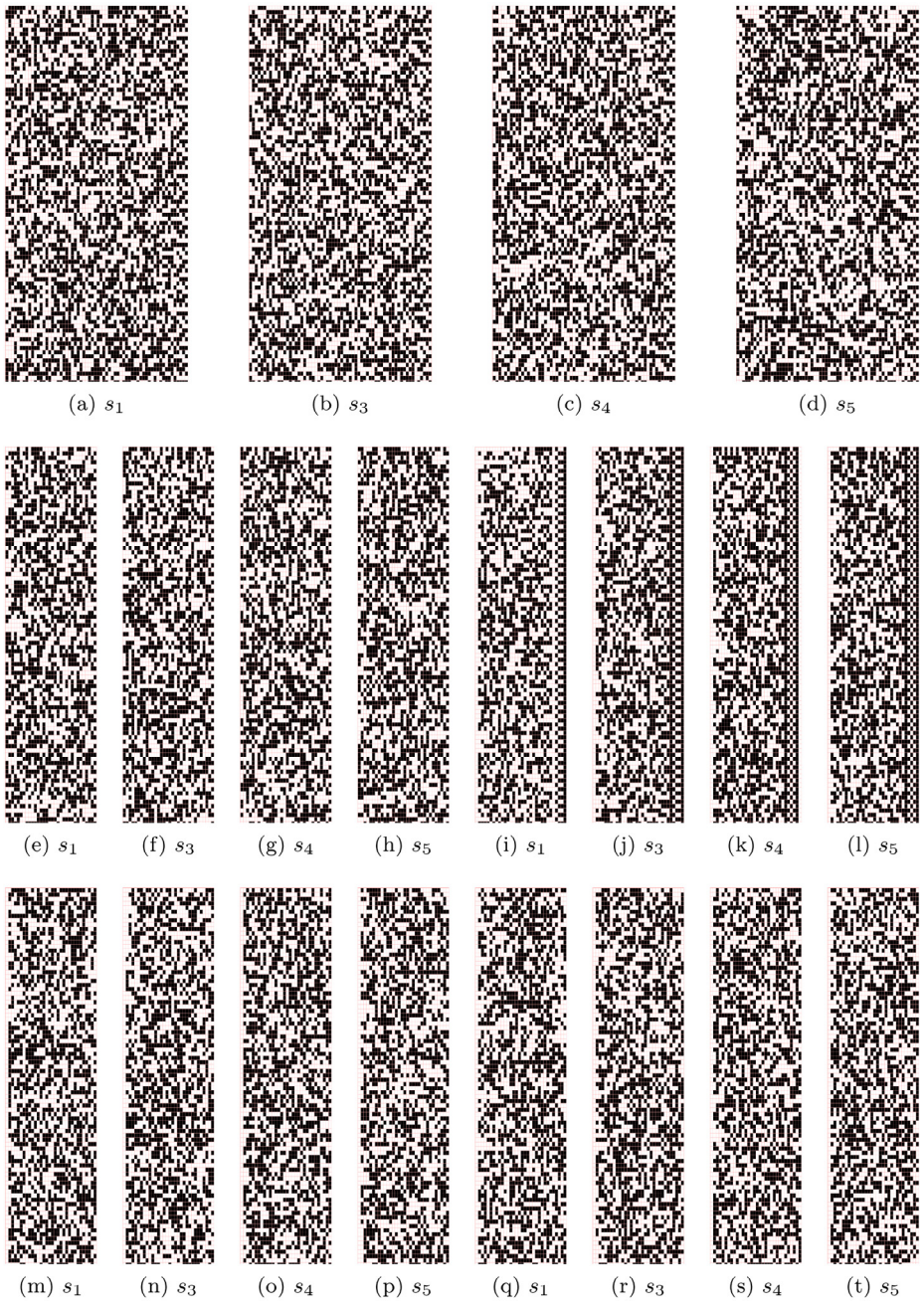


Fig. 14. Space–time diagram for Knuth's MMIX 14(a) to 14(d), Borland's LCG 14(e) to 14(h) and minstd_rand 14(i) to 14(l), rand 14(m) to 14(p) and lrand 14(q) to 14(t) of UNIX.

4.2. 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 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 of a PRNG is expected to be very large, so, for every graphical test also, all numbers of a period cannot be used; rather a set of numbers needs 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.

4.2.1. 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.

