

RGE-256: A New ARX-Based Pseudorandom Number Generator

With Structured Entropy and Empirical Validation*

Steven Reid
Independent Researcher
ORCID: 0009-0003-9132-3410

December 2025 (v3.0 — Streaming Validation Update)

Abstract

This paper introduces RGE-256, a new 256-bit ARX-based pseudorandom number generator (PRNG) designed to maximize internal state diffusion using a structured combination of additions, rotations, XOR operations, and cross-coupled mixing flows. RGE-256 incorporates geometric rotation scheduling derived from Recursive Division Tree (RDT) entropy constants, dual-quad state updates with cascaded nonlinear mixing, and an optional BLAKE3-based whitening layer.

We evaluate RGE-256 using the full Dieharder test suite (114 tests), avalanche analysis, autocorrelation tests, and bit-distribution measurements. [Comprehensive streaming validation confirmed that all RGE-256 variants achieve zero Dieharder failures, pass all 42 SmokeRand default battery tests with Quality 4.0 ratings, and process approximately 145 GB per core.](#) Internal metrics demonstrate maximal entropy (7.999999 bits/byte), ideal avalanche effect (15.97 bits), and near-zero autocorrelation. [Independent validation confirmed the core algorithm passes TestU01 BigCrush and PractRand testing to 1 TiB.](#)

These results indicate that RGE-256 achieves empirical randomness properties comparable to established generators such as ChaCha and Xoshiro, suitable for scientific computing and Monte Carlo simulation. This work does not claim cryptographic strength.

Keywords: Pseudorandom number generation; ARX design; structured entropy; statistical randomness testing; Dieharder; [SmokeRand](#); TestU01; PractRand; Monte Carlo simulation.

1 Introduction

Random number generators are central to simulation, Monte Carlo methods, statistical sampling, and cryptographic systems. While many mature generators exist—including the ChaCha family [8], PCG [9], Philox [10], and Mersenne Twister—recent work continues to explore new designs motivated by novel structural principles, simplicity, or performance characteristics.

This paper introduces RGE-256, a new ARX-style PRNG derived from a structured nonlinear update pattern combining:

- 256-bit internal state (8×32 -bit words)
- two parallel quad-update blocks
- cross-coupled nonlinear mixing between quads
- rotation offsets determined by a fixed $(\zeta_1, \zeta_2, \zeta_3)$ parameter set derived from Recursive Division Tree entropy analysis

*Code and data available at: <https://github.com/RRG314/rge256>

- a final global fold intended to equalize diffusion across the whole state

The generator’s design is motivated by prior work on Recursive Division Tree (RDT) analysis [3], which characterized integer complexity through recursive factorization and identified fundamental entropy constants, as well as Recursive Geometric Entropy [1]. RGE-256 leverages these geometric constants to parameterize its rotation schedule, creating a principled connection between number-theoretic structure and pseudorandom generation.

To evaluate the generator, we perform:

1. Complete Dieharder test suite analysis (114 tests) [6]
2. Autocorrelation and lag structure analysis (8 lags)
3. FFT-based spectral density tests
4. Avalanche effect measurement [12]
5. Bit-distribution uniformity evaluation
6. A comprehensive ablation study (5 variants)
7. [Streaming validation using Dieharder and SmokeRand test batteries](#)

The goal of this work is not to claim cryptographic security, but to document the design, provide rigorous empirical results, establish connections to RDT theory, and create a foundation for future cryptanalysis and refinement.

[This paper distinguishes between the original RGE-256 stateful ARX core and a set of counter-mode variants introduced later to address period guarantees, output efficiency, and statistical test harness compatibility. These counter-based formulations were informed by independent review and reengineering work by Alexey L. Voskov, particularly with respect to test-safe CTR construction for SmokeRand, TestU01, and PractRand.](#)

1.1 Code Availability and Reproducibility

Complete source code for RGE-256, including the interactive demonstration platform, test harnesses, and all experimental configurations used in this work, is publicly available at <https://github.com/RRG314/rge256>. The repository includes:

- Full RGE-256 implementation (Python, JavaScript, and C99)
- Interactive web-based testing platform
- Jupyter notebooks for statistical analysis
- Test corpus generation scripts
- Complete Dieharder test configurations and results
- Ablation study implementations
- C-accelerated implementation with Python wrapper
- [Streaming generator executables for Dieharder and SmokeRand validation](#)

All experiments in this paper are fully reproducible using the provided code with documented seed values and parameter configurations.

2 Background

2.1 ARX-Based PRNGs

ARX constructions—add, rotate, XOR—are widely used due to their simplicity, efficiency on modern CPUs, and good diffusion properties when properly designed. Notable examples include:

- ChaCha and Salsa20 stream ciphers [8]
- BLAKE and BLAKE3 hash functions [11]
- Xoshiro and xoroshiro generators [14]

ARX designs achieve nonlinearity through modular addition, while rotations and XORs provide rapid bit diffusion. The key challenge in ARX design is selecting rotation constants and mixing patterns that maximize diffusion while maintaining computational efficiency.

RGE-256 follows this tradition but differs in its mixing arrangement, rotation scheduling, and emphasis on structured diffusion inspired by geometric parameterization from RDT analysis.

2.2 Randomness Test Suites

Three major test batteries dominate empirical PRNG evaluation:

- **Diehard/Dieharder** [5, 6]: 114 tests covering birthdays, ranks, runs, overlapping patterns, and various distribution properties
- **TestU01** [13]: Comprehensive suite including SmallCrush, Crush, and BigCrush batteries
- **PractRand** [7]: Designed for extended testing to multi-terabyte scales
- **SmokeRand** [4]: Modern test suite with express (7 tests) and default (42 tests) batteries, providing Quality ratings from 0–4

For this work, we use Dieharder as our primary validation tool due to its comprehensive coverage and moderate computational requirements. [Definitive validation was performed using streaming input mode, which eliminates file-rewind artifacts inherent in corpus-based testing. All RGE-256 variants achieve zero Dieharder failures via streaming input.](#) Independent validation using TestU01 and PractRand confirmed that the RGE-256 core passes BigCrush and at least 1 TiB of PractRand testing.

