On testing pseudorandom 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 Test Suite is a set of commonly used tools for testing the quality of pseudo random generators. Wang and Nicol [Computers & Security **53**, 44–64 (2015)] pointed out that it is relatively easy to construct some generators whose output is "far" from uniform distribution, but which pass NIST tests. Authors proposed some statistical tests which are based on *Law of Iterated Logarithm* (LIL) which work very good for such generators (*i.e.*,, they do not pass the test). In this paper we propose a 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 a PRNG has similar properties as 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 PRNGs 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 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 sequences.

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 [1]. 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 [1] 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 bundle consists 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. [2]), TestU01 implemented by L'Ecuyer and Simard (see [3, 4]) and NIST Test Suite [5]. 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 [6]. NIST Test Suite SP800-22 [5] 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 the frequencies of occurrences of all k-bit overlapping patterns are. 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. [6, 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 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 the 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 [8] 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 [9]. 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 [10], authors also show that tests of Kim et al. [9] performed *properly* do not reveal any defects in Mersenne Twister PRNG. Recently, authors in [11] proposed another gambler ruin based procedure for testing PRNGs. Authors exploited formulas for winning probabilities for arbitrary sequences $p(i), q(i)$, (*i.e.,* winning and losing probabilities depend on the current fortune) which are the parameters of the algorithm.

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 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. [12]) . For a good PRNG we may expect that its output behaves as S_n. Hence, the following idea comes to mind: choose some probabilistic law that holds for truly random bit sequences and compare the theoretical distribution of the corresponding statistics with 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 the appropriate test statistics under the null hypothesis that the sequence generated by PRNG is truly random. Another concept named *statistical distance based testing* was suggested in [8], which relies on calculating statistical distances like e.g., total variation or separation distance (cf. formulas (5) and (6), respectively) between the theoretical and the empirical distribution for considered characteristics and rejecting PRNG if the distances are above some threshold. Such approach was adopted in [8], 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 2 we will recall the aforementioned stochastic laws for random walks, whereas the testing methods will be described in Section 3. Through the paper we will denote by $[n]$ the set $\{1, \ldots, n\}$.

Pseudo random generators. The intuition behind *pseudo random number generator* is clear. However, let us give a strict definition roughly following Asmussen and Glynn [13].

Definition 1.1. *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, $s_0 \in E$ is a so-called seed, i.e., initial state in the 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 the 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 $|E|$.

Recall that LCG (linear congruential generator) is a generator which updates its states according to formula $s_n = (as_{n-1} + c) \bmod M$, thus it is defined by three integers: a modulus M, a multiplier a, and an additive constant c. It is shortly denoted as $LCG(M, a, c)$. In case $c = 0$, the generator is called MCG (multiplicative congruential generator), shortly denoted as $MCG(M, a)$. For a detailed description of commonly used PRNGs see surveys [14], [15], [16] or a book [17].

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. We say that the sequence of bits is *truly random* if it is a realization of 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 a case the generator is more likely to outputs 0s 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.

2. 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

$$X_i = 2B_i - 1, \quad S_0 = 0, \quad S_n = \sum_{i=1}^{n} X_i.$$

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 than $[-n, n]$. *Weak* and *Strong Law of Large Numbers* imply that $\frac{S_n}{n} \xrightarrow{P} 0$, and even $\frac{S_n}{n} \xrightarrow{a.s.} 0$, where $\xrightarrow{P}$ denotes *convergence in probability* and $\xrightarrow{a.s.}$ denotes *almost sure convergence*. Thus, the deviations of S from 0 grow much slower than 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 2.1 ([18], cf. also Chapter VIII.5 in [12])**.** *For a random walk S_n we have*

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

Thus, to normalize S_n: dividing by n is *too strong* and dividing by $\sqrt{n}$ is *too weak*, the fluctuations of S_n from 0 grow proportionally to $\sqrt{2n\log\log n}$.

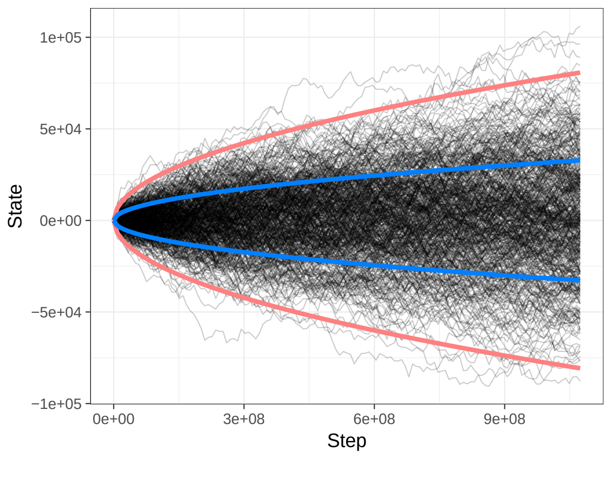

Figure 1: 500 trajectories of random walks of length 2^{30}. Blue plot: $\pm\sqrt{n}$, red plot: $\pm\sqrt{2n\log\log n}$

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 [8] for designing a novel method of testing random number generators (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 3.2.

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.

2.1. Arcsine Law

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 2.2 below (see e.g. [12]). Before we formulate the theorem, let us first introduce some notations. Let $\mathbb{1}(expr)$ is equal to 1 if *expr* is true, and 0 otherwise. For a sequence $X_1, X_2, \ldots$ let

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

D_k is equal to 1 if number of ones exceeds number of zeros either at step k or at step $k-1$, 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 2.2 (Arcsine Law for random walk)**.** *Define* $L_n = \sum_{k=1}^{n} D_k$. *For* $x \in (0,1)$ *we have*

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

$\mathbb{P}\left(L_n \le x \cdot n\right)$ 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*, it has density $f(t) = \frac{1}{\pi}\sqrt{t(1-t)}$ and cdf $F(t) = \frac{2}{\pi}\arcsin\sqrt{t}$. 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.

