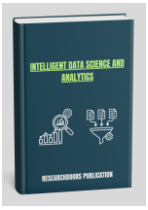
RESEARCH DOORS

Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8



A Novel Hybrid Pseudo Random Number Generator – Switch Shift PRNG

Md Mahbubur Rahman[1]
[1]Faculty of Computer Science and Engineering, Patuakhali Science and Technology University Patuakhali, Bangladesh
Corresponding Email: mahbub.cse@pstu.ac.bd

Md Atikqur Rahaman[2]
[2]Faculty of Computer Science and Engineering, Patuakhali Science and Technology University, Patuakhali, Bangladesh
Email: atik.csit@pstu.ac.bd

Chinmay Bepery[3]
[3]Faculty of Computer Science and Engineering, Patuakhali Science and Technology University Patuakhali, Bangladesh
Email: chinmay.cse@pstu.ac.bd

Hind Biswas[4]
[4]Faculty of Computer Science and Engineering, Patuakhali Science and Technology University Patuakhali, Bangladesh
Email: ug2302016@cse.pstu.ac.bd

ABSTRACT

Pseudo Random Number Generators (PRNGs) are good at generating number sequences that only look random but are in fact deterministic and periodic. Hybrid Pseudo Random Number Generators (HPRNGs) address some of these limitations by using time-based seeding with a modified Linear Congruential Generator (LCG). This approach improves upon the deterministic nature of the generator but fails to address the problem of periodicity and dependency on a single seed. This study addresses the deterministic nature and periodicity of PRNGs by proposing an improved HPRNG model, making it more suitable for various applications without losing significant performance.

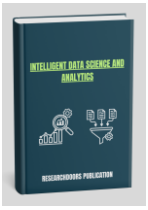
Keywords:

Pseudo Random Number Generator, Reseeding, Entropy, Kolmogorov-Smirnov, Chi-square, LCG, Hybrid PRNG

1 INTRODUCTION

Many real-life applications in the modern world, such as science, art, statistics, cryptography, gaming, gambling, and other fields, require random numbers. This demands a reliable system to generate satisfactory and seemingly random data every time. This requirement led to the development of various methods for generating random numbers, and thus Random Number Generators (RNGs) are employed. RNGs are systems, processes, or algorithms that can generate a sequence of numbers or symbols that cannot be reasonably predicted better than by random chance. There are principal classes of generators i.e. True Random Number Generators (TRNGs) sometimes known as Hardware Random-Number Generators (HRNGs), and Pseudo Random

Received: 14th April, 2025 | Accepted: 22nd June, 2025 | Published: 24th June, 2025

RESEARCH DOORS
Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8


Number Generators (PRNGs) [1]. A new class of RNG is now being studied that can have features from both TRNGs and PRNGs which is called the Hybrid Pseudo Random Number Generator (HPRNG) and is the focus of this study.

The best source for these random numbers is the TRNGs [2] which offers real randomness. They provide numbers that are truly random which ensures high security as there is no periodicity, is non-deterministic, and has high entropy. To achieve this true randomness, they extract the dynamic entropy from random and microscopic fluctuations in physical processes (e.g. thermal noise, shot noise, avalanches, clock drift, jitter, atmospheric noise, external electromagnetics, quantum phenomena, etc.). Generating numbers based on physical phenomena is naturally slower than algorithmic methods. Not only that, but it also requires specialized hardware, which may not always be available or portable. This also makes them costly and hard to implement or integrate. Since the randomness depends on the external source, performance may degrade under environmental conditions that disrupt the physical processes.

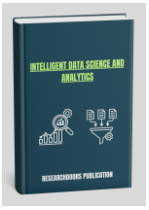
On the other hand, PRNGs e.g. LCG, Mersenne Twister (MT19937) [3], Xorshift Generators, Linear Feedback Shift Register (LFSR) [4], etc. are algorithmic solutions that are way faster than their counterparts (TRNGs). They can also generate long sequences of numbers with predictable performance, can be implemented across various platforms without reliance on hardware, and have lower resource requirements because the work is based on a predefined mathematical formula that works on a predefined input which is called the seed. But this means that the same seed and algorithm will always produce an identical sequence of numbers. At the same time, the mathematical formula leads to periodicity, where the output eventually repeats, potentially compromising security if the seed is discovered or the sequence is rendered useless once it repeats if it has lower periodicity than that is required.

Although periodicity and determinism may not be problems in most applications and, sometimes are even useful for reproducibility, they could pose a significant risk for those applications that demand more security and randomness. To address these vulnerabilities, researchers have turned to Hybrid Pseudo Random Number Generators （HPRNGs） [5] that uses seeding methods such as using epoch timestamps [6] — to introduce additional entropy into the generation process. While this method addresses determinism, periodicity still afflicts it, especially when the seed time is predictable or known. Furthermore, poor or predictable seed choice (from poor timing) can cause poor randomness as well as lower periodicity. For examle, Origines et al. [6] Demonstrated that while epoch-based methods can diversify the output sequence, the underlying periodicity of traditional PRNG algorithms remains a challenge to be solved.