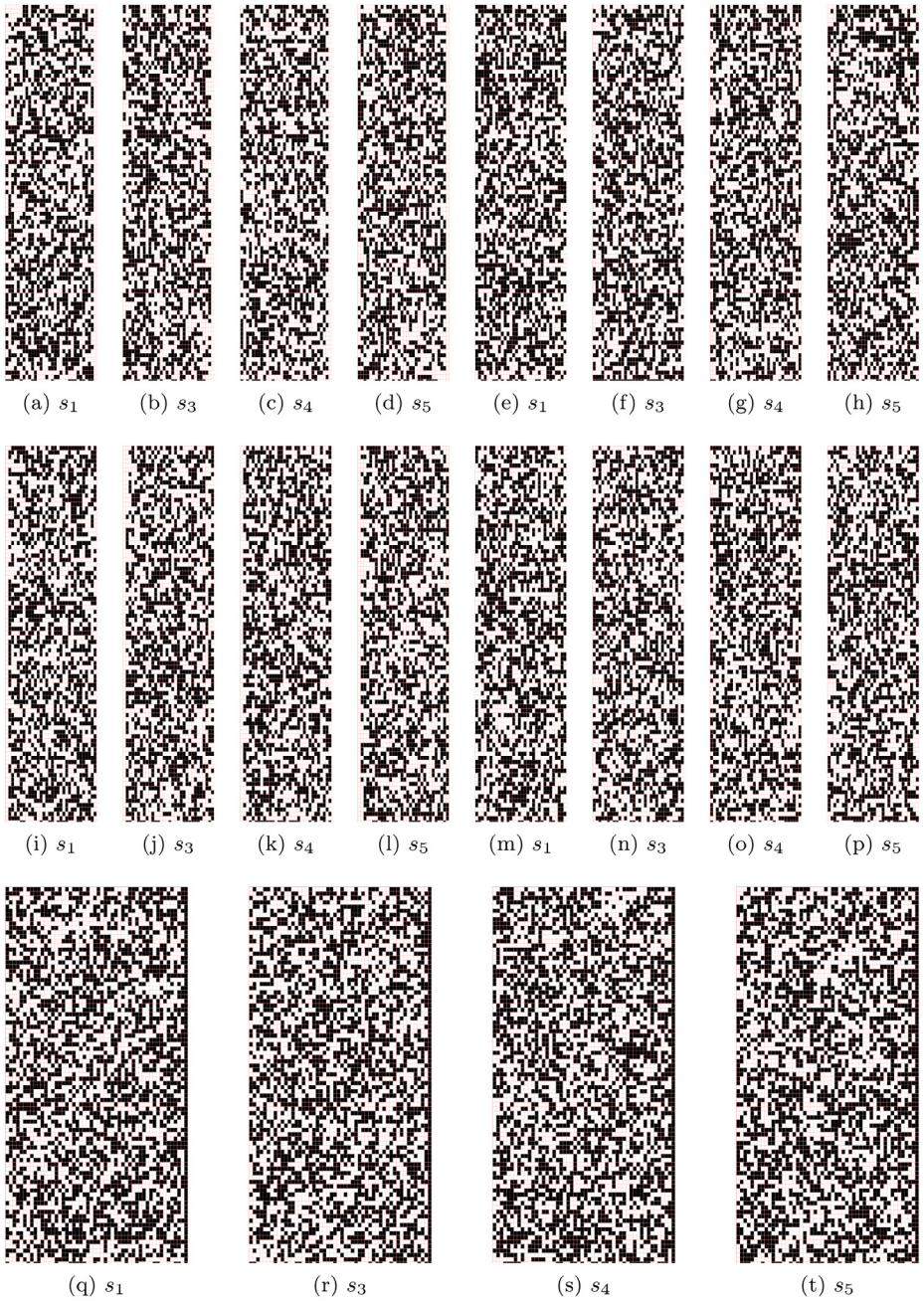


Fig. 15. Space–time diagram for MRG31k3p 15(a) to 15(d) and PCG 32-bit 15(e) to 15(h), random 15(i) to 15(l), Tauss88 15(m) to 15(p) and dSFMT19937 64 bit 15(q) to 15(t).

4.2.2. 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 [83]. 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 each seed 1000 and 250 numbers are generated for testing using the lattice test and space–time

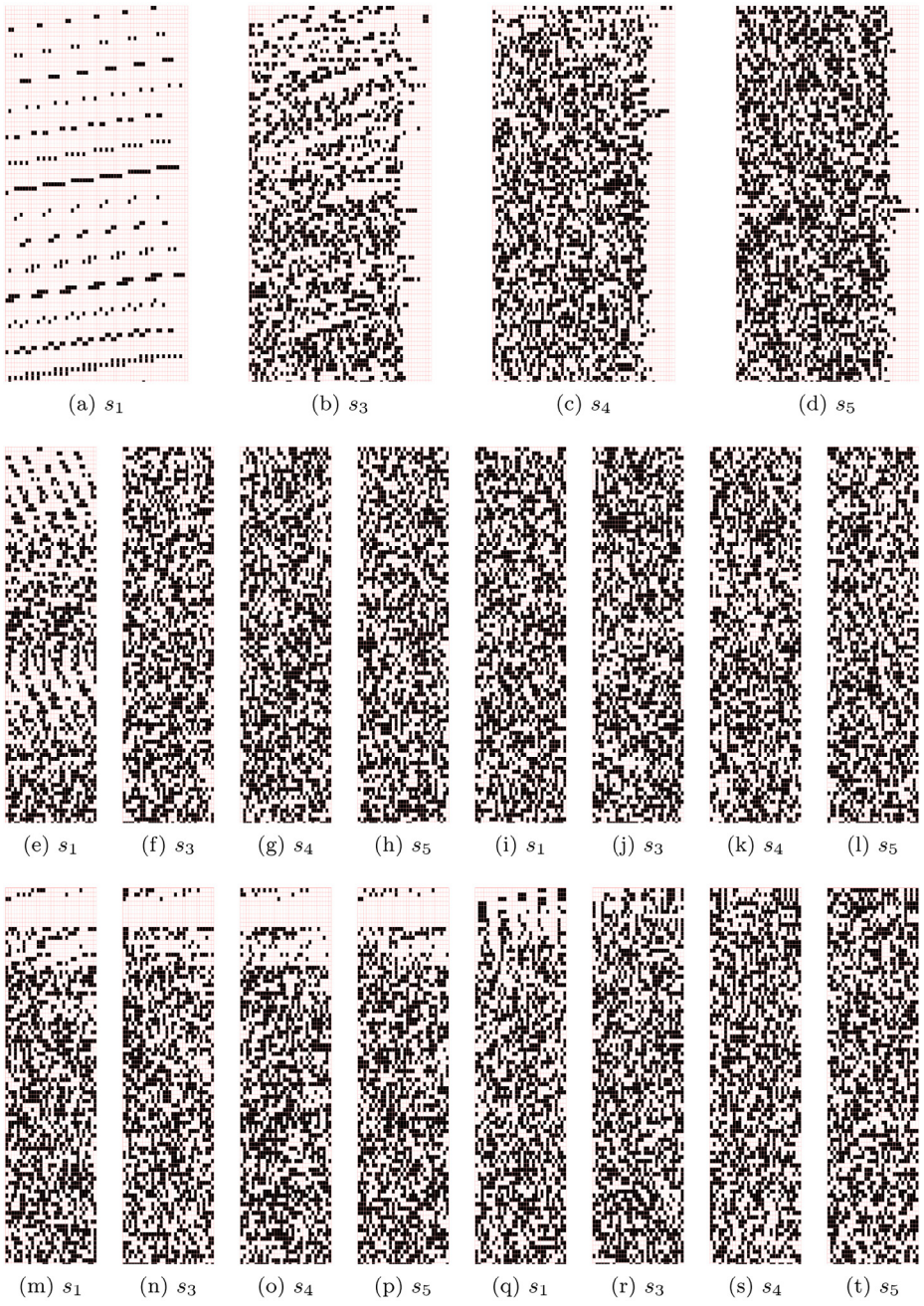


Fig. 16. Space–time diagram for LFSR258 16(a) to 16(d), LFSR113 16(e) to 16(h) and xorshift 16(i) to 16(l), WELL512 16(m) to 16(p) and WELL1024a 16(q) to 16(t).

diagram test respectively. In case of space–time diagrams, these numbers are directly represented, whereas, for lattice tests, pairs (resp. triplets) of consecutive numbers are plot as each point in the 2-D (resp. 3-D) plane.

5. Empirical facts

A usual claim of a PRNG is, it is better than others! Mostly, this claim is based on the performance of the PRNGs in the well-known test-suites. In this section, we verify the claim of the existing PRNGs to find which are really better than others with respect to randomness quality. To do so, we choose the PRNGs of Table 2 and apply empirical tests of Section 4. Although failure in any test of a testbed means the PRNG is non-random, but for the sake of uniformity, 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. However, we do not trust the Blind tests completely and give a preference to the results of graphical tests whenever there is discrepancy.