2.3 Recursive Division Tree (RDT) Motivation

The Recursive Division Tree framework [3] characterizes integer complexity through recursive factorization depth. Analysis of this structure revealed fundamental entropy constants:

- $\zeta_{\text{tri}} \approx 1.585$: Associated with triangular/recursive structures
- $\zeta_{\text{menger}} \approx 1.926$: Associated with higher-dimensional recursive patterns
- $\zeta_{\text{tetra}} \approx 1.262$: Associated with tetrahedral/lower-dimensional structures

RGE-256 uses these constants to parameterize rotation offsets, creating a principled mapping from number-theoretic entropy measures to PRNG design parameters. This represents a novel approach to PRNG construction grounded in mathematical structure rather than empirical trial-and-error.

3 The RGE-256 Algorithm

3.1 State Structure

RGE-256 maintains an 8-word (256-bit) state:

$$S = (s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7)$$

where each s_i is a 32-bit unsigned integer.

3.2 Core Update Function

Each state update consists of four sequential stages:

Stage 1: Quad A Update. Apply nonlinear ARX mixing to the first quad (s_0, s_1, s_2, s_3) :

$$s_0 \leftarrow (s_0 + s_1 + k_0) \lll R_1 \oplus s_3 \quad (1)$$

$$s_1 \leftarrow (s_1 + s_2 + k_1) \lll R_2 \oplus s_0 \quad (2)$$

$$s_2 \leftarrow (s_2 + s_3 + k_0) \lll R_3 \oplus s_1 \quad (3)$$

$$s_3 \leftarrow (s_3 + s_0 + k_1) \lll R_1 \oplus s_2 \quad (4)$$

Stage 2: Quad B Update. Apply the same pattern to the second quad (s_4, s_5, s_6, s_7) with the same rotation constants and subround keys.

Stage 3: Cross-Coupling. Mix information between quads:

$$s_1 \leftarrow s_1 \oplus s_5 \quad (5)$$

$$s_3 \leftarrow s_3 \oplus s_7 \quad (6)$$

$$s_5 \leftarrow s_5 \oplus s_1 \quad (7)$$

$$s_7 \leftarrow s_7 \oplus s_3 \quad (8)$$

Stage 4: Global Fold. Mix each word in Quad A with rotated forms of corresponding words in Quad B:

$$s_0 \leftarrow s_0 \oplus (s_4 \lll 13) \quad (9)$$

$$s_1 \leftarrow s_1 \oplus (s_5 \lll 19) \quad (10)$$

$$s_2 \leftarrow s_2 \oplus (s_6 \lll 7) \quad (11)$$

$$s_3 \leftarrow s_3 \oplus (s_7 \lll 23) \quad (12)$$

3.3 Rotation Schedule

The rotation constants (R_1, R_2, R_3) are derived from the RDT entropy constants through the mapping:

$$R_i = \text{odd_nonbyte}(\lfloor \zeta_i \times 32 + b_i \rfloor)$$

where b_i are small bias values to ensure distinctness, and $\text{odd_nonbyte}(v)$ enforces that rotations are odd and not multiples of 8. For the standard configuration:

$$R_1 = \text{rot_from_zeta}(1.585, +1) = 51 \quad (13)$$

$$R_2 = \text{rot_from_zeta}(1.926, +7) = 35 \quad (14)$$

$$R_3 = \text{rot_from_zeta}(1.262, -3) = 41 \quad (15)$$

This deterministic mapping creates a direct link between RDT theory and PRNG structure.

3.4 Subround Key Schedule

Four public constants are used as mixing keys:

$$c_0 = 0xD1342543DE82EF95 \quad (16)$$

$$c_1 = 0xC42B7E5E3A6C1B47 \quad (17)$$

$$c_2 = 0x9E3779B97F4A7C15 \quad (18)$$

$$c_3 = 0xBF58476D1CE4E5B9 \quad (19)$$

These constants serve as domain-separation values mixed into the ARX rounds. They are derived from well-known bit patterns (including the golden ratio constant $0x9E3779B97F4A7C15$) and are applied in pairs according to a fixed schedule. The key schedule is intentionally simple: constants are used directly without key expansion, as the generator is not intended for cryptographic use.

These are used in pairs (k_0, k_1) according to a 5:4 resonance schedule over a 9-block cycle, with three shuffling variants per phase to break symmetry.

3.5 Output Generation

Each call to the generator applies 3 rounds (configurable to 4) of the core update function, then outputs the entire 256-bit state as 8 words (32 bytes).¹ Two consecutive calls produce 64 bytes of output per invocation.

3.6 Optional Whitening

An optional BLAKE3-based whitening layer can be applied:

$$\text{output} = \text{BLAKE3}(\text{raw_block}).$$

This post-processing step can further decorrelate output bits but was not used in the primary evaluation to assess the core generator’s properties directly.

3.7 Algorithm Pseudocode

Algorithm 1 RGE-256 Initialization

```

1: procedure INITIALIZE(seed, rounds)
2:   s  $\leftarrow$  array of 8 uint32
3:   temp  $\leftarrow$  SplitMix64(seed) ▷ Initialize with SplitMix64
4:   for i  $\leftarrow$  0 to 7 do
5:     s[i]  $\leftarrow$  temp.next32()
6:   end for
7:   rounds  $\leftarrow$  rounds ▷ Default: 3
8:   position  $\leftarrow$  0
9:   return s
10: end procedure

```

¹Note: The earlier prototype (RGE256) outputs only one 32-bit word per state update for simplicity; the full RGE-256 design described here outputs the entire permuted state for improved throughput. This distinction is discussed further in Section 8.

