applied sciences

MDPI

Article

Statistical Quality of Number Sequences Produced by Pseudorandom Number Generators Built into Contemporary Programming Languages

Marcel Bieżuński, Michał Dulek, Mieczysław Jessa and Łukasz Matuszewski *

Institute of Multimedia Telecommunications, Faculty of Computing and Telecommunications, Poznan University of Technology, ul. Jacka Rychlewskiego 1, 61-131 Poznan, Poland; marcel.biezunski@put.poznan.pl (M.B.); mdulek0401@gmail.com (M.D.); mieczyslaw.jessa@put.poznan.pl (M.J.)
* Correspondence: lukasz.matuszewski@put.poznan.pl

Abstract

The article presents the statistical quality of number sequences produced by pseudorandom number generators (PRNGs) built into or available in libraries of contemporary programming languages. Since PRNGs are foundational to many sciences, including cryptography, statistics, and simulation methods, high-quality PRNGs are crucial for reliable results. The study evaluated 143 pseudorandom generators from 17 popular programming languages. A widely recognized suite of 15 statistical tests developed by the National Institute of Standards and Technology (NIST), was used to assess randomness, and the research's high accuracy was ensured by generating 1000 sequences for each generator, each one million bits long. The tests were performed for two variants: one with a fixed seed equal to 1 and one with a randomly selected seed. Thanks to these studies, the user can become aware of the limitations of the tools used and, if necessary, replace a weak generator with a better one. The results show that 113 of the 143 generators did not fail any test in either scenario, and 66 of them passed all tests without a single borderline result. However, four legacy linear congruential generators available in the GNU Scientific Library and both generators built into Delphi failed most of the tests, while the standard rand() function of C and C++ consistently failed the Discrete Fourier Transform test. Most borderline results were observed for the Non-overlapping Template Matching test.

Keywords: programming languages; pseudorandom number generators; statistical quality of pseudorandom number generators

Check for updates

Academic Editor: Christos Bouras

Received: 13 August 2026
Revised: 4 October 2026
Accepted: 6 October 2026
Published: 9 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Pseudorandom number generators (PRNGs) are used in various fields of science. They serve as the basis for methods for modeling random systems and techniques for simulating the behavior of various objects, and are utilized in statistics and cryptography. The high statistical quality of PRNG-generated sequences is crucial for obtaining reliable results across all design phases of many systems. Even small deviations from the desired statistical quality can have catastrophic consequences when simulating so-called rare events, for example, from the "tail" of the Gaussian distribution.

Currently, users have access to many programming languages and, practically everyone, can use built-in pseudorandom number generators or those available in libraries. Unfortunately, the literature lacks information on the quality of sequences produced by these generators. Users often assume that because a generator is available in a widely used

programming language, the statistical quality of the generated sequences is sufficient. Our research indicates that this popular belief may not be true in all cases.

In this work, we describe the results of studying the statistical properties of number sequences produced by pseudorandom generators available in Python 3.12, C++, C, C#, Arduino, Visual Basic, Matlab, Java, JavaScript, Go, Kotlin, Julia, Delphi, PHP, Ruby, Rust, R, as well as in libraries dedicated to these environments. For each environment, in addition to the default implemented generators, one additional library implementing a pseudorandom generator was selected (e.g., NumPy for Python, GSL for C++, etc.). Table 1 lists the libraries used, along with their URLs.

Table 1. The list of libraries used in the research together with their URL addresses.

Programming Language	Library	URL
C++	GNU Scientific Library GSL	https://www.gnu.org/software/gsl/doc/html/rng.html (accessed on 15 March 2026)
C	PCG Library	https://www.pcg-random.org/download.html (accessed on 25 April 2026)
C#	XoshiroPRNG.Net	https://sourceforge.net/p/xoshiroprng-net/code/ci/default/tree/ (accessed on 4 April 2026)
Visual Basic	Math.NET Numerics	https://numerics.mathdotnet.com/ (accessed on 24 March 2026)
Rust	Rand	https://docs.rs/rand/latest/rand/index.html (accessed on 23 March 2026)
Python	Numpy	https://numpy.org/ (accessed on 16 March 2026)
Python	RandomGen	https://pypi.org/project/randomgen/ (accessed on 23 March 2026)
JavaScript	SeedRandom	https://cdnjs.com/libraries/seedrandom (accessed on 12 May 2026)
Ruby	OpenSSL	https://ruby-doc.org/3.0.5/exts/openssl/OpenSSL.html (accessed on 12 January 2026)
Java	Apache Commons Math	https://commons.apache.org/proper/commons-math/ (accessed on 21 March 2026)

Our goal is not to investigate why one generator passes statistical tests, and another does not, but to give users useful knowledge about the statistical quality of sequences produced by a given generator. The user can either use such a pseudorandom number generator consciously or attempt to replace it with another generator with better statistical properties.

The statistical quality of a PRNG is not merely of theoretical interest. A well-known example is Monte Carlo simulations of the Ising model, essential for studying phase transitions in magnetic materials, performed with generators then considered of good quality that yielded systematically incorrect results [1]. The processes most sensitive to such deficiencies are those that consume very large amounts of random numbers or depend on the tails of distributions, such as Monte Carlo simulations, rare-event estimation, resampling methods in statistics, and stochastic procedures in machine learning. In cryptographic applications, good statistical properties are necessary, though not sufficient, for a generator's security.

