On testing pseudo random generators via statistical tests based on arcsine law[☆]

Paweł Lorek[a], Grzegorz Łoś[b], Filip Zagórski[c], Karol Gotfryd[c]

[a] *Mathematical Institute, University of Wrocław, pl. Grunwaldzki 2/4, 50-384, Wrocław, Poland*
[b] *Institute of Computer Science, University of Wrocław, Joliot-Curie 15, 50-383, Wrocław, Poland*
[c] *Department of Computer Science, Faculty of Fundamental Problems of Technology, Wrocław University of Science and Technology, Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland*

Abstract

NIST SP800-22 [1, 2] is a set of commonly used tools for testing the quality of pseudo random generators. Recently Wang and Nicole [3] pointed out that it is relatively easy to construct some generators whose output is "far" from uniform distribution, but which passes NIST tests. Authors proposed some statistical tests which are based on *Law of Iterated Logarithm* (LIL) which works very good for such generators (*i.e.*,, they do not pass the test). In this paper we propose test which is based on *Arcsine Law* (ASIN). The quality of ASIN is comparable to LIL, however we point out some generators which are obviously flawed and what is detected by ASIN but not by LIL.

Keywords: Arcsine law, Random walks, Pseudo random number generator, Statistical testing

1. Introduction

Random numbers are key ingredients in various applications, e.g., in cryptography (e.g., for generating a secret/public key) or in simulations (e.g., in Monte Carlo methods), just to mention a few. No algorithm can produce truly random numbers. Pseudo random number generators (PRNGs) are used instead. These are deterministic algorithms which produce numbers which we expect to resemble truly random ones *in some sense*. In order to evaluate whether a given PRNG is sound and can be applied for purposes where strong pseudorandomness is required, various tests are applied to its output. The main goal of these tests is to check if the sequence of numbers $\mathbf{U} = (U_1, U_2, \ldots, U_n)$ (or bits, depending on the actual implementation) produced by PRNG resembles a sequence of elements generated independently and uniformly at random. Due to popularity and significance of the problem, a variety of testing procedures have been developed in recent years. Such statistical tests are generally aimed at detecting various deviations in generated sequences what allows for revealing flawed PRNG implementations producing predictable output. These techniques can be divided into several groups according to the approach used for verifying the quality of generators.

The first class of procedures encompasses classical tools from statistics like Kolmogorov-Smirnov test or Pearson's chi-squared test which are used for comparing the theoretical and empirical distributions of appropriate statistics calculated for PRNG's output. It is also possible to adapt tests of normality like Anderson-Darling or Shapiro-Wilk tests for appropriately transformed pseudorandom sequence.

Another category of testing methods relies on the properties of sequences of independent and identically distributed random variables. Based on the original sequence $\mathbf{U}$ returned by the examined PRNG we are able to obtain realizations of other random variables with known theoretical distributions. Some examples of probabilistic laws used in practice in this kind of tests can be found e.g., in [4]. They include such procedures

[☆] Work supported by NCN Research Grant DEC-2013/10/E/ST1/00359

Email addresses: Pawel.Lorek@math.uni.wroc.pl (Paweł Lorek), grzegorz314@gmail.com (Grzegorz Łoś), Filip.Zagorski@pwr.edu.pl (Filip Zagórski), Karol.Gotfryd@pwr.edu.pl (Karol Gotfryd)

like gap test, permutation test and coupon collector's test, just to name a few (see [4] for more detailed treatment). These methods have also the advantage that they implicitly test independence of generator's output.

The main issue with such methods is that a single statistical test looks only at some specific property that holds for sequences of truly random numbers. If a PRNG passes given test, it only implies that generated sequences have this particular property. Hence, for practical purposes bundles of diverse tests are created. Such a test bunches consist of a series of individual procedures based on various stochastic laws from probability theory. A PRNG is then considered as good if the pseudorandom sequences it produces pass all tests in a given bundle. Some examples of such test suites are Marsaglia's Diehard Battery of Tests of Randomness from 1995, Dieharder developed by Brown et al. (cf. [5]), TestU01 implemented by L'Ecuyer and Simard (see [6]) and NIST Test Suite [1]. The last one, designed by National Institute of Standard and Technology is currently considered as one of the state-of the art test bundles and thus widely used for practical purposes. For that reason we shall briefly summarize its most important features. Comprehensive description of all its testing procedures together with additional technical and implementation details can be found in [2]. NIST Test Suite SP800-22 [1] comprises of 15 specific tests for determining whether pseudorandom sequences passed as the input can be treated as good. NIST SP800-22 includes such methods like monobit test for checking what the proportion of zeros and ones in tested sequence is, runs test taking into account the lengths of subsequences consisting of runs of identical bits and serial test for checking for given k what are the frequencies of occurrences of all k-bit overlapping patterns. The procedures in NITS SP800-22 proceed as follows. In each case the null hypothesis is that the tested sequence is a realization of truly random bit sequence. For each test m sequences are generated using tested PRNG. Then for each sequence the test is carried out with selected significance level α (by default $\alpha = 0.01$) resulting in outputting the corresponding p-value. The tested sequence is considered to be random if $p \geq \alpha$. A PRNG is then accepted as sound if the fraction of all input sequences passing the test is about $(1 - \alpha)$. The results can be further inspected by e.g., analyzing the distribution of obtained p-values, as suggested by NIST (cf. [2, 7]).

Although this approach seems to be reliable, it has, however, some inherent limitations. Namely, the key issue is that NIST tests focus merely on characteristics of single binary strings returned by tested PRNG rather that treating them as a whole. This problem can be explained by means of the following example. Suppose that we have some generator G_1 which is considered as good by the NIST Test Suite and let us consider another PRNG G_2 such that every hundredth sequence it produces will be some fixed or highly biased bit string (e.g., with number of 1s highly exceeding number of 0s), and in all remaining cases it returns the output of G_1. G_2 will be likely recognized by NIST SP800-22 as strong PRNG despite its obvious flaws (its output is easily distinguishable from that of truly random generator). This drawback are formalized and carefully discussed by Wang and Nicol [3] in Section 3. They also reported that some PRNGs used in practice which are known to be weak, like Debian Linux (CVE-2008-0166) PRNG based on OpenSSL 0.9.8c-1 or standard C linear congruential generator, pass the tests from NIST SP800-22 using testing parameters recommended by NIST. As we shall see, the method we propose is capable of detecting such flawed generators.

A very interesting approach for testing PRNGs was presented in [8]. The concept of their tests is based on the properties of random walk on $\mathbb{Z}_n$ with 0 being an absorbing state (also known as gambler ruin problem), more precisely, on the time till absorption. The authors propose three different variants of the test. The general idea of the basic procedure is the following. For fixed $p \in (0, 1)$ and $x \in \mathbb{Z}_n$, the output $\mathbf{U} = (U_i)$ of a PRNG is treated as numbers from unit interval and used to define a random walk starting in x such that if $U_i < p$ and the process is in state s, then it moves to $(s + 1) \mod n$, otherwise to $(s - 1) \mod n$. The aim of this test is to compare theoretical and empirical distributions of time to absorption in 0 when starting in x. Based on the values of testing statistics, the PRNG is then either accepted or rejected. The authors reported some "hidden defects" in the widely used Mersenne Twister generator. However, one has to be very careful dealing with randomness: it seems like re-seeding PRNG with a fixed seed was an error which led to wrong conclusions. The criticism was raised by Ekkehard and Grønvik [9], authors also show that tests of Kim et al. [8] performed *properly* do not reveal any defects in Mersenne Twister PRNG.

In recent years a novel kind of testing techniques has been introduced for more careful verification of generators. The core idea of this class of methods is based on the observation that the binary sequence (B_i)

produced by a PRNG, after being properly rescaled, can be interpreted as a one-dimensional random walk $(S_i)_{i\in\mathbb{N}}$ with $S_n = \sum_{i=1}^{n} X_i$, where $X_i = 2B_i - 1$. For random walks defined by truly random binary sequences a wide range of statistics has been considered over the years and a variety of corresponding stochastic laws has been derived (see e.g. [10]) . For a good PRNG we may expect that its output behave as S_n. Hence, the following idea comes to mind: choose some probabilistic law that holds for truly random bit sequences and check the theoretical distribution of the corresponding statistics against the empirical distribution calculated for m sequences produced by examined PRNG in m independent experiments. This comparison can be done e.g., by computing the p-value of appropriate test statistics under the null hypothesis that the sequence generated by PRNG is truly random. Yet another concept named *statistical distance based testing* was suggested in [3], which relies on calculating statistical distances like e.g., total variation or separation distance (cf. formula 10 and 11, respectively) between theoretical and empirical distribution for considered characteristics and rejecting PRNG as weak if the distances are above some threshold. Such approach was adopted in [3], where the authors derive their test statistics from Law of Iterated Logarithm for random walks. The procedure we propose uses similar methodology and is based on Arcsine Law. In the following Section 3 we will recall the aforementioned stochastic laws for random walks, whereas the testing methods will be described in Section 4. Through the paper we will denote by $[n]$ the set $\{1, \ldots, n\}$.