For all these PRNGs, we have collected the standard implementation and used them to generate numbers. To maintain uniformity in testing, we have tested the stream of binary numbers generated in sequence by these standard implementations of 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. If the generated numbers are normalized, then, first we convert

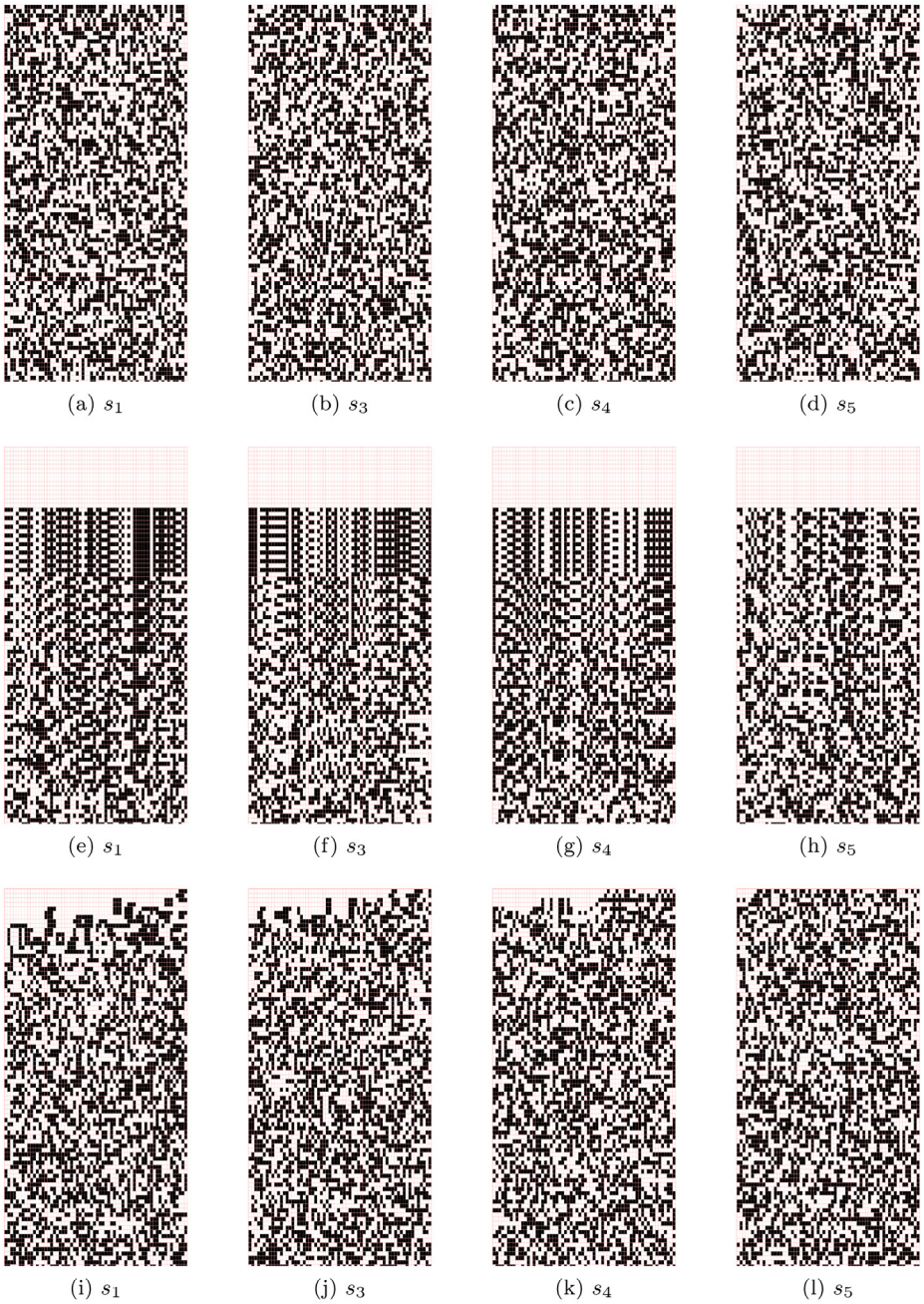


Fig. 17. Space–time diagram for xorshift64* 17(a) to 17(d), xorshift1024* 17(e) to 17(h) and xorshift128+ 17(i) to 17(l).

fractional part of each number into its binary equivalent and then add these bits to the .bin file.

5.1. 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 a test run. 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 PRNGs for fixed seeds, we also want to observe the rough estimate of the general 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. Some PRNGs, such as `LFSR113`, `LFSR258`, etc. require more than one (non-zero) seed to initialize its components. However, we supply here the same seed to all of its components.

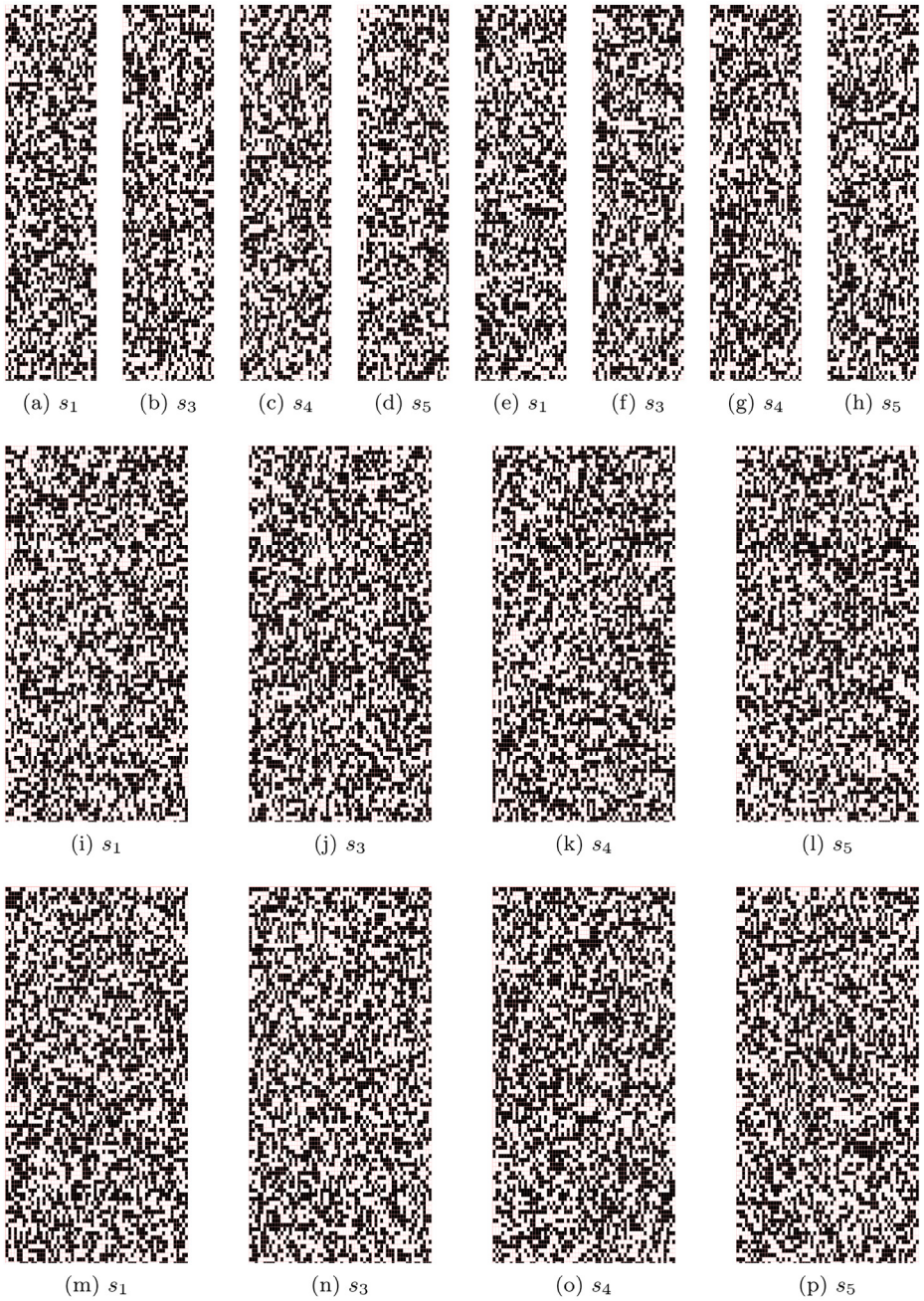


Fig. 18. Space–time diagram for MT19937 32 bit 18(a) to 18(d), SFMT19937 32 bit 18(e) to 18(h), MT19937 64 bit 18(i) to 18(l) and SFMT19937 64 bit 18(m) to 18(p).

