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.[4] This paper introduces `ra_prng2`, a PRNG architecture based on array internal state permutation, rather than conventional scalar or counter-based generation. It employs two internal (32×256) arrays that act both as state and as an entropy reservoir. Through continuous in-place mutation using bitwise rotation, XOR, shifting, and nonlinear indexing, the generator unifies RNG and array-shuffling logic into a single process. This structural approach eliminates overhead in shuffling scenarios, achieves competitive throughput with a large theoretical period, quality, efficiency, and represents a fundamentally different direction for high-performance 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 [3]. Statistical evaluations confirm that `ra_prng2` passes all NIST STS, Dieharder, PractRand, and BigCrush entropy tests, exhibits 49.9% bit-flip sensitivity (avalanche effect), and maintains a near-ideal Shannon entropy profile. Its estimated theoretical period reaches approximately $2^{4969.65}$, and has been extended to $2^{16447.65}$ 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 ensuring deterministic reproducibility. These properties make it especially suited for modern AI/ML pipelines involving data augmentation, parallel preprocessing, and reproducible simulations.

CCS Concepts

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

CSAI 2025, Beijing, China

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Keywords

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

ACM Reference Format:
Hamas A. Rahman. 2025. 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, 8 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.[13]

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.

$$\mathbb{S} \;=\; \mathbb{Z}_{2^{32}}^{256} \times \mathbb{Z}_{2^{32}}^{256} \times \mathbb{Z}_{2^{32}} \times \mathbb{Z}_{2^{32}}$$

where we denote an internal state at discrete step t as

$$S_t = \big(L_t,\ M_t,\ it_t, cons_t\big), \qquad L_t, M_t \in \mathbb{Z}_{2^{32}}^{256}\ it_t, cons_t \in \mathbb{Z}_{2^{32}};.$$

The algorithm defines a deterministic state transition operator

$$F : \mathbb{S} \to \mathbb{S}, \qquad S_{t+1} = F(S_t),$$

which can be decomposed conceptually as a composition of primitive operators

$$F = \text{Mutate} \circ \text{Permute} \circ \text{Reseed},$$

where Permute denotes index permutations (swaps) on L, Mutate denotes bitwise nonlinear transformations on array elements, and Reseed computes the next scalar `cons` via a hash-like function on M.

$$\text{rotl}_{32}(x, r) : \big((x \ll r) \vee (x \gg (32 - r))\big) \bmod 2^{32},$$

and we denote bitwise XOR by $\oplus$, left shift by $\ll$ and right shift by $\gg$.

For a outer iteration t, and for each inner index $i \in \{255, \ldots, 0\}$, the algorithm computes intermediate variables a, b, c, d, o. The value of o from M is formalized as

$$o_{(0)} := 0, \qquad o_{(e+1)} := o_{(e)} \oplus \big(M_t[(i{+}e) \bmod 256] \ll e\big), \quad e = 0, \ldots, 7,$$

and finally $o := o_{(8)}$.

The core transforms per index i are then:

$$\begin{aligned} a' &= \text{rotl}_{32}\left(b \oplus o,\ d\right) \oplus \left(cons_t + a\right), \\ b' &= \text{rotl}_{32}\left(cons_t + a',\ i\right) \oplus \left(o + d\right), \\ o' &= \left(\text{rotl}_{32}(a' \oplus o,\ i) \ll 9\right) \oplus (b' \gg 18), \end{aligned}$$

and the output register update

$$c' \;=\; \text{rotl}_{32}\Big(\big(o' + (c \ll 14)\big) \oplus (b' \gg 13) \oplus a',\ b'\Big).$$

The produced output for this inner step is

$$O_{t,i} \;=\; c'.$$

The random index d is computed using Lemire's fast reduction; for 64-bit intermediate multiplication we write:

$$d \;=\; \left\lfloor \frac{\big((O_{t,i} \bmod 2^{64}) \times (i+1)\big) \bmod 2^{64}}{2^{32}} \right\rfloor$$

which in implementation reduces to the commonly used 64bit to 32bit reduction as described in [10].

Swapping the array entries at indices i and d implements the permutation:

$$L_t^{'} \;=\; \text{swap}\big(L_t,\ i,\ d\big), \quad \text{i.e.} \quad L_t^{'}[i] = L_t[d],\ L_t^{'}[d] = L_t[i],$$

and all other indices unchanged. This defines one elementary permutation applied inside Permute.