2. Pseudo random generators

2.1. Definition and some standard PRNGs

The intuition behind *pseudo random number generator* is clear. However, we start with a strict definition roughly following Asmussen and Glynn [11].

Definition 2.1. *Pseudo random number generator (PRNG) is a 5-tuple $< E, V, s_0, f, g >$, where E is a finite state space, V is a set of values returned by generator, s_0 is a so-called seed, i.e., initial state in the output sequence $(s_i)_{i=0}^{\infty}$, function $f : E \to E$ describes transition between consecutive states $s_n = f(s_{n-1})$ and $g : E \to V$ maps generator's state into output.*

Usually $V = (0, 1)$ or $V = \{0, 1, \ldots, M\}$ for some $M \in \mathbb{N}$, the latter one is used throughout the paper. Finiteness of state space E implies that for each generator there exists d such that $s_{k+d} = s_d$. Such minimal d is called a *period* of a PRNG. Good PRNG should have long period, optimally equal to $|E|$.

Let us quickly review few known PRNGs.

LCG. Linear congruential generators (LCG) update their state according to the following recursive formula

$$s_n = (as_{n-1} + c) \mod M \ . \tag{1}$$

Generators from this class are defined by three integer parameters – a modulus M, a multiplier a and an additive constant c and are shortly denoted as $LCG(M, a, c)$. Notice that $LCG(M, a, c)$ fulfills the definition 2.1 of PRNG with $E = [M]$, $V = [M]$, $f(x) = (ax + c) \mod M$ and $g(x) = x$. It turns out that choosing the parameters M, a, c such that obtained PRNG has good properties is not so easy. However, there exists a criterion that allows for ensuring the resulting LCG will have possibly long period. It is given by the following Theorem 2.2 from [12].

Theorem 2.2. *Under the following conditions $LCG(M, a, c)$ has a full period (i.e., of length M).*

- *c and M are relatively prime,*
- *if p is a prime number and $p|M$, then $p|(a-1)$,*
- *if $4|M$, then $4|(a-1)$.*

Let us notice, however, that finding LCG with full period does not guarantee that such PRNG will produce good pseudorandom numbers. It is easy to notice the following

Fact 2.3. *Let $M = 2^k$. Then d least significant bits of the numbers returned by $LCG(M, a, c)$ have period at most 2^d.*

In this case it is not possible to achieve independence of pseudorandom numbers outputted by LCG. To partially circumvent this problem, some software packages implementing LCG as the procedure for random number generation truncate the output and return only the most significant bits of the original PRNG's output.

MCG. Multiplicative congruential generator (MCG) is also known as Lehmer generator or Park-Miller PRNG. It is a particular case of LCG with $c = 0$, that is the consecutive states are given by the recurrence relation

$$s_n = as_{n-1} \mod M \ . \tag{2}$$

MCG with parameters M and a will be shortly denoted $MCG(M, a)$. For $MCG(M, a)$ to have good pseudorandomness properties, M is required to be either a prime or a prime power, a has to generate $\mathbb{Z}_M^*$ and the seed s_0 should be relatively prime with M.

GLCG. The above-mentioned PRNGs share some inherent limitation, namely they both have relatively short periods. In the case when we need generators with much longer periods, one may find useful to make use of generalized linear congruential generators (GLCG). They follow the recurrence

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + \ldots + a_k x_{n-k}) \mod M \ . \tag{3}$$

GLCG computing its states as given by the formula above is denoted by $GLCG\left(M, (a_i)_{i=1}^k\right)$. It is indeed a PRNG according to Definition 2.1, with $E = [M]^k$, $s_n = \langle x_n, x_{n-1}, \ldots x_{n-k+1}\rangle$, $g(s_n) = x_n$. A careful choice of generator's parameters may lead to a PRNG with period $M^k - 1$.

Mixed PRNGs. Another possibility to construct a PRNG with longer period is to combine multiple generators into a single one. Suppose we have k PRNGs $\langle E_j, V_j, s_{j,0}, f_j, g_j\rangle$, $1 \leq j \leq k$, where j^{th} generator changes its state according to some update function f_j, i.e.

$$s_{j,n} = f_j(s_{j,n-1}) \ .$$

We are able to merge these PRNGs into a mixed generator such that the sequence of its states will be defines as

$$s_n = \langle s_{1,n}, s_{2,n}, \ldots s_{k,n}\rangle \ .$$

Moreover, let d_j be the period of a PRNG being the j^{th} component of mixed generator. As shown by L'Ecuyer (see Lemma 2 in [13]), such combined generator has period $d = \operatorname{lcm}(d_1, d_2, \ldots, d_k)$.

A special case of mixed generators are combined multiplicative congruential generators (CMCGs). They consist of k generators $MCG(M_j, a_j)$, where M_j are primes. Hence, the transition function between the states is

$$s_{j,n} = a_j s_{j,n-1} \mod M_j \ .$$

The output of such mixed generator is determined by the formula

$$g(s_n) = \left(\sum_{j=1}^{k} (-1)^{j-1} s_{j,n}\right) \mod M_1 - 1 \ .$$

Furthermore, if the numbers $\frac{M_j - 1}{2}$ are coprime, then CMCG has an optimal period of length $\frac{1}{2^k}(M_1 - 1) \cdot \ldots \cdot (M_k - 1)$.

LFSR. Linear feedback shift register (LFSR) is an example of generator where produced pseudorandom numbers are the output of boolean circuit consisting of binary gates (usually XOR gates) and shift register containing a sequence of bits. Due to its design, there exist both software and hardware-based implementations of LFSRs. The internal state of such PRNG is the content of shift register. The state is updated by shifting the binary sequence in register with input bit derived from its previous state. Both the next input bit and generators' output are linear functions of its state (usually a combination of exclusive-ORs of bits from some predefined positions). LFSR is seeded with some initial bit sequence which determines all its successive outputs. Because all operations are deterministic transformations of the internal state from finite state space, the state of LFSR will eventually be repeated, thus resulting in cyclic repetition of the outputted sequence. A careful choice of the function defining feedback for LFSR allows for achieving a PRNG with relatively long period. Individual LFSRs could be combined in more complex circuits allowing to produce nonlinear output. For more formal treatment of LFSRs see *e.g.,* Chapter 6.2 in [14].

Mersenne Twister. Mersenne Twister (MT19937) was proposed by Matsumoto and Nashimura in [15]. This is an example of specific PRNG with predefined parameters rather than a family of generators, which has gained particular popularity. Implementations of Mersenne Twister have been widely used in practice as standard PRNGs in many programming languages, including standard library of C++11, R, Python, Matlab or Julia.

In the generators' description we will use the following notations. Let $d[i..j]$ denotes bits on positions from i to j in fixed-length binary representation of number d (starting from LSB). Bitwise XOR and AND will be denoted by $\oplus$ and $\&$, respectively. We will use $\ll$ and $\gg$ for left and right bitwise shift operators.

The state of PRNG is defined by 624 32-bits numbers

$$x_k, x_{k+1}, \ldots x_{k+623}.$$

Subsequent states of the generator are calculated according to the formula

$$x_{k+624} = \begin{cases} x_{397+k} \oplus (0, x_k[0], & \\ \quad x_{k+1}[1..30]) & \text{if } x_{k+1}[31] = 0 \\ x_{397+k} \oplus (0, x_k[0], & \\ \quad x_{k+1}[1..30]) \oplus a & \text{if } x_{k+1}[31] = 1, \end{cases}$$

where $a = (9908B0D)_{16}$. k^{th} call to MT19937 results in outputting the value of $t(x_{623+k})$ for

$$t(x) = y_3 \oplus (y_3 \gg 18),$$

where

$$\begin{aligned} y_3 &= y_2 \oplus ((y_2 \ll 15) \ \& \ (EFC60000)_{16}), \\ y_2 &= y_1 \oplus ((y_1 \ll 7) \ \& \ (9D2C5680)_{16}), \\ y_1 &= x \oplus (x \gg 11) \ . \end{aligned}$$

The construction described above leads to a PRNG with period $2^{19937} - 1$, what explains its abbreviated name.

2.2. Generating random bits

In this section we will briefly discuss some issues on generating random binary sequences.

It is clear that both the input and the output of a random number generator can be viewed as a finite sequence of bits. For a PRNG to be considered as sound, the outputted sequences should have some particular property, namely each returned bit has to be generated independently with equal probability of being 0 and 1. For convenience let us introduce the following

Definition 2.4. *A truly random bit sequence is a Bernoulli process with success probability* $p = \frac{1}{2}$.