5.2. Results of empirical tests

The selected PRNGs (LCG-based, LFSR-based and CA-based) are tested with statistical tests as well as with the graphical tests. Here, summary of the results of these tests is documented. A library of codes and packages used for this purpose is publicly available at Ref. [131].

5.2.1. Results of statistical tests

Table 4 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.

It is observed that, none of the PRNGs can pass all tests of these blind testbeds. We also 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 (up and down) test, except `LFSR258`, which passes 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` 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` is better than `MMIX` as well as `LFSR113`, Xorshift PRNGs, WELL PRNGs and dSFMTs. In fact, performance of `PCG-32` is comparable to MTs and SFMTs.
- Performance of `LFSR113` is dependent on seed; whereas, `WELL512a` and `WELL1024a` are invariant of seeds.

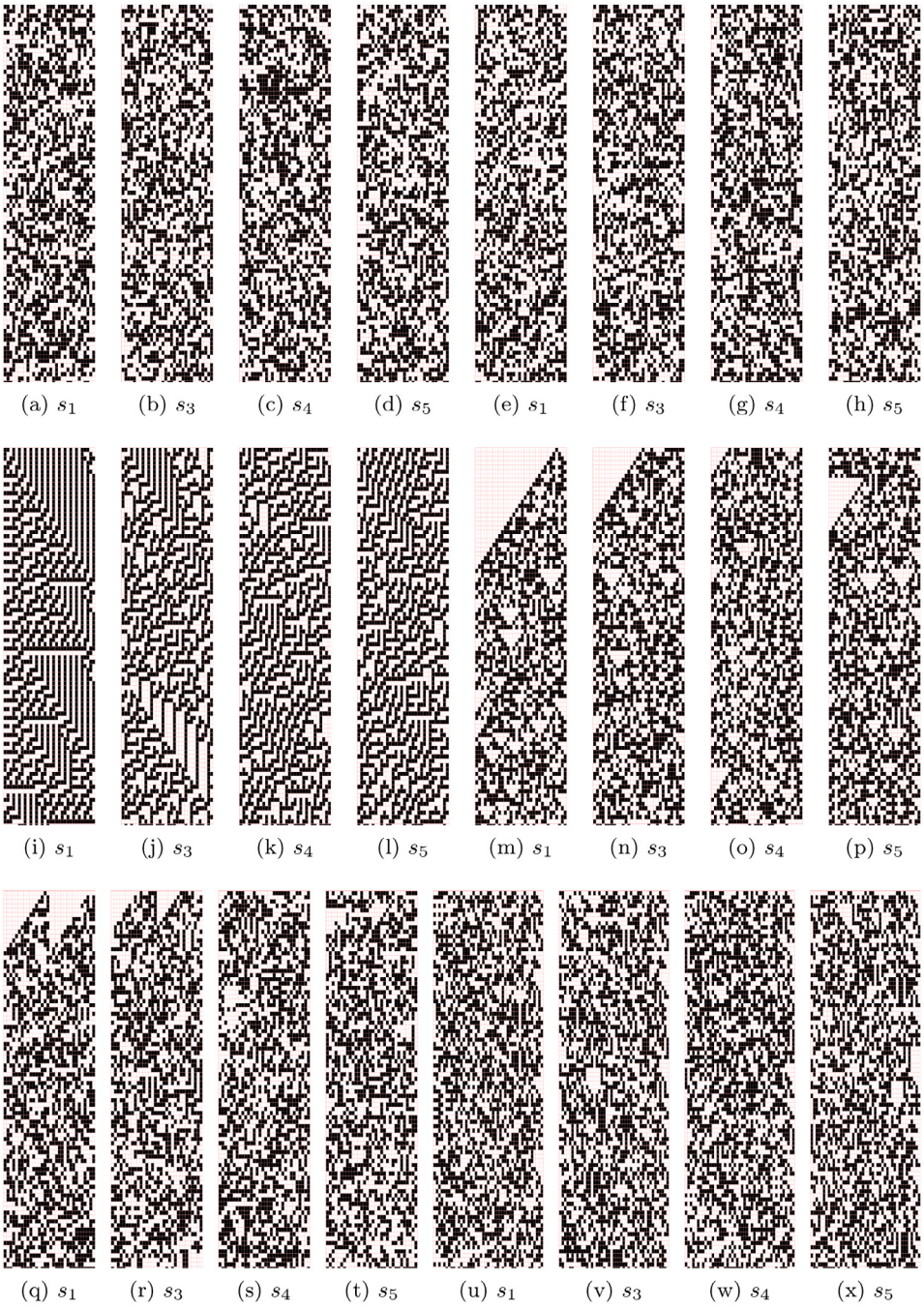


Fig. 19. Space–time diagram for dSFMT19937 32 bit 19(a) to 19(d), rule 30 19(e) to 19(h), rule 30–45 19(i) to 19(l), max-length CA with $\gamma = 0$ 19(m) to 19(p), max-length CA with $\gamma = 1$ 19(q) to 19(t) and non-linear 2-state CA 19(u) to 19(x).

- 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, Decimal CA and ECA 30 can compete with the elite group of PRNGs like Mersenne Twisters, WELL and PCG. However, other CA-based PRNGs are not so good. For all CAs, except ECA rule 30, the complete (or, a block from a) configuration of the CA is taken as a number. Among these CAs based PRNGs, performance of the max-length CA ($\gamma = 1$) is better than the non-linear 2-state 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 for fixed seeds. Last column of Table 4 shows this ranking. For this ranking, we have considered the overall tests passed in each test-suite. If two PRNGs give similar results, they have the same rank. It can be observed that, among the LCG-based PRNGs, `PCG-32` gives the best result and among the LFSR-based PRNGs, performances of SFMTs (32 and 64 bits) are the best. Moreover, among the CA-based PRNGs, Decimal CA gives the best result, whereas result of Wolfram's rule 30 is also comparable to the results of SFMTs and MTs.

- As SFMTs and Decimal CA are best performers, these are ranked as 1. Similarly, performance of `MT19937-64`, `PCG-32` and rule 30 are comparable (ranked 2), performance of

Table 4
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
	SFMT19937–32	10	25	15	9	25	15	10	25	15	9	25	15	10	25	15	1
	SFMT19937–64	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	7	11	2	5	11	3	6	11	4	5	12	4	9
	3-state CA	3	12	6	3	12	6	3	11	5	2	11	4	3	11	4	10
	Decimal CA	9	25	15	10	25	15	11	25	15	11	25	15	10	25	15	1

MT19937-32 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 we rank it lower than MT and SFMT. Similarly, LFSR113 can perform better than WELLs and non-linear 2-state 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.
- Among the other PRNGs, maximal-length CA with $\gamma = 1$ and MMIX are ranked 8, the non-linear 2-state CA based PRNG, dSFMT-52 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, lrand48, Borland's LCG, random and rule 30–45 as rank 11 and minstd_rand, MRG31k3p and LFSR258 as rank 12.