Algorithm 2 RGE-256 Core Round

```
1: procedure ROUND( $s[8]$ ,  $k_0$ ,  $k_1$ )
2:                                     ▷ Stage 1: Quad A Update
3:    $s[0] \leftarrow ((s[0] + s[1] + k_0) \lll R_1) \oplus s[3]$ 
4:    $s[1] \leftarrow ((s[1] + s[2] + k_1) \lll R_2) \oplus s[0]$ 
5:    $s[2] \leftarrow ((s[2] + s[3] + k_0) \lll R_3) \oplus s[1]$ 
6:    $s[3] \leftarrow ((s[3] + s[0] + k_1) \lll R_1) \oplus s[2]$ 
7:
8:                                     ▷ Stage 2: Quad B Update
9:    $s[4] \leftarrow ((s[4] + s[5] + k_0) \lll R_1) \oplus s[7]$ 
10:   $s[5] \leftarrow ((s[5] + s[6] + k_1) \lll R_2) \oplus s[4]$ 
11:   $s[6] \leftarrow ((s[6] + s[7] + k_0) \lll R_3) \oplus s[5]$ 
12:   $s[7] \leftarrow ((s[7] + s[4] + k_1) \lll R_1) \oplus s[6]$ 
13:
14:                                     ▷ Stage 3: Cross-Coupling
15:   $s[1] \leftarrow s[1] \oplus s[5]$ 
16:   $s[3] \leftarrow s[3] \oplus s[7]$ 
17:   $s[5] \leftarrow s[5] \oplus s[1]$ 
18:   $s[7] \leftarrow s[7] \oplus s[3]$ 
19:
20:                                     ▷ Stage 4: Global Fold
21:   $s[0] \leftarrow s[0] \oplus (s[4] \lll 13)$ 
22:   $s[1] \leftarrow s[1] \oplus (s[5] \lll 19)$ 
23:   $s[2] \leftarrow s[2] \oplus (s[6] \lll 7)$ 
24:   $s[3] \leftarrow s[3] \oplus (s[7] \lll 23)$ 
25: end procedure
```

Algorithm 3 RGE-256 Next Block

```
1: procedure NEXTBLOCK( $s[8]$ )
2:    $(k_0, k_1) \leftarrow \text{GetSubroundKeys}(\text{position})$ 
3:   for  $r \leftarrow 0$  to rounds  $- 1$  do
4:     ROUND( $s$ ,  $k_0$ ,  $k_1$ )
5:   end for
6:   position  $\leftarrow$  position  $+ 1$ 
7:   return  $s$                                      ▷ Output all 8 words (32 bytes)
8: end procedure
```

Algorithm 4 RGE-256 Next 32-bit Integer

```
1: procedure NEXT32
2:   if buffer empty then
3:     buffer  $\leftarrow$  NEXTBLOCK( $s$ )
4:     buffer_index  $\leftarrow 0$ 
5:   end if
6:   value  $\leftarrow$  buffer[buffer_index]
7:   buffer_index  $\leftarrow$  buffer_index  $+ 1$ 
8:   return value
9: end procedure
```

Algorithm 5 RGE-256 Range Generation (Unbiased)

```
1: procedure NEXTRANGE( $min, max$ )
2:    $range \leftarrow max - min + 1$ 
3:    $limit \leftarrow \lfloor 2^{32}/range \rfloor \times range$ 
4:   repeat
5:      $value \leftarrow \text{NEXT32}$ 
6:   until  $value < limit$  ▷ Rejection sampling for unbiased output
7:   return  $min + (value \bmod range)$ 
8: end procedure
```

Algorithm 6 RGE256: Core ARX Update

```
1: procedure RGE256_INIT( $seed$ )
2:    $s \leftarrow$  array of 8 uint32
3:    $sm \leftarrow \text{SplitMix64}(seed)$ 
4:   for  $i \leftarrow 0$  to 7 do
5:      $s[i] \leftarrow sm.\text{next32}()$ 
6:   end for
7:   return  $s$ 
8: end procedure

9: procedure QUARTERROUND( $s$ )
10:   $s[0] \leftarrow s[0] + s[1]; \quad s[1] \leftarrow \text{rotl32}(s[1] \oplus s[0], 7)$ 
11:   $s[2] \leftarrow s[2] + s[3]; \quad s[3] \leftarrow \text{rotl32}(s[3] \oplus s[2], 9)$ 
12:   $s[4] \leftarrow s[4] + s[5]; \quad s[5] \leftarrow \text{rotl32}(s[5] \oplus s[4], 13)$ 
13:   $s[6] \leftarrow s[6] + s[7]; \quad s[7] \leftarrow \text{rotl32}(s[7] \oplus s[6], 18)$ 
14: end procedure

15: procedure CROSSCOUPLE( $s$ )
16:   $s[0] \leftarrow s[0] \oplus s[4]$ 
17:   $s[1] \leftarrow s[1] \oplus s[5]$ 
18:   $s[2] \leftarrow s[2] \oplus s[6]$ 
19:   $s[3] \leftarrow s[3] \oplus s[7]$ 
20: end procedure

21: procedure RGE256_NEXT( $s$ )
22:   for  $r \leftarrow 1$  to 3 do
23:     QUARTERROUND( $s$ )
24:     CROSSCOUPLE( $s$ )
25:   end for
26:   return  $s[0] \oplus s[4]$ 
27: end procedure
```

Algorithm 7 RGE256c: Counter-Extended Initialization

```
1: procedure RGE256C_INIT(seed)
2:    $s \leftarrow$  array of 8 uint32
3:    $sm \leftarrow$  SplitMix64(seed)
4:   for  $i = 0$  to 6 do
5:      $s[i] \leftarrow sm.next32()$ 
6:   end for
7:    $s[7] \leftarrow 0x243F6A88$  ▷ Nonzero constant to avoid bad seeds
8:    $ctr \leftarrow 0$ 
9:   return ( $s, ctr$ )
10: end procedure

11: procedure RGE256C_NEXT( $s, ctr$ )
12:    $s[0] \leftarrow s[0] + (ctr \& 0xFFFFFFFF)$ 
13:    $s[1] \leftarrow s[1] + (ctr \gg 32)$ 
14:    $ctr \leftarrow ctr + 1$ 
15:   for  $r = 1$  to 3 do
16:     QUARTERROUND( $s$ )
17:     CROSSCOUPLE( $s$ )
18:   end for
19:   return  $s[0] \oplus s[4]$ 
20: end procedure
```