3. 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 4. For the sake of completeness we will also recall some basic facts about the method of evaluating PRNGs presented in [8]. Finally, we will perform the analysis of approximation errors that occur in our testing procedure.

3.1. 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 2.2. Let us define

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

S_n^{asin} is a fraction of time instants at which ones prevail zeros. From Arcsine Law (Theorem 2.2) we can conclude that *for large* n we have

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

(we will be more specific on "$\approx$" in Section 3.3). 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 $0 \le i \le s+1$:

$$\mu_n^{asin}\left(P_i^{asin}\right) = \mathbb{P}\left(S_n^{asin} \in P_i^{asin}\right), \tag{3}$$

$$\nu_n^{asin}\left(P_i^{asin}\right) = \frac{|\{j :\ S_{n,j}^{asin} \in P_i^{asin}, 1 \le j \le m\}|}{m}. \tag{4}$$

Measure μ_n^{asin} corresponds to the theoretical distribution and can be calculated (approximated) using (2), whereas ν_n^{asin} represents empirical distribution obtained from tests. We will measure the distance between these measures either using *total variation* or *separation* distance defined for partition $\mathcal{P}$ as follows:

$$d_{\mathcal{P}}^{tv}(\mu,\nu) = \frac{1}{2}\sum_{A\in\mathcal{P}} |\mu(A) - \nu(A)|, \tag{5}$$

$$d_{\mathcal{P}}^{sep}(\mu,\nu) = \max_{A\in\mathcal{P}}\left(1 - \frac{\mu(A)}{\nu(A)}\right). \tag{6}$$

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

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

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 (2)). Assuming null hypothesis the statistic

$$T^{asin} = \sum_{i=0}^{s+1} \frac{(O_i - E_i)^2}{E_i} \tag{7}$$

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.

3.2. LIL based testing

The procedure of testing pseudorandom number generators based on Law of Iterated Logarithm (see Theorem 2.1) was proposed in [8]. This method is similar to that based on Arcsine Law discussed in details in Section 3.1, 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

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

and use an appropriate partition of the real line.

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

$$\mathbb{P}\left(S^{lil}_n \in (a,b)\right) = \mathbb{P}\left(\frac{S_n}{\sqrt{n}} \in (a\,l(n), b\,l(n))\right) \approx \Phi(b\,l(n)) - \Phi(a\,l(n)). \tag{8}$$

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 for $i \in \{0, \ldots, s+1\}$ as follows:

$$\mu^{lil}_n\left(P^{lil}_i\right) = \mathbb{P}\left(S^{lil}_n \in P^{lil}_i\right), \tag{9}$$

$$\nu^{lil}_n\left(P^{lil}_i\right) = \frac{|\{j:\ S^{lil}_{n,j} \in P^{lil}_i, 1 \le j \le m\}|}{m}. \tag{10}$$

Theoretical distribution μ^{lil}_n of the statistic S^{lil}_n can be directly approximated from (8). 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 ASIN test.

3.3. Error analysis

Our test relies on (2), however some approximations are used there. In this Section we will show that for sequences of length at least 2^{26} the possible error is negligible.

Lemma 3.1. *For any fixed a and b such that $0 \le a < b \le 1$ and for $\mathfrak{n} \ge 2^{26}$ the approximation error of the probability $\mathbb{P}\left(S^{asin}_{\mathfrak{n}} \in (a,b)\right)$ in (2) using partition $\mathcal{P}^{asin}_{40}$ can be upper bounded by 10^{-5}, i.e.,*

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

Proof. Let us assume that $\mathfrak{n}$ is even and that $\mathfrak{n} = 2n$. 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

$$p_{2k,2n} = \binom{2k}{k}\binom{2(n-k)}{n-k}2^{-2n}. \tag{11}$$

The standard proof of Theorem 2.2 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, $0 < \theta_n \le 1$, such that

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

Plugging (12) into each factorial appearing in (11) we have

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

Thus we have

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

and

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

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

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

what implies

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

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

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

We will estimate the approximation error in (2) in two steps. First, take two numbers a, b such that $\delta \le a < b \le 1 - \delta$. Then

$$\left|\sum_{a \le \frac{k}{n} \le b} p_{2k,2n} - \sum_{a \le \frac{k}{n} \le b} d_{k,n}\right| \le \sum_{a \le \frac{k}{n} \le b} |p_{2k,2n} - d_{k,n}| \le \sum_{a \le \frac{k}{n} \le b} \frac{1}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}}$$

$$= \frac{\lceil bn - an\rceil}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}} \le \frac{b-a}{3\pi n\left(\delta(1-\delta)\right)^{\frac{3}{2}}} \le \frac{1}{3\pi n\left(\delta(1-\delta)\right)^{\frac{3}{2}}} =: \eta.$$

The second source of errors in probability estimation in (2) 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 maximum and minimum value of f on that interval, respectively. Using the Lagrange's mean value theorem we obtain

$$\left|\int_a^b f(x)dx - \sum_{a\le\frac{k}{n}\le b}\frac{1}{n}f(x_k)\right| \le \sum_{a\le\frac{k}{n}\le b}\frac{1}{n}(M_i-m_i) = \sum_{a\le\frac{k}{n}\le b}\frac{1}{n^2}|f'(\xi_i)| \le \sum_{a\le\frac{k}{n}\le b}\frac{1}{n^2}\sup_{a\le x\le b}|f'(x)|$$

$$= \frac{\lceil bn-an\rceil}{n^2}\sup_{a\le x\le b}|f'(x)| \le \frac{2(b-a)}{n}\sup_{a\le x\le b}|f'(x)|.$$

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 $(a,b)\subseteq(\delta,1-\delta)$ we have