After the inner loop completes (for all i), the mutation of M is an elementwise XOR with the permuted L:

$$M_{t+1}[j] \;=\; M_t[j] \oplus L_t^{'}[j], \qquad j = 0, \ldots, 255.$$

The reseed scalar is computed by a hash-like reduction ra_hash : $\mathbb{Z}_{2^{32}}^{256} \to \mathbb{Z}_{2^{32}}^{8}$,

$$\text{tmp8} \;=\; \text{ra\_hash}\big(M_{t+1}\big), \qquad new\_cons \;=\; \bigoplus_{e=0}^{7} \big(\text{tmp8}[e] \ll e\big),$$

and the next scalar state is

$$cons_{t+1} \;=\; new\_cons.$$

Thus one full outer transition can be summarized compactly by

$$S_{t+1} \;=\; F(S_t) \;=\; \big(L_t^{'},\ M_{t+1},\ cons_{t+1}\big),$$

and the observable output stream is the sequence $\{O_{t,i}\}$ over iterations t and inner indices i.

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

Algorithm 1. Seed and array initiation phase.

```
Initialize: seed ← any positive integer (1 in default)
for i ← 0 to 255 do
    L[i] ← i × 0x06a0dd9b + 0x06a0dd9b;
    M[i] ← i × 0x9e3779b7 + 0x9e3779b7;
    *[r]Initiating 256 random item per array
```

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.

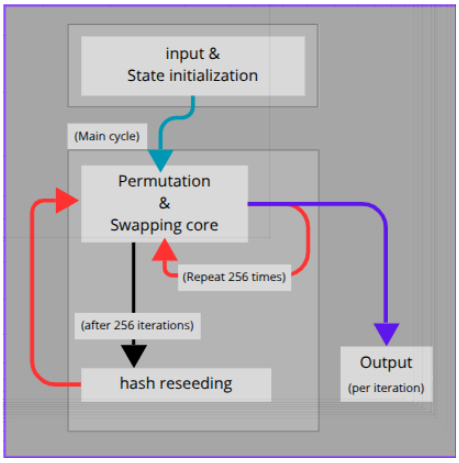

Figure 1: Flow of ra_prng2 algorithm.

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 XOR and bit shift-based hash function.

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[10] 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.[6].

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 [6]. 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).

Algorithm 2. Main function of `ra_prng2`

Initialize: cons ← seed; it ← 0; iterations ← any positive integer

for it ← 0 **to** *iterations*−1 **do**
  // Initialize variables
  $a \leftarrow$ cons ; // stable entropy
  $b \leftarrow$ it ; // volatile entropy
  $c \leftarrow 0$; // output accumulator
  $d \leftarrow 0$; // random index buffer
  for $i \leftarrow 255$ 0 **do**
    $o \leftarrow 0$; // main entropy source
    for $e \leftarrow 0$ **to** 7 **do**
      $o \leftarrow o \oplus (M[(i+e) \bmod 256] \ll e)$
    $a \leftarrow \mathrm{ROTL}_{32}(b \oplus o, d) \oplus (\mathrm{cons} + a)$;
    $b \leftarrow \mathrm{ROTL}_{32}(\mathrm{cons} + a, i) \oplus (o + d)$;
    $o \leftarrow (\mathrm{ROTL}_{32}(a \oplus o, i) \ll 9) \oplus (b \gg 18)$;
    // Generate RNG output
    $c \leftarrow \mathrm{ROTL}_{32}(((o + (c \ll 14)) \oplus (b \gg 13) \oplus a), b)$;
    // Get random index using Lemire's fast reduction
    $d \leftarrow \lfloor ((c \times (i+1)) \bmod 2^{64})/2^{32} \rfloor$;
    swap($L[i], L[d]$);
  // Mutate array M after full shuffle
  for $i \leftarrow 0$ **to** 255 **do**
    $M[i] \leftarrow M[i] \oplus L[i]$
  // Regenerate constant cons
  tmp8 ← ra_hash(M) ;
  new_cons ← 0 ;
  for $e \leftarrow 0$ **to** 7 **do**
    new_cons ← new_cons ⊕ (tmp8[e] ≪ e)
  cons ← new_cons ;

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.

Algorithm 3. `ra_prng2` implementation in shuffling tool

// Generate RNG output
$c \leftarrow \mathrm{ROTL}_{32}(((o + (c \ll 14)) \oplus (b \gg 13) \oplus a), b)$;
idx ← $\lfloor ((c \times (\text{count+1})) \bmod 2^{64})/2^{32} \rfloor$;
// Get random index for the data
$d \leftarrow \lfloor ((c \times (i+1)) \bmod 2^{64})/2^{32} \rfloor$;
// Get random index for internal state
swap($L[i], L[d]$);
swap(data[idx], data[count]);

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 [17] on a 1 GiB sample yielded an entropy value of 7.99999 bits/byte, nearly the theoretical maximum of 8 bits/byte [15]. 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.