On the other hand, the predictability and the possibility of cycle formation might make hybrid models—even those that use time-based seeding—ineffective in high-security settings. The necessity for more innovation in this field is emphasized in recent research. One of the approaches to improve the generated random number sequence is to use Chaos Maps [7] (e.g. Logistic Map [8], Tent Map, Chebyshev Map [9] etc.). PRNGs can be made more unpredictable by taking advantage of chaotic systems' complicated, non-linear behavior and sensitive reliance on initial conditions [10]. But it also comes with its fair share of problems. The first and most obvious is its slow execution speed [11]. On top of that, they are not much research on its behavior and enough empirical data to back the randomness of Chaos Maps and thus Counter based RNGs are more preferred [12]. Research on parameter switching techniques [9], for instance, has demonstrated how dynamically changing the parameters of a chaotic system can obfuscate any deterministic structure and disturb periodic patterns. Furthermore, research on counter-mode deterministic generators [12] offers a standard by which to measure randomness, demonstrating that incorporating several entropy sources can greatly improve security. Something like these two techniques can be improvised and implemented to improve the generators without using Chaos Maps.

In this study, we propose an enhanced HPRNG algorithm that combines multiple techniques as well as uses a modified and better equation to generate better random number sequence. To provide a statistically

RESEARCH DOORS
Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8



superior random number output, this technique aims to address issues with both determinism and periodicity. Our approach provides a solid solution for security-focused applications where good randomness quality is crucial, building on well-established theoretical frameworks and current experimental findings in the field.

2 METHODOLOGY

The two major problems faced by any traditional PRNG are its deterministic nature and periodicity. Using Epoch Timestamp [6] or System Time [5] as Entropy Sources in HPRNGs helps solve deterministic nature up to an extent. But if the exact time is known somehow or leaked, the model fails at solving the deterministic nature. At the same time, the problem with periodicity remains. To solve this, we have introduced the Periodic Reseeding Model as well as some other strategies [13].

The proposed algorithm—hereafter referred to as *Switch Shift PRNG (SSRNG)*—proceeds by iterating a modified linear-congruential recurrence. It takes 4 parameters:

1. **Modulus m:** range of outputs (0 to m). It would be best to take where '*e'* is the computer's word length and subtracting 1 from it would result in the highest prime number. *(Lehmer's algorithm* [14]), the higher the period.
2. **Count n:** number of values to generate
3. **Initial multiplier a**
4. **Weight w:** Value for periodic reseeding

The algorithm of the proposed model can be described as follows:

2.1 Choosing Seed and Initialization

The algorithm uses system time or epoch time (in nanoseconds) as the initial seed/state. This ensures that each time the generator runs, it uses a new seed instead of relying on predetermined seeds.

2.2 Periodic Reseeding

This is the new model we have introduced to counteract the inherent periodicity of digital chaotic systems and other PRNGs. Since every PRNG has a period, if the algorithm is reseeded after its assumed worst-case period, the problem with periodicity is solved. To inject fresh entropy and avoid short cycles, the generator periodically updates x every T_{period} iterations, where

$$T_{\text{period}} = \text{round}(\text{w} \times \text{m})$$

and w is a small weight (e.g., 0.01 or 1%) that is the assumed worst-case period in percentile while m defines the modulus range from a typical LCG algorithm. After each T_{period}, the seed x is refreshed using the current system time. This systematic reseeding injects fresh entropy, effectively breaking any emerging cycles and ensuring the randomness remains robust over prolonged sequences.

2.3 Iteration:

For each output, the generator iterates over the formula for i from 1 to 'n', each time updating the state with shift, multiplication, and modular reduction:

a. **Check for Reseeding:** If the i (number of iterations till n) reaches a repeating value of the predetermined worst-case period (the maximum safe interval; $i \left(mod\, T_{period}\right) \equiv 0$, Refresh the state by drawing a fresh seed from the current time (e.g., $x = \lfloor \text{time\_ms} \rfloor$). This periodic reseeding injects new entropy and breaks any long-term cycle in the state.
b. **Bit-shift mixing:** Compute a left shift by 5 bits of the state multiplied by a: let $x = (x \cdot a) \ll 5$. This operation effectively mixes the bit positions by moving the high-order bits to a larger range. Then that new bit-shifted state is multiplied with the multiplicative term a.
c. **Modulus:** Then the state is multiplied by n^2 and kept under the given range by modulus operation $(\text{mod}\, m + 1)$.

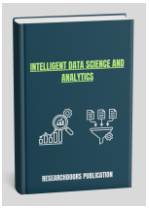
This iteration runs till the i reaches n i.e. it has generated n random numbers.

