RA_PRNG2 AND BEYOND: A NOVEL LIGHTWEIGHT PRNG ARCHITECTURE FOR HIGH-PERFORMANCE RANDOM DATA GENERATION

Hamas A. Rahman
Independent Researcher
Banjarnegara
nexthamas95@gmail.com

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 call per index, thereby creating a bottleneck in large-scale data processing [1].

This paper introduces `ra_prng2`, a lightweight PRNG architecture designed to produce up to 255 32-bit numbers in a single iteration. Unlike conventional scalar or counter-based generators, `ra_prng2` is built upon two internal 32×256 arrays that function both as state and as an integrated entropy reservoir. Through continuous in-place mutation using bitwise rotation, XOR, shifting, and nonlinear indexing, the generator effectively fuses RNG and array-shuffling logic into a unified process. This structural novelty, where randomization is achieved via state-internal permutation, offers a fundamentally different approach to high-throughput random number generation.

The relatively small state size (2 KB) allows the entire structure to reside in L1 cache, enabling highly efficient execution with IPC consistently above 3 [2]. Statistical evaluations confirm that `ra_prng2` passes all NIST STS and Dieharder entropy tests, exhibits 49.9% bit-flip sensitivity (avalanche effect), and maintains a near-ideal Shannon entropy profile. Its estimated theoretical period reaches 2^{9907}, and has been extended to 2^{32832} in the next-generation version `ra_prng3`, which also achieves significantly higher throughput.

The advantages of this PRNG lie in its ability to generate large batches of decorrelated outputs without reseeding overhead, while maintaining deterministic reproducibility. These properties make it especially suited for modern AI/ML pipelines involving data augmentation, parallel preprocessing, and reproducible randomized simulations.

Keywords cryptography · PRNG · information · mathematics · AI · ML · Fisher–Yates · entropy · Lemire's fast reduction

1 Introduction

1.1 Pseudo-Random Number Generators and `ra_prng`

A pseudo-random number generator (PRNG) is a fundamental component in modern computing, used across domains ranging from cryptography and simulation to statistical experiments and machine learning (AI/ML). In the AI/ML realm, PRNGs are employed to shuffle datasets before training, randomly initialize neural network weights, and perform data augmentation. Therefore, a PRNG's quality must be evaluated not only by its entropy and distribution but also by its speed, determinism (reproducibility), and efficiency at large data scales.

However, conventional array-shuffling algorithms like the Fisher–Yates shuffle have structural limitations: they depend on separate RNG calls for each element being shuffled. When applied to large-scale datasets—as in training modern AI models—repeated RNG function calls become a bottleneck in the data pipeline [3].

This paper introduces `ra_prng2`, a PRNG algorithm designed with a fundamentally different architecture: a PRNG built around array permutation as its core internal state, rather than merely a single-number generator function. With this design, array shuffling is not just an application of the PRNG but an integral part of the RNG process itself. The shuffling operation can be performed on-the-fly during RNGing without external function calls.

In `ra_prng2`, the internal arrays undergo mutation and permutation directly in each RNG iteration through rotations, index shuffling, and nonlinear operations such as XOR, shifts, and rotates. Since the RNG output is taken directly from these internal transformations, data shuffling can occur simultaneously with RNG generation, eliminating the overhead of separate calls and making this PRNG well suited for data-shuffling applications.

This design implicitly blurs the boundary between the RNG function and the shuffling operation. In `ra_prng2`, both run in a single unified process. This is the main advantage over conventional counter-based RNGs, ciphers, or simple mathematical generators that do not functionally integrate with the data array's structure.

1.2 Structure and Design of the `ra_prng2` Algorithm

The `ra_prng2` algorithm is built upon two internal arrays (L and M) of dimensions 256×32-bit, which are continuously mutated and shuffled via bit-level operations and nonlinear indexing.

1.2.1 State List and Seed Initialization

RNGing begins by constructing two 32-bit arrays, `L[256]` and `M[256]`. Both are populated with specific constants according to predetermined values. By default, the lists are initialized as shown below. Additionally, the seed is initialized, although it is optional because the entropy contained in these two lists is sufficient to generate a fresh seed in subsequent iterations. In other words, the seed may be any value from 0 to $2^{32} - 1$ without affecting output quality.

```
for (uint8_t i; i < 255; ++i) {
M[i] = i * 0x06a0dd9b + 0x06a0dd9b;
L[i] = i * 0x9e3779b7 + 0x9e3779b7;
}
seed = 1;
```

Listing 1: Initialization of lists M and L.