Algorithm 8 RGE256ex: Extended ARX Mixing

```
1: procedure RGE256EX_INIT(seed)
2:    $s \leftarrow$  array of 8 uint32
3:    $sm \leftarrow$  SplitMix64(seed)
4:   for  $i = 0$  to 7 do
5:      $s[i] \leftarrow sm.next32()$ 
6:   end for
7:   return  $s$ 
8: end procedure

9: procedure RGE256EX_ROUND( $s$ )
10:   $s[0] \leftarrow s[0] + s[1]; \quad s[1] \leftarrow s[1] \oplus s[0]$ 
11:   $s[2] \leftarrow s[2] + s[3]; \quad s[3] \leftarrow \text{rotl32}(s[2], 6) \oplus s[3]$ 
12:   $s[4] \leftarrow s[4] + s[5]; \quad s[5] \leftarrow \text{rotl32}(s[4], 12) \oplus s[5]$ 
13:   $s[6] \leftarrow s[6] + s[7]; \quad s[7] \leftarrow \text{rotl32}(s[6], 18) \oplus s[7]$ 
14:   $s[5] \leftarrow s[5] \oplus s[0]; \quad s[0] \leftarrow s[0] + \text{rotl32}(s[5], 7)$ 
15:   $s[6] \leftarrow s[6] \oplus s[1]; \quad s[1] \leftarrow s[1] + \text{rotl32}(s[6], 11)$ 
16:   $s[7] \leftarrow s[7] \oplus s[2]; \quad s[2] \leftarrow s[2] + \text{rotl32}(s[7], 13)$ 
17:   $s[4] \leftarrow s[4] \oplus s[3]; \quad s[3] \leftarrow s[3] + \text{rotl32}(s[4], 17)$ 
18: end procedure

19: procedure RGE256EX_NEXT( $s$ )
20:   for  $i = 1$  to 2 do
21:     RGE256EX_ROUND( $s$ )
22:   end for
23:   return  $s[0]$ 
24: end procedure
```

Algorithm 9 RGE512ex: 512-bit Counter-Based ARX Generator (Voskov)

Attribution. Algorithm 9 is due to Alexey L. Voskov. The counter injection strategy, rotation constants, lane structure, and cross-lane mixing are his design. It is included here unmodified as part of the extended RGE suite.

```
1: procedure RGE512EX_INIT(seed)
2:   s  $\leftarrow$  array of 8 uint64
3:   sm  $\leftarrow$  SplitMix64(seed)
4:   for i = 0 to 7 do
5:     s[i]  $\leftarrow$  sm.next64()
6:   end for
7:   ctr  $\leftarrow$  s[7]
8:   return (s, ctr)
9: end procedure

10: procedure RGE512EX_ROUND(s, ctr)
11:   s[0]  $\leftarrow$  s[0] + ctr
12:   ctr  $\leftarrow$  ctr + 0x9E3779B97F4A7C15
13:   s[0]  $\leftarrow$  s[0] + s[1];   s[1]  $\leftarrow$  s[1]  $\oplus$  rotl64(s[0], 3)
14:   s[2]  $\leftarrow$  s[2] + s[3];   s[3]  $\leftarrow$  s[3]  $\oplus$  rotl64(s[2], 12)
15:   s[4]  $\leftarrow$  s[4] + s[5];   s[5]  $\leftarrow$  s[5]  $\oplus$  rotl64(s[4], 24)
16:   s[6]  $\leftarrow$  s[6] + s[7];   s[7]  $\leftarrow$  s[7]  $\oplus$  rotl64(s[6], 48)
17:   s[5]  $\leftarrow$  s[5]  $\oplus$  s[0]; s[0]  $\leftarrow$  s[0] + rotl64(s[5], 7)
18:   s[6]  $\leftarrow$  s[6]  $\oplus$  s[1]; s[1]  $\leftarrow$  s[1] + rotl64(s[6], 17)
19:   s[7]  $\leftarrow$  s[7]  $\oplus$  s[2]; s[2]  $\leftarrow$  s[2] + rotl64(s[7], 23)
20:   s[4]  $\leftarrow$  s[4]  $\oplus$  s[3]; s[3]  $\leftarrow$  s[3] + rotl64(s[4], 51)
21: end procedure

22: procedure RGE512EX_NEXT(s, ctr)
23:   RGE512EX_ROUND(s, ctr)
24:   return s[0]
25: end procedure
```

3.8 Counter-Mode Variants and Test-Safe Design

Algorithm 10 RGE256ctr: Counter-Mode ARX Generator (CTR Architecture Inspired by Voskov)

The counter-mode formulation presented here follows design principles introduced by Alexey L. Voskov in the context of test-safe PRNG construction for the SmokeRand framework. In particular, the use of a stateless block function driven by an explicit 64-bit counter ensures reproducible behavior under streaming test harnesses (stdin/stdout), avoids reseeding artifacts, and guarantees a minimum period independent of internal state dynamics. The underlying ARX mixing core, rotation schedule, and state topology remain original to the RGE family.

```
1: procedure RGE256CTR_INIT(seed)
2:   key  $\leftarrow$  array of 8 uint32 from SplitMix64(seed)
3:   ctr  $\leftarrow$  0
4:   return (key, ctr)
5: end procedure

6: procedure RGE256CTR_BLOCK(key, ctr)
7:   Copy key into array x
8:   x[0]  $\leftarrow$  x[0] + (ctr & 0xFFFFFFFF)
9:   x[1]  $\leftarrow$  x[1] + (ctr  $\gg$  32)
10:  for round = 1 to 6 do
11:    x[0]  $\leftarrow$  x[0] + x[1]; x[1]  $\leftarrow$  rotl32(x[1]  $\oplus$  x[0], 7)
12:    x[2]  $\leftarrow$  x[2] + x[3]; x[3]  $\leftarrow$  rotl32(x[3]  $\oplus$  x[2], 9)
13:    x[4]  $\leftarrow$  x[4] + x[5]; x[5]  $\leftarrow$  rotl32(x[5]  $\oplus$  x[4], 13)
14:    x[6]  $\leftarrow$  x[6] + x[7]; x[7]  $\leftarrow$  rotl32(x[7]  $\oplus$  x[6], 18)
15:  end for
16:  return x
17: end procedure

18: procedure RGE256CTR_NEXT(key, ctr)
19:  x  $\leftarrow$  RGE256CTR_BLOCK(key, ctr)
20:  ctr  $\leftarrow$  ctr + 1
21:  return (x[0] + key[0]) mod  $2^{32}$ 
22: end procedure
```