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

In our tests we use the partition $\mathcal{P}_{40}^{asin}$ of the real line, thus we set $\delta=\frac{1}{80}$. Moreover, we assumed that $\mathfrak{n}\ge 2^{26}$, *i.e.,* $n\ge 2^{25}$. For these parameters' values

$$\eta \le \frac{77.4}{n} \le 2.31\cdot 10^{-6} \quad\text{and}\quad \kappa \le \frac{226.3}{n} \le 6.75\cdot 10^{-6}.$$

Finally,

$$\left|\int_a^b f(x)dx - \sum_{a\le\frac{k}{n}\le b} p_{2k,2n}\right| \le \left|\int_a^b f(x)dx - \sum_{a\le\frac{k}{n}\le b} d_{k,n}\right| + \left|\sum_{a\le\frac{k}{n}\le b} d_{k,n} - \sum_{a\le\frac{k}{n}\le b} p_{2k,2n}\right|$$

$$=\eta+\kappa \le 9.06\cdot 10^{-6} \le 10^{-5},$$

what justifies the approximation (2) 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

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

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

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

3.4. Notes on Takashima's method for testing PRNG and arcsine test implementation from TestU01

The idea of using arcsine law for testing PRNGs was proposed earlier by Takashima in [19] and further developed in [20, 21]. In this series of articles, a test statistics built upon discrete arcsine law for random walks was introduced and used for testing some particular kind of PRNGs, namely maximum-length linearly recurring sequences (m-sequences). The presented in these papers results of experiments clearly show that the examined generators generate biased output, hence proving that Takashima's methods seem to be useful and could be applied in practice as a building block of test suites provided by software tools for testing RNGs.

The method introduced by Takashima [20] can be briefly described as follows. At the beginning, the tested PRNG is initialized and a sequence of $2NL$ bits is generated.[1] This sequence is then divided into N subsequences of length $2L$. From each subsequence, a random walk is constructed (starting at 0) and 200 the test statistic based on arcsine law is calculated. From this N samples, an empirical distribution of the time spent by the random walk above the OX axis (author calls it *sojourn time*) is then derived and compared with the theoretical distribution. This comparison is done in the following steps. Denoting by t^j_{2L} the value of the test statistic for for j^{th} subsequence, $j = 0, \dots, N-1$, for each $m = 0, \dots, L$, the values $f_{2m} = |\{j : t^j_{2L} = 2m\}|$ are calculated. Then, the χ^2-test is applied to the values f_{2m}. The whole procedure of generating random bits and calculating test statistic is repeated M times, resulting in the set of statistics χ^2_k, $k = 0, \dots, M-1$. At the final stage, the numbers of χ^2_k values falling between 90^{th} and 95^{th} percentile and bigger than 95^{th} percentile of χ^2 distribution with L degrees of freedom are calculated. These two counts are then the basis for deciding if the null hypothesis (that the pseudorandom bits are generated independently and uniformly at random) should be rejected.

The author considers also another slightly modified variant of the procedure, where the chi-square test is combined with Kolomogorov-Smirnov test. Namely, for each $k = 0, \dots, M-1$, 30 repetitions of the χ^2 test are performed and the Kolmogorov-Smirnov test statistics are calculated. Finally, the numbers of samples between 95^{th} and 99^{th} percentile and bigger than 99^{th} percentile are determined as a result of the testing method. Let $\mathfrak{L}_{2n} = \sup\{2k : k \leq n, S_{2k} = 0\}$ be the last visit to 0 in a walk of length $2n$. It is known (cf. [12]) that $\mathfrak{L}_{2n}$ has the same distribution as sojourn time, thus it also follows arcsine law. Takashima [21] exploited this fact to construct another test for PRNGs (beside different statistic, the method is very similar to that one described in [20]). Note that in our simulations we used (in most cases) sequences of length 2^{34}, thus Takashima's tests would require huge amount of memory to store values of $f_{2m}, m = 0, \dots, L$.

As these tests were proven to be useful in detecting flaws of some families of PRNGs (see the results presented and discussed in [19, 20, 21]), the method based on arcsine law was implemented in the TestU01 library (see [4]). This tool, developed by L'Ecuyer and Simard, provides a big variety of functions for empirical testing of PRNGs. One of the test modules, `swalk`, provides a procedure `swalk_RandomWalk1` which applies, among others, a test based on arcsine law. This function generates N random walks from the tested sequence of random bits and for each walk computes the corresponding test statistic. Then, it compares the empirical distribution derived from these N samples with the theoretical one using χ^2 test. The procedure implemented in `TestU01` is very similar to ours. The main difference is that the partition size s is a parameter in our case, whereas the partition used therein is calculated automatically, and it is dependent on the numbers to be tested. If there are too few observations within some intervals, the intervals are merged. In addition, we calculate total variation and separation distances between the theoretical and the empirical distributions. Besides the single-level test, one can also use TestU01 software to perform the second-order test, where M independent repetition of the basic procedure is executed and either the obtained statistics are then combined (i.e. the distribution of the sum of the statistics' values is analyzed) or the goodness of fit tests (like Kolmogorov-Smirnov test) are applied to those results. The implementation of the `swalk_RandomWalk1` procedure is quite general. In particular, it allows for constructing the random walk from bits extracted in many different ways from the random numbers produced by the tested generator. Hence, it can be viewed as a generalization of the methods described in the series of Takashima's articles.

4. 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, ASIN and LIL tests on generators including different implementations of standard C/C++ linear congruential generators,

[1] In the original paper each bit is the most significant bit of the number produced by the tested PRNG, although the proposed method can be applied to any sequence of pseudorandom bits.