Given a PRNG G outputting integers from the set $\bar{M} = \{0, 1, \ldots, M-1\}$, we may obtain a pseudorandom binary sequence with any given length using the following simple procedure. Namely, while the bit sequence s is not sufficiently long, generate the next pseudorandom number a and concatenate its binary representation (on $\lceil \log_2 M \rceil$ bits) with the current content of s. In the ideal model with G being truly random number generator, such algorithm produces truly random bit sequences provided that M is a power of 2. Indeed, for $M = 2^k$ there is one to one correspondence between k-bit sequences and the set $\bar{M}$. Hence, when each number is generated independently with uniform distribution on $\bar{M}$, then each combination of k bits is equally likely and therefore each bit of the outputted sequence is independent and equal to 0 or 1 with probability $\frac{1}{2}$.

However, this is not true for $M \neq 2^k$. It is easy to observe that in such case the generator is more likely to outputs 0's and generated bits are no longer independent. To overcome this issue one may simply discard the most significant bits. This still results in non-uniform distribution, though the deviations are relatively small. Another approach is to take d first bits from binary representation of $\frac{a}{M}$ for some fixed d instead of simply outputting bits of a. Such method has the advantage that it can be easily adopted for the underlying generator returning numbers from the unit interval, what is common for many PRNG implementations.

3. Stochastic laws for random walks

Let $(B_i)_{i\geq 0}$ be a *Bernoulli process* with parameter $p \in (0, 1)$, *i.e.,* the sequence of independent random variables with identical distribution $P(B_1 = 1) = 1 - P(B_1 = 0) = p$. A good PRNG should *behave* as a generator of Bernoulli process with $p = 1/2$ (what we assume from now on). It will be more convenient to consider

$$X_i = 2B_i - 1, \quad S_n = \sum_{i=1}^{n} X_i. \tag{4}$$

The sequence X_i is $\{-1, +1\}$-valued, the process $(S_i)_{i\in\mathbf{N}}$ is called a random walk.

Of course $|S_n| \leq n$. However, large values of $|S_n|$ have small probability and in practice the values of S_n are in a much narrower interval then $[-n, n]$. *Weak* and *Strong Law of Large Numbers* imply that $\frac{S_n}{n} \xrightarrow{P} 0$, and even $\frac{S_n}{n} \overset{a.s.}{=} 0$, where $\xrightarrow{P}$ denotes *convergence in probability* and $\overset{a.s.}{\to}$ denotes *almost sure convergence*. Thus, the deviations of S from 0 grow much slower then linearly. On the other hand *Central Limit Theorem* states that $\frac{S_n}{n} \xrightarrow{D} \mathcal{N}(0, 1)$ (where $\xrightarrow{D}$ denotes convergence in distribution), what is in some sense a lower bound on fluctuations of S_n – they will leave interval $[-\sqrt{n}, \sqrt{n}]$ since we have $\limsup_{n\to\infty} \frac{S_n}{\sqrt{n}} = \infty$ (implied by 0-1 Kolmogorov's Law). It turns out that the fluctuations can be estimated more exactly:

Theorem 3.1 ([16], cf. also Chapter VIII.5 in [10])**.** *For a random walk S_n we have*

$$\begin{aligned} \mathbb{P}\left(\liminf_{n\to\infty} \frac{S_n}{\sqrt{2n\log\log n}} = -1\right) &= 1, \\ \mathbb{P}\left(\limsup_{n\to\infty} \frac{S_n}{\sqrt{2n\log\log n}} = +1\right) &= 1. \end{aligned}$$

The observations are summarized in Table 1. Thus, to normalize S_n: dividing by n is *too much* and dividing by $\sqrt{n}$ is *too weak*, the fluctuations of S_n from 0 grow proportionally to $\sqrt{2n\log\log n}$.

In Figure 1 500 trajectories of random walks of length 2^{30} are depicted, the darker the image the higher the density of trajectories. We can see that $\pm\sqrt{2n\log\log n}$ roughly corresponds to the fluctuations of S_n. However, few trajectories after around billion steps are still outside $[-\sqrt{2n\log\log n}, \sqrt{2n\log\log n}]$. The Law of Iterated Logarithm tells us that for *appropriately* large n the trajectories will not leave $[-\sqrt{2n\log\log n}, \sqrt{2n\log\log n}]$ with probability 1. The conclusion is that n must be much larger.

One could think that the following is a good test for randomness: Fix some number, say 100 and classify PRNG as "good" if a difference between number of ones and zeros never exceeds 100. The large difference could suggest that zeros and ones have different probabilities of occurring. However, LIL shows us that this reasoning is wrong, we *should* expect some fluctuations, the absence of which means that PRNG does not

produce bits which can be considered random. This property of random walks was used by the authors in [3] for designing a novel method of testing random number generators (as mentioned in Section 2.2, the output of any PRNG can be considered as a binary sequence, thus it defines some one-dimensional random walk). We shall shortly recall the details of their technique in Section 4.2.

There is yet one more interesting property. Define $S_n^{LIL} = \frac{S_n}{\sqrt{2n \log\log n}}$. LIL implies that S_n^{LIL} does not converge pointwise to any constant. However, it converges to 0 in probability. Thus, let us fix some small $\varepsilon > 0$. For almost all n with arbitrary high probability $p < 1$ the process S_n^{LIL} will not leave $(-\varepsilon, \varepsilon)$. On the other hand it tells us that the process will be outside this interval infinitely many times. This apparent contradiction shows how unreliable our intuition can be on phenomena taking place at infinity.

3.1. Arcsine Law

The sequence X_i defined in (4) can be treated as a result of the following game. Two players are tossing a *perfect* coin. If Head – first player receives one dollar from the other, otherwise it gives a dollar to the other player. Then S_n corresponds to profit of, say, first player after n games. The observations described in previous section imply that *averaging everything* S_n will spend half of its time above the 0-axis and half of its time below. However, the typical situation is counter-intuitive (at first glance): typically the random walk will either spend most of its time above or most of its time below 0-axis. This is expressed in the Theorem 3.2 below (see e.g. [10], Chapter XII.8 and [17], Chapter III.4). Before we formulate the theorem, let us first introduce some notations. For given $X_1, \ldots, X_n$ let

$$D_k = \mathbb{1}\left(S_k > 0 \vee S_{k-1} > 0\right), k = 1, 2, \ldots, n. \tag{5}$$

D_k is equal to 1 if number of ones prevails number of zeros until k-th step and 0 otherwise (in case of ties, i.e., $S_k = 0$, we look at the previous step letting $D_k = D_{k-1}$). In other words, $D_k = 1$ corresponds to the situation when the line segment of the trajectory of random walk between steps $k-1$ and k is above the 0-axis.

Theorem 3.2 (Arcsine Law for random walk)**.** *Define* $L_{2n} = \sum_{k=1}^{2n} D_k$. *For* $x \in (0, 1)$ *we have*

$$\mathbb{P}\left(L_{2n} \leq x \cdot 2n\right) \xrightarrow[n\to\infty]{} \frac{1}{\pi} \int_0^x \frac{dt}{\sqrt{t(1-t)}} = \frac{2}{\pi} \arcsin\sqrt{x}\ .$$

$\mathbb{P}\left(L_{2n} \leq x \cdot 2n\right)$ is the chance that the random walk was above 0-axis for at most x fraction of time. The limiting distribution is called *arcsine distribution* and has density $f(t) = \frac{1}{\pi}\sqrt{t(1-t)}$ and cdf $F(t) = \frac{2}{\pi} \arcsin\sqrt{t}$. Both $f(t)$ and $F(t)$ are depicted in Fig. 2. The shape of pdf $f(t)$ clearly indicates that the fractions of time spent above and below 0-axis are more likely to be unequal than close to each other.

Remark. The intuition suggests that S_n should be more or less half of the time above 0-axis. It seems reasonable: it is known that number of visits to 0 is infinite in infinitely long game. Moreover, Strong Law of Large Numbers *seems* suggesting that the fraction of time spent above 0-axis should be similar to the fraction of time spent below 0-axis. The conclusion would be that around half of the time should be spent above/below this axis. However, SLLN is for random variables with finite mean, that is why it is not applicable in this case.

4. Testing PRNGs based on Arcsine Law

In this Section we will show how to exploit the theoretical properties of random walks discussed in preceding sections to design a practical routine for testing PRNGs. We briefly describe the approach based on Arcsine Law we employ in experiments presented in Section 5. For the sake of completeness we will also recall some basic facts about the method of evaluating PRNGs presented in [3]. Finally, we will perform the analysis of approximation errors that occur in our testing procedure.