In this ranking, however, there are many ties. To improve the ranking, we next use 1000 seeds, as mentioned before, and observe the rough estimate of the performance of PRNGs in Diehard battery.

Figs. 10 and 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 5.

We find rough estimate of performance by calculating $\frac{\sum_i f_i \times t_i}{1000}$, where f_i is the frequency of a PRNG to pass t_i number of tests for ith seed. The column 'Range' in Table 5 indicates minimum and maximum number of tests passed in the whole experiment for a particular PRNG. The new ranks are discussed below.

- In terms of (rough) estimate of tests passed and range, rule 30 beats all other PRNGs. However, considering results of NIST and TestU01, Decimal CA and SFMTs still hold the first rank, while rule 30 is ranked 2. As estimated tests passed by MT19937-64 is better than PCG-32, it is ranked 3.
- Estimated tests passed by PCG-32, MT19937-32, 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 (rank 5), WELL512a and xorshift64* (rank 6). We have observed that, Tauss88 sometimes fails to pass any tests of Diehard. So, it is put in the same group (rank 7) with LFSR113, which has a very good estimate of pass count despite having a not-so-good performance for the fixed seeds.
- xorshift1024* has better rank (rank 8) than non-linear 2-state CA and MMIX (rank 9), because of performance in NIST and TestU01 library.
- The next rank holders are xorshift32 and dSFMT-52 (rank 10).
- As estimated pass count of maximal-length CA with $\gamma = 1$ is low, its rank is degraded. It is put in the same group as the 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, lrand48 and random are ranked 13.

Table 5
Summary of Statistical test results for different seeds.

Name of the PRNGs		Fixed seeds			Random seeds		Previous rank	2nd level rank
		Diehard	TestU01	NIST	Estimate	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
	SFMT19937–32	9–10	25	15	9.5	5–12	1	1
	SFMT19937–64	9–11	25	15	9.52	6–12	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	5–7	11	3–4	5.85	2–8	9	9
	3-state CA	2–3	11–12	4–6	2.7	1–4	10	11
	Decimal CA	9–11	25	15	9.59	6–12	1	1

- 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.

As mentioned in the introduction, blind tests have inherent incompleteness. Therefore, to deal with these shortcomings, in the next section, graphical tests are further incorporated on these PRNGs to verify whether this ranking is justified by visualization (human intervention) as well as to break the tie between intra-class PRNGs whenever possible.

5.2.2. Results of graphical tests

As mentioned, we use two types of graphical tests — lattice tests (2-D and 3-D) and space–time diagram test. Here, each of the PRNGs is tested using only the five fixed seeds, which are renamed as following for ease of presentation: seed 7 as s_1, 1234 as s_2, 12345 as s_3, 19650218 as s_4 and seed 123456789123456789 as s_5. 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 and co-relate with the blind test results. The result of these tests are shown as 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 Decimal CA, MTs, SFMTs and WELL, the plots are relatively filled. For example, see Fig. 12 for output of `MMIX`, `Tauss88`, `WELL1024a` and rule 30. We have observed that, if a PRNG performs badly, then in the corresponding lattice test diagrams, there is correlation between the numbers. Hence, these plots of lattice test verify our previous ranking based on blind tests. However, this test fails to further enhance or modify the ranking shown in Table 5. So, we avoid supplying all the images of lattice test but move to space–time diagram.
2. **Result of Space–time Diagram Test:** For space–time diagram, a set of 250 numbers are generated from each seed and printed on $X - Y$ plane. The space–time diagrams of 4 seeds s_1, s_3, s_4, s_5 for each PRNG, are shown in Figs. 13 to 19. From these figures, we can observe the following:

   - 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`, 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 whereas max-length CAs have dependency on seed up to some initial configurations. For non-linear 2-state CA, there are some minute cluster of black and white boxes. However, the figures for Decimal CA, ECA rule 30 and 3-state CA appear relatively random with no dependency on seed.
   - For the LCGs, the dependency on seed is less visible than the LFSRs.
   - If observed closely, every PRNG has some kind of clubbing of white boxes and black boxes, that is, none

Table 6
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	Estimate	Range			1st level	2nd 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
	SFMT19937–32	9–10	25	15	9.5	5–12	Relatively Filled	Independent of seed	1	1	2
	SFMT19937–64	9–11	25	15	9.52	6–12	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	5–7	11	3–4	5.85	2–8	Relatively Filled	Less dependency on seed	9	9	13
	3-state CA	2–3	11–12	4–6	2.7	1–4	Relatively Filled	Independent of seed	10	11	13
	Decimal CA	9–11	25	15	9.59	6–12	Relatively Filled	Independent of seed	1	1	1

of the figures is actually free of pattern. However, for the good PRNGs, these patterns are non-repeating.

5.3. Final ranking and remark

Many time the blind test results and space–time diagram results do not accord with each other. For example, one can see `xorshift1024*` (Fig. 17 and Table 4). In that case, although this PRNG passes many blind tests but it clearly shows pattern in space–time diagram as initial couple of numbers generated by it are fixed. This shows an evidence of flaw in the blind tests which can be detected by graphical tests that allow human intervention in the decision. So, in such cases, we change the ranking of the PRNG depending on its behavior in space–time diagram. Further, we can also distinguish between intra-class PRNGs using this tool, for example, `WELL1024a` and `MT19937-32` had same rank in Table 5. But, as `WELL1024a` as visible patterns (see Fig. 16), it loses its status to be included in the same group as `MT19937-32`. Hence, using the space–time diagrams along with the statistical tests, we can further improve the rankings of the PRNGs –

- `SFMT19937-64` holds the first position as it appears more random than `SFMT19937-32`. Decimal CA gives a tough competition to `SFMT19937-64`, sometimes even outperforming it which proves that it is at least at par with it (if not better!). So both of them are ranked 1.
- Rule 30 holds the 3rd rank, whereas `MT19937-64` holds rank 4. `PCG-32` is better than `MT19937-32` and `dSFMT-32`. So, it holds rank 5. The next rank holder is `MT19937-32`.
- `dSFMT-32` has less dependency on seed than `WELL1024a` and `xorshift128+`. So, it is ranked on the 7th 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` has less dependency on seeds than non-linear 2-state CA based PRNG and max-length CA with $\gamma = 1$. So, it holds rank 12.
- Non-linear 2-state CA based PRNG, max-length CA with $\gamma = 1$ based PRNG and 3-state CA 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 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.

Based on the empirical tests, we finally rank the selected 30 PRNGs into 24 groups. This final ranking is shown in Table 6.