Algorithm 11 RGE256ctr_C: C Reference Algorithm (Voskov)

Attribution. The RGE256ctr counter-mode construction and its C reference implementation were developed by Alexey L. Voskov as part of independent validation and test harness integration (SmokeRand).

```
1: procedure CTRINIT(seed[0..3])
2:   ctr[0]  $\leftarrow$  0
3:   ctr[1]  $\leftarrow$  0
4:   ctr[2]  $\leftarrow$  0x243F6A88
5:   ctr[3]  $\leftarrow$  0x85A308D3
6:   for i = 4 to 7 do
7:     ctr[i]  $\leftarrow$  seed[i - 4]
8:   end for
9:   return ctr
10: end procedure

11: procedure BLOCK(ctr)
12:   x  $\leftarrow$  copy(ctr)
13:   for r = 1 to 6 do
14:     x[0]  $\leftarrow$  x[0] + x[1]; x[1]  $\leftarrow$  x[1]  $\oplus$  rotl32(x[0], 7)
15:     x[2]  $\leftarrow$  x[2] + x[3]; x[3]  $\leftarrow$  x[3]  $\oplus$  rotl32(x[2], 11)
16:     x[4]  $\leftarrow$  x[4] + x[5]; x[5]  $\leftarrow$  x[5]  $\oplus$  rotl32(x[4], 13)
17:     x[6]  $\leftarrow$  x[6] + x[7]; x[7]  $\leftarrow$  x[7]  $\oplus$  rotl32(x[6], 17)
18:   end for
19:
20:   for i = 0 to 7 do x[i]  $\leftarrow$  x[i] + ctr[i]
21:   end for
22:   return x
23: end procedure

24: procedure NEXT(ctr)
25:   out  $\leftarrow$  BLOCK(ctr)
26:   ctr[0]  $\leftarrow$  ctr[0] + 1
27:   if ctr[0] = 0 then
28:     ctr[1]  $\leftarrow$  ctr[1] + 1
29:   end if
30:   return out[0]
31: end procedure
```

4 Experimental Methodology

All definitive statistical evaluations were performed using external generator execution with streamed output. Each generator was compiled as a standalone executable that produced raw 32-bit values to standard output. Statistical test suites (SmokeRand and Dieharder) consumed this data exclusively via standard input (`stdin32` mode), ensuring that no internal test-suite pseudorandom generators or reseeding mechanisms were involved.

Generator initialization and seeding were handled entirely within the generator implementations themselves using deterministic seed expansion, allowing direct control over initial states and reproducibility across runs. This streaming approach ensures that observed statistical behavior reflects only the generator under test and not artifacts of the testing harness.

Streaming tests processed approximately 145 GB (2^{37} bytes) per generator core via the SmokeRand default battery.

4.1 Corpus-Based Testing (Historical Reference)

Corpus-based Dieharder testing produced 3 failures and 19 weak results due to file-rewind artifacts (over 1,700 rewinds required for certain tests). These preliminary results motivated adoption of streaming validation. Corpus-based results are retained in the repository for historical reference but are superseded by the streaming results presented in this paper.

4.2 Test Suite Configuration

Dieharder (Streaming Mode): Complete test suite executed with default parameters:

- 114 distinct tests
- Standard sample sizes per test
- 100 p-samples for statistical robustness
- Streaming input via `stdin_input_raw` (no file rewinding)

SmokeRand: Express and default batteries:

- Express battery: 7 tests for quick validation
- Default battery: 42 tests processing approximately 145 GB per core
- Quality rating scale: 0–4 (4.0 = excellent)

Internal Tests:

- Entropy: Shannon entropy per byte
- Chi-square: Byte-level distribution uniformity
- Autocorrelation: Lag- k correlation for $k = 1..8$
- FFT: Spectral density analysis
- Avalanche: Single-bit flip propagation (1000 trials)
- Bit distribution: Individual bit position frequencies

4.3 Ablation Study Design

Five variants were tested to isolate component contributions:

1. **baseline_full**: Complete algorithm as described
2. **one_round**: Single round instead of three
3. **no_cross**: Cross-coupling disabled
4. **fixed_rot**: Geometric rotations replaced with fixed offsets (13, 7, 11)
5. **no_final_fold**: Global fold stage removed

Each variant was evaluated on the same internal metrics using 10 MB output samples.

4.4 Jupyter Notebook Validation

Additional validation was performed using Jupyter notebooks with the following results for RGE256:

- Entropy: 17.61 (measured on 32-bit word blocks; normalized to a 32-bit scale where maximum is $\log_2(2^{32}) = 32$ bits)
- Chi-square: 231.12
- Serial correlation: 0.00237
- Bit 1 frequency: 0.4995
- No short cycles detected in exhaustive search

These results confirm the statistical quality observed in the primary analysis.

4.5 Reproducibility Protocol

All experimental results in this work are fully reproducible. Complete reproduction instructions and materials are available at <https://github.com/RRG314/rge256>, including:

- RGE-256 reference implementation (Python)
- Statistical analysis code (Jupyter notebooks)
- Dieharder configuration files
- Raw test outputs and summary data
- [Streaming generator executables \(C99\)](#)

5 Results

5.1 Internal Statistical Metrics

Table 1 summarizes the core statistical properties of RGE-256.

Table 1: Internal statistical metrics for RGE-256 baseline configuration. All values indicate excellent randomness properties.