The main findings of this study can be summarized as follows. Most contemporary generators, including those based on the Mersenne Twister, PCG, xoshiro/xoroshiro, ChaCha, and Philox families, produce sequences of high statistical quality: 113 of the 143 tested generators did not fail any of the NIST Statistical Test Suite (STS) described in NIST publication

SP 800-22, regardless of the seeding scenario. At the same time, several generators that are still readily available to users exhibit serious deficiencies. The legacy linear congruential generators, available in the GNU Scientific Library, failed between 10 and 15 of the 15 tests, and both generators built into the Delphi environment failed 8 to 9 tests. Moreover, the rand() function in the C and C++ standard libraries, probably the most frequently used generator in practice, failed the DFT test in both scenarios. We also observed that the same algorithm may yield different results depending on the implementation environment.

The main contributions of this paper are as follows:

1. An empirical evaluation of the statistical quality of 143 pseudorandom number generators available in 17 popular programming languages and their libraries, performed with a uniform methodology based on the NIST SP 800-22 suite.
2. A comparison of the results obtained for two seeding scenarios, a fixed seed and a random seed, which reflects the typical ways in which generators are used in practice.
3. The identification of generators that are still readily available to users but produce sequences with statistical deviations from randomness, as well as the observation that the same algorithm may yield different results depending on the environment in which it is implemented.
4. The publication of the complete numerical results for all tested generators, which can serve as a reference for users and for further research.

2. Testing Methodology

Several generally available statistical test suites can be utilized to assess the randomness of sequences produced by pseudorandom generators. The most popular include the statistical test suite STS developed by the National Institute of Standards and Technology described in SP 800-22 [2], the TestU01 statistical test suite [3], and the DIEHARD suite [4]. In this study, we chose the NIST STS for two main reasons. First, STS was developed and is maintained by a U.S. government agency, and it is recognized worldwide as a basic reference tool for evaluating the randomness of binary sequences, whereas other suites, although valuable in research, do not have the status of a standard. Second, it provides explicitly defined acceptance criteria for both the proportion of sequences passing each test and the uniformity of the p-value distribution, allowing an unambiguous pass/fail classification of a large number of generators. However, the results presented in the article should not be viewed as exhaustive—that is, as detecting every possible anomaly in pseudorandom number sequences. Packages such as TestU01 or DIEHARD can provide additional statistical information about the generated sequences; in particular, the Crush and BigCrush test batteries in the TestU01 suite can detect subtle correlations or structural defects in PRNGs. TestU01 should be the tool of choice when evaluating a specific algorithm for producing l-bit pseudorandom numbers, prior to its publication. Although NIST STS targets cryptographic-level randomness, passing the tests from this package does not mean that the tested generator is suitable for cryptographic applications. The same applies to interpreting results from the TestU01 and DIEHARD test suites. Passing all tests from all test batteries does not prove that a generator is truly random or cryptographically secure. They provide statistical evidence that the observed output is consistent with the expected behavior of a random sequence. For cryptographic PRNGs, statistical testing should therefore be combined with an analysis of the generator's design, state size, and resistance to prediction.

The NIST SP 800-2 test suite contains 15 tests designed to verify whether the selected generator actually generates number sequences that can be classified as random. The tests examine various aspects of the randomness of a number sequence, from the simplest features, such as the frequencies of zeros and ones, to more complex patterns. Each NIST test verifies two hypotheses: the null hypothesis (H_0), which assumes the sequence is

random, and the alternative hypothesis (H_1), which states the sequence is not random. Acceptance of the H_0 hypothesis occurs when the test statistic does not exceed a critical value. This value is determined based on certain statistics of random sequences [2]. The rejection of the H_0 hypothesis in favor of the alternative hypothesis H_1 indicates that the critical value has been exceeded and the rejection region has been entered, which is a sign of sequence non-randomness.

During testing, there is a chance of accepting an incorrect hypothesis. We then commit either a Type I error, rejecting the null hypothesis when it is true, or a Type II error, accepting the null hypothesis when, in reality, the sequence is not random from the perspective of a given test. The probability α of a Type I error (also called the significance of the test) and the probability β of a Type II error are correlated and affect, among other things, the required length of the tested sequence [2]. The number α is chosen so that events whose probability of occurrence is no greater than α are considered unlikely. The significance level a is typically equal to 0.01. The result of the test is the so-called p-value, which reflects the probability that the observed result is due to chance. For p-value = 1, we obtain a perfectly random sequence from the point of view of the given test. The test can be considered passed when p-value $\geq \alpha$ [2].

The STS described in SP 800-22 consists of 15 tests, some of which include two or more subtests. They are (Table 2):

Table 2. The list of NIST SP800-22 tests.

No.	Test	What the Test Checks
1.	Frequency (Monobit)	Whether the proportion of ones and zeros in the entire sequence is close to 1/2.
2.	Frequency within a Block	Whether the proportion of ones within M-bit blocks is close to M/2.
3.	Cumulative Sums *	Whether the maximum excursion of the random walk formed by the cumulative sum of the sequence (with bits mapped to ± 1) matches that of a random sequence; computed in the forward and backward directions.
4.	Runs	Whether the total number of runs (uninterrupted sequences of identical bits) is as expected, i.e., whether the oscillation between zeros and ones is neither too fast nor too slow.
5.	Longest Run of Ones in a Block	Whether the length of the longest run of ones within M-bit blocks is consistent with a random sequence.
6.	Binary Matrix Rank	The rank of disjoint 32 × 32 binary matrices built from the sequence, to detect linear dependence among fixed-length substrings.
7.	Discrete Fourier Transform (Spectral)	The peak heights in the discrete Fourier transform of the sequence, to detect periodic features.
8.	Non-overlapping Template Matching *	The number of occurrences of predefined aperiodic 9-bit patterns; after a match, the search window is moved past the found pattern.
9.	Overlapping Template Matching	The number of occurrences of a 9-bit run of ones; after a match, the window slides by only one bit.