6. 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 using conventional testbeds and check how they actually perform in similar platform with the same seeds. We have used three empirical test-beds — Diehard, battery *rabbit* of 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 results of the rough estimate behavior of the PRNGs on Diehard battery of tests for 1000 seeds. However, as the underlying approximations about distributions in the testbeds and the thresholds are not always reliable, all blind tests have inherent incompleteness. To address this, we further use two graphical tests — lattice tests and space–time diagram test that incorporate human intervention to decide further. We have observed that many times the visual behavior of a PRNG do not agree with the blind test results. Therefore, giving preference to the visualization given by the space–time diagrams, a final ranking has been done in Table 6. According to our tests, Decimal CA based PRNGs and SFMT-64 bit generator are at par and the best pseudo-random number generators among all our selected PRNGs. Nevertheless, this ranking is not absolute and can be changed based on different metric. Further, the disagreement between blind test and space–time diagram results indicates the requirement of developing better metric that can detect non-randomness and rank the PRNGs by theoretical analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors are grateful to Mr. Krishnendu Maity for his contribution in testing the generators and studying the results. We also thank the anonymous reviewers for their comments and suggestions which have improved the work.

References

[1] F. Galton, Dice for statistical experiments, Nature 42 (1070) (1890) 13–14.
[2] L. Tippett, Random Sampling Numbers, in: Tracts for Computers, Cambridge University Press, 1927.
[3] M.G. Kendall, B. Babington-Smith, Randomness and random sampling numbers, J. R. Stat. Soc. 101 (1) (1938) 147–166.
[4] M.G. Kendall, B. Babington-Smith, Second paper on random sampling numbers, Suppl. J. R. Stat. Soc. 6 (1) (1939) 51–61.
[5] J. Von Neumann, 13. Various techniques used in connection with random digits, Appl. Math. Comput. 12 (1951) 36–38.
[6] P. L'Ecuyer, Maximally equidistributed combined Tausworthe generators, Math. Comp. 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. Software 32 (1) (2006) 1–16.
[11] M. Saito, M. Matsumoto, SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator, in: 7th International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, Springer Berlin Heidelberg, 2008, pp. 607–622.
[12] M. Saito, M. Matsumoto, A PRNG specialized in double precision floating point numbers using an affine transition, in: 8th International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, Springer Berlin Heidelberg, 2009, pp. 589–602.
[13] R.C. Tausworthe, Random numbers generated by linear recurrence modulo two, Math. Comp. 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, Adv. Appl. Math. 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 Gener. Comput. Syst. 16 (2–3) (1999) 291–305.
[17] S. Vigna, An experimental exploration of marsaglia's xorshift generators, scrambled, ACM Trans. Math. Software 42 (4) (2016) 30:1–30:23.
[18] S. Wolfram, Origins of randomness in physical systems, Phys. Rev. Lett. 55 (1985) 449–452.
[19] 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. Comput.-Aided Des. Integr. Circuits Syst. 8 (8) (1989) 842–859.
[20] P.D. Hortensius, R.D. McLeod, H.C. Card, Parallel random number generation for VLSI systems using cellular automata, IEEE Trans. Comput. C-38 (10) (1989) 1466–1473.
[21] P.H. Bardell, Analysis of cellular automata used as pseudorandom pattern generators, in: Proceedings International Test Conference, 1990, pp. 762–768.
[22] A. Compagner, A. Hoogland, Maximum-length sequences, cellular automata, and random numbers, J. Comput. Phys. 71 (2) (1987) 391–428.
[23] M. Sipper, M. Tomassini, Generating parallel random number generators by cellular programming, Internat. J. Modern Phys. C 7 (2) (1996) 180–190.
[24] K. Cattell, M. Serra, The analysis of one dimensional multiple-valued linear cellular automata, in: Proceedings of the Twentieth International Symposium on Multiple-Valued Logic, 1990, pp. 402–409.
[25] R. Alonso-Sanz, L. Bull, Elementary cellular automata with minimal memory and random number generation, Complex Syst. 18 (2) (2009) 195–213.
[26] S. Das, B.K. Sikdar, A scalable test structure for multicore chip, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 29 (1) (2010) 127–137.
[27] S. Das, Theory and Applications of Nonlinear Cellular Automata In VLSI Design (Ph.D. thesis), Bengal Engineering and Science University, Shibpur, India, 2007.
[28] S. Guan, S. Zhang, An evolutionary approach to the design of controllable cellular automata structure for random number generation, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 7 (1) (2003) 23–36.
[29] S. Guan, S.K. Tan, Pseudorandom number generation with self-programmable cellular automata, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 23 (7) (2004) 1095–1101.
[30] M. Tomassini, M. Sipper, M. Perrenoud, On the generation of high-quality random numbers by two-dimensional cellular automata, IEEE Trans. Comput. 49 (10) (2000) 1146–1151.
[31] S. Guan, S. Zhang, T. Quieta, 2-D CA variation with asymetric neighborship for psedorandom number generation, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 23 (3) (2004) 378–388.
[32] S.M. Hosseini, H. Karimi, M.V. Jahan, Generating pseudo-random numbers by combining two systems with complex behaviors, J. Inf. Secur. Appl. 19 (2) (2014) 149–162.
[33] G. Marsaglia, DIEHARD: A battery of tests of randomness, 1996, http://stat.fsu.edu/~geo/diehard.html.