Table 1: Results of ASIN test for initial version of PRNG from BSD libc. `Time: 69m 10s`

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.0424	0.0499	0.0495	0.0464	0.0448	0.0411
sep1	0.1798	0.2036	0.2303	0.2065	0.2093	0.1868
sep2	0.1919	0.2276	0.1729	0.1793	0.2172	0.1897
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

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 ASIN 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_r^{asin} or S_r^{lil} were calculated both for each sequence ($r = n$) 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 [8] *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 3.1 and
250 3.2. 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` [22] and each sequence was generated using different seed. Total running times needed for generating and calculating statistics are given in tables. Each simulation was performed on one thread on twin Intel Xeon E5-2630 v2 CPUs (12 physical cores @2.60GHz) with 64 GB RAM. Note that instead of simulating 10000 paths on one thread, it is possible to simulate *e.g.,* 1000 paths on 10 threads and then combine the results. It would significantly decrease the total running times. The source code of our implementation is available at `https://github.com/lorek/PRNG_Arcsine_test`.

In the following sections we provide the outcomes of performed tests as well as discussion and analysis of the 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 (3) or (9) and the empirical measure ν_{n_k} according to (4) or (10) 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 (7). We provide both, $d^{sep}(\mu_{n_k}, \nu_{n_k})$ and $d^{sep}(\nu_{n_k}, \mu_{n_k})$, since – despite phrase "distance" in its name – separation distance $d^{sep}(\cdot, \cdot)$ is not symmetric.

We also calculated `swalk_RandomWalk1` statistics from `TestU01` for 10000 sequences of length 2^{26} of each PRNG we tested. The following parameters for `swalk_RandomWalk1` were used: $N = 10000, n = 64, r = 0, s = 32, L0 = L1 = 1048576$. The result are given in Table 11. For each Statistic H, M, J, R and C, four p-values were obtained: using *Kolmogorov-Smirnov+*, *Kolmogorov-Smirnov-* (upper part of cells) and *Anderson-Darling*, *Chi-square* (lower part of cells) statistics. For convenience, p-values of ASIN and LIL tests are also included in the Table.

4.1. BSD libc rand()

Function `rand` from BSD system standard library has used originally $LCG(2^{31}, 1103515245, 12345)$ and 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 1 and 2 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. Also all the statistics of `swalk_RandomWalk1` from `TestU01` conducted for sequences of length 2^{26} indicate that this is not a good PRNG (see Table 11). The reason behind that is the LCG uses modulus 2^{31}, what implies that the period of d least significant bits equals to

Table 2: Results of LIL test for initial version of PRNG from BSD libc. `Time: 65m 40s`

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.1828	0.2152	0.2381	0.2437	0.2565	0.2613
sep1	0.9903	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.3629	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 3: Results of ASIN test for PRNG in MS Visual C++. `Time: 377 hrs`

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	0.0148	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 4: Results of LIL test for PRNG in MS Visual C++. `Time: 397 hrs`

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0348	0.0510	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	0.1400	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

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 slightly more evident in the outcomes of tests based on S_n^{lil} characteristic, which outperforms those based on S_n^{asin} in this case.

Similarly, the results (not included here) for RANDU (which is defined as $MCG(2^{31}, 65539)$) are comparable to those for `rand` from BSD system standard library.

4.2. Microsoft Visual C++ rand()

Function `rand` in Microsoft Visual C++ is based on $LCG(2^{32}, 214013, 2531011)$ linear congruential generator). It differs from an ordinary LCG in that it returns only the bits on last 15 positions. This PRNG is one of those tested in [8]. As the authors in [8], 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 from 23 to 30 of the number generated by LCG). The results of performed tests are presented in Tables 3 and 4.

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 $d^{tv}(\mu_n^{lil}, \nu_n^{lil}) \approx 0.95$.

300 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 (for this n, abouth half of p-values of `swalk_RandomWalk1` from `TestU01` suggest rejecting the hypothesis that it is a good PRNG). 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 [5], as pointed out in [8].

We also conducted (not included here) the experiments for the procedure `rand` from standard library in Borland C/C++ (which implements $LCG(2^{32}, 22695477, 1)$). The outcomes are very akin to those for standard PRNG in MS Visual C++.

Table 5: Results of ASIN test for GNU C standard library generator. `Time: 313.2 hrs`

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0238	0.0266	0.0286	0.0247	0.0178	0.0279	0.0232	0.0257	0.0255
sep1	0.2079	0.1187	0.1666	0.1186	0.1481	0.1780	0.1879	0.1514	0.1405
sep2	0.1491	0.1566	0.1785	0.1606	0.1063	0.1482	0.1744	0.1421	0.1961
p-val	0.4731	0.2100	0.1758	0.3663	0.9744	0.1206	0.4593	0.1563	0.2389

Table 6: Results of LIL test for GNU C standard library generator. `Time: 295.4 hrs`

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0215	0.0272	0.0226	0.023	0.021	0.0212	0.0325	0.0235	0.0220
sep1	0.3828	0.1802	0.2838	0.1978	0.1508	0.1853	0.3451	0.3737	0.2696
sep2	0.1449	0.2568	0.2735	0.1970	0.1831	0.2521	0.3456	0.2284	0.1327
p-val	0.6052	0.0714	0.5685	0.5729	0.7853	0.5988	0.0019	0.4231	0.7914

4.3. 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, $0 \le s \le 2^{31}$, and its initial state is given by

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

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 $x_{k+343} \gg 1$ (where $\gg$ is a right logical shift operator).

For our tests we took all 31 bits outputted by PRNG. From the results gathered in Tables 5 and 6 one may conclude that ASIN 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. Similarly, the statistics of `swalk_RandomWalk1` from `TestU01` in few cases (mainly Statistic J and R) also suggest that this PRNG may contain some flaws (for sequences of length 2^{26}, see Table 11). 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.