Table 2. *Cont.*

No.	Test	What the Test Checks
10.	Maurer's "Universal Statistical"	The number of bits between matching patterns, to detect whether the sequence can be significantly compressed.
11.	Approximate Entropy	The frequencies of all overlapping m-bit and (m + 1)-bit patterns, compared with those expected for a random sequence.
12.	Random Excursions *	The number of visits to states −4 to +4 (excluding 0) within cycles of the cumulative-sum random walk.
13.	Random Excursions Variant *	The total number of visits to states −9 to +9 (excluding 0) of the cumulative-sum random walk.
14.	Serial *	Whether all overlapping m-bit patterns occur with approximately the same frequency.
15.	Linear Complexity	The length of the shortest linear feedback shift register that generates each block of the sequence; too short a register indicates non-randomness.

The * symbol indicates that a given test includes two or more subtests.

The testing process can be performed using software provided by NIST [2] or a dedicated application, Random Bitstream Tester [5]. When publishing test results marked with an asterisk, we report the worst result among all subtests as the final test result.

A full evaluation of the generator itself requires an exhaustive test, i.e., examining all sequences with a given number of elements produced by the generator. In practice, this is impossible, so the null hypothesis, which in the case of generator evaluation is that the tested generator produces random sequences, is assessed based on a sample from the general population. For a typical significance level of 0.01, we examined $N = 1000$ different sequences produced by the generator, each 10^6 bits long, aiming to ensure sufficient accuracy in estimating the proportions of sequences that pass and fail the test. Since an ideal random generator can produce not only sequences that pass the test but also sequences that fail it (the ideal random number generator does not "prefer" any sequence), we cannot demand that the generator produce only sequences that pass all statistical tests. This fact is reflected in the acceptable proportion R of strings that pass a given test [2].

The proportion of sequences passing the test is not the only criterion that a generator must meet to be considered a random generator for a given significance level. Another important criterion is the distribution of p-values obtained for $N = 1000$ sequences. Because ideally, all output sequences are equally probable, the distribution of p-values calculated for a given test should be uniform. In the NIST STS, this property is verified with the chi-square goodness-of-fit test as follows. The interval [0, 1] is divided into 10 equal subintervals C_1, C_2, ..., C_{10}, and the number of p-values falling into each subinterval is counted. For a uniform distribution, each subinterval should contain approximately $N/10$ p-values, i.e., 100 for $N = 1000$. The chi-square statistic measures how much the observed counts deviate from this expected value, and on its basis a second-level p-value, denoted as P_T, is computed. Thus, P_T does not refer to a single sequence but describes the uniformity of the p-values obtained for all N sequences. If $P_T \geq 0.0001$, the distribution of p-values for a given test is considered uniform [2]. Accepting a generator as pseudorandom requires meeting both of the requirements described. This is particularly important in cryptography and in simulating the rare events mentioned in the Introduction. Small deviations from randomness can lead to successful attacks or, for example, the earlier occurrence of events

whose probabilities were underestimated by using a generator that does not meet the criteria mentioned in this paper.

We have conducted the tests in two scenarios: with a fixed seed set to 1, and with a randomly selected seed for each generator run. The method of producing a random seed depended on the tested programming environment. Table 3 lists the functions used and their descriptions.

Table 3. Generation of a random seed.

Program	Function	Description
Python	random.seed(None)	Exemplary function used
C++	srand(time(NULL)	Unix time
C	srand(time(NULL))	Unix time
C#	new Random()	Random seed based on system time
Arduino	analogRead(A0)	Sampling noise from an analog input pin
Visual Basic	new Random()	Random seed based on system time
Matlab	currentTime = datetime('now'); seed = mod(round(posix-time(currentTime) * 1 × 10^3), 2^{32})	Getting the current time Using modulo to limit the range to $[0, 2^{32} - 1]$
Java	new Random()	Uses System.nanoTime() + seedUniquifier()
JavaScript	Date.now().toString()	Random seed based on system time
Go	Funkcja time.Now().UnixNano()	Unix time
Kotlin	val random = Random() System.nanoTime()	Random seed selected automatically using System.nanoTime() mixed with an internal JVM counter
Julia	RandomDevice()	Reads bytes from the OS kernel
Delphi	Randomize	Random seed based on system time
PHP	random_bytes()	Reads bytes from the OS kernel
Ruby	Random.new	System time + counter
Rust	from_entropy()	Generates a seed based on the system's entropy
R	set.seed(as.numeric(Sys.time()))	Using the current time

The testing procedure applied to each generator can be summarized in the following steps:

1. The generator was initialized with a fixed seed equal to 1 or with a random seed obtained as described in Table 3.
2. N = 1000 binary sequences of 10^6 bits each were generated.
3. Each sequence was subjected to the 15 tests of the NIST STS with the default parameters.
4. For each test, the proportion R of sequences with a p-value $\geq \alpha = 0.01$ and the uniformity of the p-value distribution (P_T) were determined.
5. Each test result was classified as pass ($R \geq 980$ and $P_T \geq 0.0001$), borderline ($R = 978$ or 979), or fail (otherwise). For tests with several subtests, we reported the worst result.
6. Steps 1–5 were repeated for the second seeding scenario.

3. Research Results

This section presents the results of statistical tests for 143 PRNGs of various types available in 17 popular programming environments. Given the vast number of software implementations of PRNGs, selecting specific ones for rigorous evaluation is extremely challenging. In each programming language discussed, it is possible to implement numerous PRNGs—either of different types or of the same type but with slightly varying parameters. The study focused on generators built into the respective programming environments and those available in popular libraries for these environments. For some platforms, the number of built-in PRNG solutions is very limited. For Arduino, for instance, only the random() function is provided. Naturally, other generators—including those we examined, such as Xorshift32 or PCG—can also be implemented. The choice of studied generators is broad enough to indicate at least one generator for a given environment that passes all randomness tests for proportion and distribution of *p*-values. An exception is the Arduino environment, for which we tested only one generator—the one built into the environment. We do not describe the methods for generating pseudorandom numbers, as these are known and covered in many publicly available review articles or books (see, e.g., [6–21] and the References therein). Due to space limitations, the paper includes a graphical version of the results. The numerical values for the tests performed for all 143 generators are available in Supplementary Materials: https://github.com/lmatuszewski-cmyk/PRNG (accessed on 10 September 2026).