[34] P. L'Ecuyer, R. Simard, TestU01: A C library for empirical testing of random number generators, ACM Trans. Math. Software 33 (4) (2007) 22:1–22:40.
[35] 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.
[36] P. L'Ecuyer, Random numbers for simulation, Commun. ACM 33 (10) (1990) 85–97.
[37] O. Goldreich, Foundations of Cryptography, Volume 2, Cambridge University Press, 2003.
[38] D.H. Lehmer, Mathematical methods in large-scale computing units, Ann. Comput. Lab. Harv. Univ. 26 (1951) 141–146.
[39] D.E. Knuth, The Art of Computer Programming – Seminumerical Algorithms, Vol. 2, third ed., Pearson Education, 2000.
[40] R. McGrath, et al., GNU C Library, Free Software Foundation, Inc, https://www.gnu.org/software/libc/manual/html_node/Pseudo_002dRandom-Numbers.html#index-pseudo_002drandom-numbers.
[41] P.A.W. Lewis, A.S. Goodman, J.M. Miller, A pseudo-random number generator for the System/360, IBM Syst. J. 8 (2) (1969) 136–146.
[42] G.S. Fishman, Multiplicative congruential random number generators with modulus 2^{β}: an exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$, Math. Comp. 54 (189) (1990) 331–344.
[43] G. Fishman, L.R. Moore III, An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$, SIAM J. Sci. Stat. Comput. 7 (1) (1986) 24–45.
[44] S.K. Park, K.W. Miller, Random number generators: Good ones are hard to find, Commun. ACM 31 (10) (1988) 1192–1201.
[45] S.K. Park, K.W. Miller, P.K. Stockmeyer, Technical correspondence: Response, Commun. ACM 36 (7) (1993) 105–110.
[46] R.P. Brent, On the periods of generalized fibonacci recurrences, Math. Comp. 63 (207) (1994) 389–401.
[47] G. Marsaglia, A. Zaman, A new class of random number generators, Ann. Appl. Probab. 1 (3) (1991) 462–480.
[48] C. Koç, Recurring-with-carry sequences, J. Appl. Probab. 32 (4) (1995) 966–971.
[49] R. Couture, P. L'Ecuyer, Distribution properties of multiply-with-carry random number generators, Math. Comput. Am. Math. Soc. 66 (218) (1997) 591–607.
[50] J. Eichenauer, J. Lehn, A non-linear congruential pseudo random number generator, Stat. Hefte 27 (1) (1986) 315–326.
[51] J. Eichenauer-Herrmann, Construction of inversive congruential pseudo-random number generators with maximal period length, J. Comput. Appl. Math. 40 (3) (1992) 345–349.
[52] J. Eichenauer-Herrmann, Statistical independence of a new class of inversive congruential pseudorandom numbers, Math. Comp. 60 (201) (1993) 375–384.
[53] P. L'Ecuyer, Efficient and portable combined random number generators, Commun. ACM 31 (6) (1988) 742–751.
[54] B.A. Wichmann, I.D. Hill, Algorithm AS 183: An efficient and portable pseudo-random number generator, J. R. Stat. Soc. C 31 (2) (1982) 188–190.
[55] P. L'Ecuyer, Combined multiple recursive random number generators, Oper. Res. 44 (5) (1996) 816–822.
[56] P. L'Ecuyer, Good parameters and implementations for combined multiple recursive random number generators, Oper. Res. 47 (1) (1999) 159–164.
[57] R.E. Nance, C. Overstreet Jr., Some experimental observations on the behavior of composite random number generators, Oper. Res. 26 (5) (1978) 915–935.
[58] P. Bratley, B.L. Fox, L.E. Schrage, A Guide to Simulation, Springer Science & Business Media, 1987.
[59] 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, 2000, pp. 683–689.
[60] M.E. O'Neill, PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation (HMC-CS-2014-0905), Harvey Mudd College, Claremont, CA, 2014.
[61] J.P.R. Tootill, W.D. Robinson, D.J. Eagle, An asymptotically random tausworthe sequence, J. ACM 20 (3) (1973) 469–481.
[62] 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.
[63] P. L'Ecuyer, Random Number Generators, DIRO, Université de Montréal, 2017, http://www-labs.iro.umontreal.ca/~simul/rng/.
[64] M. Matsumoto, Y. Kurita, Twisted GFSR generators, ACM Trans. Model. Comput. Simul. 2 (3) (1992) 179–194.
[65] M. Matsumoto, Y. Kurita, Twisted GFSR generators II, ACM Trans. Model. Comput. Simul. 4 (3) (1994) 254–266.
[66] M. Saito, M. Matsumoto, SIMD-oriented fast mersenne twister (SFMT): twice faster than mersenne twister, 2017, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/#dSFMT.
[67] G. Marsaglia, Xorshift RNGs, J. Stat. Softw. 8 (14) (2003) 1–6.
[68] R.P. Brent, Note on Marsaglia's xorshift random number generators, J. Stat. Softw. 11 (5) (2004) 1–5.
[69] S. Vigna, Further scramblings of Marsaglia's xorshift generators, J. Comput. Appl. Math. 315 (Supplement C) (2017) 175–181.
[70] P. L'Ecuyer, J. Granger-Piché, Combined generators with components from different families, Math. Comput. Simulation 62 (3–6) (2003) 395–404.
[71] J. Kari, Theory of cellular automata: A survey, Theoret. Comput. Sci. 334 (1–3) (2005) 3–33.
[72] K. Bhattacharjee, N. Naskar, S. Roy, S. Das, A survey of cellular automata: types, dynamics, non-uniformity and applications, Nat. Comput. 19 (2) (2020) 433–461.
[73] S. Wolfram, Statistical mechanics of cellular automata, Rev. Modern Phys. 55 (3) (1983) 601–644.
[74] M. Matsumoto, Simple cellular automata as pseudorandom m-sequence generators for built-in self-test, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 31–42.
[75] P. Pal Chaudhuri, D. Roy Chowdhury, S. Nandi, S. Chattopadhyay, Additive Cellular Automata – Theory and Applications, Vol. 1, IEEE Computer Society Press, USA, ISBN: 0-8186-7717-1, 1997.
[76] H. Card, P. Hortensius, R. McLeod, Parallel random number generation for VLSI systems using cellular automata, IEEE Trans. Comput. 38 (10) (1989) 1466–1473.
[77] M. Saraniti, S.M. Goodnick, Hybrid fullband cellular automaton/Monte Carlo approach for fast simulation of charge transport in semiconductors, IEEE Trans. Electron Devices 47 (10) (2000) 1909–1916.