4.4. Minstd

Minstd (abbr. from *minimal standard generator*) is based on MCG with parameters recommended by Park and Miller in [23]. The authors' goal was to develop a simple generator, not necessarily perfect,

Table 7: Results of ASIN 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 8: 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 9: Results of arcsine test for C++11 implementation of MT19937-64 generator. `Time: 309.1 hrs`

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0272	0.0245	0.0226	0.0279	0.0273	0.0223	0.0273	0.0233	0.0252
sep1	0.1421	0.1666	0.1167	0.1824	0.1938	0.1571	0.1760	0.1397	0.1573
sep2	0.1133	0.1195	0.1056	0.1399	0.1209	0.1315	0.1391	0.1065	0.1444
p-val	0.2548	0.6674	0.8907	0.1014	0.1287	0.7727	0.1002	0.7447	0.2523

Table 10: Results of LIL test for C++11 implementation of MT19937-64 generator. `Time: 291.5 hrs`

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0229	0.0243	0.0234	0.0207	0.0251	0.0297	0.0287	0.0309	0.0225
sep1	0.1606	0.2400	0.1689	0.2890	0.1458	0.1853	0.2712	0.2318	0.1795
sep2	0.2106	0.1458	0.2017	0.1228	0.1839	0.2492	0.1951	0.1748	0.1559
p-val	0.5440	0.6035	0.4630	0.8106	0.4609	0.0653	0.0593	0.0077	0.7745

but fast, simple to implement and suitable for most common applications. As a result they proposed $MCG(2^{31}-1, 16807)$. In their later work from 1993, Park and Miller [24] suggested that it would be better to use the multiplier 48271. However, changing the multiplier does not affect our tests much. In Tables 7 and 8 the results for 48271 are presented, and those for 16807 (not included here) are similar.

For the ASIN 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. This can also be seen in statistics of `swalk_RandomWalk1` from `TestU01` conducted for sequences of length 2^{26}. Seven out of twenty p-values are smaller or equal to 0.01, see Table 11.

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).

4.5. Mersenne Twister

Mersenne Twister generator was tested using the built-in implementation in C++11. We chose 64-bit version of the PRNG *i.e.,* class `std::mt19937_64`. In tests all 64 bits returned by a single call to the generator were used.

The results presented in Tables 9 and 10 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 (maybe only in case of S^{LIL} there is some deviation from randomness

Table 11: Results (p-values) of ASIN, LIL and `swalk_RandomWalk1` statistics from `TestU01` for $n = 2^{26}$.

PRNG\Test	ASIN	LIL	Statistic H	Statistic M	Statistic J	Statistic R	Statistic C
BSD libc	0.0000	0.0000	0.000; 0.000	0.000; 0.000	0.000; 0.000	0.000; 0.000	0.000; 0.000
			0.000; 0.000	0.000; 0.000	0.000; 0.000	0.000; 0.000	0.000; 0.000
MS Visual C++	0.0148	0.0001	0.920; 0.000	0.950; 0.000	0.860; 0.000	0.230; 0.060	0.860; 0.020
			0.000; 0.002	0.000; 0.000	0.000; 0.002	0.120; 0.530	0.020; 0.110
GNU C	0.4731	0.6052	0.080; 0.330	0.380; 0.170	0.008; 0.008	0.660; 0.003	0.630; 0.007
			0.490; 0.740	0.320; 0.550	0.008; 0.990	0.030; 0.010	0.110; 0.230
Minstd 48271	0.0115	0.0000	0.000; 0.310	0.013; 0.007	0.290; 0.000	0.200; 0.270	0.490; 0.010
			0.000; 0.000	0.010; 0.994	0.010; 0.530	0.280; 0.130	0.180; 0.080
MT19937-64	0.2548	0.5540	0.070; 0.007	0.580; 0.004	0.340; 0.002	0.210; 0.270	0.150; 0.460
			0.030; 0.670	0.010; 0.080	0.050; 0.029	0.760; 0.570	0.630; 0.270
Flawed	0.0000	0.5558	0.540; 0.430	0.930; 0.370	0.210; 0.910	0.270; 0.800	0.260; 0.550
			0.750; 0.310	0.570; 0.180	0.330; 0.900	0.460; 0.750	0.470; 0.700

in case $n = 2^{33}$). This can also the case with in statistics of `swalk_RandomWalk1` from `TestU01` conducted for sequences of length 2^{26}, see Table 11.

It is worth noting that the experiments for CMRG (not included here) gave similar results. Nevertheless, one should note that in [9] the authors provide strong evidence that the output of CMRG is not sound.

4.6. 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 a single binary sequence produced by a PRNG 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. distinguishable from uniform bit strings. This issue was carefully discussed in Section 3 of [8], where some specific examples of such PRNGs were presented.

The following example gives an insight on how this kind of weaknesses in generator's output may be detected by the ASIN test.

Before introducing the PRNG we need to define *Dyck paths* and say how to simulate one randomly.

Definition 4.1. *A sequence of $2n$ bits $B_1, \ldots, B_{2n}$ is called a Dyck path if the corresponding walk fulfills: $S_k = \sum_{i=1}^{k}(2B_i - 1) \geq 0, k = 1, \ldots, 2n-1$ and $S_{2n} = 0$. A set of all Dyck paths of length $2n$ is denoted by $\mathcal{D}_{2n}$.*

Thus, a Dyck path of length $2n$ corresponds to valid grouping of n pairs of parantheses, we have $|D_{2n}| = C_n = \frac{1}{n+1}\binom{2n}{n}$ (the Catalan number). Let $\mathcal{P}^{-1}_{2n+1}$ be the set of sequences of bits $B_1, \ldots, B_{2n+1}$ such that the corresponding walk ends at -1, *i.e.,* $S_{2n+1} = -1$. It is relatively easy to simulate uniformly at random $P \in \mathcal{P}$: make a random permutation $(0, \ldots, 0, 1, \ldots, 1)$, where we have $n+1$ "zeros" and n "ones".

