# ra_prng2 and Beyond: An Array-Based PRNG Architecture for Efficient Random Generation

Hamas A. Rahman
nexthamas95@gmail.com
State Polytechnic of Malang
Malang, East Java, Indonesia

## Abstract

In many applications, such as shuffling datasets for AI/ML training and large-scale random data processing, efficiency, high entropy, determinism, and speed are critically important. The classic Fisher–Yates array-shuffling algorithm has limitations because it relies on one RNG(random number generator) call per index, thereby creating a bottleneck in large-scale data processing.[3]

## CCS Concepts

• **Computing methodologies** → *Machine learning*; • **Mathematics of computing** → *Mathematical analysis*; • **Theory of computation** → **Pseudorandomness and derandomization**;

## Keywords

PRNG, information, mathematics, AI, ML, Fisher–Yates, entropy, Lemire's fast reduction, random, optimization, algorithm

**ACM Reference Format:**
Hamas A. Rahman. 2018. ra_prng2 and Beyond: An Array-Based PRNG Architecture for Efficient Random Generation. In *Proceedings of The 9th International Conference on Computer Science and Artificial Intelligence (CSAI 2025).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

### 1.1 Pseudo-Random Number Generators and `ra_prng`

A pseudorandom number generator (PRNG) is an important component in modern computing, widely used in many applications such as mathematical experiments, simulations, artificial intelligence (AI) training, cryptography, and beyond. The quality of a PRNG is not evaluated only by its statistical properties, but also by its security, speed, efficiency, and determinism in specific use cases.

Here, examples and explanations for most classical and common algorithms such as PCG, Xoshiro, and Mersenne Twister are designed using modular arithmetic, bit shifting, hash-based functions, or other linear formulas. They do not consider array shuffling as the core of the generator's state, rather than merely as an external application.

However, conventional data shuffling algorithms such as Fisher–Yates shuffle still rely on repeated external RNG calls to obtain random indices. This dependency causes computational overhead and reduces the overall speed of the shuffling process, especially in large-scale computing scenarios.[11]

This problem raises the question: is it possible to design a PRNG that does not require repeated reseeding and external index lookups, but instead treats shuffling itself as its main process? Such a design would be increasingly relevant in modern computational workflows, such as AI training pipelines, where even minor overhead can cause significant performance impacts.

This paper introduces `ra_prng2`, an algorithm designed using a fundamentally different form from conventional PRNG design by making permutation and array shuffling as its core mechanism of random number generation, rather than producing random values through a single functional recurrence. Using this design, array shuffling is not just an implementation of the PRNG, but a core part of the random number generation process itself. This random generation occurs simultaneously with array permutation, without external function calls or costly reseeding.

In `ra_prng2`, the internal state undergoes direct mutation and permutation at each generation step through bit rotations, internal indices shuffling, and nonlinear operations such as XOR, bit shifts, and rotates. Since the output is extracted directly from these transformations, external dataset shuffling can occur in tandem with RNG operations, eliminating overhead typically caused by external random function calls.

This design effectively reduces the separation between random number generation and data shuffling. `ra_prng2` unifies both processes in a single mechanism, providing a distinct advantage over counter-based, cipher-based, or purely mathematical generators that lack direct structural integration with array-based data operations.

### 1.2 Structure and Design of the `ra_prng2`

The structure of `ra_prng2` is built from a large internal state consisting of 2 arrays containing 256 32-bit numbers each, which continuously undergo permutation and randomization through bitwise operations and nonlinear indexing.