2.4 Formula

Using a shift by 5 and a multiplier mixes the bits of x *(*much like XOR shift generators do with bitwise shifts and XORs) while the modulo operation confines the result to the range [0, m) as in a standard LCG:

RESEARCH DOORS

Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8



$$x_{i+1} = (n^2 + (x_i \cdot a) \ll 5) \bmod (m+1)$$

The numbers generated in each iteration are then appended to the output sequence and returned at the end.

Figure 1 Entropy Reseeding Model

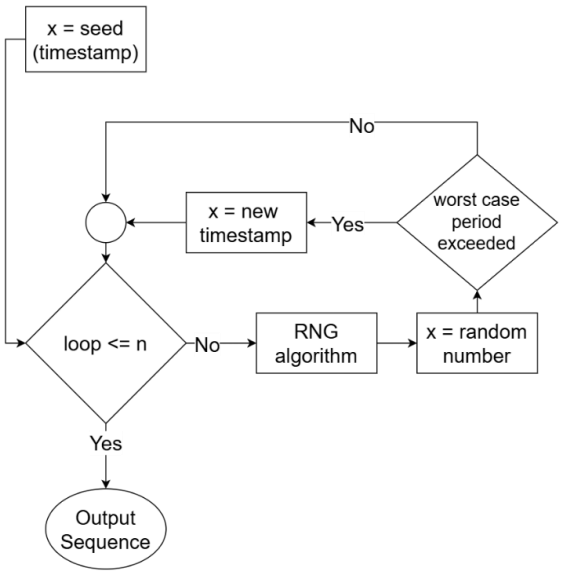
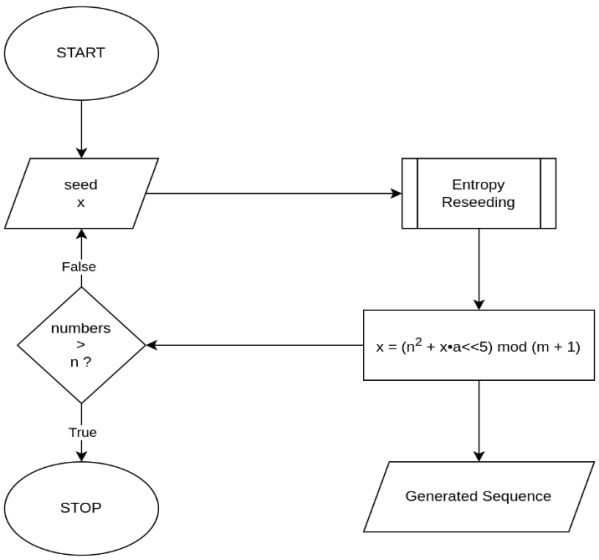


Figure 2 Switch Shift PRNG



3 FLOW CHART AND IMPLEMENTATION

The algorithm has 2 main parts i.e., Entropy Reseeding Model, and the generator formula. We have chosen these different unrelated models because:

"An acceptable random generator must combine at least two (ideally, unrelated) methods. The methods combined should evolve independently and share no state. The combination should be by simple operations that do not produce results less random than their operands." [15]

3.1 *Entropy Reseeding Model*

The first proposed model is the Entropy Reseeding Model (Figure 1). It works by taking a worst-case period, T_{period} and reseeding with new entropy seed after every T_{period} times generations.

3.2 *Main Generator*

In the main generator a modified LCG as shown in **Error! Reference source not found.** is used.

3.3 *Code Implementation*

The implementation of the algorithm is done Python v3.13.3.

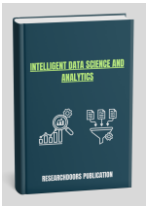
4 STATISTICAL TESTS

An extensive statistical test was performed to ensure the quality and consistency of the pseudo-random numbers produced by our proposed SSRNG. The purpose of these tests is to determine whether the output sequences follow the expected uniform distribution, which is a crucial prerequisite for simulations and applications involving cryptography.

We implemented a rigorous testing methodology in our experimental setup that involved 1000 threads, each of which ran 10 tests for each algorithm with various parameter configurations (i.e., different values of m and a). As a result, $1000 \times 10 = 10{,}000$ separate tests were produced. We created thorough statistical reports, rejection heatmaps, random number distribution profiles, and rejection rates for the tests as well as execution time taken by each algorithm.

RESEARCH DOORS

Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8



4.1 *Test Methodology*

To rigorously evaluate the randomness of our proposed Switch, Shift Pseudo-Random Number Generator (SSRNG), we employed two primary statistical tests: the Chi-Square Test [16] and the Kolmogorov – Smirnov (KS) Test [17]. These tests assess different aspects of the generated sequences to ensure they meet the criteria for uniformity and randomness.