4.1. Arcsine Law based testing

The general idea of tests is clear: take a sequence of bits generated by PRNG (treat them as -1 and $+1$) and compare empirical distributions with distributions for truly random numbers (which are known). More exactly: we will compare it to Arcsine Law given in Theorem 3.2. Let us define

$$S_n^{asin} = \frac{1}{n}\sum_{k=1}^{n} D_k. \tag{6}$$

S_n^{asin} is a fraction of time instants at which ones prevail zeros. From Arcsine Law (Theorem 3.2) we can conclude that *for large* n we have

$$\mathbb{P}\left(S_n^{asin} \in (a,b)\right) \approx \frac{1}{\pi}\int_a^b \frac{dt}{\sqrt{t(1-t)}} = \frac{2}{\pi}\arcsin(\sqrt{b}) - \frac{2}{\pi}\arcsin(\sqrt{a}). \tag{7}$$

To test PRNG we generate m sequences of length n. We obtain thus m realizations of a variable S_n^{asin}, the j-th replication is denoted by $S_{n,j}^{asin}$. We fix some partition of a real line and count the number of realizations of S_n^{asin} which are in a given interval. In our tests we will use $(s+2)$ element partition $\mathcal{P}_s^{asin} = \{P_0^{asin}, P_1^{asin}, \ldots, P_{s+1}^{asin}\}$, where

$$\begin{aligned} P_0^{asin} &= \left(-\infty, -\frac{1}{2s}\right), \\ P_i^{asin} &= \left[\frac{2i-3}{2s}, \frac{2i-1}{2s}\right), \quad 1 \le i \le s, \\ P_{s+1}^{asin} &= \left[1 - \frac{1}{2s}, \infty\right). \end{aligned}$$

Now we define two measures on $\mathcal{P}_s^{asin}$, for $0 \le i \le s+1$:

$$\mu_n^{asin}\left(P_i^{asin}\right) = \mathbb{P}\left(S_n^{asin} \in P_i^{asin}\right), \tag{8}$$

$$\nu_n^{asin}\left(P_i^{asin}\right) = \frac{|\{j:\ S_{n,j}^{asin} \in P_i^{asin}, 1 \le j \le m\}|}{m}. \tag{9}$$

Measure μ_n^{asin} corresponds to theoretical distribution and can be calculated from (7), whereas ν_n^{asin} represents empirical distribution obtained from tests. If the PRNG is "good" then the measures should be close one to each other, what we will measure in various of ways. First, we will calculate *total variation* and *separation* distances defined for partition $\mathcal{P}$ as follows:

$$d_{\mathcal{P}}^{tv}(\mu,\nu) = \frac{1}{2}\sum_{A\in\mathcal{P}} |\mu(A) - \nu(A)|, \tag{10}$$

$$d_{\mathcal{P}}^{sep}(\mu,\nu) = \max_{A\in\mathcal{P}}\left(1 - \frac{\mu(A)}{\nu(A)}\right). \tag{11}$$

Another approach is based on hypothesis testing. Using statistical terminology $S_{n,j}^{asin}$ will be called observations. Null hypothesis is that observations have distribution μ_n^{asin} , *i.e.,* that PRNG produces random bits. Define

$$O_i = |\{j:\ S_{n,j}^{asin} \in P_i^{asin}, 1 \le j \le m\}|, \quad 0 \le i \le s+1, \tag{12}$$

the number of observations within P_i^{asin} interval. Let E_i denote expected number of observations within this interval, *i.e.,* $E_i = m \cdot \mathbb{P}\left(S_n^{asin} \in P_i^{asin}\right)$ (what is calculated from (7)). Assuming null hypothesis the statistic

$$T^{asin} = \sum_{i=0}^{s+1} \frac{(O_i - E_i)^2}{E_i} \tag{13}$$

has approximately $\chi^2(s+1)$ distribution. Large values of T^{asin} mean that PRNG is not good.

Summarizing, for given PRNG we generate m sequences of length n each. Then we calculate $d^{tv}_{\mathcal{P}}(\mu^{asin}_n, \nu^{asin}_n)$, $d^{sep}_{\mathcal{P}}(\mu^{asin}_n, \nu^{asin}_n)$ and T^{asin}. Instead of fixing the threshold for d^{tv}, d^{sep} and T^{asin}, we simply calculate these measures/statistics and compare several PRNGs relatively.

4.2. LIL based testing

The procedure of testing pseudorandom number generators based on Law of Iterated Logarithm (see Theorem 3.1) was proposed in [3]. This method is similar to that based on Arcsine Law discussed in details in Section 4.1, hence we will point out only the most significant differences. The LIL based testing procedure merely differs in that instead of S^{asin}_n we calculate the characteristic

$$S^{lil}_n = \frac{S_n}{\sqrt{2n \log\log n}} \tag{14}$$

and use an appropriate partition of the real line.

Theoretical distribution of S^{lil}_n for truly random bit sequence can be obtained using CLT. Denoting by Φ the cdf of standard normal distribution $\mathcal{N}(0,1)$ and letting $l(n) = \sqrt{2\log\log n}$, we have

$$\begin{aligned} \mathbb{P}\left(S^{lil}_n \in (a,b)\right) &= \mathbb{P}\left(\frac{S_n}{\sqrt{n}} \in (a\,l(n), b\,l(n))\right) \\ &\approx \Phi(b\,l(n)) - \Phi(a\,l(n)). \end{aligned} \tag{15}$$

As the support of S^{lil}_n differs from that of S^{asin}_n, we have to adjust the partition of real line, namely we will use $\mathcal{P}^{lil}_s = \{P^{lil}_0, P^{lil}_1, \ldots, P^{lil}_{s+1}\}$, where

$$\begin{aligned} P^{lil}_0 &= (-\infty, -1), \\ P^{lil}_i &= \left[-1 + \frac{2(i-1)}{s}, -1 + \frac{2i}{s}\right), \quad 1 \le i \le s, \\ P^{lil}_{s+1} &= [1, \infty). \end{aligned}$$

The theoretical and empirical measures on $\mathcal{P}^{lil}_s$ in this case are defined as... for $0 \le i \le s+$!

$$\mu^{lil}_n\left(P^{lil}_i\right) = \mathbb{P}\left(S^{lil}_n \in P^{lil}_i\right), \tag{16}$$

$$\nu^{lil}_n\left(P^{lil}_i\right) = \frac{|\{j :\ S^{lil}_{n,j} \in P^{lil}_i, 1 \le j \le m\}|}{m}. \tag{17}$$

Theoretical distribution μ^{lil}_n of the statistic S^{lil}_n can be directly approximated from (15). The measure ν^{lil}_n describes empirical distribution derived from the outcomes of tests (with $S^{lil}_{n,j}$ being the j^{th} realization of random variable S^{lil}_n calculated for the j^{th} generated sequence). All the remaining stages of this testing procedure are the same as for arcsine test.

4.3. Error analysis

Our test relies on (7), however some approximation "$\approx$" is used there. Here we will show that for $n \ge 2^{25}$ possible error is negligible.

Lemma 4.1. *For any fixed a and b such that $0 \le a < b \le 1$ and for $n \ge 2^{25}$ the approximation error of the probability $\mathbb{P}\left(S^{asin}_n \in (a,b)\right)$ in (7) can be upper bounded by 10^{-5}, i.e.*

$$\left|\mathbb{P}\left(S^{asin}_n \in (a,b)\right) - \frac{1}{\pi}\int_a^b \frac{dt}{\sqrt{t(1-t)}}\right| \le 10^{-5}\ . \tag{18}$$

Proof. Let $p_{2k,2n}$ denote the probability that during $2k$ steps in first $2n$ steps the random walk was above 0-axis, *i.e.,* $p_{2k,2n} = \mathbb{P}(L_{2n} = 2k)$. Standard results on simple random walk show that

$$p_{2k,2n} = \binom{2k}{k}\binom{2n-k}{n-k}2^{-2n}. \tag{19}$$

The standard proof of Theorem 3.2 shows that $p_{2k,2n}$ is close to $d_{k,n} = \frac{1}{\pi\sqrt{k(n-k)}}$. We will show that $|p_{2k,2n} - d_{k,n}|$ is small. A version of Stirling's formula states that for each n there exists θ_n, $0 < \theta_n \leq 1$, such that

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \exp\left\{\frac{\theta_n}{12n}\right\}. \tag{20}$$

Plugging (20) into each factorial appearing in (19) we have

$$p_{2k,2n} = \frac{1}{\pi\sqrt{k(n-k)}} \exp\left\{\frac{\theta_{2k} - 4\theta_k}{24k} + \frac{\theta_{2(n-k)} - 4\theta_{n-k}}{24(n-k)}\right\}. \tag{21}$$

Thus we have

$$\frac{p_{2k,2n}}{d_{k,n}} \leq \exp\left\{\frac{1}{24k} + \frac{1}{24(n-k)}\right\} = \exp\left\{\frac{n}{24k(n-k)}\right\}$$

and

$$\frac{p_{2k,2n}}{d_{k,n}} \geq \exp\left\{\frac{-4}{24k} + \frac{-4}{24(n-k)}\right\} = \exp\left\{-\frac{n}{6k(n-k)}\right\}.$$