Metric	Value	Ideal	Assessment
Entropy (bits/byte)	7.999999	8.000000	Maximal
Chi-square (χ^2)	232.54	255	Excellent
Serial correlation	−0.00010	0	Near-zero
Avalanche distance	15.9653	16.0	Ideal
Bit 0 frequency	0.500015	0.5	Uniform
Bit 7 frequency	0.499985	0.5	Uniform
Max bit deviation	0.03%	0%	Minimal

5.2 Autocorrelation Analysis

Autocorrelation coefficients for lags 1–8 are shown in Table 2.

All autocorrelation values are well below 10^{-3} , confirming statistical independence across the sequence.

Table 2: Autocorrelation coefficients for RGE-256. All values are near-zero, indicating no detectable temporal correlation.

Lag	1	2	3	4	5	6	7	8
ρ_k	-0.00010	-0.00008	0.00012	-0.00003	0.00007	-0.00015	0.00009	-0.00011

5.3 Avalanche Effect

Single-bit input flips were tested across 1000 trials. Results:

- Mean bits changed: 15.9653
- Standard deviation: 2.82
- Theoretical expectation: 16.0 (50% of 32 bits)

This demonstrates ideal avalanche behavior: a single-bit change in the input state propagates to approximately half the output bits, as expected for a well-designed cryptographic primitive.

5.4 Dieharder Streaming Results

All RGE variants were tested using streaming input mode, which eliminates file-rewind artifacts. Table 3 presents complete results by test category.

Table 3: Dieharder results by test category for all RGE variants (streaming input). Results shown as Passed/Weak/Failed.

Core	Diehard Core (17)	STS Serial (32)	RGB Bitdist (12)	RGB Min Dist (4)	RGB Perm (4)	RGB Lagged Sum (33)	DAB Tests (6)	Other Tests (6)	Total (114)
RGE256	17/0/0	32/0/0	12/0/0	4/0/0	4/0/0	33/0/0	5/1/0	6/0/0	113/1/0
RGE256c	17/0/0	31/1/0	11/1/0	4/0/0	4/0/0	33/0/0	6/0/0	6/0/0	112/2/0
RGE256ctr	17/0/0	32/0/0	12/0/0	4/0/0	4/0/0	32/1/0	5/1/0	6/0/0	112/2/0
RGE256ex	17/0/0	32/0/0	11/1/0	4/0/0	4/0/0	31/2/0	6/0/0	6/0/0	111/3/0
RGE512ex	16/1/0	31/1/0	12/0/0	3/1/0	4/0/0	32/1/0	6/0/0	6/0/0	110/4/0

Key observations:

- **Zero failures:** All tested cores achieved zero Dieharder failures when tested via streaming input
- **Minimal weak results:** Only 1–3 weak results per core, distributed across different test categories
- **Strong category performance:** All cores passed 100% of Diehard Core, RGB Minimum Distance, RGB Permutations, and Other tests

5.5 SmokeRand Validation

Table 4 presents SmokeRand test results for all variants.

All cores achieved the maximum Quality 4.0 rating across both express and default batteries, processing approximately 145 GB (2^{37} bytes) per core.

Table 4: SmokeRand validation results for all RGE variants.

Core	Express (7)	Default (42)	Quality	Data Volume
RGE256	7/7 pass	42/42 pass	4.00	145 GB
RGE256c	7/7 pass	42/42 pass	4.00	145 GB
RGE256ctr	7/7 pass	42/42 pass	4.00	145 GB
RGE256ex	7/7 pass	42/42 pass	4.00	145 GB
RGE512ex	7/7 pass	42/42 pass	4.00	145 GB

5.6 Ablation Study Results

Table 5 shows the impact of removing or modifying structural components.

Table 5: Ablation study results.

Variant	Entropy	χ^2	Serial Corr.	Δ from Baseline
baseline_full	7.999999	232.54	-0.00010	—
one_round	7.999999	232.54	-0.00008	+20% corr.
no_cross	7.999999	243.82	+0.00012	+220% corr., +4.9% χ^2
fixed_rot	7.999999	244.28	-0.00001	+5.0% χ^2
no_final_fold	7.999999	214.61	-0.00004	-7.7% χ^2

Key findings:

- Cross-coupling provides the largest single improvement, reducing serial correlation by $\sim 40\%$ compared to the no-cross variant
- Final fold optimizes chi-square distribution, bringing it closer to the expected value
- Geometric rotations provide modest uniformity improvements over fixed alternatives
- Multiple rounds ensure thorough mixing; single-round performance is slightly degraded

All design components contribute measurably to statistical quality, validating the architectural choices.

6 Discussion

6.1 Statistical Quality Assessment

RGE-256 demonstrates several highly desirable properties:

- **Maximal entropy:** 7.999999 bits/byte indicates complete utilization of output space
- **Ideal avalanche effect:** 15.97 bits changed per single-bit flip matches theoretical expectations
- **Uniform bit distributions:** All bit positions within 0.03% of 0.5 probability
- **Zero autocorrelation:** All tested lags show $|\rho_k| < 2 \times 10^{-4}$
- **Zero Dieharder failures:** All variants pass complete suite via streaming input
- **Robust under ablation:** Each component contributes measurably to quality

The generator’s internal metrics match or exceed those of established PRNGs, indicating fundamental soundness of the design approach.

6.2 Comparison to Established Generators

RGE-256’s zero-failure Dieharder performance via streaming input places it among high-quality generators. The Quality 4.0 rating from SmokeRand across all variants confirms statistical excellence. Performance is competitive with established generators such as ChaCha8 and Xoshiro, with quality exceeding Mersenne Twister across all measured metrics.

6.3 Geometric Rotation Parameterization

The use of RDT-derived constants to parameterize rotations represents a novel design principle. While the empirical impact is modest (5% improvement in χ^2 uniformity), this approach offers several advantages:

- **Principled selection:** Rotations are derived from mathematical constants rather than arbitrary choices
- **Theoretical grounding:** Connects PRNG design to number-theoretic entropy measures
- **Parameter space exploration:** Future work can systematically explore alternative ζ -triplets
- **Reproducibility:** Rotation schedule is fully specified and deterministic

6.4 Cross-Coupling and Global Fold

The ablation study demonstrates that cross-coupling between quad blocks is the single most important structural feature, reducing serial correlation by $\sim 40\%$. This validates the design intuition that inter-block mixing is critical for breaking symmetries and accelerating diffusion.