For the assumed test significance level of $\alpha = 0.01$, the minimum acceptable proportion value was 980/1000, i.e., 0.98 [2]. The exceptions are the random walk tests (RandomExcursions* and RandomExcursionsVariant*), for which the minimum acceptable proportion can change. If the generator fails a test for R or P_T, it is counted as one failed test. Borderline results are considered those that achieve an R score of no less than 2 below the acceptance level. If the minimum pass ratio for tests is 980 per 1000 sequences, then scores of 978 and 979 are considered borderline. Lower values are considered a failed test. A borderline test result suggests the need for further research, particularly for cryptographic applications. For readability, we partitioned the complete 143 × 15 result matrix into six panels (Figures 1–6) such that the generators in each panel group belong to a closely related language. Within each panel, the left half shows the results obtained with a fixed seed (seed = 1), and the right half shows the results obtained with a random seed. The test ordering is identical in both halves, allowing you to compare homologous cells at a glance. Every cell aggregates 1000 independent sequences of 10^6 bits each, and the classification follows the standard NIST acceptance criteria. A test is reported as pass when the proportion of accepted sequences satisfies $R \geq 980/1000$, and the uniformity p-value satisfies $P_T \geq 10^{-4}$. Borderline results are those that achieve an R score no less than 2 below the acceptance level (980). Fail when R or P_T are violated.

Most generators available in contemporary programming languages, including the default generators of Python, Java, Go, Rust, and R, can be safely used in typical simulation and statistical applications. However, some generators that are still readily available and often used by default, such as the rand() function of C and C++, the random() function of Arduino, and the generators built into Delphi, produce sequences with statistical deviations from randomness, which may bias the results of Monte Carlo simulations, the modeling of rare events, or randomized algorithms. This is particularly relevant in education and embedded systems, where these environments are widely used, and the default generator is rarely questioned. PRNGs built into Delphi, or other generators that fail statistical tests, often have worse statistical quality because traditional generators were designed for speed, simplicity, and compatibility rather than modern high-quality statistical simulation or cryptography. Older PRNGs were also developed before comprehensive batteries such as TestU01 became

widely used. Consequently, a generator may pass simple tests but fail more demanding tests designed to detect subtle dependencies. Moreover, since the same algorithm may yield different results depending on the environment, users should verify not only the choice of algorithm but also its specific implementation, and in cryptographic applications they should use only dedicated cryptographically secure generators. The fact that sequences produced by a PRNG pass all statistical tests does not mean it is automatically cryptographically secure. For cryptographic applications, statistical testing must be supplemented with an analysis of the PRNG's forward and backward security for all possible seed values.

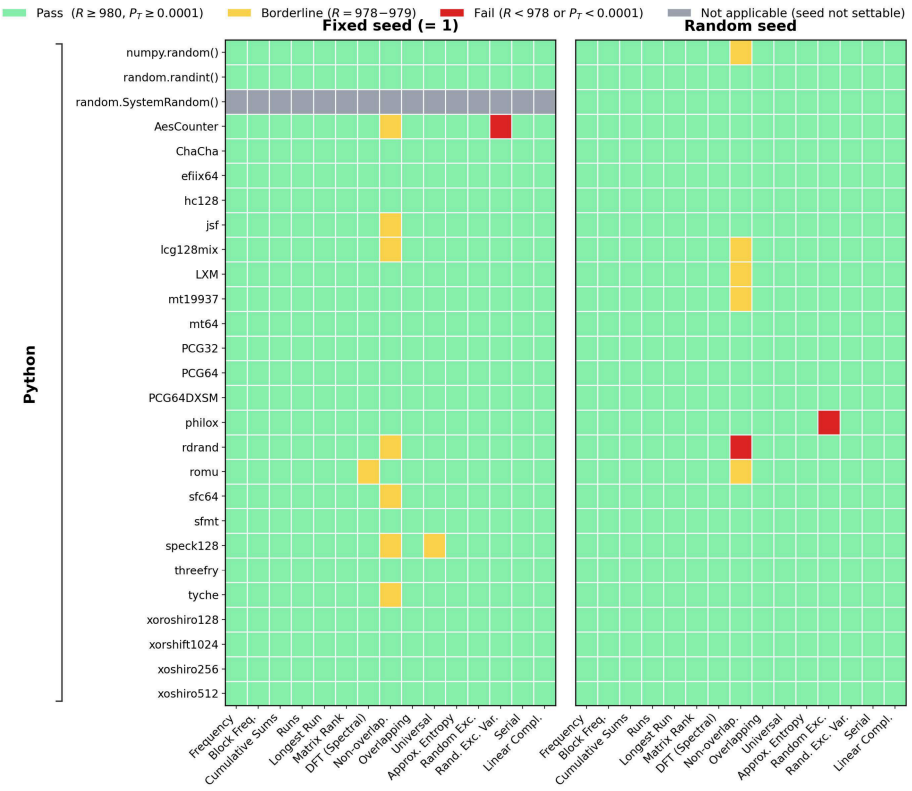


Figure 1. The results for the Python language (27 generators).