We can obtain a Dyck path of length $2n$ from $P \in \mathcal{P}^{-1}_{2n+1}$ in the following way (the trick is known as *Cycle Lemma* [25]). Let $P = (B_1, \ldots, B_{2n+1}) \in \mathcal{P}$ and take $s = \text{argmin}\{S_k\}$ (the lowest level). Then, $f(P) = (B_{s+1}, \ldots, B_{2n+1}, B_1, \ldots, B_{s+1})$ is a Dyck path, see Fig. 2 We have

Lemma 4.2. *For any $P \in \mathcal{P}^{-1}_{2n+1}$ the path $f(P)$ is a Dyck path. Moreover, any Dyck path in $\mathcal{D}_{2n}$ is the image of exactly $2n+1$ paths in $\mathcal{P}^{-1}_{2n+1}$.*

The procedure for *random sampling of a Dyck path* of length $2n$ is now straightforward. Take a sequence of $n+1$ "zeros" and n "ones"; perform a permutation of these bits; perform $f(\cdot)$ on the resulting bits.

Now we are ready to present our *Flawed* PRNG. It takes as an input:

- *rng* – another PRNG (*e.g.,* Mersenne Twister MT19937-64),

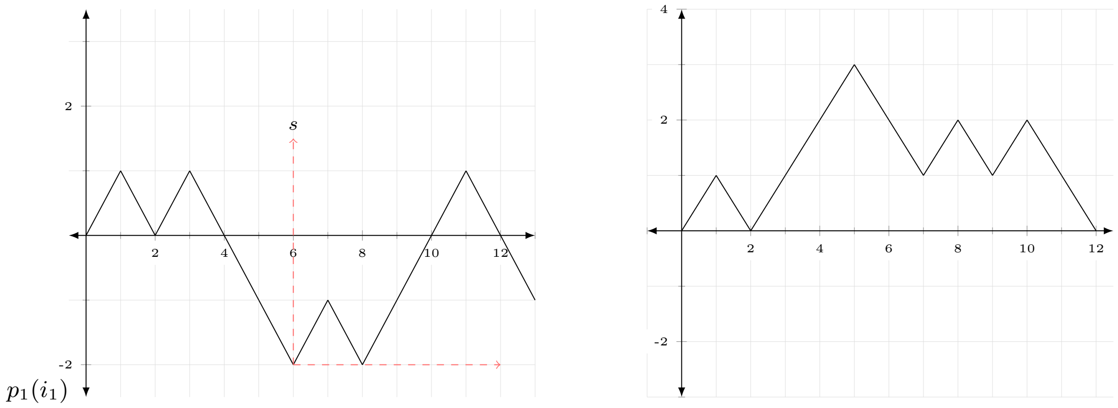

Figure 2: A path $P \in \mathcal{P}_{13}^{-1}$ (left) and corresponding Dyck path $f(P)$ of length 12 (right).

- N – the maximal number of bits returned by the generator is limited by 2^N,
- m – parameter which decides how often the output is modified,
- *seed* – the seed.

Flawed works in the following way.

Algorithm 1 Flawed$(rng, N, m, seed)$

```
 1: if $seed = 0 \bmod m$ then
 2: return $rng(seed)$
 3: else
 4:     $z_1 \dots z_{2^{N-2}} \leftarrow rng(seed)$
 5:     $\pi \leftarrow \mathsf{RandPerm}(2^{N-2}, rng(seed))$
 6:     for $i = 1$ to $2^{N-2}$ do
 7:         $z_{2^{N-2}+i} := 1 - z_{\pi(i)}$
 8:     end for
 9:     $z_{2^{N-1}+1 \dots z_{2^N}} \leftarrow \mathsf{DyckPaths}(N, z_1 \dots z_{2^{N-1}}, rng(seed))$
10: return $z_1 \dots z_{2^N}$
11: end if
```

The following is an informal description of the procedure DyckPaths$(n, r_1 \dots r_{2^{n-1}}, rng)$:

1. Denote the current half of the output as $\mathbf{r} = (r_1, \dots, r_{2^{n-1}})$, the corresponding random walk is given by $S_n = \sum_{i=1}^{n}(2r_i - 1)$. Recall (1): $D_k = \mathbb{1}\,(S_k > 0 \vee S_{k-1} > 0)$.
2. Define $O_0 := [l_0^O, r_0^O] := [0,0]$ and $O_i = \{l_i^O, \dots, r_i^O\}$, where $l_i^O = \min\{l_{i-1}^O < k \leq 2^{n-1} : D_k = 1\}, r_i^O = \max\{l_i^O < k \leq 2^{n-1} : D_k = 1\}$, $n_O = \min\{i : l_i^O = \emptyset\} - 1, |O_i| = r_i^O - l_i^O + 1$.
3. Similarly, $U_0 := [l_0^U, r_0^U] := [0,0]$ and $U_i = \{l_i^U, \dots, r_i^U\}$, where $l_i = \min\{l_{i-1}^U < k \leq 2^{n-1} : D_k = 0\}, r_i = \max\{l_i^U < k \leq 2^{n-1} : D_k = 0\}$, $n_U = \min\{i : l_i^U = \emptyset\} - 1, |U_i| = r_i^U - l_i^U + 1$.