*1.2.1 Array State and Seed Initialization.* random generation begins by constructing two 32-bit arrays, `L[256]` and `M[256]`. Both are filled with specific constants based on predetermined values. By default, the arrays are initialized as shown below. In addition, the seed is initialized at the same time; however, in fact, seed initialization is optional because the entropy weight derived from the two lists is sufficient to create random data and the seed in the next

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CSAI 2025, Beijing, China*
© 2018 ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

iterations. In other words, the seed value can be between 0 to $2^{32} - 1$ without affecting the output quality.

```
{Initiating 256 random item per array}
FOR i := 0 TO 255 DO
  M[i] := i * 0x06a0dd9b + 0x06a0dd9b
  L[i] := i * 0x9e3779b7 + 0x9e3779b7
END FOR
seed := 1
```

**Listing 1: Seed and array initiation phase.**

These constants are presented as fixed multipliers, derived from the golden ratio value, which is also used by other PRNGs such as PCG or Xoshiro256, to reduce collisions, dead states, and short cycles by applying the following transformations repeatedly.

*1.2.2 Core Algorithm: the* `ra_core` *Loop.* Execution structure of `ra_prng2` is visualized in this Figure 1, showing the crucial flow from the beginning to the very end of the algorithm.

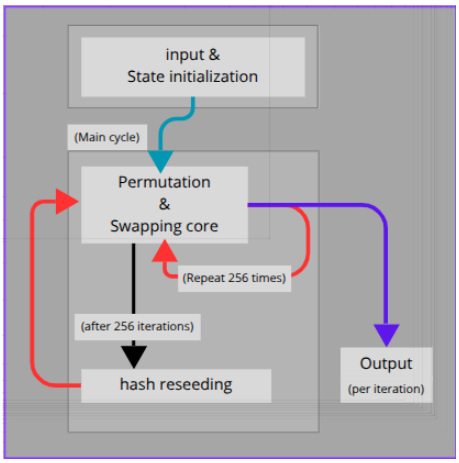


**Figure 1: Flow of ra_prng2 algorithm.**

The main function of `ra_core` executes a number of iterations that act as internal state randomizers, provide output, and supply entropy for the next iterations, as well as generate random data. Each iteration changes the shape of arrays `L` and `M` through a combination of: bitwise rotation (rot32), XOR operation, bit shifting, and index swapping between `L[i]` and `L[d]`, based on the index that has been calculated from the random data created inside the loop. Followed by XOR between all items in arrays `L` and `M` as entropy diffusion, and reseeding (cons) through a simple hash function.

```
iterations := x {any}
cons := seed {any seed}

FOR it := 0 TO iterations - 1 DO
  {Variables initiation}
  a := cons {stable entropy}
  b := it {volatile entropy}
  c := 0 {for output RNG}
  d := 0 {for random index}

  FOR i := 255 DOWN_TO 0 DO
    o := {main entropy source}

    FOR e := 0 TO 7 DO
      o := o XOR (M[(i + e) mod 256] << e)

    a := ROTL32(b XOR o, d) XOR (cons + a)
        b := ROTL32(cons + a, i) XOR (o + d)
        o := ROTL32(a XOR o, i) << 9 XOR (b >> 18)

        {this c variable is the output}
        c := ROTL32((o + c << 14) XOR (b >> 13) XOR a, b)

        {random indexing and calculating fair random index}
        {using Lemire's Fast Reduction}
        d := floor(((c * (i + 1)) mod 2**64) / 2**32)

        swap(L[i], L[d])

    {After all item shuffled, reseed the PRNG}
    {mutate the M array}
    FOR i := 0 TO 255 DO
      M[i] := M[i] XOR L[i]
  END FOR

  {update the cons}
  tmp8 := ra_hash(M)
  new_cons := 0

  FOR e := 0 TO 7 DO
    new_cons = new_cons XOR (tmp8[e] << e)
  END FOR

  cons := new_cons
END FOR
```

**Listing 2: Main part of ra_prng2 algorithm.**

*1.2.3 Technical Purpose & Significance.*

- the value of variable `c` in each loop, before being used for random indexing of array `L` through Lemire's Reduction function, is the value taken as the output of random generation. Lemire's fast modulo reduction[8] efficiently indexes $n$-bit integers to a smaller size without the common statistical bias that arises in costly modulo operations($n$ mod $i$).
- Swapping `L[i]` with `L[d]` implements internal nonlinear index permutation, making it a PRNG capable of shuffling its own internal state.
- All operations of this algorithm are branchless, speeding up, compacting execution, and optimizing CPU pipeline

## 1.3 Output Production: 32-bit numbers

In each iteration $i$, the internal loop runs backward from 255 to 1, and each loop produces one valid value (c) as 32-bit random output. Thus, one iteration can handle up to 256 random generating before reseeding for the next iteration.[5].