Many generators are on the borderline of the passing criterion. Unfortunately, this group also includes a considerable number of generators described as secure, meaning they should show very good results. A borderline result in NIST STS should not be interpreted as 'the PRNG is almost random' in a universal sense. It means that, for the particular finite sample, test, parameters, and significance level, the output is close to the statistical-test decision boundary. NIST itself emphasizes that STS is a useful first step, not a certificate of suitability because statistical testing cannot replace cryptanalysis. Do not use a borderline or general-purpose PRNG for cryptographic purposes. In some non-cryptographic applications, borderline results can be acceptable because a borderline p-value is not necessarily evidence that the generator has a real defect. Statistical tests have false positives and false negatives, and the outcome depends on sample size and significance level.

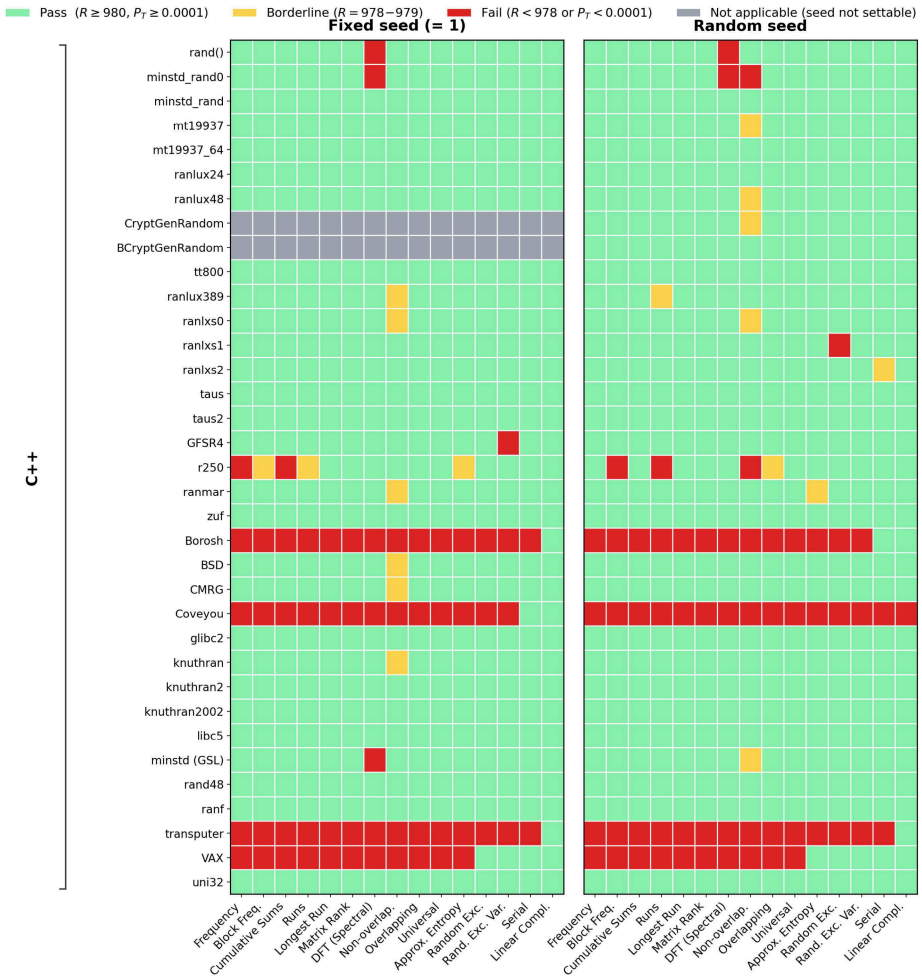


Figure 2. The results for the C++ (35 generators).

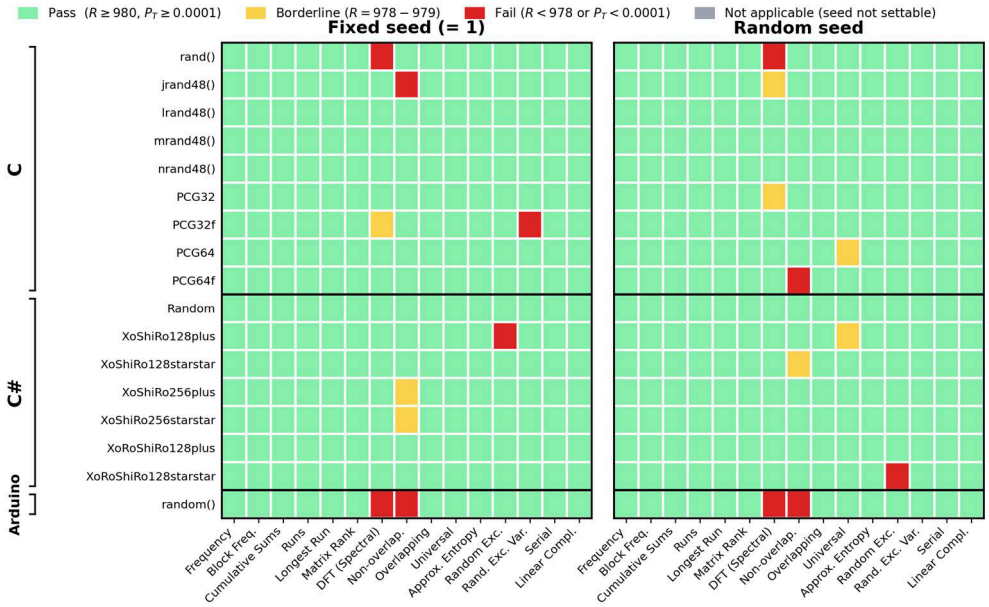


Figure 3. The results for C, C#, and Arduino (17 generators).