4.2 *Chi-square Test:*

This test assesses how well the observed frequency distribution of numbers aligns with the expected uniform distribution. The Chi-square statistic is computed as:

$$X^2 = \sum_{i}^{n} \frac{(\mathrm{O}_i - \mathrm{E}_i)^2}{\mathrm{E}_i}$$

where O_i and E_i are the observed and expected frequencies for the iii-th bin, respectively. A p-value is derived from the statistic, and if the p-value is below the significance level ($\alpha = 0.05$), the test rejects the hypothesis that the distribution is uniform.

[N.B. It's important to note that the Chi-Square Test requires a sufficiently large sample size to ensure that the expected frequency in each bin is adequate, typically at least 5, to validate the test's assumptions.]

4.3 *Kolmogorov–Smirnov (KS) Test.:*

This non-parametric test compares the empirical distribution of the generated numbers to a theoretical uniform distribution. It computes the maximum deviation (D) between the two distributions and evaluates the null hypothesis H_0 that the numbers are uniformly distributed.

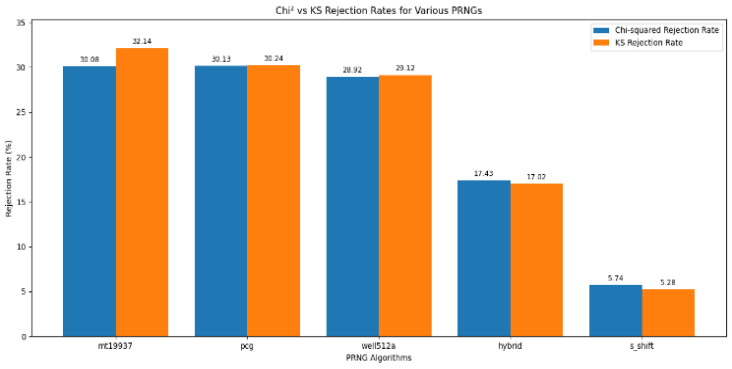
The KS Test is advantageous because it does not require binning of data and is effective for continuous distributions. However, it is more sensitive near the center of the distribution than at the tails.

4.4 *Implementation and Test Procedures*

The testing framework was implemented in Python v 3.13.3 using the SciPy (v1.14.1) library for statistical analysis. The data has been stored in SQLite database. Key steps in the process include:

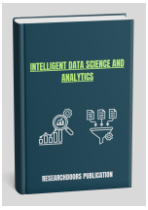
- **Normalization:** The raw output from the PRNG is normalized to the $[0,1]$ interval to facilitate comparison with the uniform distribution.
- **Test Execution:** For each algorithm, a sequence of N random numbers is generated and normalized. The Chi-square and KS tests are then applied to this sequence. The entire testing routine is executed repeatedly (100 iterations per algorithm) to compute average test statistics and execution times.
- **Data Aggregation:** Test results—including the Chi-square statistic, KS statistic, corresponding p-values, and execution times—are stored. Aggregation scripts then compile rejection counts and performance metrics across multiple threads.
- **Visualization:** Results are visualized using Python libraries such as Pandas and Seaborn, providing a clear comparative overview of the rejection rates and execution times for each algorithm.
- **Algorithms Tested:** A total of five algorithms were tested for comparison. Amongst them, SSRNG, our proposed model has been compared with the base model HPRNG (named 'hybrid' in the tests) and other popular PRNG algorithms i.e. MT19937, PCG, WELL512

Figure 3 Rejection Rates on different algorithm. Blue is the Chi2 rejection rate and Yellow is the KS Test rejection rate. The rate of rejection is on the y-axis and the algorithms on the x-axis. The algorithms from left to right respectively are: Mersenne Twister, PCG, Well512a, HPRNG and proposed SSRNG



RESEARCH DOORS

Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8



5 RESULTS

We conducted the tests on our proposed CHPRNG and base model algorithm, HPRNG. We compiled the result based on 4 parameters, i.e., Rejection Rate, Rejection Heat map, Random Number Distribution, and Test Statistics.

*[**N.B.** in the result figures, HPRNG has been used as 'hybrid' and SSRNG as 's_shift']*

5.1 Rejection Rate

Table 1 Rejection Rate

Algorithm	Chi²	KS	Total Tests
MT19937	30.08%	32.14%	10000
PCG	30.13%	30.24%	10000
WELL512	28.92%	31.12%	10000
HPRNG (base)	17.43%	17.02%	10000
SSRNG (proposed)	05.74%	05.28%	10000

The rejection rate measures the proportion of generated sequences that fail the uniformity tests at a significance level of $\alpha = 0.05$. The results of the Chi-squared and Kolmogorov – Smirnov (KS) tests for both the proposed SSRNG and the baseline HPRNG, some other algorithms (Mersenne Twister, PCG, Well512a) are summarized in **Error! Reference source not found.**. This means the lower the rejection rate, the less the model fails the uniformity test across different values of the parameters and the better the model is. The rate has been calculated based on, how many tests the model rejects the null hypothesis among all the tests.