Using the fact that $e^x - 1 \leq 2x$ for sufficiently small $x > 0$ and $1 - e^{-x} \leq x$ we have

$$\begin{aligned} p_{2k,2n} - d_{k,n} &\leq d_{k,n}\left(\exp\left\{\frac{n}{24k(n-k)}\right\} - 1\right) \leq d_{k,n}\frac{n}{12k(n-k)} \\ d_{k,n} - p_{2k,2n} &\leq d_{k,n}\left(1 - \exp\left\{-\frac{n}{6k(n-k)}\right\}\right) \leq d_{k,n}\frac{n}{6k(n-k)} \end{aligned}$$

what implies

$$|p_{2k,2n} - d_{k,n}| \leq d_{k,n}\frac{n}{6k(n-k)} = \frac{n}{6\pi\left(k(n-k)\right)^{\frac{3}{2}}}.$$

Fix $\delta > 0$ and assume furthermore that $\delta \leq \frac{k}{n} \leq 1 - \delta$. The function $k \mapsto (k(n-k))^{3/2}$ achieves minimal value on the border of considered interval, thus

$$|p_{2k,2n} - d_{k,n}| \leq \frac{n}{6\pi\left(\delta n(n - \delta n)\right)^{\frac{3}{2}}} = \frac{1}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}}.$$

We will estimate the approximation error in (7) in two steps. First, take two numbers a, b such that $\delta \leq a < b \leq 1 - \delta$. Then

$$\begin{aligned} &\left|\sum_{a\leq\frac{k}{n}\leq b} p_{2k,2n} - \sum_{a\leq\frac{k}{n}\leq b} d_{k,n}\right| \leq \sum_{a\leq\frac{k}{n}\leq b} |p_{2k,2n} - d_{k,n}| \\ &\leq \sum_{a\leq\frac{k}{n}\leq b} \frac{1}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}} = \frac{\lceil bn - an\rceil}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}} \\ &\leq \frac{b-a}{3\pi n\left(\delta(1-\delta)\right)^{\frac{3}{2}}} \leq \frac{1}{3\pi n\left(\delta(1-\delta)\right)^{\frac{3}{2}}}. \end{aligned} \tag{♦}$$

The second source of errors in probability estimation in (7) is approximating the sum by integral. Let us consider an arbitrary function f differentiable in the interval (a, b). Split (a, b) into subintervals of length $\frac{1}{n}$

and let x_k be an arbitrary point in the interval containing $\frac{k}{n}$. Denote by M_k and m_k maximal and minimal value of f on that interval, respectively. Using the Lagrange's mean value theorem we obtain

$$\begin{aligned}\left|\int_a^b f(x)dx - \sum_{a\le\frac{k}{n}\le b}\frac{1}{n}f(x_k)\right| &\le \sum_{a\le\frac{k}{n}\le b}\frac{1}{n}(M_i-m_i)\\ &= \sum_{a\le\frac{k}{n}\le b}\frac{1}{n^2}|f'(\xi_i)| \le \sum_{a\le\frac{k}{n}\le b}\frac{1}{n^2}\sup_{a\le x\le b}|f'(x)|\\ &= \frac{\lceil bn-an\rceil}{n^2}\sup_{a\le x\le b}|f'(x)| \le \frac{2(b-a)}{n}\sup_{a\le x\le b}|f'(x)|.\end{aligned}$$

For $f(x)=\frac{1}{\pi\sqrt{x(1-x)}}$ we have $f'(x)=\frac{2x-1}{2\pi(x(1-x))^{3/2}}$ and $\frac{1}{n}f\left(\frac{k}{n}\right)=d_{k,n}$. Hence, in the considered interval $(a,b)\subseteq(\delta,1-\delta)$

$$\begin{aligned}\left|\int_a^b f(x)dx - \sum_{a\le\frac{k}{n}\le b} d_{k,n}\right| &\le \frac{2}{n}\sup_{\delta<x<1-\delta}|f'(x)|\\ &= \frac{1-2\delta}{\pi n(\delta(1-\delta))^{\frac{3}{2}}}. \qquad (\bigstar)\end{aligned}$$

In our tests we use the partition $\mathcal{P}_{40}^{asin}$ of the real line, thus we set $\delta=\frac{1}{80}$. Moreover, in all tests $n\ge 2^{25}$. For these parameters' values

$$(\blacklozenge)\le\frac{77.4}{n}\le 2.31\cdot 10^{-6} \text{ and } (\bigstar)\le\frac{226.3}{n}\le 6.75\cdot 10^{-6}$$

Finally,

$$\begin{aligned}\left|\int_a^b f(x)dx - \sum_{a\le\frac{k}{n}\le b} p_{2k,2n}\right| &\le \left|\int_a^b f(x)dx - \sum_{a\le\frac{k}{n}\le b} d_{k,n}\right| + \left|\sum_{a\le\frac{k}{n}\le b} d_{k,n} - \sum_{a\le\frac{k}{n}\le b} p_{2k,2n}\right|\\ &=(\blacklozenge)+(\bigstar)\le 9.06\cdot 10^{-6}\le 10^{-5},\end{aligned}$$

what justifies the approximation (7) when $\delta\le a<b\le 1-\delta$.

To complete the analysis we need to investigate the errors "on the boundaries" of unit interval, i.e. for $(0,\delta)$ (and, by symmetry, for $(1-\delta,\delta)$). We get

$$\begin{aligned}\left|\int_0^\delta f(x)dx - \sum_{0\le\frac{k}{n}<\delta} p_{2k,2n}\right| &= \left|\int_0^{\frac{1}{2}} f(x)dx - \int_\delta^{\frac{1}{2}} f(x)dx - \sum_{0\le\frac{k}{n}\le\frac{1}{2}} p_{2k,2n} + \sum_{\delta\le\frac{k}{n}\le\frac{1}{2}} p_{2k,2n}\right|\\ &= \left|\frac{1}{2} - \int_\delta^{\frac{1}{2}} f(x)dx - \frac{1}{2} + \sum_{\delta\le\frac{k}{n}\le\frac{1}{2}} p_{2k,2n}\right| = \left|\int_\delta^{\frac{1}{2}} f(x)dx - \sum_{\delta\le\frac{k}{n}\le\frac{1}{2}} p_{2k,2n}\right| \le 10^{-5},\end{aligned}$$

where the last inequality follows directly from the preceding calculations. Thus, the above analysis shows that the approximation error can be neglected. □

5. Experimental results

In this section we present and discuss a series of experimental results of testing some widely used PRNGs implemented in standard libraries in various programming languages. We performed both arcsine and LIL based tests on generators including different implementations of standard C/C++ linear congruential generator, the standard generator RAND from GNU C Library, Mersenne Twister, Minstd and CMRG generators. As our last example we propose some hypothetical weak PRNG which is clearly identified by our arcsine test as non-random, whereas the LIL test fails to detect its obvious flaws.

Each considered PRNG was tested by generating $m = 10000$ sequences of length $n = 2^{34}$ in most cases. The values of characteristics $S_\bullet^{asin}$ or $S_\bullet^{lil}$ were calculated both for each sequence as well as for subsequences of length $n/2, \ldots, n/2^k$ for some fixed value of k depending on tested PRNG (usually $k = 8$). This approach, called in [3] *snapshot testing* at points $2^n, \ldots, 2^{n/2^k}$, allows for observing and comparing the values of analyzed parameters on different stages. Finally, for each PRNG respective statistics and distances between theoretical and empirical distributions were calculated, as described in detail in Sections 4.1 and 4.2. In our experimental analysis for tests based on characteristics S_n^{asin} and S_n^{lil} we used partitions $\mathcal{P}_{40}^{asin}$ and $\mathcal{P}_{40}^{lil}$, respectively.

In the experiments we used our own implementations of tested PRNGs (except MT19937). The generators were initialized with random seeds from `http://www.random.org` [18] and each sequence was generated using different seed.

In the following sections we provide the outcomes of performed tests as well as discussion and analysis of the obtained results. The outcomes of experiments for each PRNG are summarized in tables in the following way. The first row denotes the length n of generated subsequences. For a subsequence of length n_k the theoretical probability measure μ_{n_k} was calculated according to Formula (8) or (16) and the empirical measure ν_{n_k} according to (9) or (17) depending on considered characteristic. Successive columns contain the values of $d^{tv}(\mu_{n_k}, \nu_{n_k})$, $d^{sep}(\mu_{n_k}, \nu_{n_k})$, $d^{sep}(\nu_{n_k}, \mu_{n_k})$ and p-value of the statistic (5.9).

5.1. RANDU

Let us start with an example of PRNG designed in the early 1960s which is no longer used. RANDU was mentioned in Section 2.1 as an example of a generator definitely failing the spectral test. This generator is defined as $MCG(2^{31}, 65539)$. Selecting 65539 as the multiplicative constant seemed to be good as $65539 = 2^{16} + 3$ what allows for performing fast multiplication on hardware.

The results of testing RANDU are shown in Tables 2 and 3.

As we can observe, there is no need to generate very long sequences of pseudorandom bits to observe that the output of RANDU is far from truly random. This explains why this PRNG was quickly superseded by other algorithms.

