

# BBRES-RNG

## Bit-Based Randomized Entropy System Scheduler-Based Random Number Generator

Technical Documentation • Architecture • API Reference • Statistical Validation

Developed by Mehul Singh

mehul.engineer | hello@mehul.engineer

Version 1.0 • Java JDK 8+ • 45/45 Statistical Tests Passed

# 1. Overview

BBRES-RNG (Bit-Based Randomized Entropy System) is a multi-threaded random number generator implemented in Java that uses the non-deterministic behaviour of the operating system thread scheduler as its primary entropy source. Unlike conventional pseudo-random number generators (PRNGs) that rely on deterministic mathematical sequences seeded with an initial value, BBRES-RNG derives randomness from real-time system chaos: the unpredictable order in which the OS schedules competing threads.

The result is an RNG whose output is not reproducible even when run on identical hardware with the same inputs, because thread scheduling is influenced by CPU load, interrupt timing, memory pressure, and hundreds of other factors that cannot be controlled or predicted at application level.

## 1.1 Design Philosophy

The fundamental insight behind BBRES-RNG is that concurrency is inherently non-deterministic. When multiple threads compete for execution time, the OS scheduler makes real-time decisions that no application-level algorithm can fully predict or replicate. By deliberately manufacturing race conditions and recording their outcomes, BBRES-RNG converts system-level entropy into usable random bits - conceptually analogous to hardware RNGs that harvest physical noise, except the noise source here is the operating system itself.

## 1.2 Key Characteristics

- Non-deterministic: No seed, no mathematical formula, no reproducible state
- Multi-threaded: Uses concurrent thread races as its entropy mechanism
- Statistically robust: Passes 45/45 tests including all NIST SP 800-22 core tests
- Pure Java: No native libraries or OS-level APIs required
- Configurable: Thread count (n) and generation technique (randTech) are user-tunable
- Rejection-sampled: Range output uses rejection sampling - not modulo - to eliminate bias

## 1.3 Statistical Highlights

Metric	Value	Theoretical Maximum
Shannon Entropy (8-bit)	7.999860	8.000000
Min-Entropy (8-bit)	7.944862	8.000000
Transition Rate (bit flips)	0.500064	0.500000
Bit Balance (1s : 0s)	49.988 : 50.012	50.000 : 50.000
NIST Core Tests Passed	9 / 9	9 / 9
Total Statistical Tests	45 / 45	45 / 45



## 2. Architecture

BBRES-RNG uses a layered, multi-tier threading architecture. Each random bit generation cycle involves spawning a group of worker threads into a controlled race condition and observing the order in which they complete - this order encodes the entropy. Two generation algorithms (G1 and G2) are provided, differing in how threads are assigned IDs before the race.

### 2.1 Component Overview

Class	Package	Role
RNG.java	bbresRNG	Public API. Entry point for all random number generation.
randomBitGeneratorModifiedRoot.java	bbresRNG	Core orchestrator. Manages G1 and G2 bit generation cycles.
modRandomBitGenRNG.java	bbresRNG	Worker thread group manager. Segments threads and collects timing data.
randomBitGenRNG.java	bbresRNG	Base worker thread. Races to record its ID; the arrival order is entropy.
RNGv1.java	bbresv1	Version 1 API. Used internally by G2 for its shuffle step.
randomBitGenRNGPrntv1.java	bbresv1	V1 parent thread. Analogous to modRandomBitGenRNG.
randomBitGenRNGv1.java	bbresv1	V1 worker thread. Analogous to randomBitGenRNG.

### 2.2 Pipeline

Generating a single random bit proceeds through four stages:

#### Stage 1 - Thread ID Assignment

**G1 (Sequential):** An integer array of size  $n$  is populated with IDs 0 through  $n-1$  in order. Thread groups of up to 32 are formed and launched.

**G2 (Shuffled):** The same ID array is first randomly shuffled using a Fisher-Yates shuffle (bootstrapped via RNGv1) before the threads are launched, adding an additional layer of unpredictability in ID assignment.