The rejection rates for the Mersenne Twister, PCG, and Well512a turned out to be roughly around 30%. For the HPRNG algorithm, the rejection rates were 17.43% and 17.02% for the Chi-squared and KS tests, respectively. In contrast, the proposed SSRNG achieved lower rejection rates of 5.74% (Chi-squared) and 5.28% (KS). This indicates an improvement in statistical uniformity and randomness quality in the proposed algorithm compared to the baseline:

Figure 4 Rejection Heatmap 1 across different values of m. For each algorithm, rejection placed for KS and Chi-square side by

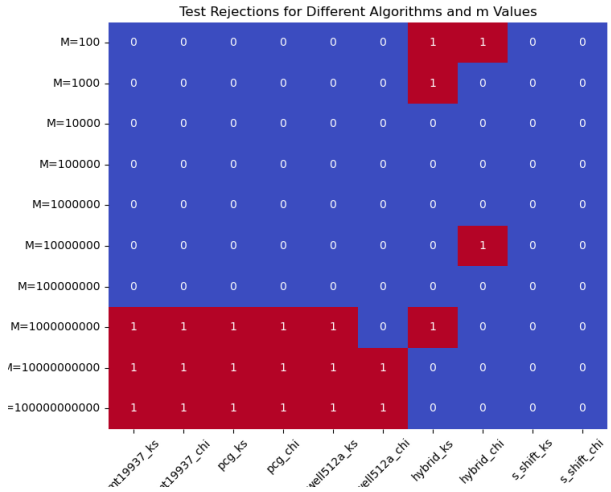
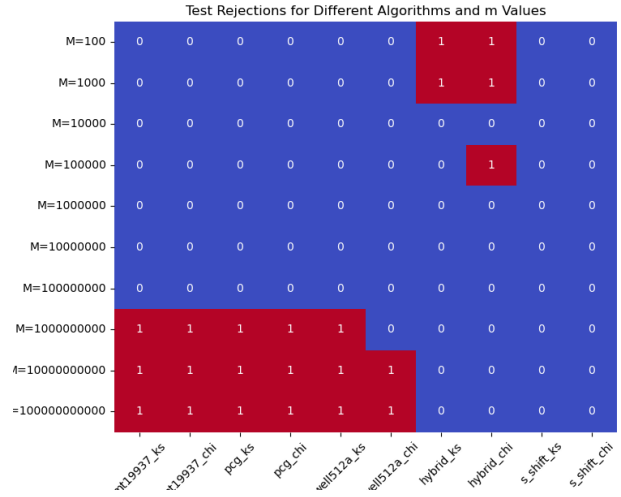


Figure 5 Rejection Heatmap 2



The results from Table 1 suggest that CHPRNG exhibits enhanced randomness properties, as evidenced by lower rejection rates across both statistical tests i.e. Approx. 68.67% and 68.98% improvement in Chi-square and KS Tests respectively compared to the base model of HPRNG and 82% (on average) better than the other models.

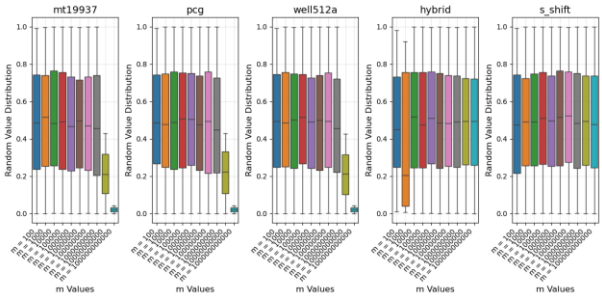
RESEARCH DOORS
Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8

5.2 *Rejection Heat map*

To further illustrate how rejection rates vary with different sample sizes, we present the 2 rejection heatmaps in **Error! Reference source not found.** and **Error! Reference source not found.**. Each row corresponds to a specific m value (ranging from 10^2 to 10^6), while each column represents a particular combination of algorithms and statistical tests (Chi-square and KS). A cell, shaded in red (=1), indicates that the corresponding test rejects the null hypothesis, whereas a blue cell (=0) signifies no rejection.

This heat map reveals some interesting characteristics of the models. It is seen that the base model HPRNG, rejects the null hypothesis at the lower values of m. Sometimes it fails at higher values too but not all of them. On the other hand, the other models reject at a higher value of m but perform well at lower values. Whereas our proposed model remains uniform for almost every value of m. So, it is more suitable for generating random number sequences over any range of numbers as the value of m is the range of the generated random numbers.

Figure 2 Generated Number Distribution for base model HPRNG



5.3 *Random Number Distribution*

The Figure 2 compares the distributions of generated random values for Mersenne Twister (mt19937), PCG, Well512a, HPRNG, and the proposed SSRNG algorithm (from left to right) across a range of sample sizes. In each subplot, the horizontal axis shows the increasing sample sizes of m (10^2 to 10^{11}) and the vertical axis shows the range of generated values, normalized to the interval $[0,1]$: the central line denotes the median of the draws, the box boundaries mark the first and third quartiles (interquartile range, IQR), and the whiskers extend to the minimum and maximum observed values.