[78] J.M. Comer, J.C. Cerda, C.D. Martinez, D.H. Hoe, Random number generators using cellular automata implemented on FPGAs, in: Proceedings of Southeastern Symposium on System Theory, 2012, pp. 67–72.
[79] J.M. Comer, J.C. Cerda, C.D. Martinez, D.H.K. Hoe, Random number generators using Cellular Automata implemented on FPGAs, in: Proceedings of the 2012 44th Southeastern Symposium on System Theory (SSST), 2012, pp. 67–72.
[80] S. Wolfram, Cryptography with cellular automata, in: Advances in Cryptology - Crypto'85, Vol. 218, 1986, pp. 429–432.
[81] J. Machicao, A.G. Marco, O.M. Bruno, Chaotic encryption method based on life-like cellular automata, Expert Syst. Appl. 39 (16) (2012) 12626–12635.
[82] Z. Eslami, J. Zarepour Ahmadabadi, A verifiable multi-secret sharing scheme based on cellular automata, Inform. Sci. 180 (15) (2010) 2889–2894.
[83] S. Wolfram, A New Kind of Science, Wolfram-Media, 2002.
[84] W. Pries, A. Thanailakis, H.C. Card, Group properties of cellular automata and VLSI applications, IEEE Trans. Comput. C-35 (12) (1986) 1013–1024.
[85] M. Serra, T. Slater, J.C. Muzio, D.M. Miller, The analysis of one-dimensional linear cellular automata and their aliasing properties, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 9 (7) (1990) 767–778.
[86] K. Cattell, J.C. Muzio, Synthesis of one-dimensional linear hybrid cellular automata, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 15 (3) (1996) 325–335.
[87] K. Cattell, S. Zhang, Minimal cost one-dimensional linear hybrid cellular automata of degree through 500, J. Electron. Test.: Theory Appl. 6 (2) (1995) 255–258.
[88] P.H. Bardell, W.H. McAnney, J. Savir, Built-in Test for VLSI: Pseudorandom Techniques, Wiley-Interscience, New York, NY, USA, 1987.
[89] P.H. Bardell, Primitive polynomials of degree 301 through 500, J. Electron. Test. 3 (2) (1992) 175–176.
[90] 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: Proceedings of International Conference on Cellular Automata, Research and Industry (ACRI), 2006, pp. 204–214.
[91] K. Bhattacharjee, D. Paul, S. Das, Pseudo-random number generation using a 3-state cellular automaton, Internat. J. Modern Phys. C 28 (06) (2017) 1750078.
[92] K. Bhattacharjee, S. Das, Random number generation using decimal cellular automata, Commun. Nonlinear Sci. Numer. Simul. 78 (2019) 104878.
[93] L. Petrica, FPGA optimized cellular automaton random number generator, J. Parallel Distrib. Comput. 111 (2018) 251–259.
[94] T. Purkayastha, D. De, K. Das, A novel pseudo random number generator based cryptographic architecture using quantum-dot cellular automata, Microprocess. Microsyst. 45 (2016) 32–44.
[95] M. Tomassini, M. Perrenoud, Cryptography with cellular automata, Appl. Soft Comput. 1 (2) (2001) 151–160.
[96] M. Tomassini, M. Perrenoud, Stream cyphers with one- and two-dimensional cellular automata, in: M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J.J. Merelo, H.-P. Schwefel (Eds.), Parallel Problem Solving from Nature PPSN VI, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 722–731.
[97] D.R. Chowdhury, I. Sengupta, P.P. Chaudhuri, A class of two-dimensional cellular automata and their applications in random pattern testing, J. Electron. Test. 5 (1) (1994) 67–82.
[98] C. Torres-Huitzil, M. Delgadillo-Escobar, M. Nuno-Maganda, Comparison between 2D cellular automata based pseudorandom number generators, IEICE Electron. Express 9 (17) (2012) 1391–1396.
[99] B. Kang, D. Lee, C. Hong, High-performance pseudorandom number generator using two-dimensional cellular automata, in: 4th IEEE International Symposium on Electronic Design, Test and Applications (Delta 2008), 2008, pp. 597–602.
[100] B. Shackleford, M. Tanaka, R.J. Carter, G. Snider, FPGA implementation of neighborhood-of-four cellular automata random number generators, in: Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays, 2002, pp. 106–112.
[101] C.G. Langton, Studying artificial life with cellular automata, Physica D 22 (1–3) (1986) 120–149.
[102] E. Dong, M. Yuan, S. Du, Z. Chen, A new class of Hamiltonian conservative chaotic systems with multistability and design of pseudo-random number generator, Appl. Math. Model. 73 (2019) 40–71.
[103] H. Xu, X. Tong, X. Meng, An efficient chaos pseudo-random number generator applied to video encryption, Optik 127 (20) (2016) 9305–9319.
[104] A.A. Rezk, A.H. Madian, A.G. Radwan, A.M. Soliman, Reconfigurable chaotic pseudo random number generator based on FPGA, AEU - Int. J. Electron. Commun. 98 (2019) 174–180.
[105] M. Meranza-Castillón, M. Murillo-Escobar, R. López-Gutiérrez, C. Cruz-Hernández, Pseudorandom number generator based on enhanced Hénon map and its implementation, AEU - Int. J. Electron. Commun. 107 (2019) 239–251.
[106] A.V. Tutueva, E.G. Nepomuceno, A.I. Karimov, V.S. Andreev, D.N. Butusov, Adaptive chaotic maps and their application to pseudo-random numbers generation, Chaos Solitons Fractals 133 (2020) 109615.
[107] Y. Wang, Z. Liu, J. Ma, H. He, A pseudorandom number generator based on piecewise logistic map, Nonlinear Dynam. 83 (4) (2016) 2373–2391.
[108] B. Li, X. Liao, Y. Jiang, A novel image encryption scheme based on improved random number generator and its implementation, Nonlinear Dynam. 95 (3) (2019) 1781–1805.
[109] X. Lv, X. Liao, B. Yang, A novel pseudo-random number generator from coupled map lattice with time-varying delay, Nonlinear Dynam. 94 (1) (2018) 325–341.
[110] I. Öztürk, R. Kılıç, A novel method for producing pseudo random numbers from differential equation-based chaotic systems, Nonlinear Dynam. 80 (3) (2015) 1147–1157.
[111] M.L. Sahari, I. Boukemara, A pseudo-random numbers generator based on a novel 3D chaotic map with an application to color image encryption, Nonlinear Dynam. 94 (1) (2018) 723–744.
[112] R. Hamza, A novel pseudo random sequence generator for image-cryptographic applications, J. Inf. Secur. Appl. 35 (2017) 119–127.
[113] M. Garcia-Bosque, A. Pérez-Resa, C. Sánchez-Azqueta, C. Aldea, S. Celma, Chaos-based bitwise dynamical pseudorandom number generator on FPGA, IEEE Trans. Instrum. Meas. 68 (1) (2019) 291–293.
[114] P. Ayubi, S. Setayeshi, A.M. Rahmani, Deterministic chaos game: A new fractal based pseudo-random number generator and its cryptographic application, J. Inf. Secur. Appl. 52 (2020) 102472.
[115] S. Cang, Z. Kang, Z. Wang, Pseudo-random number generator based on a generalized conservative Sprott-A system, Nonlinear Dynam. 104 (1) (2021) 827–844.
[116] S. Chen, B. Li, C. Zhou, FPGA implementation of SRAM PUFs based cryptographically secure pseudo-random number generator, Microprocess. Microsyst. 59 (2018) 57–68.
[117] E. Avaroğlu, I. Koyuncu, A.B. Özer, M. Türk, Hybrid pseudo-random number generator for cryptographic systems, Nonlinear Dynam. 82 (1) (2015) 239–248.
[118] F. Neugebauer, I. Polian, J.P. Hayes, S-box-based random number generation for stochastic computing, Microprocess. Microsyst. 61 (2018) 316–326.
[119] I. Ullah, N.A. Azam, U. Hayat, Efficient and secure substitution box and random number generators over Mordell elliptic curves, J. Inf. Secur. Appl. 56 (2021) 102619.
[120] K.G. Savvidy, The MIXMAX random number generator, Comput. Phys. Comm. 196 (2015) 161–165.
[121] M. Bakiri, C. Guyeux, J.-F. Couchot, A.K. Oudjida, Survey on hardware implementation of random number generators on FPGA: Theory and experimental analyses, Comp. Sci. Rev. 27 (2018) 135–153.
[122] Y.-G. Yang, Q.-Q. Zhao, Novel pseudo-random number generator based on quantum random walks, Sci. Rep. 6 (1) (2016) 1–11.
[123] A.A.A. EL-Latif, B. Abd-El-Atty, S.E. Venegas-Andraca, Controlled alternate quantum walk-based pseudo-random number generator and its application to quantum color image encryption, Physica A 547 (2020) 123869.
[124] P. L'Ecuyer, D. Munger, B. Oreshkin, R. Simard, Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs, Math. Comput. Simulation 135 (2017) 3–17, Special Issue: 9th IMACS Seminar on Monte Carlo Methods.
[125] R.J. McEliece, Finite Field for Scientists and Engineers, Kluwer Academic Publishers, Norwell, MA, USA, 1987.
[126] P. L'Ecuyer, R. Simard, S. Wegenkittl, Sparse serial tests of uniformity for random number generators, SIAM J. Sci. Comput. 24 (2) (2002) 652–668.
[127] P. L'Ecuyer, J.-F. Cordeau, R. Simard, Close-point spatial tests and their application to random number generators, Oper. Res. 48 (2) (2000) 308–317.
[128] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inform. Theory IT-24 (5) (1978) 530–536.
[129] 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.
[130] 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 241 (3) (1997) 579–592.
[131] K. Bhattacharjee, S. Das, PRNG Library, GitHub, 2021, https://github.com/kamalikaB/, Accessed on Nov 26, 2021.