5.2. Microsoft Visual C++ rand()

Function `rand` in Microsoft Visual C++ is based on $LCG(2^{32}, 214013, 2531011)$. It differs from an ordinary LCG in that it returns only the bits on positions 30..16. This PRNG is one of those tested in [3]. As the authors in [3], we discard the least significant 7 bits of numbers produced by the `rand` function (*i.e.,* we take into account only the MSB on positions 30..23 of the number generated by LCG). The results of performed tests are presented in Tables 4 and 5.

Notice that this LCG has period 2^{31} and from single call of `rand` we get $8 = 2^3$ bits. This implies that when generating a sequence of bits of length 2^{34} we go through the full cycle of generator. Hence, each combination of 8 bits has to be generated the same number of times, what leads to the conclusion that after 2^{34} steps the random walk always returned to 0. This is evident in test results for characteristic S_n^{lil}. For $n = 2^{34}$ all observations fall into the interval $[0, 0.05)$, for which the theoretical measure $\mu_n^{lil}([0, 0.05)) \sim 0.05$, thus $d^{tv}(\mu_n^{lil}, \nu_n^{lil}) \approx 0.95$.

It is clear that generators with short periods cannot be used for generating large amounts of pseudorandom numbers. However, observe that the PRNG analyzed in this section fails the tests even for $n = 2^{26}$ and generating a random walk of this length requires only $2^{23}/2^{31} = 1/2^8 \approx 0.4\%$ of the full period. Thus, this

PRNG cannot be considered as random even when restricting only to generators with short periods. What is **surprising**, this PRNG passes the NIST Test Suite [1], as pointed out in [3].

5.3. Borland C/C++ rand()

Procedure `rand` from standard library in Borland C/C++ implements $LCG(2^{32}, 22695477, 1)$. Similarly to `rand` in MS Visual C++, it returns only the bits on positions 30..16. Like in previous example in experiments we only use 8 most significant bits of numbers returned by tested PRNG. Tables 6 and 7 contain the outcomes of experiments carried out for Borland C/C++ stdlib `rand`. They are very akin to those for standard PRNG in MS Visual C++ and the same remarks as in Section 5.2 apply.

5.4. BSD libc rand()

Function `rand` from BSD system standard library has used originally $LCG(2^{31}, 1103515245, 12345)$ and unlike two previously discussed LCG generators it does not truncate its output, i.e., it returns all bits of generated numbers. Hence, in our experiments with BSD `rand` we also adopt the full output produced by this LCG. Tables 8 and 9 clearly depict why the original implementation of this generator has been changed.

From the obtained results one can easily notice that even for short sequences the output of BSD `rand` function turns out to be far from truly random. The reason behind that is the LCG uses modulus 2^{31}. Fact 2.3 implies, that in such case the period of d least significant bits equals to 2^d. As a result the number of 0s and 1s is fairly close to each other what is easily recognized by the tests based on properties of random walks. It is worth mentioning that this flaw is more evident in the outcomes of tests based on S_n^{lil} characteristic, which outperforms those based on S_n^{asin} in this case.

5.5. GLIBC standard library rand()

Function `rand` in GNU C Library makes use of more complicated generator than these described in the preceding sections. Its state is determined by 34 numbers $x_i, x_{i+1}, \ldots, x_{i+33}$. The PRNG is seeded with some random number s, $0 \leq s \leq 2^{31}$, and its initial state is given by

$$\begin{aligned} &x_0 = s \\ &x_i = 16807 x_{i-1} \mod (2^{31} - 1), && 0 < i < 31 \\ &x_i = x_{i-31}, && i \in \{31, 32, 33\}. \end{aligned}$$

The successive values x_i are calculated according to the formula

$$x_i = (x_{i-3} + x_{i-31}) \mod 2^{32}.$$

k^{th} call to the function `rand` results in returning $x_{k+343} \gg 1$.

For our tests we took all 31 bits outputted by PRNG. From the results gathered in Tables 10 and 11 one may conclude that arcsine test gives no reason for rejecting the hypothesis that the sequences generated by the analyzed PRNG are random. In the case of LIL test, the results for $n = 2^{34}$ and $n = 2^{33}$ may also suggest that this PRNG is good. But for $n = 2^{32}$ we can observe an evident deviation, namely the p-value is very small, only about 1/1000. This can be a matter of chance – with probability 1/1000 this could happen even for a truly random generator. However, take a look on p-values for ten data subsets of size 1000 (*i.e.,* each sample consists of 1000 sequences). We have 0.9313, 0.0949, 0.8859, 0.1739, 0.3675, 0.0334, 0.0321, 0.1824, 0.0017, 0.6205. As we can see, 4 out of 10 p-values are below 0.1. The probability that for truly random sequences at least 4 p-values fall into that interval is

$$1 - \sum_{i=0}^{3} \binom{10}{i} \left(\frac{1}{10}\right)^i \left(1 - \frac{1}{10}\right)^{10-i} \approx 0.0016 \ .$$

This suggests that the GLIBC standard library PRNG may contain some hidden flaws. However, despite the issue pointed above, the implementation of `rand` function considered in this section appears to be the best among these already tested by us.

5.6. Minstd

Minstd (abbr. from *minimal standard generator*) is based on MCG with parameters recommended by Park and Miller in [19]. The authors' goal was to develop a simple generator, not necessarily perfect, but fast, simple to implement and suitable for most common applications. As a result they proposed $MCG(2^{31} - 1, 16807)$. For generating binary sequences eight most significant bits returned by the PRNG were used. The results of experimental testing are shown in Tables 12 and 13.

In their later work from 1993, Park and Miller [20] suggested that it would be better to use the multiplier 48271. The effect of changing multiplier on the testing results can be viewed in the Tables 14 and 15.

For the arcsine tests the total variation distance between theoretical and empirical measures decreased in most cases after changing the multiplier, especially for longer sequences. But in the case of LIL tests the opposite change occurred. Thus, based on these results it is hard to say whether this modification brings significant improvement. Nevertheless, χ^2 test in both cases shows that Minstd is not a sound PRNG.

Despite its weaknesses, Minstd became a part C++11 standard library. It is implemented by the classes `std::minstd_rand0` (with multiplier 16807) and `std::minstd_rand` (with multiplier 48271).

5.7. CMRG

CMRG, which stands for *combined multiple recursive generator*, is one of so called combined generators, discussed in paragraph 2.1. This PRNG returns numbers from Z_n computed according to the following formula

$$\begin{aligned} Z_n &= X_n - Y_n \mod 2^{31} - 1 \\ X_n &= 63308X_{n-2} - 183326X_{n-3} \mod 2^{31} - 1 \\ Y_n &= 86098Y_{n-1} - 539608Y_{n-3} \\ &\quad \mod 2^{31} - 2000169 \end{aligned} \tag{22}$$

In our tests we used bits 15..8 of the number Z_n outputted by the generator. Experimental results obtained for CMRG are gathered in Tables 16 and 17.

At first glance we can see that this PRNG has rather long period and testing results give no reason for rejecting the hypothesis about the randomness of bit sequences produces by this PRNG. Nevertheless, one should note that in [8] the authors provide strong evidence that the output of CMRG is not sound.

5.8. Mersenne Twister

Mersenne Twister generator was tested using the built-in implementation in C++11. we choose 64-bit version of the PRNG, i.e. class `std::mt19937_64`. It is a minor modification of 32-bit variant characterized in Section 2.1, adapted to architectures with 8-byte words. In tests all 64 bits returned by a single call to the generator were used.

The results presented in Tables 18 and 19 give an explanation why the Mersenne Twister is one of the most popular PRNG used in practice. These outcomes give no arguments against the claim that the output of this PRNG is indeed truly random.

5.9. Hypothetical flawed PRNG

Describing NIST SP800-22 Test Suite in Section 1 we pointed out some of its inherent limitations directly related to the approach applied for statistical testing. Namely, this test suite focuses only on the quality of single binary sequences produced by PRNGs and does not take into account the set of all bit strings as a whole. Let us consider some hypothetical PRNG which usually generates sequences which look random, but with some non-negligible probability (e.g., for some subset of seeds) its output is biased and far from truly random. Such flawed sequence will be surely recognized by NIST Test Suite as non-random, but this deviation probably will not cause considering whole PRNG as "not good". However, its output will be easily distinguishable from uniform bit strings. This issue was carefully discussed in Section 3 in [3], where some specific examples of such PRNGs were presented.

The following example gives an insight how this kind of weaknesses in generator's output will be detected by the discussed distance based tests, i.e., LIL test and arcus sinus test. We will consider the following PRNG.

- For one in every hundred seeds it returns the sequence described by the regular expression 10(0110)*01, i.e., binary string starting with 10, followed by repeated runs of 0110 and ending with 01.
- Otherwise, the the output of MT19937-64 generator is returned.