These constants serve as fixed multipliers—analogous to the golden-ratio constants in other PRNGs such as PCG or Xoshiro—to avoid collisions, dead states, and short cycles when repeated transformations are applied.

1.2.2 Core Algorithm: the `ra_core` Loop

The main PRNG function (`ra_core`) executes a number of iterations that both shuffle the internal state and provide entropy for subsequent iterations, as well as generate the RNG output. Each iteration modifies L and M through a combination of:

- bitwise rotation (`rot32`)
- XOR operations
- bit shifting
- index swapping between `L[i]` and `L[d]` based on the RNG-calculated index
- XORing across all entries in M and L to diffuse entropy
- a simple re-hash to produce the next seed (`cons`)

```
for (size_t it = 0; it < iterations; ++it) {
    // Variables initiation
    uint32_t  a = cons; //entrpoy source
    uint32_t  b = it; // entropy source
    uint32_t  c = 0; // output RNG
    uint32_t  d = 0; // random index


    // Permutation step
    // RNGing process
    // Entropy list swappping
```

```
    for (uint32_t i = 255; i > 0; --i) {
        uint32_t o = 0; // entropy source

        for (uint8_t e = 0; e < 8; ++e) {
            o ^= (M[(uint8_t)(i + e)] << e);
        }

        a = (rot32(b ^ o, d) ^ (cons + a));
        b = (rot32(cons + a, i) ^ (o + d));
        o = (rot32(a ^ o, i) << 9 ^ (b >> 18));

        // c is the output
        c = rot32((o + c << 14) ^ (b >> 13) ^ a, b);

        // Lemire's Fast Reduction for random indexing
        d = (uint32_t)(((uint64_t)c * (i + 1)) >> 32);

        // Internal state swapping
        o = L[i];
        L[i] = L[d];
        L[d] = o;
    }

    // Mixing internal state
    for (uint16_t i = 0; i < 256; ++i) {
        M[i] ^= L[i];
    }

    // Hash to next seed (cons)
    ra_hash(M, tmp8);

    // Build next cons from bits of tmp8
    uint32_t new_cons = 0;
    for (uint8_t e = 0; e < 8; ++e) {
        new_cons ^= tmp8[e] << e;
    }
    cons = new_cons;
```

Listing 2: Main part of ra$_p rng2 algoritm$.

1.2.3 Technical Purpose & Significance

- The variable `c` in each iteration—before being used for indexing into L via Lemire Reduction—serves directly as a 32-bit random output. Lemire's fast modulo reduction [4] efficiently maps an n-bit integer into a smaller range without the statistical bias common in standard modulo operations ($n \bmod i$).
- Swapping `L[i]` with `L[d]` implements a nonlinear internal-index permutation, making this a self-mutating RNG.
- All operations are branchless, accelerating execution and optimizing CPU pipeline friendliness.

1.3 Output Production: 255 32-bit Numbers

Iteration `i` runs from 255 down to 1, and each iteration yields one valid `c` value as a 32-bit random output. Thus, a single call to `ra_core` can produce up to 255 random 32-bit numbers without additional RNG function calls. If a specific count of numbers is needed, the loop may be stopped early, as documented on GitHub [5].

1.4 PRNG Reseeding

After generating 255 values of `c` and XORing arrays M and L, the final step is to perform a simple nonlinear hashing of M's contents via the `ra_hash` function to produce a new seed (`cons`). This ensures that each subsequent iteration is neither cyclic nor repetitive, maximizing coverage of the state space. The full hash implementation is provided in the appendix [6]. The new seed is constructed by XORing the hash output (`tmp8`) with different bit shifts per value, preserving single-bit sensitivity (avalanche effect).