For the Mersenne Twister (mt19937), PCG, and Well512a, the medians remain tightly clustered around 0.5, and the IQR (the height of each box) remains nearly constant for the smaller values of m. However, the distribution does not remain uniform at higher values of m and shows similar drop as the whiskers do not fully span the full interval. As for the HPRNG, it shows a noticeable skew at lower value of m: the median is slightly below 0.5 (roughly around 0.3). For $m = 10^3$, the median drops to 0.2 with a wider IQR box with whiskers not extending to 1. But as the value of m increased the median shifted towards 0.5 with a more uniform distribution.

On the other hand, the SSRNG shows way better distribution. Even at lower values of m, SSRNG's distribution is very close to uniform: its median lies almost exactly at 0.5, with an IQR only marginally wider than at higher samples. There is essentially no visible change throughout every value of m. The spans and notch widths are almost identical to the large- behavior of the other generators, showing that even small draws from SSRNG result in an ideal uniform distribution.

5.4 *Tests Statistics*

The **Error! Reference source not found.** and **Error! Reference source not found.** respectively shows the Kolmogorov–Smirnov (KS) and Chi-squared test statistics for the Mersenne Twister (colored blue), PCG (colored orange), Well512a (colored green), base HPRNG (colored blue) and the proposed SSRNG algorithm (colored purple) across various sample sizes (m). A smaller statistic in both tests indicates a closer fit to the theoretical uniform distribution.

For the KS statistics, at lower values of m, the base algorithm exhibits larger values, suggesting less uniformity that gets more uniform as the value of the m increases. As for the other models, it increases sharply for higher values of m, but it was uniform for lower values. The proposed model on the other hand showed little deviation throughout.

RESEARCH DOORS
Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8

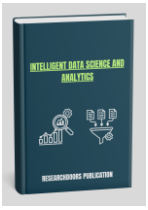
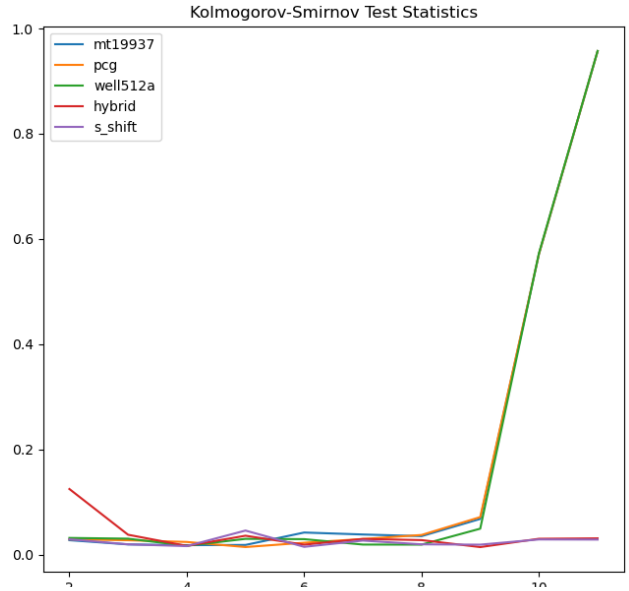
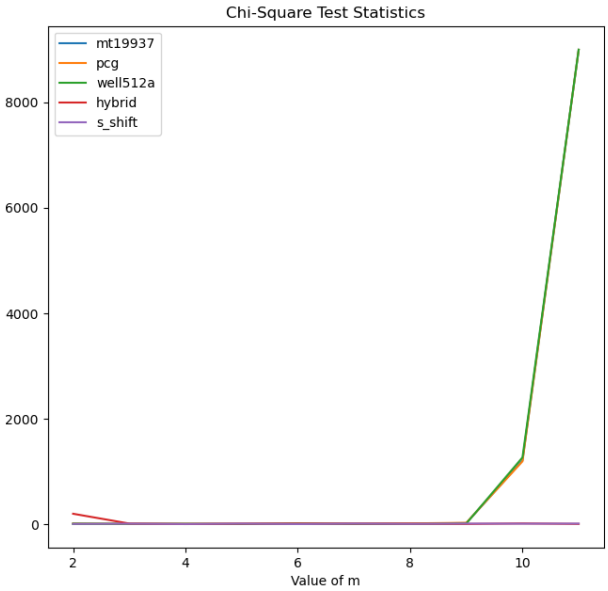


Figure 7 KS Tests Statistics



For lower values of m, the base algorithm exhibits comparatively higher Chi-squared statistics, suggesting less uniformity. As m increases, both (HPRNG and SSRNG) algorithms' test statistics decrease, reflecting an overall improvement in distribution uniformity. For the other models, they showed similar behavior to KS stats. Nevertheless, proposed algorithm consistently yields lower or comparable test statistics compared to