Across all 188 subtests in the NIST Statistical Test Suite (STS) [14], the mean p-value was 0.503 ± 0.287, consistent with the uniform [0,1] expectation for ideal random sources. The overall pass rate reached 99.4%, with no subtest below the recommended significance threshold of $p > 0.0001$. This show stable diffusion of entropy throughout the generator's state.

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 or weak 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.

This behavior aligns with the theoretical expectation of non-invertible random mappings, where each iteration acts as a many-to-one function $F : \mathcal{S} \to \mathcal{S}$ that diffuses entropy uniformly over the state space. The uniform p-value distribution across all test suites thus provides empirical evidence of the mapping's chaotic diffusion properties, consistent with the random mapping theorem [8] and the predicted cycle length $\lambda \approx 0.7824\sqrt{n}$.

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 Analysis with Multi-Seed Perturbation Models. To comprehensively evaluate the avalanche characteristics of `ra_prng2`, three complementary experiments were conducted under distinct seed perturbation models. These models assess how minor variations in the initial seed propagate through the generator's iterative process, an essential property of diffusion in well-designed pseudorandom systems [2, 18]. Each experiment begins with the base seed `0x00000001` and generates 255 sequential 32-bit outputs. At every iteration, the bitwise difference between the modified-seed output and the baseline output is measured using the Hamming distance [5], representing the number of differing bits between corresponding outputs.

Three distinct perturbation schemes were employed:

(1) **Bit-Flipping Avalanche Test.** Each modified seed is derived by flipping a single bit in the base seed:

$$\mathsf{seed}_i = \mathsf{seed}_0 \oplus (1 \ll i), \quad 0 \le i < 32$$

This configuration measures orthogonal sensitivity across bit positions, indicating how individual input bits influence overall diffusion. In classical avalanche analysis, strong diffusion is indicated when single-bit perturbations produce a 50% difference in output bits in under 3 iterations [16].

(2) **Incremental-Seed Sensitivity Test.** The initial seed increases linearly as:

$$\mathsf{seed}_i = \mathsf{seed}_0 + (i + 1), \quad 0 \le i < 32$$

This test evaluates continuity resistance, determining whether nearby seeds yield decorrelated outputs. High sensitivity under incremental variation indicates the generator's resistance to local correlation traps, common weaknesses in low-entropy PRNG designs.

(3) **Power-of-Two Seed Test.** Each seed is initialized as a single active bit at distinct positions:

$$\mathsf{seed}_i = 1 \ll (i + 1), \quad 0 \le i < 32$$

This isolates the contribution of each bit position as an independent entropy vector, revealing uniformity in how each seed bit contributes to the evolving state.

For each of the three experiments above, the per-iteration average of the Hamming distance across all 32 perturbed seeds was computed. The resulting comparison is shown in Fig. 2, where the horizontal axis represents the internal iteration index, and the vertical axis shows the average number of differing bits (0–32) between baseline and perturbed outputs. Each line represents one perturbation model: Bit-Flipping (blue), Incremental (orange), and Power-of-Two (green).

Figure 2: Average bit divergence per iteration across three seed perturbation models: Bit-Flipping, Incremental, and Power-of-Two. The ideal diffusion reference (50% bit difference ≈ 16/32 bits) is achieved after the first few iterations.

As illustrated in Fig. 2, all three perturbation models rapidly converge toward a stable diffusion level near 16 out of 32 bits (approximately 50%), consistent with the expected behavior of a high-entropy generator. The Bit-Flipping test reveals a minor delay for upper-bit flips (bit 5–32) within the first iteration, after which diffusion becomes uniform, suggesting transient local dependency in early state transitions. Conversely, both the Incremental and Power-of-Two seed tests show immediate and uniform diffusion from the first iteration, confirming that even simple arithmetic or isolated-bit seed perturbations lead to strongly decorrelated outputs. Together, these results confirm that `ra_prng2` achieves robust avalanche behavior with rapid entropy propagation and no detectable bias across different seed perturbation domains. Full source code for these tests is available in the `benchmark` folder on GitHub [7].