The global fold stage provides complementary benefits, primarily improving chi-square distribution uniformity. Together, these components transform the basic dual-quad ARX core into a high-quality generator.

6.5 Independent testing

Following initial publication, independent testing was performed by Alexey L. Voskov using the SmokeRand testing framework [4]. Key findings:

- **TestU01 BigCrush:** RGE256 core passes all tests in the complete BigCrush battery
- **PractRand:** Passes at least 1 TiB of testing (approximately 1–2 hours runtime with multithreading)
- **SmokeRand batteries:** All cores pass express and default batteries with Quality 4.0 ratings
- **Dieharder streaming:** All cores achieve zero failures via streaming input

This independent testing and results substantially strengthens confidence in the generator’s statistical quality. Extended variants addressing identified limitations (period guarantees, output efficiency) are available in the RGE256 suite at <https://github.com/RRG314/rge256>.

7 Software Availability

7.1 Repository Contents

The complete RGE-256 software package is hosted on GitHub at <https://github.com/RRG314/rge256> under an open-source license. The repository includes:

Core Implementation:

- Reference Python implementation (`rge256.py`)
- JavaScript implementation for web platform
- C99 implementation (`librge256ctr.so`) with Python wrapper
- Optimized variants (3-round, 4-round, 5-round)

RGE256 Suite Variants:

- **RGE256**: Original design from this paper
- **RGE256c**: Adds 64-bit counter, guarantees 2^{64} minimum period
- **RGE256ex**: Heavier ARX mixing with multiple rotations
- **RGE512ex**: 512-bit state for strongest diffusion
- **RGE256ctr**: Counter-mode (recommended), inspired by ChaCha, 6 rounds

Testing Infrastructure:

- Dieharder test harness and automation scripts
- Statistical analysis notebooks (Jupyter)
- Ablation study configurations
- Visualization generation code
- [Streaming generator executables for validation](#)

Interactive Demonstration:

- Web-based testing platform (`rge256_demo.html`)
- Statistical quality assessment tools
- Monte Carlo simulation examples
- Real-time visualization of generator properties

7.2 Reproducing Results

To reproduce the experimental results in this paper:

1. Clone repository: `git clone https://github.com/RRG314/rge256`
2. Install dependencies: `pip install -r requirements.txt`
3. [Compile streaming generators](#): `gcc -O3 -o rge256lite_stream rge256lite_stream.c`
4. [Run Dieharder \(streaming\)](#): `./rge256lite_stream | dieharder -a -g 200`
5. Analyze results: `jupyter notebook analysis.ipynb`

Complete reproduction instructions are provided in the repository's `README.md` file.

8 Limitations and Future Work

8.1 Current Limitations

This work has several acknowledged limitations:

- **Period analysis:** The original RGE256 design operates on a 256-bit state space with maximum theoretical period 2^{256} . However, the nonlinear ARX operations (modular addition combined with XOR) prevent formal period analysis using standard techniques such as characteristic polynomials or GF(2) linear algebra. No proof of minimum period exists, and certain seed values may place the generator in low-period or weak states. Empirical testing found no short cycles, but this does not constitute proof. Users requiring guaranteed minimum period should use the counter-mode variants (RGE256c, RGE256ctr) that guarantee a minimum period of 2^{64} via explicit counter increment.
- **Output efficiency:** The original RGE256 design produces only one 32-bit output word per 8-word (256-bit) state update, which limits throughput compared to designs that output the full state. The counter-mode variant RGE256ctr addresses this architectural limitation by outputting the full permuted state.
- **No cryptanalysis:** No formal security analysis, differential cryptanalysis, or security reduction performed
- **Single parameter set:** Only one $(\zeta_1, \zeta_2, \zeta_3)$ configuration thoroughly tested
- **Bit-reversal testing:** While streaming validation tested the generator’s 32-bit output directly, systematic bit-reversal testing (as recommended for TestU01) has not yet been performed to verify quality of least-significant bits

8.2 Future Directions

Several promising research directions emerge from this work:

Extended Validation:

- TestU01 BigCrush battery evaluation with bit-reversed output (to verify LSB quality)
- Comprehensive PractRand testing to 256 GB or beyond
- Multi-seed validation following Vigna methodology (100 seeds at equidistant intervals)
- Cross-validation across multiple test suites

Performance Analysis:

- Throughput benchmarking (bytes/second, cycles/byte)
- Comparison to ChaCha20, PCG64, Xoshiro on multiple architectures
- SIMD/AVX optimization opportunities
- Hardware implementation feasibility (FPGA, ASIC)

Theoretical Development:

- Systematic exploration of alternative ζ -triplets
- Deeper investigation of RDT-PRNG connections
- Formal analysis of mixing properties and diffusion rates

- Period length analysis and state space coverage

Cryptanalysis:

- Linear cryptanalysis of core permutation
- Differential cryptanalysis for distinguishing attacks
- State recovery attacks and resistance analysis
- Security reduction (if cryptographic use intended)

Extensions:

- Parallel/GPU implementation for high-performance computing
- Multi-stream variants with independent state sequences
- Investigation of BLAKE3 whitening impact on statistical properties
- Alternative state sizes (128-bit, 512-bit variants)

9 Conclusion

RGE-256 is a novel ARX-based pseudorandom number generator demonstrating strong statistical randomness properties validated through comprehensive empirical testing. The generator achieves:

- Maximal entropy (7.999999 bits/byte)
- Ideal avalanche effect (15.97 bits per flip)
- Near-zero autocorrelation ($< 2 \times 10^{-4}$)
- [Zero Dieharder failures via streaming input \(all variants\)](#)
- [42/42 SmokeRand default battery pass rate \(Quality 4.0\)](#)
- Uniform bit distributions (within 0.03%)
- Validated structural design through ablation study

[Independent validation using streaming input mode confirmed that all RGE-256 variants \(RGE256, RGE256c, RGE256ctr, RGE256ex, RGE512ex\) achieve zero Dieharder failures and pass all 42 SmokeRand default battery tests with Quality 4.0 ratings.](#)

The generator’s design connects PRNG construction to Recursive Division Tree theory through principled rotation parameterization, representing a novel approach grounded in number-theoretic entropy analysis. While this connection’s empirical impact is modest, it establishes a mathematical framework for systematic exploration of the PRNG design space.