## 1.4 PRNG Reseeding

After 255 internal loops and array swapping are performed, this means the entire entropy of array `M` has been used up. Therefore, the next step is to update all values of `M` by performing XOR against array `L` which has been fully shuffled before, and reseed the constant value through the `ra_hash` function to maximize state space coverage. The full implementation of the hash function is provided in the appendix [5]. The new constant is constructed by XORing

all output values of the hash (tmp8) with varying bit shifts for each value, preserving bit sensitivity (avalanche effect).

### 1.5 Data-Shuffling Application

In specific applications of data randomization, random indexing can be treated with the same technique as shuffling an array L. The shuffling code is broadly written as follows; for more detailed code, see the GitHub repository.

```
c := ROTL32((o + c << 14) XOR (b >> 13) XOR a, b)

{random fair data indexing calculation}
idx := floor(((c * (count + 1)) mod 2**64) / 2**32)

{random internal state indexing calculation}
d := floor(((c * (i + 1)) mod 2**64) / 2**32)

swap(data[idx], data[count])
swap(L[i], L[d])
```

**Listing 3: ra_prng2 implementation in shuffling tool.**

## 2 Performance Evaluation, Statistical Analysis, and Periodicity

### 2.1 Testing Methodology

To ensure comparability with prior works, the performance evaluation employed four categories of tests that are commonly adopted in PRNG benchmarking.

(1) Classic randomness tests using ent, NIST STS, and Dieharder.
(2) Entropy tests and avalanche-effect measurement to assess input-output sensitivity.
(3) PRNG periodicity analysis.
(4) Shuffling performance benchmarking, and practical aplication comparison.

These test suites (NIST STS, Dieharder, ent) are widely recognized and commonly adopted in the evaluation of PRNGs, as seen in

*2.1.1 Entropy, Distribution, Dieharder, and NIST STS Results.* An initial entropy test of ra_prng2 output using the ent tool [15] on a 1 GiB sample yielded an entropy value of 7.99999 bits/byte, nearly the theoretical maximum of 8 bits/byte [13]. The byte distribution showed a mean of 127.5018, very close to the ideal random expectation of 127.5. A Monte Carlo estimation of $\pi$ produced 3.14133013 with a relative error of 0.01%. The serial correlation coefficient was measured at 0.000001, indicating negligible correlation between samples.

In the NIST Statistical Test Suite (STS) [12], across 188 subtests, ra_prng2 performed excellently: 12 subtests scored 9/10, 25 scored 9/9, 1 scored 8/9, 2 scored 8/10, and the rest achieved a perfect 10/10. No subtest fell below the recommended statistical thresholds.

Meanwhile, in Dieharder [1] testing (total of 114 subtests), ra_prng2 passed 112 subtests (PASSED) and yielded 2 WEAK results, with zero FAILED outcomes. The WEAK subtests were rgb_lagged_sum 24 (p-value = 0.00044347) and sts_serial 3 (p-value = 0.99644404). These values still lie within the extreme tails of the random p-value distribution and do not directly imply structural weaknesses.

Furthermore, the generator passed all 160 tests in BigCrush without failure, confirming the absence of detectable bias across a comprehensive range of statistical probes, including spacing tests, autocorrelation, spectral dependencies, and Hamming weight transitions.