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 Analysis

A central property of any pseudo-random number generator (PRNG) is its *periodicity*, the number of iterations before the generator returns to a previously visited internal state. 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. In practice, perfect mixing is rare, and actual periods may be limited by local loops, unreachable states, or noninvertible processes [9, 11]. The following outlines the rough upper-bound estimation of the period before returning to the initial state. Formally, let $\mathcal{S}$ denote the finite state space of the generator, and let $F : \mathcal{S} \to \mathcal{S}$ represent the deterministic update operator defining one iteration of the generator. For discrete time steps $t \in \mathbb{N}$, the internal state evolves as

$$S_{t+1} = F(S_t),$$

with the orbit of an initial state S_0 defined as

$$O(S_0) = \{S_0, F(S_0), F^2(S_0), \dots\}.$$

Since $\mathcal{S}$ is finite, every trajectory is eventually periodic: there exist integers $\mu \geq 0$ and $\lambda \geq 1$ such that

$$F^{\mu+\lambda}(S_0) = F^{\mu}(S_0),$$

where λ is called the *period length*. The goal of this section is to characterize λ for `ra_prng2`.

Non-invertible structure. Unlike classical generators such as Mersenne Twister or xoshiro, which are designed as bijective or semi-bijective linear recurrences over $\mathbb{F}_2$, the core update operator of `ra_prng2` is *non-invertible*. This non-invertibility arises from the internal step that generates the constant term `cons` through a hash-like reduction:

$$\texttt{cons}_{t+1} = H(M_t),$$

where H combines multiple words of M_t using nonlinear XOR-aggregation. Because many distinct input states can yield the same 32-bit output, H acts as a many-to-one mapping, compressing information and breaking strict reversibility. Consequently, multiple distinct states can converge to the same future orbit, reducing the maximal achievable period compared to a perfectly invertible mapping.

Contributions to state size. To estimate upper bounds on possible cycle lengths we separate the state into two classes of contributions: (i) components whose bits are *fully mutable* under the update (i.e., each bit may change through XOR/add/rotate operations), and (ii) components that are *permutative only* (their values are rearranged but not bitwise mutated).

In our `ra_prng2` instantiation the fully mutable components are the array `M` (256 words of 32 bits each), the per-iteration constant `cons` (32 bits), and the iteration counter `it` (32 bits). Hence the mutable bit count is

$$B_{\text{mut}} = 256 \cdot 32 + 32 + 32 = 8256.$$

The array `L` contributes only by permutations of its 256 entries; therefore its combinatorial contribution is 256!, which satisfies (using Stirling's approximation)

$$\log_2(256!) \approx 1684.05.$$

Raw upper bound. Multiplying the two contributions yields a raw upper bound on the number of distinct internal states:

$$|\mathcal{S}| \approx 2^{8256} \times 256! \approx 2^{8256} \times 2^{1684.05} \approx 2^{9940}.$$

We stress that this is a structural upper bound on the number of possible states (assuming full variability of the mutable components) and *not* a proven period.

Heuristic theoretical periodicity for non-invertible mappings. Because `ra_prng2` includes a hash-like reduction when generating `cons` (a many-to-one step that compresses multiple words into a 32-bit value), the overall update map behaves non-invertibly. This non-invertibility implies that the mapping $F : \mathcal{S}_t \to \mathcal{S}_{t+1}$ cannot

be represented as a bijection, since multiple distinct internal states can converge to the same successor.

Therefore, its long-term dynamics are more accurately modeled as a random many-to-one mapping rather than a deterministic permutation. Under this assumption, the random mapping theory provides an appropriate heuristic framework for estimating the expected cycle length and the statistical behavior of orbit convergence.[8] We approximate the theoretical periodicity using this heuristic,

$$\lambda \approx 0.7824\sqrt{|\mathcal{S}|} \approx 0.7824 \cdot 2^{9940/2} = 0.7824 \cdot 2^{4970}.$$

Equivalently,

$$\lambda \approx 2^{4970+\log_2(0.7824)} \approx 2^{4969.65},$$

which should be interpreted as a statistical/heuristic estimate rather than a rigorous period proof.

Comparison to MT19937 and ra_prng3. For reference, the Mersenne Twister MT19937 has period $2^{19937} - 1$ [12]. Under the calculations above the raw upper bound exponent of ra_prng2 ($\approx$ 9940) and the heuristic period exponent ($\approx$ 4969.65) are both *smaller* than the MT19937 exponent (19937). Thus MT19937's period remains substantially larger than the heuristic theoretical periodicity for ra_prng2.