1.5 Data-Shuffling Application

In dedicated data-shuffling applications, random indexing can be performed using the same technique as sampling random indices for list `L`. The shuffler code is implemented accordingly; for more detailed code, see the GitHub repository.

```
c = rot32(((o + c) << 14) ^ (b >> 13) ^ a, b) & 0xFFFFFFFF;
// random index calculation using Lemire's fast reduction
uint32_t idx = (uint32_t)((uint64_t)c * (count + 1) >> 32);

// random L index for swapping internal state
d = (uint32_t)((uint64_t)c * (i + 1) >> 32);

// swap in scrambled_tokens and in table L
uint32_t tmp_tok = scrambled_tokens[count];
scrambled_tokens[count] = scrambled_tokens[idx];
scrambled_tokens[idx] = tmp_tok;

o = L[i];
L[i] = L[d];
L[d] = o;
```

Listing 3: ra_prng2 implementation in shuffling tool.

2 Performance Evaluation, Statistical Analysis, and Periodicity

2.1 Testing Methodology

Performance evaluation was conducted using four types of tests: (1) classic randomness tests using ENT, NIST STS, and Dieharder; (2) entropy and avalanche-effect measurements to assess input–output sensitivity; (3) throughput benchmarking and IPC efficiency; and (4) PRNG periodicity analysis.

2.1.1 Entropy, Distribution, Dieharder, and NIST STS Results

An initial entropy test of `ra_prng2` output using the `ent` tool [7] on a 1 GiB sample yielded an entropy value of 7.99999 bits/byte, nearly the theoretical maximum of 8 bits/byte [8]. The byte distribution showed a mean of 127.5018, very close to the ideal random expectation of 127.5. A Monte Carlo estimation of π 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) [9], 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 [10] 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.

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 $\texttt{seed} \oplus (1 \ll i)$ for $1 \leq i \leq 31$, yielding 32 seed variations. This method is commonly used to analyze output diffusion in random systems [11, 12].

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 [13] for each output pair at the same iteration. The results are visualized in Figure 1, showing the distribution of bit differences across all iterations and all flipped-bit positions. Source code is available in the `benchmark` folder on GitHub [14].

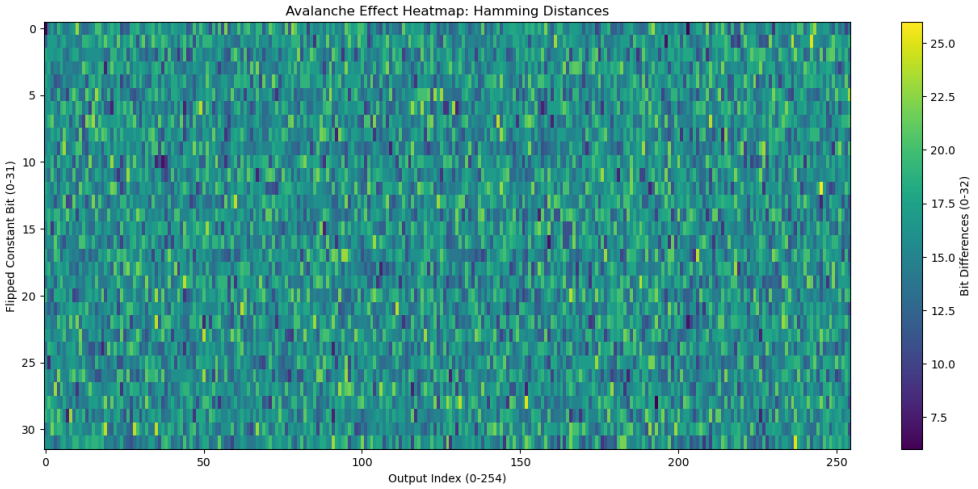

Figure 1: Hamming-distance heatmap between the output of the original seed `0x1` and 31 single-bit–flipped seeds. The y-axis represents the flipped-bit position, and the x-axis indicates the output index. Color intensity shows the number of differing bits in each output.

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 [15]. Additional tests with various random seeds produced similar behavior without any discernible patterns.

2.2 Performance and Throughput

`ra_prng2` achieves a throughput of 8,160 bits per iteration by outputting 255 32-bit values in a single function call. In microbenchmarks, the algorithm recorded an IPC (instructions per cycle) of 3.72, without relying on L2 cache or branch prediction. In shuffling applications, `ra_prng2` outperformed `/dev/urandom` by up to 10×, since `/dev/urandom` is bottlenecked by repeated RNG system calls and I/O overhead [16].

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 [17, 18]. In practice, perfect mixing is rare, and actual periods may be limited by local loops, unreachable states, or noninvertible processes [17, 18]. The following outlines the rough upper-bound estimation of the period before returning to the initial state.