And also, the generator was subjected to PractRand stream-based testing up to 128 GB, exhibiting no failure results. This places the generator's output quality within the statistical performance class of high-end PRNGs used in scientific computing and AI training workflows.

Although the entropy quality of this algorithm is good and it passes all tests, this does not directly mean that the algorithm is secure for cryptography. The algorithm is designed with a new structure and has not yet been tested, with no specific evaluations against various types of attacks, so it still falls into the category of general-purpose.

*2.1.2 Avalanche-Effect Test with Hamming-Distance Heatmap.* To evaluate the avalanche effect of ra_prng2, an experiment was conducted by flipping a single bit in the initial seed at positions 1 through 31. The base seed used was 0x00000001; each test seed was generated as seed $\oplus$ $(1 \ll i)$ for $1 \leq i \leq 31$, yielding 32 seed variations. This method is commonly used to analyze output diffusion in random systems [2, 16].

Each seed produced 255 sequential pseudorandom values, which were compared bitwise to the outputs from the original seed. Bit differences were computed via the Hamming distance [4] for each output pair at the same iteration. The results are visualized in Figure 2, showing the distribution of bit differences across all iterations and all flipped-bit positions. Source code is available in the benchmark folder on GitHub [6].

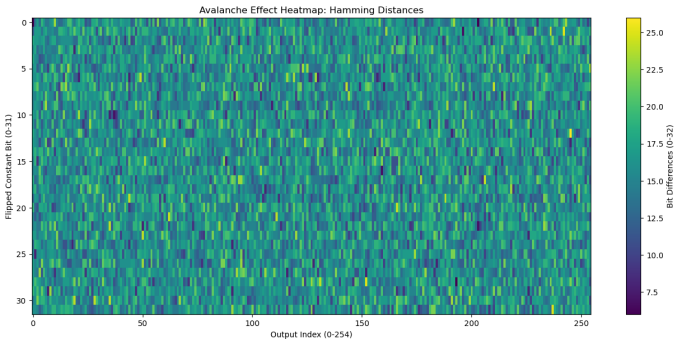


**Figure 2: Hamming-distance heatmap between the output of the original seed 0x1 and 31 single-bit–flipped seeds. The $y$-axis represents the flipped-bit position, and the $x$-axis indicates the output index. Color intensity shows the number of differing bits in each output.**

The average bit difference reached 15.98 out of 32 bits (49.9 %), with small deviation across iterations. No dominant pattern was observed in the distribution, indicating effective diffusion of input information across all output bits, akin to random distribution. This confirms that ra_prng2 exhibits a strong avalanche effect, even from minimal seed changes [14]. Additional tests with various random seeds produced similar behavior without any discernible patterns.

## 2.2 Performance and Throughput

`ra_prng2` achieved a throughput of up to 8.16 Gbit in a single iteration completed within 1.073 seconds, equivalent to 745.57 Megabytes/s. In microbenchmarks, the algorithm reached an IPC (Instructions per cycle) of 3.72, which is relatively high, without relying on the L2 cache and with minimal branch prediction. In shuffling applications, `ra_prng2` outperformed `chacha20` by up to $1,615\times$ faster in no-buffer call mode. This is due to the denser instructions of `ra_prng2`; although the operational cost is higher, the absence of I/O overhead keeps the generation process efficient.

## 2.3 Periodicity

The theoretical period can be estimated from the number of reachable PRNG states, which generally depends on the PRNG's ability to traverse every state without omission [7, 9]. In practice, perfect mixing is rare, and actual periods may be limited by local loops, unreachable states, or noninvertible processes [7, 9]. The following outlines the rough upper-bound estimation of the period before returning to the initial state.