   O_i and U_i are the intervals for which random walk is over x-axis and under x-axis respectively. For example, if
   $$(D_1, \dots, D_{16}) = (1,1,0,0,1,1,1,1,0,0,0,0,0,0,1,1),$$
   then $O_1 = [1,2], O_2 = [5,8], O_3 = [15,16], U_1 = [3,4], U_2 = [9,15]$.
4. In the second half, the bits will be chosen so that the walk will be $\sum_i |O_i|$ steps *under* x-axis and $\sum_i |U_i|$ over this axis in the following way. For this we will generate randomly?? Dyck paths of lengths $|U_1|, \dots, |U_{n_U}|$ and reflected (simply replacing bits $0 \leftrightarrow 1$) Dyck paths of lengths $|O_1|, \dots, |O_{n_O}|$. A

Table 12: Results of ASIN test for the *Flawed* generator described in Section 4.6.

n	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.0222	0.0301	0.0213	0.0232	0.0230	0.0258	0.0296	0.0301
sep1	0.1011	0.1529	0.1754	0.1305	0.1029	0.1186	0.1567	0.1594
sep2	0.1022	0.1635	0.1256	0.1404	0.1348	0.1709	0.1536	0.3582
p-val	0.9401	0.0567	0.8188	0.4730	0.7640	0.3532	0.0498	0.0000

Table 13: Results of LIL test for the *Flawed* generator described in Section 4.6.

n	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.0203	0.0192	0.0260	0.0212	0.0233	0.0208	0.0276	0.023
sep1	0.2047	0.1929	0.2834	0.1541	0.2018	0.2327	0.1586	0.1631
sep2	0.2309	0.0998	0.1459	0.1663	0.1299	0.1381	0.1648	0.2106
p-val	0.7220	0.9829	0.2290	0.9229	0.5943	0.8944	0.1497	0.5558

random permutation $\pi \leftarrow \mathsf{RandPerm}(n_O + n_U, rng(seed))$ of $n_O + n_U$ numbers will indicate their relative ordering.

5. Return computed bits.

10 sample trajectories of *Flawed* (all from *Dyck path*-based part of the Algorithm) are given in Fig. 3.

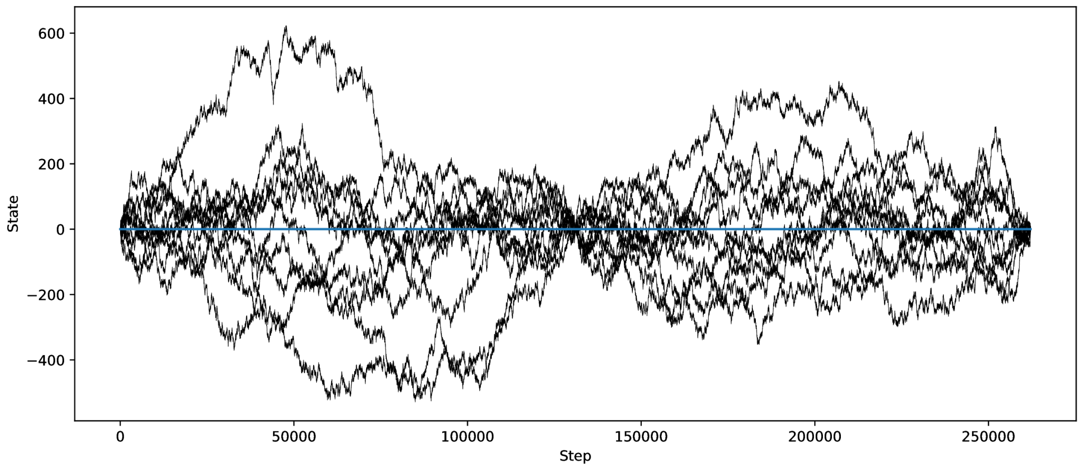

Figure 3: 10 trajectories of length 2^{18} of *Flawed* generator

4.6.1. Results of ASIN and LIL tests for Flawed

Testing results for this PRNG which produced $m = 10000$ sequences of length 2^{26} are shown in Tables 12 and 13. As we expected, for $n = 2^{26}$ the outcomes of ASIN test clearly indicate that the generator is flawed and its output cannot be recognized as random. Note that this is not the case for LIL test, where the obtained p-values give no sufficient evidence for rejecting this PRNG. The reason behind the differences in the behavior of ASIN and LIL tests is that the every hundredth trajectory of *Flawed* is exactly half of the time above x-axis, and half of the time below it. Therefore, the fraction of time when number of 1s exceeds number of 0s (*i.e.,* the random walk is above x-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 $0.01 \cdot m$ 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 values which belong to very likely intervals in the partition $\mathcal{P}^{lil}$ (first quater of bits are from MT19937-64, then the walk is forced (however randomly) to go to 0 in the

Table 14: Partial results from the `finalAnalysisReport.txt` file produced by NIST Test Suite applied for 10^4 sequences of length 2^{15} of flawed generator.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
976	1043	972	1007	943	1053	1029	960	924	1093	0.001907	9907/10000	Frequency
1015	999	927	972	1015	971	991	983	1029	1098	0.032705	9905/10000	BlockFrequency
956	982	1066	996	928	1028	1005	996	956	1087	0.008428	9901/10000	CumulativeSums
987	1027	951	967	1052	950	1011	977	938	1140	0.000101	9918/10000	CumulativeSums
1042	930	972	1008	997	986	993	995	957	1120	0.004239	9886/10000	Runs
1013	1015	1037	1014	1001	1008	947	988	989	988	0.812723	9809/10000 *	LongestRun
1006	869	1136	1016	1262	671	1466	864	778	932	0.000000 *	9803/10000 *	Rank
1146	928	987	1055	868	995	1076	949	993	1003	0.000000 *	9778/10000 *	FFT
1078	939	1008	999	1048	1027	978	1006	976	941	0.041709	9787/10000 *	OverlappingTemplate
2017	1305	1154	1083	969	911	755	700	596	510	0.000000 *	9576/10000 *	ApproximateEntropy
5	9	2	6	6	3	5	4	6	8	0.494392	53/54	RandomExcursions
6	4	5	4	6	5	7	4	6	7	0.971699	54/54	RandomExcursions
9	5	2	1	5	6	4	8	4	10	0.075719	54/54	RandomExcursions
11	4	7	3	6	6	2	5	9	1	0.040108	51/54	RandomExcursionsVariant
10	6	5	3	5	5	2	7	6	5	0.455937	53/54	RandomExcursionsVariant
8	9	4	1	6	6	7	4	5	4	0.350485	53/54	RandomExcursionsVariant
9	8	3	1	10	9	4	2	6	2	0.011791	53/54	RandomExcursionsVariant
1201	935	1038	983	1005	949	971	951	978	989	0.000000 *	9682/10000 *	Serial
1214	980	1037	1013	886	1001	934	925	1025	985	0.000000 *	9697/10000 *	Serial
1112	875	931	933	1038	1057	1057	1073	966	958	0.000000 *	9731/10000 *	LinearComplexity

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the random excursion (variant) test is approximately =
9870 for a sample size = 10000 binary sequences.