If the design is extended to a 64-bit word size and all state words become fully mutable (the `ra_prng3` variant), then the total bit count becomes

$$B_3 = 2 \cdot 256 \cdot 64 + 64 + 64 = 32896,$$

giving a raw upper bound 2^{32896} and a corresponding heuristic many-to-one theoretical periodicity

$$\lambda_3 \approx 0.7824 \cdot 2^{32896/2} = 0.7824 \cdot 2^{16448} \approx 2^{16447.65}.$$

Note that while the raw bound (2^{32896}) exceeds the MT19937 exponent, the random-mapping heuristic (which takes the square root) yields an exponent ($\approx$ 16447.65) that remains below 19937.

Caveat. All expressions above are estimations: the raw bounds count combinatorial possibilities, while the $\sqrt{n}$ heuristic applies when the effective mapping on reachable states behaves like a random many-to-one map. A rigorous proof of exact period for the full-size generator is computationally infeasible.

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)
xoshiro256	3597.1
PCG32	3196.7
ra_prng3 (dev)	1539.6
Philox4x32	1339.8
ra_prng2	728.0
ChaCha20	564.8

2.4.1 Shuffling Performance Evaluation. 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 executed 30 times and measured using `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 amortize random number generation by using a small block of precomputed random values.
The `ra_prng` family was evaluated only in the per-call setting, since its design intrinsically integrates shuffling within its state transition, making additional buffering redundant. In each iteration, `ra_prng2` performs internal swaps across the array state, rendering buffering irrelevant for this application.
Results for both configurations are shown in Table 2. In the per-call configuration, `ra_prng2` and its successor `ra_prng3` require approximately twice the time to shuffle compared to the fastest algorithms, with an average shuffle time of around 0.12 seconds. Nevertheless, this performance gap is considerably smaller than in raw throughput benchmarks; for instance, `ra_prng2` is about 4.98× slower than `xoshiro256` in random number generation throughput, but only 1.99× slower in practical shuffling.
When buffering is introduced, most conventional PRNGs show only marginal improvements, typically less than 0.1 ms. However, `ChaCha20` exhibits a notable speedup: its shuffle time decreases from 0.196 seconds to 0.064 seconds, nearly a threefold improvement. As expected, the `ra_prng` family was not included in this comparison due to its integrated design, which inherently performs array mutation during each state transition rather than relying on separate random calls.

Table 2: Shuffle time for 1,000,000 items under both configurations (per-call and buffered), "Dev" refers to standard deviation across 30 runs.

	Per-call (No Buffering)			Buffered (16-word)		
Algorithm	Times	Dev	IPC	Times	Dev	IPC
xoshiro256	0.06101	2.81%	1.60	0.06085	3.51%	1.63
PCG32	0.05992	1.60%	1.59	0.05931	2.06%	1.64
ra_prng3	0.12159	1.36%	3.47	-	-	-
Philox4x32	0.06622	3.75%	1.43	0.06249	4.43%	1.53
ra_prng2	0.12159	1.36%	3.46	-	-	-
ChaCha20	0.19645	2.09%	1.95	0.06452	2.73%	1.64

- **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 3: Dieharder Test Summary (114 subtests) and Estimated Theoretical Periodicity

Algorithm	PASSED	WEAK	FAILED	Theoretical Period
xoshiro256	111	3	0	2^{256}
PCG32	112	2	0	2^{64}
ra_prng3	-	-	-	$2^{16447.65}$
Philox4x32	112	1	1	2^{128}
ra_prng2	112	2	0	$2^{4969.65}$
ChaCha20	110	4	0	2^{512}

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.

2.4.2 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.

This high IPC value with low and consistent deviation indicates that instructions are not only executed densely and efficiently, but the operations are also executed consistently. This efficiency largely stems from the branchless design, which effectively minimizes pipeline overhead and prediction errors in modern CPUs. In other words, the algorithm achieves good performance with high operations per CPU cycle, demonstrating optimal utilization of hardware execution units. Future research will also be directed towards simplifying bitwise operations while maintaining instruction density and the branchless nature to preserve the high IPC. Consequently, a lighter structure will yield higher throughput and shuffling performance, enabling it to compete effectively with other classic PRNGs.

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 non buffered shuffle-speed test for 1,000,000 data items (Table 2), 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 $2^{4969.65}$ in its second version, and continues to be extended in `ra_prng3` toward $2^{16447.65}$, 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] Dejazzer Lecture. 2023. Advanced Cache Optimizations: Fast Hit Time in L1 Caches. Lecture PDF.

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

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

[6] 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.

[7] 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.

[8] B. Harris. 1960. Probability Distributions Related to Random Mappings. *The Annals of Mathematical Statistics* 31, 4 (1960), 1045–1062. doi:10.1214/aoms/1177705677

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

[10] 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

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

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

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

[14] 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

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

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

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

[18] 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