Testing results for this PRNG are shown in Tables 20 and 21. The outcomes of arcsine test clearly indicate that the generator is flawed and its output cannot be recognized as random. What is interesting, however, this is not the case for LIL test, where the obtained p-values give no sufficient evidence for rejecting this PRNG as "weak". The reason behind the differences in the behavior of arcsine and LIL tests is that the trajectory (surely non-)random walk described by pattern 10(0110)*01 resembles a triangle wave oscillating around the 0-axis. Therefore, the fraction of time when number of 1's exceeds number of 0's (i.e. the random walk is above 0-axis) is exactly $\frac{1}{2}$. Hence, the value of characteristic S_n^{asin} falls into the least probable interval in the partition $\mathcal{P}^{asin}$ in almost $0.01\,.m$ trials more than for uniform bit sequences. This suffices for significant increase of the value of statistic . On the other hand, the characteristic S_n^{lil} takes for this sequence value 0 belonging to the most likely interval in the partition $\mathcal{P}^{lil}$. Thus, it is more difficult for χ^2 test to detect such deviation. Nevertheless, if we use e.g. $m = 100000$ sequences in the experiments above, LIL based test will also identify this weakness of considered generator. This leads to the conclusion that in order to detect such kinds of flaws in PRNGs implementations one should increase the number of sequences tested rather than the length of individual binary string.

6. Conclusions

In this paper we proposed a novel method of testing PRNGs which is based on arcsine law for random walks. From classical approaches it mainly differs on that PRNG's output is considered as a bit string rather than sequence of numbers. This allows for designing various testing procedures making use of properties of random walks. Our method is an example of statistical distance based testing techniques, where the quality of PRNG is measured by statistical distance between the empirical distribution of considered characteristic for generated pseudorandom output and its theoretical distribution for truly random binary sequences.

The experimental results presented in this paper show that our testing procedure can be used for detecting weaknesses in many common PRNGs implementations. Likewise LIL test from [3], ASIN test also has revealed some flaws and regularities in generated sequences not necessarily being identified by other current state of the art tools like NIST SP800-22 testing suite. Thus, this kind of testing techniques seems to be very promising, as it allows also for recognition of different kinds of deviations from those detected by existing tools. Nevertheless, like other statistical tests, ASIN test is not universal and encompasses only one from immense range of characteristics of random bit strings and does not capture all known flaws. Therefore, the distance based testing procedures relying on properties of random walks like ASIN or LIL tests should be used along with other tests bundles for more careful assessment of pseudorandom generators. This issue is well depicted by the provided example of obviously predictable generator for which LIL test has failed to detect its weaknesses, but ASIN test has turned out to be very sensitive for that kind of deviations. Hence, the important line of further research should be developing another novel statistical distance based tests utilizing various properties of random walks. Such tests should together be capable of detecting more hidden dependencies between consecutive bits in sequences generated by PRNGs. This should lead to designing more robust test suite for evaluating quality of random numbers generated by both new PRNGs implementation and those being already in use, especially for cryptographic purposes.

References

[1] NIST.gov - Computer Security Division - Computer Security Resource Center, NIST Test Suite, csrc.nist.gov/groups/ST/toolkit/rng/index.html (2010).
URL http://csrc.nist.gov/groups/ST/toolkit/rng/index.html

[2] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo, A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, Tech. Rep. Rev. 1a, NIST (2010). doi:10.6028/NIST.SP.800-22r1a.
URL http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf

[3] Y. Wang, T. Nicol, On statistical distance based testing of pseudo random sequences and experiments with PHP and Debian OpenSSL, Computers & Security 53 (2015) 44–64. doi:10.1016/j.cose.2015.05.005.
URL http://linkinghub.elsevier.com/retrieve/pii/S0167404815000693
[4] D. E. Knuth, The art of computer programming, Volume 2: Seminumerical Algorithms, 3rd Edition, Addison-Wesley Pub. Co, 1997.
[5] R. G. Brown, D. Eddelbuettel, D. Bauer, Dieharder: A Random Number Test Suite, www.phy.duke.edu/~rgb/General/dieharder.php.
URL http://www.phy.duke.edu/~rgb/General/dieharder.php
[6] P. L'Ecuyer, R. Simard, TestU01: A C library for empirical testing of random number generators, ACM Transactions on Mathematical Software 33 (4) (2007) 22–es. doi:10.1145/1268776.1268777.
URL http://portal.acm.org/citation.cfm?doid=1268776.1268777
[7] E. Barker, J. Kelsey, DRAFT NIST Special Publication 800-90A , Rev . 1 - Recommendation for Random Number Generation Using Deterministic Random Bit Generators, Tech. rep., NIST (2014).
[8] C. Kim, G. H. Choe, D. H. Kim, Tests of randomness by the gambler's ruin algorithm, Applied Mathematics and Computation 199 (1) (2008) 195–210. doi:10.1016/j.amc.2007.09.060.
URL http://linkinghub.elsevier.com/retrieve/pii/S0096300307009873
[9] H. Ekkehard, A. Grø nvik, Re-seeding invalidates tests of random number generators, Applied Mathematics and Computation 217 (1) (2010) 339–346. doi:10.1016/j.amc.2010.05.066.
URL http://dx.doi.org/10.1016/j.amc.2010.05.066
[10] W. Feller, An Introduction to Probability Theory and Its Applications, Volume 1, 3rd Edition, John Wiley & Sons, 1968.
[11] S. r. Asmussen, P. Glynn, Stochastic Simulation: Algorithms and Analysis, Springer, 2007.
[12] T. E. Hull, A. R. Dobell, Random Number Generators, SIAM Review 4 (3) (1962) 230–254. doi:10.1137/1004061.
URL http://epubs.siam.org/doi/10.1137/1004061
[13] P. L'Ecuyer, Efficient and portable combined random number generators, Communications of the ACM 31 (6) (1988) 742–751. doi:10.1145/62959.62969.
URL http://portal.acm.org/citation.cfm?doid=62959.62969
[14] A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone, Handbook of applied cryptography, CRC Press, 1997.
[15] M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation 8 (1) (1998) 3–30. doi:10.1145/272991.272995.
URL http://dl.acm.org/citation.cfm?id=272991.272995
[16] A. Khintchine, Über einen Satz der Wahrscheinlichkeitsrechnung, Fundamenta Mathematicae 6 (1) (1920) 9–20.
URL https://eudml.org/doc/214283
[17] W. Feller, An introduction to probability theory and its applications, 1971.
URL https://books.google.pl/books/about/An_introduction_to_probability_theory_an.html?id=NFNQAAAAMAAJ&pgis=1
[18] RANDOM.ORG - True Random Number Service, https://www.random.org/.
URL https://www.random.org/
[19] S. K. Park, K. W. Miller, Random number generators: good ones are hard to find, Communications of the ACM 31 (10) (1988) 1192–1201. doi:10.1145/63039.63042.
URL http://portal.acm.org/citation.cfm?doid=63039.63042
[20] S. K. Park, K. W. Miller, P. K. Stockmeyer, Technical correspondence, Communications of the ACM 36 (7) (1993) 105. doi:10.1145/159544.376068.
URL http://portal.acm.org/citation.cfm?doid=159544.376068

	conv. in prob.	a.s. conv.	lim sup a.s	lim inf a.s
LLN	$\frac{S_n}{n} \xrightarrow{P} 0$	$\frac{S_n}{n} \xrightarrow{a.s.} 0$	$\limsup_{n\to\infty} \frac{S_n}{n} = 0$	$\liminf_{n\to\infty} \frac{S_n}{n} = 0$
LIL	$\frac{S_n}{2n\log\log n} \xrightarrow{P} 0$	$\frac{S_n}{2n\log\log n} \xrightarrow{a.s.} 0$	$\limsup_{n\to\infty} \frac{S_n}{2n\log\log n} = 1$	$\liminf_{n\to\infty} \frac{S_n}{2n\log\log n} = -1$
CLT	$\forall x \frac{S_n}{n} \xrightarrow{P} x$	$\forall x \frac{S_n}{n} \xrightarrow{a.s.} x$	$\limsup_{n\to\infty} \frac{S_n}{n} = \infty$	$\liminf_{n\to\infty} \frac{S_n}{n} = -\infty$

Table 1: Conclusions for random walks based on LLN, LIL and CLT laws

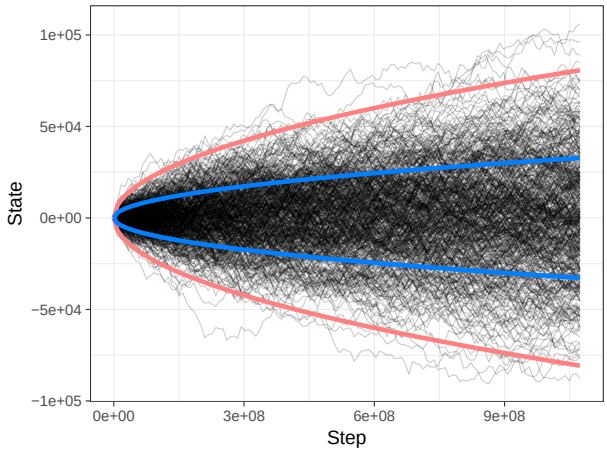

Figure 1: 500 trajectories of random walks of length 2^{30}. Blue plot: $\pm\sqrt{n}$, red plot: $\pm\sqrt{2n\log\log n}$