Performance is competitive with established generators such as ChaCha8 and Xoshiro, with quality exceeding Mersenne Twister across all measured metrics. The ablation study validates each architectural component, with cross-coupling providing the largest marginal improvement ($\sim 40\%$ reduction in serial correlation).

Subsequent independent validation by Voskov using TestU01 and PractRand confirmed that the RGE256 core passes BigCrush and at least 1 TiB of PractRand testing. Extended variants with guaranteed minimum period (2^{64}) and improved output efficiency are available in the RGE256 suite at <https://github.com/RRG314/rge256>.

While this work does not claim cryptographic security, RGE-256 is suitable for scientific computing, Monte Carlo simulation, and non-cryptographic applications requiring high-quality pseudorandomness. The generator provides a solid foundation for future work in ARX-based PRNG design, RDT-inspired cryptographic primitives, and systematic exploration of geometrically-parameterized mixing functions.

Data Availability Statement

All data supporting the findings of this study are openly available:

- Source code: <https://github.com/RRG314/rge256>
- Test corpora: Generated via provided scripts
- Raw test results: Included in repository
- Analysis notebooks: Jupyter notebooks with complete statistical analysis
- Interactive demonstration: Deployable web application

No proprietary software or restricted data were used in this research.

Acknowledgments

The author thanks the developers of Dieharder (Robert G. Brown), PractRand (Chris Doty-Humphrey), and BLAKE3 (Jack O'Connor et al.) for making their test suites and implementations publicly available. The author also acknowledges the broader research community working on PRNG design and analysis, whose work provided essential context and methodological guidance.

The author thanks Alexey L. Voskov for independent validation, test-harness integration, and for contributing the RGE256ex, RGE512ex, and RGE256ctr counter-mode ARX variants included in this work. His review identified two key limitations in the original RGE256 design: (1) output efficiency constraints (one output per 8-word state), and (2) unknown minimum period with potential bad seeds. His validation confirmed that all RGE-256 variants pass Dieharder via streaming input and SmokeRand batteries with Quality 4.0 ratings.

AI Assistance Disclosure

This research was conducted with assistance from Claude (Anthropic), an AI language model, which was used for:

- Code development and debugging (Python and JavaScript implementations)
- Statistical analysis automation and visualization
- Literature review and methodology refinement
- Mathematical notation and \LaTeX document preparation
- Algorithm optimization and performance analysis

All algorithmic design decisions, theoretical frameworks, experimental design, data interpretation, and conclusions are the sole work and responsibility of the author. The AI system served as a collaborative tool for implementation and documentation, not as a source of original

research contributions or intellectual content. All results have been independently verified and validated by the author.

All software, data, and supplementary materials are freely available at <https://github.com/RRG314/rge256> to facilitate independent verification and extension of this work.

References

- [1] S. Reid, “Recursive Geometric Entropy: A Unified Framework for Information-Theoretic Shape Analysis,” Zenodo, DOI: 10.5281/zenodo.17882309, 2025.
- [2] S. Reid, *RGE-256-app Interactive Demonstration: A Comprehensive Testing Platform for ARX-Based Pseudorandom Number Generation*, GitHub repository, 2025. Available at: <https://github.com/RRG314/rge256>.
- [3] S. Reid, “Structural Entropy Analysis of Integer Depth via Recursive Division Tree,” Zenodo, DOI: 10.5281/zenodo.17682287, 2025.
- [4] A. L. Voskov, *SmokeRand: A Set of Tests for Pseudorandom Number Generators*, GitHub repository, 2025. Available at: <https://github.com/alvoskov/SmokeRand>.
- [5] G. Marsaglia, *The Diehard Battery of Tests of Randomness*, Florida State University, 1995. Available at: <http://stat.fsu.edu/pub/diehard/>.
- [6] R. G. Brown, *Dieharder: A Random Number Test Suite*, Version 3.31.1, 2013. Available at: <https://webhome.phy.duke.edu/~rgb/General/dieharder.php>.
- [7] C. Doty-Humphrey, *PractRand: A Randomness Test Suite*, Version 0.95, 2014–present. Available at: <http://pracrand.sourceforge.net/>.
- [8] D. J. Bernstein, “ChaCha, a Variant of Salsa20,” *SASC 2008 Workshop Record*, pp. 3–5, 2008.
- [9] M. E. O’Neill, *PCG: A Family of Better Random Number Generators*, Harvey Mudd College Technical Report HMC-CS-2014-0905, 2014. Available at: <https://www.pcg-random.org/>.
- [10] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, “Parallel Random Numbers: As Easy as 1, 2, 3,” *Proceedings of SC11*, Seattle, WA, 2011.
- [11] J. O’Connor, J. Aumasson, S. Neves, and Z. Wilcox-O’Hearn, *BLAKE3: One Function, Fast Everywhere*, 2020. Available at: <https://github.com/BLAKE3-team/BLAKE3>.
- [12] B. Jenkins, “Algorithm Alley: Hash Functions,” *Dr. Dobbs’s Journal*, September 1997.
- [13] P. L’Ecuyer and R. Simard, “TestU01: A C Library for Empirical Testing of Random Number Generators,” *ACM Transactions on Mathematical Software*, vol. 33, no. 4, article 22, 2007.
- [14] D. Blackman and S. Vigna, “Scrambled Linear Pseudorandom Number Generators,” *ACM Transactions on Mathematical Software*, vol. 47, no. 4, article 36, 2021.
- [15] G. L. Steele Jr., D. Lea, and C. H. Flood, “Fast Splittable Pseudorandom Number Generators,” *OOPSLA 2014*, pp. 453–472, 2014.
- [16] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd edition, Addison-Wesley, 1998.