Figure 8 Chi-squared Tests Statistics

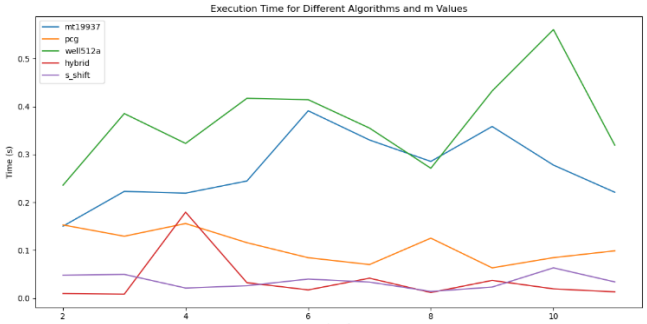


HPRNG and other models, indicating more robust performance across the examined sample sizes.

5.5 *Execution Time*

For the proposed model as well as other models, we have tested the execution time by running the generators multiple times. Here, in Figure 3, we can see how much time each model took to generate random number sequence over different values of m. The Well512a and the Mersenne Twister took the most amount of time. PCG took around half the amount. Whereas the Base model HPRNG and proposed SSRNG took the least amount of time.

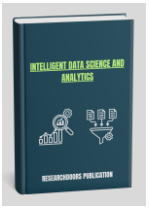
Figure 3 Execution Time: Mersenne Twister (colored blue), PCG (colored orange), Well512a (colored green), base HPRNG (colored blue) and the proposed SSRNG algorithm (colored purple)



6 OBSERVATION AND DISCUSSION

The proposed algorithm implements a modified multiplicative LCG with bit shift and entropy reseeding. This showed great improvement than well-known RNGs as well as the base model HPRNG in our rigorous testing. Through every parameter, the SSRNG showed great results. It has the lowest rejection rate throughout different ranges of m and a. When observed in the heatmap, we can see that the well-known algorithm performed uniformly great at lower values of m but constantly failed at higher ranges. The opposite happened with the HPRNG as it failed at lower values of m. We can see this more clearly when looking at the test statistics and the distribution boxplots. At a medium range of m, all the models were similarly uniform and performed well. As for our proposed model, it was not only uniform

RESEARCH DOORS
Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8



throughout every value of m and showed a huge improvement in rejection rate. Along with better performance, it was also equally fast and computationally cheap.

7 CONCLUSION

In this work, we have introduced a novel and enhanced Pseudo Random Number Generator – The Switch Shift Pseudo Random Number Generator or SSRNG – expanding or improving over the Hybrid Pseudo Random Number Generator. Our algorithm directly addresses two major limitations of the conventional PRNGs i.e. their deterministic nature and finite-period behavior. By using an epoch-based seeding mechanism at initialization and a systematic "periodic reseeding" model, SSRNG injects fresh entropy into the generator at carefully chosen intervals i.e. expected worst-case period, effectively "breaking" any potential long-term recurrence and thus solves the issue of periodicity. In every iteration, a lightweight bit-shift and multiplicative mixing step prepares a new number reduced to a value ranging between 0 to m [0, m) by modular operation. This combination of shift-multiplication-modulus along with periodic switching of seed, not only preserves shows high computational efficiency but also raises the effective period such that it exceeds the requirements of the most demanding applications.

Our extensive empirical tests —spanning distributional boxplots, Pearson chi-square tests, Kolmogorov–Smirnov metrics, and end-to-end execution-time measurements—show that SSRNG surpasses the performance of the classical generators (Mersenne Twister, PCG, and Well512a) as well as the earlier base model Hybrid PRNG (HPRNG). Our model does not fail at lower ranges of m like the HPRNG as well as at higher ranges of m like the classical generators. So, the uniformity performance is way better than that of previous models. On top of that, it is not deterministic, as even if we feed a known seed, due to it automatically switching the seed during execution based on the current time and not on the initial state/seed, it will produce different values every time. This gets rid of the deterministic nature of the PRNGs and behaves more like TRNGs. Along with solving the deterministic nature, the periodic reseeding keeps adding entropy regularly, this disrupts the periodicity of the sequence. And the result can be seen from the test data.

Beyond statistical quality, SSRNG's run-time performance is outstanding. Alongside producing better, non-deterministic, and non-periodic random number sequences, it is still computationally cheap. From the execution time graph, we can see that it takes a little more time than that of the HPRNG yet solves the two most fundamental issues of classical PRNGs.