The minimum pass rate for the random excursion (variant) test is approximately = 51 for a sample size = 54 binary sequences.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

middle, then oscilates around zero in a similar way as in first half). Thus, it is more difficult for χ^2 test to detect such deviation.

Note also that *Flawed* passes all the tests of `swalk_RandomWalk1` from `TestU01`, see Table 11. The p-values for *Flawed* are even "better" than the ones for Mersenne Twister. At first glance it may be surprising, since *Flawed* is strongly based on Mersenne Twister (and is non-random). But this is because - in a sense - a walk (randomly enough, but) "oscilates" around x-axis and tests similar to LIL cannot detect flaws.

4.6.2. Results of `TestU01` for Flawed

TO DO:

4.6.3. NIST Test Suite

For $m = 10000$ sequences of length $n = 2^{15}$, the ASIN test already returns p-value 0.000. For comparison, for this parameters we ran we ran a battery NIST Test Suite. Results are given in Table 14. For each sequence the corresponding p-value is was calculated. NIST adopted two ways to interpret a set of p-values: a) examination of proportion of sequences that pass a certain test; b) testing the uniformity of p-values (for truly random sequences they should be uniformly distributed in $[0, 1]$).

Values in the columns C1,..., C10 represent number of p-values falling into intervals $[0.0, 0.1), [0.1, 0.2), \ldots, [0.9, 1]$. Values in P-VALUE column represent the results for uniformity testing of p-values, value in PROPORTION column represent proportion of sequences that pass a given test. The character $*$ denotes that the PRNG failed given test (more exactly: given interpretation of a set of p-values). For detailed description of results' interpretation see [6]. Note that some tests from NIST Test Suite detected the flaw, but most did not.

5. Conclusions

In this paper we analyzed method for 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 [8], 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 or `TestU01`. 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 non-random 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 the quality of random numbers generated by both new PRNGs implementation and those being already in use, especially for cryptographic purposes.

Acknowledgements

We thank two anonymous reviewers whose suggestions helped improve and clarify this manuscript.

References

[1] D. E. Knuth, The art of computer programming, Volume 2: Seminumerical Algorithms, 3rd Edition, Addison-Wesley Pub. Co, 1997.

[2] R. G. Brown, D. Eddelbuettel, D. Bauer, Dieharder: A Random Number Test Suite, www.phy.duke.edu/~rgb/General/dieharder.php.

[3] 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.

[4] P. L'Ecuyer, R. Simard, TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators. User's guide, detailed version, 23 April 2017.

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

[6] 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).

[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] 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.

[9] 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.

[10] H. Ekkehard, A. Grønvik, Re-seeding invalidates tests of random number generators, Applied Mathematics and Computation 217 (1) (2010) 339–346.

[11] P. Lorek, M. Słowik, F. Zagórski, Statistical testing of PRNG: Generalized gambler's ruin problem, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 10693 LNCS, 2017, pp. 425–437.

[12] W. Feller, An Introduction to Probability Theory and Its Applications, Volume 2, 3rd Edition, John Wiley & Sons, 1968.

[13] S. Asmussen, P. Glynn, Stochastic Simulation: Algorithms and Analysis, Springer, 2007.

[14] D. P. Kroese, T. Taimre, Z. I. Botev, Handbook of Monte Carlo Methods, John Wiley & Sons, Inc., Hoboken, NJ, USA, 2011.

[15] P. L'Ecuyer, History of uniform random number generation, in: 2017 Winter Simulation Conference (WSC), IEEE, 2017, pp. 202–230.

[16] H. Niederreiter, Quasi-Monte Carlo methods and pseudo-random numbers6, Bulletin of the American Mathematical Society 84 (6) (1978) 957–1041.

[17] M. Denker, W. A. Woyczynski, Introductory statistics and random phenomena : uncertainty, complexity, and chaotic behavior in engineering and science, Birkhäuser Boston, 1998.

[18] A. Khintchine, Über einen Satz der Wahrscheinlichkeitsrechnung, Fundamenta Mathematicae 6 (1) (1924) 9–20.
[19] K. Takashima, Sojourn time test for maximum-length linearly recurring sequences with characteristic primitive trinomials 7 (1994) 77–87.
[20] K. Takashima, Sojourn time test of m-sequences with characteristic pentanomials, Journal of the Japanese Society of Computational Statistics 8 (1995) 37–46.
[21] K. Takashima, Last visit time tests for pseudorandom numbers, Journal of the Japanese Society of Computational Statistics 9 (1) (1996) 1–14.
[22] RANDOM.ORG - True Random Number Service, https://www.random.org/.
[23] S. K. Park, K. W. Miller, Random number generators: good ones are hard to find, Communications of the ACM 31 (10) (1988) 1192–1201.
[24] S. K. Park, K. W. Miller, P. K. Stockmeyer, Technical correspondence, Communications of the ACM 36 (7) (1993) 105.
[25] A. Dvoretzky, T. Motzkin, A problem of arrangements, Duke Mathematical Journal 14 (2) (1947) 305–313.