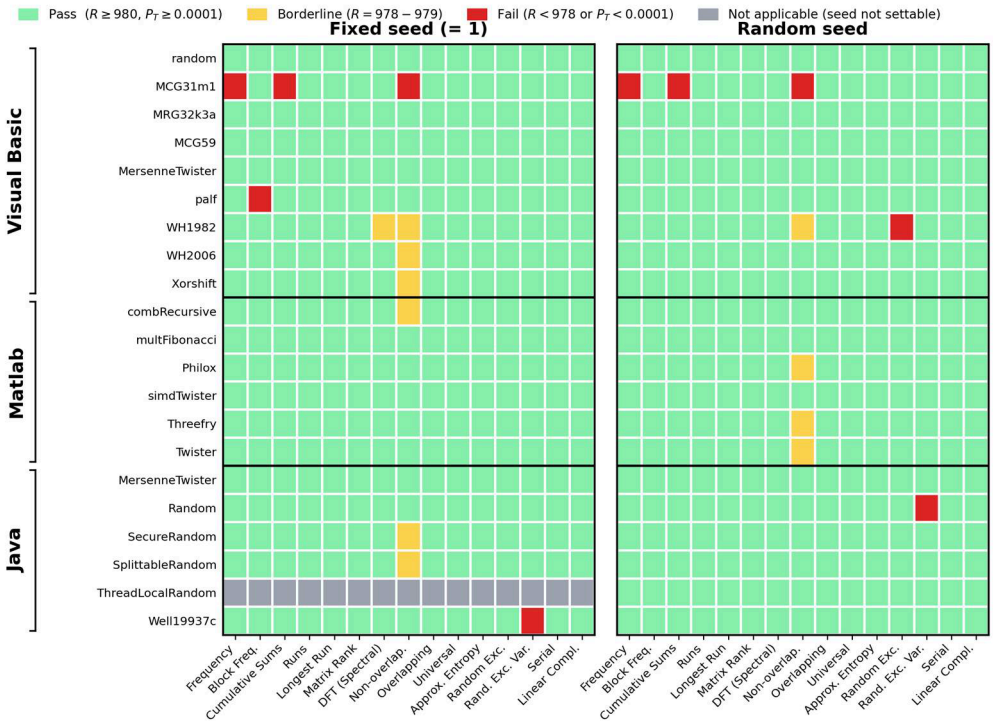


Figure 4. The results for Visual Basic, Matlab, and Java (21 generators).

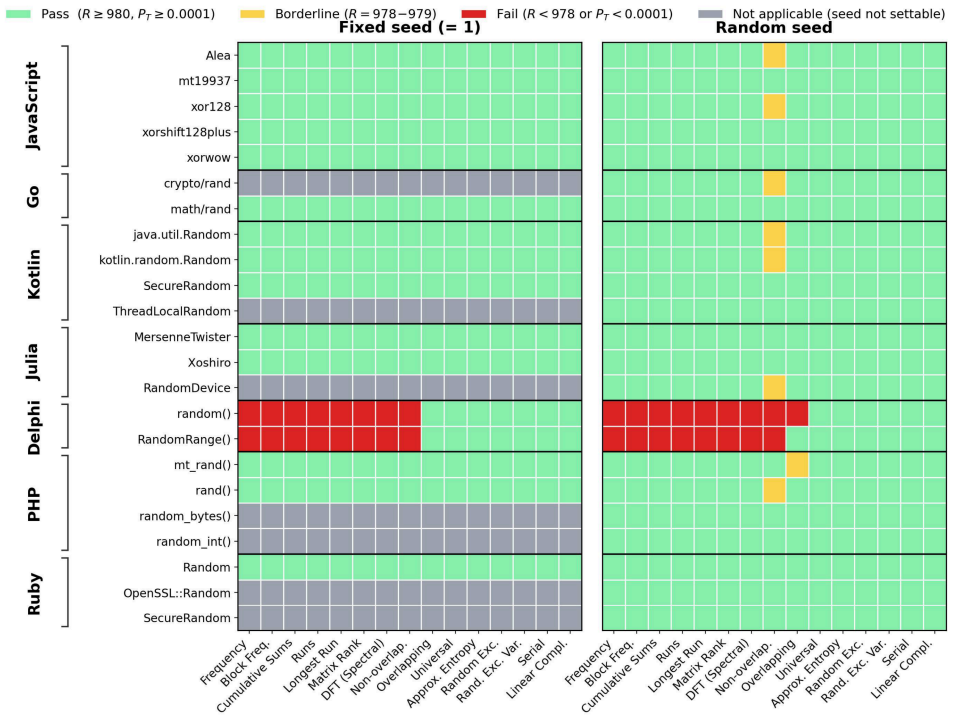


Figure 5. The results for JavaScript, Go, Kotlin, Julia, Delphi, PHP, and Ruby (23 generators).

For Monte Carlo simulation, marginal test results do not automatically invalidate a PRNG. If the PRNG has a long period, known good statistical properties, and the simulation's estimator is insensitive to the detected defect, the results can still be useful. Such a generator may be acceptable, but we should validate the simulation output, preferably with an independent, high-quality generator and multiple seeds.