First, the period contribution from array `M` (assuming 255 perfectly mutable 32-bit words) is

$$2^{256\times 32} = 2^{8192}.$$

The second-largest contributor is array `L`, which holds 256 words but only supports index permutation (not value mutation). Its period contribution is

$$\log_2(256!) \approx 2^{1683}.$$

Multiplying by the 32-bit seed gives an overall upper bound of

$$2^{8192} \times 2^{1683} \times 2^{32} \approx 2^{9907},$$

An astronomically large value that almost exceeds half of the Mersenne Twister's period [10]. In the extended version, `ra_prng3`, the upper bound of periodicity increases significantly up to $2^{32832}$, with only minor structural modifications and by expanding the word size to 64-bit. Consequently, both arrays become fully permutable, which drastically enlarges the periodicity range.

## 2.4 Comparison with Other Algorithms

This section compares the statistics of `ra_prng2` with several other algorithms in terms of entropy quality, periodicity, RNG speed, shuffling efficiency, and internal architectural complexity. For a fair and standardized benchmark, all experiments were run in the same environment.

- **Experimental Environment**
  All experiments were conducted on a personal computer with the following specifications:
  (1) Processor: Intel Core i3-1115G4 (2 cores, 4 threads, up to 4.10 GHz)
  (2) RAM: 8 GB DDR4
  (3) Operating System: Arch Linux (kernel 6.8.7-arch1-1)
  (4) Device: Acer Aspire 5 (model A514-54)
  The algorithm implementations were written in C and compiled with `gcc` version 15.1.1 using the `-O3` flag. Statistical tests were performed with NIST STS version 2.1.2 and Dieharder version 3.31.1.
- **Random-Number Generation Speed**
  The table below shows throughput in MB/s. The `ra_prng3` algorithm is a development version of this stable release but has not yet been further analyzed for other aspects. Measurements were taken by timing how fast the PRNG can generate 1 GiB of output.

**Table 1: PRNG Speed Benchmark**

| | Algorithm | Throughput (MB/s) |
|---|---|---|
| 1 | xoshiro256 | 3597.1 |
| 2 | PCG32 | 3196.7 |
| 3 | ra_prng3 (dev) | 1539.6 |
| 4 | Philox4x32 | 1339.8 |
| 5 | ra_prng2 | 728.0 |
| 6 | ChaCha20 | 564.8 |

- **Entropy-Test Results** Tests were performed with NIST STS and Dieharder. The following table summarizes whether each algorithm passed all subtests in the respective tools.

**Table 2: Dieharder Test Summary (114 subtests)**

| Algorithm | PASSED | WEAK | FAILED |
|---|---|---|---|
| xoshiro256 | 111 | 3 | 0 |
| PCG32 | 112 | 2 | 0 |
| ra_prng2 | 112 | 2 | 0 |
| Philox4x32 | 112 | 1 | 1 |
| ChaCha20 | 110 | 4 | 0 |

*Notes on the WEAK Flag in Dieharder.* Some algorithms show a `WEAK` flag in a small number of Dieharder subtests. It is important to note that `WEAK` does not automatically imply a critical weakness or failure in randomness quality; it typically appears when a p-value is near the extreme tail (too high or too low), which can occur even for statistically strong random data generators. A fair evaluation should consider the overall distribution of results and the number of subtests passed.

- **xoshiro256**: `WEAK` on `diehard_opso` and two variants of `rgb_lagged_sum` (p-values 0.9966 and 0.0013).
- **PCG32**: `WEAK` on two `rgb_lagged_sum` subtests (p-values > 0.998).
- **ra_prng2**: `WEAK` on `sts_serial 3` and `rgb_lagged_sum 24` (extreme p-values).
- **Philox4x32**: `FAILED` on `rgb_lagged_sum 31` (p = 9e-8) and `WEAK` on one `marsaglia_tsang_gcd` subtest.
- **ChaCha20**: `WEAK` on `parking_lot`, `3dsphere`, and two `rgb_lagged_sum` subtests (e.g., p = 0.00029 and 0.9986).

- **Theoretical Periodicity** Periodicity estimates are based on state length and the cyclical behavior of the internal system. Since `ra_prng2` and `ra_prng3` use nonlinear array structures that can be dynamically permuted and rotated, their periods greatly exceed those of scalar PRNGs.