#### Stage 2 - Controlled Race Condition

All  $n$  worker threads are spawned and immediately enter a spin-wait loop, polling a shared atomic flag. The parent thread sets this flag, releasing all workers simultaneously. Each worker races to write its ID into the first available slot of a shared flag array using a synchronized method. The order in

which threads win and record their IDs is determined entirely by OS thread scheduling - a source of real-time, non-deterministic timing entropy.

### Stage 3 - Bitwise Mixing

Once all workers complete, the parent reads the flag array and XORs together the IDs at the outermost 20% positions (top 10% and bottom 10% by index). This selects a subset of the timing-encoded data and reduces it to a single integer. That integer is then passed through a three-operation xorshift mixing sequence:

```
xor ^= xor << 10;  
xor ^= xor >>> 23;  
xor ^= xor << 7;
```

This finalization step diffuses bit patterns and removes any residual structure before the bit is extracted.

### Stage 4 - Bit Extraction

The least significant bit of the mixed integer (`xor & 1`) is taken as the output bit. This single bit is appended to a growing binary string. Stages 1-4 repeat once per required bit, determined by the bit-length of the requested range.

## 3. API Reference

All public API methods are in the RNG class (package bbresRNG). Instantiate once; methods can be called repeatedly on the same object.

### 3.1 Constructors & Method Signatures

Method Signature	Description
bbresRNG()	Random long in [0, 10]. Default n=71, technique G1.
bbresRNG(long max)	Random long in [0, max]. Default n=71, G1.
bbresRNG(long min, long max)	Random long in [min, max]. Default n=71, G1.
bbresRNG(long min, long max, int n)	Full range with custom thread count n.
bbresRNG(long min, long max, int n, int randTech)	Full control. randTech=1 for G1, randTech=2 for G2.

### 3.2 Parameters

Parameter	Type	Valid Range	Default	Description
min	long	Any long	0	Lower bound of output range (inclusive).
max	long	Any long > min	10	Upper bound of output range (inclusive).
n	int	3 – 1000	71	Number of worker threads per bit generation cycle. Higher values increase entropy depth but reduce performance.
randTech	int	1 or 2	1	Bit generation algorithm. 1 = G1 (sequential IDs), 2 = G2 (shuffle-bootstrapped IDs).

### 3.3 Usage Examples

```
import bbresRNG.RNG;

RNG rng = new RNG();

// Basic: random number in [0, 10]
long result = rng.bbresRNG();

// Custom range: [5, 15]
long result = rng.bbresRNG(5, 15);
```

```
// Custom concurrency: [1, 100] with 50 threads
long result = rng.bbresRNG(1, 100, 50);

// Full control: G2, n=35, range [0, 10000]
long result = rng.bbresRNG(0, 10000, 35, 2);
```

### 3.4 Range Handling & Bias Elimination

BBRES-RNG uses rejection sampling to map generated bits to the requested range, eliminating modulo bias. The generated binary string is converted to a long integer; if the result exceeds the requested range, the cycle repeats until a valid value is produced. This ensures a perfectly uniform distribution across all values in [min, max].

```
do {
    randomNo = getInteger(getRandomBinary(range));
    randomNo = randomNo & Long.MAX_VALUE;
} while (randomNo > range);
return min + randomNo;
```

## 4. Configuration Guide

### 4.1 Choosing n (Thread Count)

The parameter `n` controls how many worker threads compete in each race. A higher `n` increases the entropy pool - more threads means more timing variation to harvest - but also increases overhead, as each bit generation cycle creates and joins `n` threads.

n Value	Threads per bit	Use Case	Approx. Relative Speed
3 – 20	3–20	Low-entropy, fast generation. Prototype/testing only.	Very Fast
21 – 50	21–50	Light use. Reasonable quality for non-critical applications.	Fast
71 (default)	71	Balanced quality and performance. General purpose.	Moderate
100 – 200	100–200	Higher entropy depth. Suitable where quality is prioritized.	Slow
200 – 1000	200–1000	Maximum entropy. Research / security-critical use only.	Very Slow

### 4.2 Choosing randTech (Generation Algorithm)

**G1 (randTech=1, default):** Thread IDs are assigned sequentially (0 to `n-1`). The race order is the sole source of entropy. Faster to initialize.