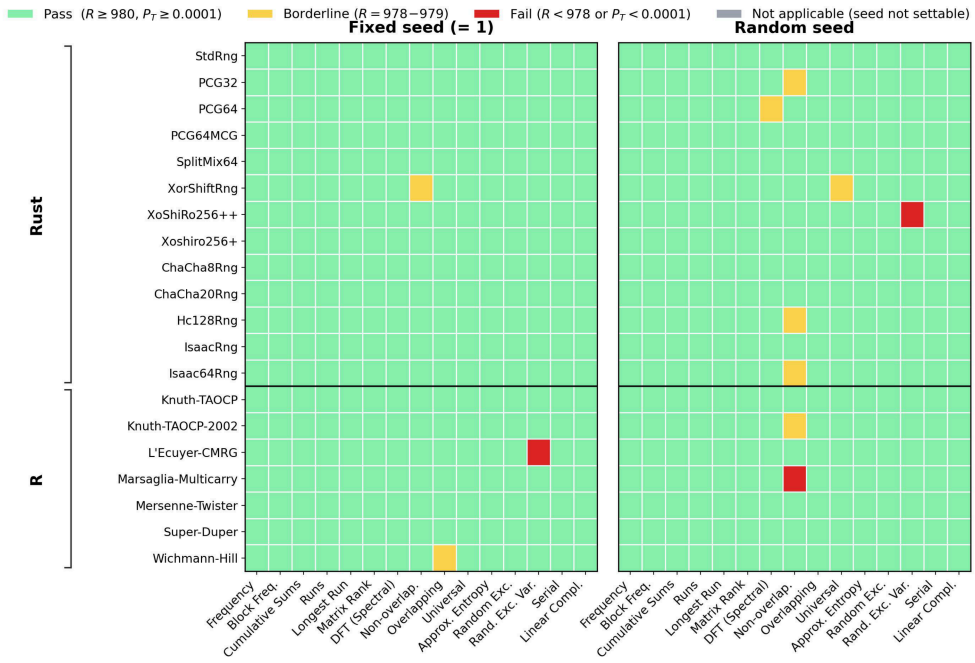


Figure 6. The results for Rust and R (20 generators).

In numerical simulations or scientific computing, we do not recommend using PRNGs that produce borderline statistical test results because small correlations or structural artifacts can interact with an algorithm being studied in ways that ordinary NIST STS do not expose. The danger increases for very large simulations, quasi-deterministic algorithms, parallel streams, or simulations known to be sensitive to correlations. A significantly better solution is to choose a well-established, thoroughly tested generator rather than deliberately choosing a borderline one.

In rare events simulation, using a generator with borderline statistical performance is problematic. The quantity of interest may depend on rare event probabilities. The precise probability value depends on the physics of the simulated phenomenon. A tiny systematic defect in the random sequence can have an outsized effect on the estimated tail probability, and the rare events algorithm may deliberately amplify particular regions of the sample space. When simulating rare events, avoid borderline-quality sequences, use a generator with strong empirical/theoretical validation, and independently validate the rare event estimator.

The high statistical quality of the pseudorandom numbers produced by the generators under study is not the only factor in choosing a specific generator. Another, often crucial factor is selecting a solution with acceptable computational complexity and low computational cost.

The computational complexity of pseudorandom number generators depends on their underlying mathematical operations per generated word. Nearly all modern PRNGs run in $O(1)$ time complexity per generated word and require $O(1)$ spatial memory complexity, proportional to their state size. For a fixed generator configuration, all studied generators produce one output in $O(1)$ time. However, they differ significantly in terms of computational cost. This cost varies substantially by implementation, word size, and whether you count only the state transition or also seeding or output transformations [6–21]. A useful comparison is therefore relative cost per generated word. The cheapest generators require only one to seven operations per output. This group includes linear congruential and multiplicative congruential generators (LCG, MCG), multiply-with-carry generators (MWC), lagged Fibonacci generators (LFG, GFSR), Xorshift, and Romu. Most modern

general-purpose generators require approximately seven to fifteen operations per output. This group includes the XoShiRo and XoRoShiRo families, PCG, JSF, SFC, the Mersenne Twister, and its Single Instruction, Multiple Data (SIMD) oriented variant SFMT. In practice, speed differences within these two groups are small. The computational cost increases noticeably for generators based on arithmetic modulo a prime number, such as combined LCG, MRG, and combined multiple recursive generator (CMRG). Such generators require modular reductions, which are slower than generators using arithmetic modulo 2^l, where l is the number of bits encoding pseudorandom numbers. Cipher-based and cryptographically secure generators perform many rounds of mixing per output block, which further increases computation cost. A useful grouping of the methods of producing pseudorandom numbers may be the following:

- Very low-cost: MCG, LCG, LFSR, GFSR, Xorshift.
- Low-cost: XoShiRo, XoRoShiRo, SFC, JSF, PCG, Romu, MWC
- Moderate cost: MRG, LFG, Combined LCG, CMRG, WELL, SFMT
- Moderate-high cost: Mersenne Twister, ISAAC, Ranlux
- Highly implementation-dependent: CBPRNG, CSPRNG

The classification presented here, while practically useful, is only a rough approximation. It does not account for memory latency, vectorization, compiler optimizations, or a generator's actual throughput that depends on the platform and should be measured in the target environment. To determine the precise computational cost requires full knowledge of the generator's implementation details, the hardware it runs on, and the programming environment. The proposed classification may also change with hardware acceleration for certain calculations—for instance, by using a processor with SIMD capabilities. Another particularly important point is that computational cost should not be considered independently of statistical quality. An LCG may be several times cheaper than PCG, but PCG's extra few integer operations can be an excellent trade-off when the random numbers are used for numerical simulation requiring parallel generation of multiple pseudorandom number streams. If high speed is a priority, suitable choices include generators employing methods such as XoShiRo, XoRoShiRo, Romu, SFC, JSF, or a high-quality Xorshift-family generator. If we need both high pseudorandom number generation speed and excellent statistical quality, a good choice is a generator based on PCG, XoShiRo, or XoRoShiRo. In turn, the Mersenne Twister yields sequences with extremely long periods.