**Table 3: Estimated Theoretical Periodicity**

| Algorithm | Theoretical Period |
|---|---|
| xoshiro256 | $2^{256}$ |
| PCG32 | $2^{64}$ |
| ra_prng3 (dev) | $2^{32832}$ |
| Philox4x32 | $2^{128}$ |
| ra_prng2 | $2^{9907}$ |
| ChaCha20 | $2^{512}$ |

- **Shuffling Application Speed** The RNG implementations for shuffling are available in the GitHub repository. Benchmarks were performed by shuffling a dataset of 1,000,000 token items using the Fisher–Yates algorithm. Each experiment was run 30 times and measured with `perf stat -r 30`. Two experimental settings were used: (1) *baseline per-call*, where each random index is generated by invoking the RNG function once without buffering, and (2) *buffered*, where PRNGs are allowed to amortize random number generation by using a small block of precomputed random values. The `ra_prng` family is only evaluated in the per-call setting since its design integrates shuffling into its state transition, making additional buffering redundant. Furthermore, this algorithm designed to swap the array in every iteration, this cause buffering not relevant to be implemented in this application.

**Table 4: Shuffle time for 1,000,000 items without buffering (per-call).**

| Algorithm | Shuffle Time (s) | Deviation | IPC |
|---|---|---|---|
| xoshiro256 | 0.06101 | 2.81% | 1.60 |
| PCG32 | 0.05992 | 1.60% | 1.59 |
| ra_prng3 (dev) | 0.12159 | 1.36% | 3.47 |
| Philox4x32 | 0.06622 | 3.75% | 1.43 |
| ra_prng2 | 0.12159 | 1.36% | 3.46 |
| ChaCha20 | 0.19645 | 2.09% | 1.95 |

The results show that `ra_prng2` and `ra_prng3` require about twice as long to shuffle compared to the fastest algorithms, with an average shuffle time of around 0.12 seconds. Nevertheless, the performance gap is much smaller than in raw throughput benchmarks: for example, `ra_prng2` is about 4.98× slower than `xoshiro256` in random number generation throughput, but only about 1.99× slower in practical shuffling.

**Table 5: Shuffle time for 1,000,000 items using a 16-word buffer (amortized).**

| Algorithm | Shuffle Time (s) | Deviation | IPC |
|---|---|---|---|
| xoshiro256 | 0.06085 | 3.51% | 1.63 |
| PCG32 | 0.05931 | 2.06% | 1.64 |
| ra_prng3 (dev) | - | - | - |
| Philox4x32 | 0.06249 | 4.43% | 1.53 |
| ra_prng2 | - | - | - |
| ChaCha20 | 0.06452 | 2.73% | 1.64 |

When buffering is introduced, the performance of most PRNGs improves only marginally, typically by less than 0.1 ms. However, `ChaCha20` shows a significant benefit: its shuffle time drops from 0.196 seconds to 0.064 seconds, nearly a threefold speedup. For the `ra_prng` family, buffering was not applied since their design already integrates shuffling within the internal state transition.

*2.4.1 Discussion and Future Work.* An interesting observation is that although `ra_prng3` achieves nearly twice the raw throughput of `ra_prng2`, both algorithms produce nearly identical shuffling performance. This is due to significant bit waste in Lemire's fast reduction step, where large portions of the generated bits cannot be used for indexing. This suggests a future direction for optimization: redesigning output usage to minimize discarded bits.

The buffering results suggest that efficient entropy pooling could be explored in future versions to maximize bit usage without unnecessary state growth. Interestingly, `ra_prng2` and `ra_prng3` also show the highest IPC ($\approx 3.5$) with the lowest deviation ($\approx 1.3\%$), suggesting that their structural design executes instructions densely and consistently across runs. While this does not directly translate into faster shuffling time, it indicates that the algorithm may exploit parallel execution paths efficiently, which could correlate with robustness in structural complexity.