Table 2: Results of arcsine test for RANDU generator.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.4604	0.4616	0.4637	0.4670	0.4695	0.4697
sep1	0.6017	0.6646	0.6229	0.6138	0.5934	0.6453
sep2	0.8659	0.8662	0.8667	0.8675	0.8681	0.8682
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 3: Results of LIL test for RANDU generator.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.4955	0.496	0.4965	0.4969	0.4973	0.4977
sep1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 4: Results of arcsine test for PRNG in MS Visual C++.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0289	0.0387	0.0615	0.0768	0.0848	0.0928	0.0965	0.1870	0.2093
sep1	0.2030	0.1548	0.1952	0.2605	0.3664	0.4210	0.5528	0.6533	0.8163
sep2	0.1780	0.2260	0.2524	0.2364	0.2841	0.3160	0.4336	0.6112	0.4328
p-val	0.0128	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

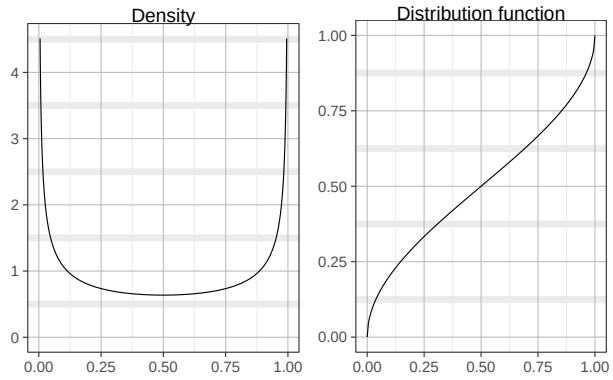

Figure 2: Arcsine distribution.

Table 5: Results of LIL test for PRNG in MS Visual C++.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0350	0.0512	0.0938	0.1234	0.1672	0.2423	0.3100	0.4991	0.9500
sep1	0.2831	0.7419	0.9731	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.2670	0.1416	0.2635	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 6: Results of arcsine test for PRNG in Borland C/C++ standard library.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0384	0.0502	0.0697	0.0861	0.1394	0.1562	0.1231	0.1466	0.2148
sep1	0.1293	0.1840	0.2513	0.3733	0.5485	0.5149	0.5696	0.6873	0.8219
sep2	0.2024	0.2120	0.2096	0.3137	0.3702	0.3730	0.3799	0.5167	0.4009
p-val	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 7: Results of LIL test for PRNG in Borland C/C++ standard library.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0684	0.1002	0.1395	0.1940	0.3187	0.4136	0.4685	0.5911	0.9500
sep1	0.6752	0.7357	0.9193	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.1610	0.2565	0.2978	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 8: Results of arcsine test for initial version of PRNG from BSD libc.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.0883	0.0980	0.0862	0.0951	0.0936	0.1081
sep1	0.3402	0.3979	0.4111	0.4980	0.4346	0.4751
sep2	0.3634	0.4198	0.3053	0.2805	0.3424	0.4070
p-val	0.1829	0.0326	0.1874	0.1767	0.1181	0.0051

Table 9: Results of LIL test for initial version of PRNG from BSD libc.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.1956	0.2191	0.2472	0.2539	0.2464	0.2707
sep1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 10: Results of arcsine test for GNU C standard library generator.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0287	0.0217	0.0264	0.0228	0.0202	0.0289	0.0237	0.0254	0.0230
sep1	0.1157	0.1249	0.1629	0.1186	0.1661	0.1717	0.1306	0.1316	0.1405
sep2	0.1754	0.1909	0.1436	0.1869	0.1110	0.1344	0.1786	0.1262	0.1879
p-val	0.0649	0.5511	0.2565	0.4887	0.7967	0.1115	0.5880	0.2599	0.3930

Table 11: Results of LIL test for GNU C standard library generator.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0185	0.0243	0.0242	0.0244	0.0210	0.0220	0.0330	0.0237	0.0221
sep1	0.2593	0.1640	0.1844	0.2573	0.2010	0.1853	0.2882	0.4129	0.2563
sep2	0.0807	0.2133	0.2299	0.2060	0.1839	0.2361	0.3456	0.2078	0.1405
p-val	0.9627	0.4878	0.5225	0.3022	0.6978	0.5382	0.0009	0.4903	0.7901

Table 12: Results of arcsine test for Minstd generator with multiplier 16807.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0321	0.0321	0.0402	0.0725	0.0917	0.1116	0.1648	0.2003	0.2109
sep1	0.1172	0.1686	0.1407	0.2985	0.2880	0.6670	0.5203	0.7345	0.8486
sep2	0.1302	0.1490	0.2420	0.3986	0.4051	0.3735	0.3822	0.5352	0.3894
p-val	0.0360	0.0203	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 13: Results of LIL test for Minstd generator with multiplier 16807.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0370	0.0622	0.0823	0.1120	0.1877	0.2002	0.3538	0.3194	0.9500
sep1	0.5035	0.5465	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.1444	0.2366	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 14: Results of arcsine test for Minstd generator with multiplier 48271.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0318	0.0430	0.0477	0.0982	0.1303	0.0918	0.0848	0.1131	0.2089
sep1	0.1607	0.1467	0.1901	0.3803	0.4280	0.3845	0.4583	0.3975	0.8079
sep2	0.1828	0.1719	0.2120	0.2548	0.3449	0.3179	0.3192	0.3186	0.4579
p-val	0.0115	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 15: Results of LIL test for Minstd generator with multiplier 48271.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0372	0.0563	0.0590	0.2573	0.2879	0.3408	0.3167	0.3434	0.9500
sep1	0.3298	0.4839	0.9328	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.3843	0.1850	0.2207	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 16: Results of arcsine test for CMRG.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0191	0.0234	0.0235	0.0239	0.0232	0.0228	0.0218	0.0268	0.0272
sep1	0.1135	0.2090	0.1187	0.1450	0.1607	0.1191	0.1232	0.1592	0.2084
sep2	0.1402	0.0900	0.1084	0.1477	0.1934	0.1666	0.0931	0.1353	0.1158
p-val	0.9752	0.4101	0.8062	0.7402	0.4043	0.6841	0.9821	0.2394	0.1343

Table 17: Results of LIL test for CMRG.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0218	0.0241	0.0272	0.0203	0.0231	0.0251	0.0330	0.0233	0.0234
sep1	0.2336	0.3273	0.1803	0.2551	0.3085	0.1690	0.3146	0.3310	0.2829
sep2	0.3450	0.2105	0.1234	0.1032	0.1901	0.1658	0.2302	0.2486	0.1761
p-val	0.6046	0.2689	0.2030	0.8653	0.4581	0.3966	0.0128	0.4106	0.6803

Table 18: Results of arcsine test for C++11 implementation of MT19937-64 generator.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0271	0.0245	0.0226	0.0281	0.0276	0.0228	0.0271	0.0230	0.0255
sep1	0.1421	0.1606	0.1167	0.1824	0.1938	0.1571	0.1760	0.1397	0.1573
sep2	0.1133	0.1195	0.1056	0.1399	0.1256	0.1287	0.1391	0.0994	0.1444
p-val	0.2755	0.6823	0.8804	0.0801	0.1267	0.7343	0.1094	0.7596	0.2029

Table 19: Results of LIL test for C++11 implementation of MT19937-64 generator.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0254	0.0264	0.0238	0.0234	0.0266	0.0265	0.0290	0.0299	0.0215
sep1	0.1853	0.2147	0.1689	0.3093	0.1887	0.2015	0.2877	0.2318	0.1953
sep2	0.2841	0.1696	0.1917	0.1282	0.2017	0.2492	0.1842	0.1541	0.1716
p-val	0.2483	0.3690	0.5965	0.5066	0.2196	0.1495	0.0499	0.0189	0.8170

Table 20: Results of arcsine test for the flawed generator described in Section 5.9.

n	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
tv	0.0326	0.0241	0.0281	0.0285	0.0298	0.0339	0.0298	0.0285	0.0276
sep1	0.1466	0.1754	0.1529	0.1124	0.1249	0.1752	0.1594	0.1666	0.1257
sep2	0.3255	0.334	0.345	0.4212	0.3608	0.3582	0.3556	0.3925	0.3901
p-val	0.0000	0.0031	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 21: Results of LIL test for the flawed generator described in Section 5.9.

n	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
tv	0.0239	0.0303	0.0255	0.0269	0.0241	0.0280	0.0248	0.0294	0.0229
sep1	0.1929	0.2834	0.1541	0.2018	0.2327	0.1586	0.1631	0.2654	0.1689
sep2	0.1625	0.1517	0.1663	0.1469	0.1381	0.1648	0.2106	0.1667	0.1815
p-val	0.4073	0.0199	0.2584	0.1250	0.5742	0.1250	0.2703	0.0436	0.5789