4. Conclusions

The aim of the work was to evaluate the quality of pseudorandom number generators available in various programming environments, which essentially means verifying to what extent different generators meet the criteria of randomness. During testing, the National Institute of Standards and Technology guidelines and resources were applied, meaning all conclusions are constrained by the NIST test suite. The research evaluated 143 pseudorandom generators using 24 methods of generating pseudorandom sequences from 17 programming environments. The analyses identified the weak and strong traits of specific generators, which determine their potential utility in various fields. Comparing the results within this framework shows that although most modern generators provide very good randomness quality for basic applications, advanced practice may require deeper analysis and more careful generator selection. Borderline-quality sequences are clearly inappropriate for cryptographic use and should also be avoided for rare-event simulations. For routine Monte-Carlo simulation, borderline STS behavior is not automatically fatal, but there is usually little reason to choose a borderline generator when a well-characterized alternative is available. Another important conclusion is that a greater computational cost does not, by itself, lead to better statistical quality as measured by the NIST STS. Generator

choice should consider both statistical quality and the application's requirements. For simulations and statistical applications, fast generators that pass all tests are sufficient, while the higher computational cost of cryptographically secure generators is justified only when unpredictability is required.

The results presented in this paper can serve as a reference point for further research aimed at developing more effective pseudorandom number generation algorithms and as a foundation for analyzing their applications across various fields of computer engineering. An interesting direction for further research would be to compare results obtained using the NIST methodology with those from studies using the TestU01 and DIEHARD suites for a large group of 143 PRNGs representing 24 methods of generating pseudorandom sequences.

Supplementary Materials: The following supporting information can be downloaded at: https://github.com/lmatuszewski-cmyk/PRNG (accessed on 10 September 2026).

Author Contributions: Conceptualization, Ł.M. and M.J.; methodology, M.B. and M.D.; investigation, M.B. and M.D.; resources, M.B. and M.D.; writing—original draft preparation, Ł.M. and M.J.; writing—review and editing, M.J.; visualization, Ł.M.; supervision, M.J.; project administration, M.J.; funding acquisition, M.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Polish Ministry of Science, grant number 0314/SBAD/0261.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The numerical results of tests performed on all 143 generators are available at https://github.com/lmatuszewski-cmyk/PRNG (accessed on 10 September 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Ferrenberg, A.M.; Landau, D.P.; Wong, Y.J. Monte Carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Lett.* **1992**, *69*, 3382–3384. [CrossRef] [PubMed]
2. Rukhin, A.; Soto, J.; Nechvatal, J.; Smid, M.; Barker, E.; Leigh, S.; Levenson, M.; Vangel, M.; Banks, D.; Heckert, A.; et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*; National Institute of Standards and Technology: Gaithersburg, MA, USA, 2010; Special Publication 800-22.
3. L'Ecuyer, P.; Simard, R. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.* **2007**, *33*, 22.
4. Marsaglia, G. DIEHARD: A Battery of Tests of Randomness. 1996. Available online: http://stat.fsu.edu/pub/diehard/ (accessed on 10 January 2026).
5. Random Bitstream Tester. Available online: https://mzsoltmolnar.github.io/random-bitstream-tester/ (accessed on 10 January 2026).
6. Knuth, D.E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*; Addison-Wesley: Boston, MA, USA, 1969.
7. Gentle, J.E. *Random Number Generation and Monte Carlo Methods*, 2nd ed.; Springer: New York, NY, USA, 2003.
8. Johnston, D. *Random Number Generators-Principles and Practice: A Guide for Engineers and Programmers*; Walter de Gruyter: Berlin, Germany, 2018.
9. Luengo, E.A. A brief and understandable guide to pseudorandom number generators and specific models for security. *Stat. Surv.* **2022**, *16*, 137–181. [CrossRef]
10. Bhattacharjee, K.; Das, S. A search for good pseudorandom number generators: Survey and empirical studies. *Comput. Sci. Rev.* **2022**, *45*, 100471. [CrossRef]
11. Hull, T.E.; Dobell, A.R. Mixed Congruential Random Number Generators for Binary Machines. *J. ACM* **1964**, *11*, 31–40. [CrossRef]
12. Lehmer, D.H. Mathematical methods in large-scale computing units. In Proceedings of the Second Symposium on Large-Scale Digital Calculating Machinery, Cambridge, MA, USA, 13–16 September 1949.
13. O'Neill, M.E. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation, HMC-CS-2014-0905. *ACM Trans. Math. Softw.* **2014**, *204*, 46.

14. Matsumoto, M.; Nishimura, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Model. Comput. Simul.* **1998**, *8*, 3–30.
15. Golomb, W. *Shift Register Sequences*; Holden-Day: San Francisco, CA, USA, 1967.
16. Marsaglia, G. Xorshift RNGs. *J. Stat. Softw.* **2003**, *8*, 1–6. [CrossRef]
17. Blackman, D.; Vigna, S. Scrambled Linear Pseudorandom Number Generators. *ACM Trans. Math. Softw.* **2021**, *47*, 36. [CrossRef]
18. Salmon, J.K.; Moraes, M.A.; Dror, R.O.; Shaw, D.E. Parallel random numbers: As easy as 1, 2, 3. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, (SC'11), Seattle, WA, USA, 12–18 November 2011.
19. Jenkins, R.J., Jr. ISAAC. *Lect. Notes Comput. Sci.* **1996**, *1039*, 41–49. [CrossRef] [PubMed]
20. Nir, Y.; Langley, A. ChaCha20 and Poly1305 for IETF Proto cols; Internet Research Task Force (IRTF), Request for Comments: 8439. Available online: https://www.rfc-editor.org/info/rfc8439/ (accessed on 10 January 2026).
21. Aumasson, J.P. On the Pseudorandom Generator ISAAC; FHNW, 5210 Windisch, Switzerland. Available online: https://eprint.iacr.org/2006/438.pdf (accessed on 16 February 2026).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.