## 3 Conclusion

This paper introduces the `ra_prng2` algorithm, a random number generator algorithm that places the data shuffling process as the core of its design rather than treating it as one of the RNG applications. This approach can be leveraged for critical needs today, one of which is dataset shuffling for AI before training. In addition, this algorithm is deterministic, making it reproducible.

In the shuffle-speed test for 1,000,000 data items (Table 4), the `ra_prng2` algorithm outperforms `chacha20` but remains slower than `PCG32` and `xoshiro256`. Nevertheless, `ra_prng2` offers design flexibility and integrates RNG and shuffling into a single process, eliminating external PRNG call overhead.

Furthermore, the algorithm's theoretical period is extremely large—up to $2^{9907}$ in its second version—and continues to be extended in `ra_prng3` toward $2^{32832}$, without sacrificing statistical quality, and improving its random generating throughput by almost double. Its deterministic design also makes it highly suitable for AI/ML applications requiring reproducibility and precise control over randomization.

Future work will focus on improving shuffling efficiency through better bit utilization, exploring other techniques to maximize output usage, and analyzing the relationship between high IPC utilization and structural robustness. These directions point toward a family of algorithms with strong potential for use in AI/ML, embedded systems, and deterministic security–randomization frameworks.

## References


[1] Robert G. Brown. 2004. Dieharder: A Random Number Test Suite. https://webhome.phy.duke.edu/~rgb/General/dieharder.php. Accessed: 2025-06-16.

[2] Joan Daemen and Vincent Rijmen. 2002. *The Design of Rijndael: AES—The Advanced Encryption Standard.* Springer.

[3] R. Durstenfeld. 1964. Algorithm 235: Random permutation. *Commun. ACM* 7, 7 (1964), 420.

[4] Richard W. Hamming. 1950. Error Detecting and Error Correcting Codes. *Bell System Technical Journal* 29, 2 (1950), 147–160.

[5] hamzy hams. 2025. ra_prng2: Novel Pseudorandom Number Generator Structure with Internal State Mutation. https://github.com/hamzy-hams/ra_prng/tree/main/ra_prng2/src. Accessed: 2025-06-16.

[6] hamzy hams. 2025. ra_prng2: Novel Pseudorandom Number Generator Structure with Internal State Mutation. https://github.com/hamzy-hams/ra_prng/blob/main/ra_prng2/benchmark/avalanche_effect_analysis.py. Accessed: 2025-09-25.

[7] Donald E. Knuth. 1997. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* (3rd ed.). Addison-Wesley.

[8] Daniel Lemire. 2019. Fast Random Integer Generation in an Interval. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 29, 1 (2019), 3:1–3:12. doi:10.1145/3230636

[9] Pierre L'Ecuyer. 1990. A note on the period of combined multiple recursive random number generators. *Statistics & Probability Letters* 9, 4 (1990), 335–339.

[10] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. 3–30 pages.

[11] M. Penschuck et al. 2023. Engineering Shared-Memory Parallel Shuffling to Generate Random .... *arXiv preprint arXiv:2302.03317* (2023).

[12] Andrew Rukhin et al. 2001. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* Technical Report Special Publication 800-22 Revision 1a. National Institute of Standards and Technology (NIST). https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final

[13] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27, 3 (1948), 379–423.

[14] Douglas R. Stinson. 2005. *Cryptography: Theory and Practice* (3rd ed.). Chapman and Hall/CRC.

[15] John Walker. 2008. ENT: A Pseudorandom Number Sequence Test Program. https://www.fourmilab.ch/random/. Accessed: 2025-06-16.

[16] Alan F. Webster and Stafford E. Tavares. 1986. On the Design of S-Boxes. *Advances in Cryptology — CRYPTO '85* (1986), 523–534. doi:10.1007/3-540-39799-X_38