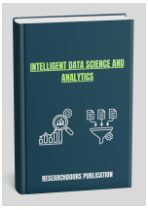
In summary, SSRNG shows a significant performance boost and advancement as a high-security, high-performance pseudorandom generator. It is truly a Hybrid Random Number Generator as it has properties of Both TRNGs (not being deterministic, reproducible, or periodic) and PRNGs (being fast, not requiring hardware, and computationally cheap). Its ability to produce statistically uniform random number sequences combined with its great performance throughout a range of different parameters, makes it highly suitable for applications ranging from cryptographic key generation to large-scale Monte Carlo simulations. At the same time, the simple design of the algorithm—merely a few shifts, multiplications, and modular arithmetic steps along with occasional time-based reseeding— makes it highly portable and easy to implement and integrate across various platforms and environments. Future research may explore further optimizations, and dynamic parameters, investigating adaptive reseeding schedules, or formally analyzing the provable lower bounds on SSRNG's cycle length under worst-case timestamp distributions. Nevertheless, even in its present form, SSRNG, not only works great with high performance with fast execution but also successfully addresses the two most fundamental characteristics of classical PRNGs i.e. deterministic nature and periodicity without sacrificing simplicity and execution time.

8 REFERENCES

[1] F. Koeune, "Pseudorandom Number Generator," in *Encyclopedia of Cryptography and Security*, H. van

RESEARCH DOORS

Intelligent Data Science and Analytics
Volume 1 Issue 2
Page no: 09-18
DoI: https://doi.org/10.63110/idsa.v1i02.8



Tilborg and S. Jajodia, Eds., Boston, MA, Springer, Boston, MA, 2011, pp. 995-996.

[2] M. Stipčević and Ç. K. Koç, "True Random Number Generators," *Open Problems in Mathematics and Computational Science*, p. 75–315, 2014.

[3] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator.," *ACM Transactions on Modeling and Computer Simulation (TOMACS).*, vol. 8, no. 1, pp. 3-30, 1998.

[4] L. Oumouss, A. Younes, A. Ahmed and A. Rguibi, "Cryptographically robust pseudo-random binary sequence generator based on the integration of LFSRs and CAs," in *2024 International Conference on Circuit, Systems and Communication (ICCSC)*, Fes, 2024.

[5] M. M. Rahman and T. Ahmed, "The Hybrid Pseudo Random Number Generator," *International Journal of Hybrid Information Technology*, vol. 9, no. 7, pp. 299-312, 2016.

[6] D. V. Origines, A. M. Sison and R. P. Medina, "A Novel Pseudo-Random Number Generator Algorithm based on Entropy Source Epoch Timestamp," in *2019 International Conference on Information and Communications Technology (ICOIACT)*, Yogyakarta, 2019.

[7] J. Amigó, "Chaos-Based Cryptography.," in *Intelligent Computing Based on Chaos*, L. Kocarev, Z. Galias and S. Lian, Eds., Berlin, Springer Berlin Heidelberg, 2009, pp. 291-313.

[8] T. H. Teo, M. Xiang, M. Elsharkawy, H. R. Leao, M. J. Andrew Calderon, J. L. Lee, S. A. Bin Rosli, H. Y. See and E. Y. Lim, "Design and Implementation of a Logistic Map-Based Pseudo-Random Number Generator on FPGA," in *2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Kuala Lumpur, 2024.

[9] İ. Öztürk and R. Kılıç, "A new pseudo random number generator based on Chebyshev maps and parameter switching," in *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, Istanbul, 2018.

[10] W. F. H. Al-shameri and M. A. Mahiub, "Some Dynamical Properties of the Family of Tent Maps," *International Journal of Mathematical Analysis*, vol. 7, no. 29, pp. 1433-1449, 2013.

[11] M. S. Kaya and K. İnce, "Benchmarking Various 1D Chaotic Maps For Lightweight Pseudo-Random Number Generation," in *2024 8th International Artificial Intelligence and Data Processing Symposium (IDAP)*, Malatya, 2024.

[12] R. I. Caran, "Comparative Analysis Between Counter Mode Deterministic Random Bit Generators and Chaos-Based Pseudo-Random Number Generators," in *2024 International Conference on Development and Application Systems (DAS)*, Suceava, 2024.

[13] I. V. Chugunkov, V. A. Gulyaev, E. A. Baranova and V. I. Chugunkov, "Method for Improving the Statistical Properties of Pseudo-random Number Generators," in *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, Saint Petersburg and Moscow, 2019.

[14] D. H. Lehmer, "Mathematical Methods in Large-Scale Computing Units," in *Proceedings of the Second Symposium on Large Scale Digital Calculating Machinery*, 1951.

[15] W. T. Vetterling, Numerical Recipes - The Art of Scientific Computing - 3rd Edition, Cambridge University Press, 1986.

[16] "Chi-Square Test," in *The Concise Encyclopedia of Statistics*, New York, Springer New York, 2008, pp. 77-79.

[17] "Kolmogorov–Smirnov Test," in *The Concise Encyclopedia of Statistics*, New York, Springer New York, 2008, pp. 283-287.