First, the period contribution from array `M` (255 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

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

an astronomically large value that even exceeds the Mersenne Twister's period [19].

2.4 Comparison with Other Algorithms

This paper 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 the PRNG while generating 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
7	/dev/urandom	402.4

- **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
dev_urandom	109	5	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 RNGs. 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).
  - **/dev/urandom**: Several `WEAK` flags, notably in `craps`, `sts_serial`, and `kstest`, indicating system entropy variations.

- **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}
dev_urandom	non-periodic

- **Shuffling Application Speed**

  The RNG implementation for shuffling is available in the GitHub repository. Benchmarking was performed by shuffling a dataset of 100,000 token items.

Table 4: Shuffle Time for 100,000 Items

Algorithm	Shuffle Time (s)
xoshiro256	0.006282048
PCG32	0.005142026
ra_prng3 (dev)	–
Philox4x32	0.007818317
ra_prng2	0.011274038
ChaCha20	0.008287736
dev_urandom	0.144705242

3 Conclusion

This paper has introduced `ra_prng2`, a lightweight, array-permutation–based PRNG algorithm designed to produce large volumes of random output efficiently, deterministically, and in parallelizable fashion. One of its main advantages is the ability to generate high entropy with a customizable internal structure, while delivering performance competitive with popular PRNG algorithms.

In the shuffle-speed test for 100,000 data items (Table 4), `ra_prng2` recorded a time of 0.0112 s, outperforming `/dev/urandom` (0.1447 s), though still behind `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 it's RNGing speed 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 increasing throughput, enhancing internal structure flexibility, and exploring deeper entropy models. Periodicity improvements remain a clear target, without compromising efficiency or output quality. With a modular approach and high parallelism potential, the `ra_prng` family holds great promise for widespread use in AI/ML, embedded systems, and deterministic security–randomization systems.

References

[1] R. Durstenfeld, "Algorithm 235: Random permutation," *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.

[2] Dejazzer Lecture, "Advanced cache optimizations: Fast hit time in l1 caches," Lecture PDF, 2023.

[3] M. Penschuck *et al.*, "Engineering shared-memory parallel shuffling to generate random ...," *arXiv preprint arXiv:2302.03317*, 2023.

[4] D. Lemire, "Fast random integer generation in an interval," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 29, no. 1, pp. 3:1–3:12, 2019.

[5] hamzy hams, "ra_prng2: Novel pseudorandom number generator structure with internal state mutation," https://github.com/hamzy-hams/ra_prng/ra_prng2/example, 2025, accessed: 2025-06-16.

[6] ——, "ra_prng2: Novel pseudorandom number generator structure with internal state mutation," https://github.com/hamzy-hams/ra_prng/ra_prng2/src, 2025, accessed: 2025-06-16.

[7] J. Walker, "Ent: A pseudorandom number sequence test program," https://www.fourmilab.ch/random/, 2008, accessed: 2025-06-16.

[8] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[9] A. Rukhin *et al.*, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards and Technology (NIST), Tech. Rep. Special Publication 800-22 Revision 1a, 2001. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final

[10] R. G. Brown, "Dieharder: A random number test suite," https://webhome.phy.duke.edu/~rgb/General/dieharder.php, 2004, accessed: 2025-06-16.

[11] A. F. Webster and S. E. Tavares, "On the design of s-boxes," *Advances in Cryptology — CRYPTO '85*, pp. 523–534, 1986.

[12] J. Daemen and V. Rijmen, *The Design of Rijndael: AES—The Advanced Encryption Standard*. Springer, 2002.

[13] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[14] hamzy hams, "ra_prng2: Novel pseudorandom number generator structure with internal state mutation," https://github.com/hamzy-hams/ra_prng/comparation/scrambling_speed, 2025, accessed: 2025-06-16.

[15] D. R. Stinson, *Cryptography: Theory and Practice*, 3rd ed. Chapman and Hall/CRC, 2005.

[16] hamzy hams, "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, 2025, accessed: 2025-06-16.

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

[18] P. L'Ecuyer, "A note on the period of combined multiple recursive random number generators," *Statistics & Probability Letters*, vol. 9, no. 4, pp. 335–339, 1990.

[19] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator," pp. 3–30, 1998.