**G2 (randTech=2):** Thread IDs are first randomized via a Fisher-Yates shuffle (using RNGv1) before the race. Adds an extra entropy layer: even the IDs being raced are unpredictable. Slower due to the shuffle bootstrap, but provides deeper entropy mixing.

## 5. Statistical Validation

BBRES-RNG was validated against a comprehensive 45-test battery, run on a sample of 10,918,505 bits and 100,000 integers in the range [0, 999]. The significance level used throughout was  $\alpha = 0.01$ .

### 5.1 NIST SP 800-22 Core Tests - 9/9

Test	p-value	Result
Frequency (Monobit)	0.441897	PASS
Block Frequency (M=128)	0.143304	PASS
Runs Test	0.670559	PASS
Longest Run of Ones	0.046389	PASS
Cumulative Sums (Forward)	0.164091	PASS
Cumulative Sums (Reverse)	0.656919	PASS
Approximate Entropy (m=2)	0.851301	PASS
Serial (m=2) delta1	0.679895	PASS
Serial (m=2) delta2	0.671131	PASS

### 5.2 NIST SP 800-22 Extended Tests - 7/7

Test	p-value	Result
Maurer's Universal Statistical	0.722786	PASS
Poker Test (m=4)	0.517531	PASS
Random Excursion Variant	0.933697	PASS
Binary Matrix Rank	0.982609	PASS
Non-Overlapping Template (m=9)	0.361738	PASS
Overlapping Template (m=9)	1.000000	PASS
Linear Complexity	0.620280	PASS

### 5.3 Adversarial & ML Attack Tests - 5/5

All three machine learning models achieved accuracy at or below 50.52% - statistically equivalent to random guessing - confirming that BBRES-RNG output cannot be predicted by standard ML classifiers even given access to prior output.

Attack	Accuracy	p-value	Result
Logistic Regression (w=16)	0.4901 (49.01%)	0.976148	PASS
Gradient Boosted Trees (w=16)	0.4935 (49.35%)	0.877536	PASS
MLP Neural Network (w=16)	0.5052 (50.52%)	0.173827	PASS
Frequency Prediction (w=8)	0.5020 (50.20%)	0.523577	PASS
Pattern Repetition (w=59)	No repetition detected	1.000000	PASS

## 5.4 Entropy Quality

Metric	BBRES-RNG	Theoretical Max	% of Maximum
Shannon Entropy (8-bit)	7.999860	8.000000	99.998%
Min-Entropy (8-bit)	7.944862	8.000000	99.311%
Transition Rate	0.500064	0.500000	99.987%
Bit Balance (1s : 0s)	49.988 : 50.012	50 : 50	99.976%

## 6. Known Limitations

---

### 6.1 Performance

Each random bit requires creating and joining  $n$  threads. For  $n=71$  (default), generating a 64-bit number requires approximately 4,544 thread lifecycle events. This makes BBRES-RNG orders of magnitude slower than seeded PRNGs like `java.util.Random` or hardware-backed sources like `SecureRandom`. It is not suitable for applications requiring high-throughput generation (e.g., simulations generating millions of values).

### 6.2 Thread-Safety of Shared State

Several fields are declared as static on the RNG class (`min`, `max`, `n`, `randTech`) and on helper classes (`randomBitGenRNG.flag[]`, `randomBitGeneratorModifiedRoot.nos`). Concurrent calls to `bbresRNG()` from multiple threads on the same or different instances will produce race conditions on these shared fields. BBRES-RNG should not be used from multiple threads simultaneously without external synchronization.

### 6.3 Platform Sensitivity

The quality and distribution of entropy produced is tied to the behavior of the JVM thread scheduler on the host platform. Behavior may vary between operating systems, JVM implementations, hardware architectures, and system load levels. Results are validated on the platform where testing was conducted; entropy characteristics may differ on constrained environments (embedded systems, containers with restricted scheduling).

### 6.4 V1 Modulo Bias

The internal G2 bootstrapping step uses RNGv1, which maps its output with a direct modulo operation (`randomNo % (max - min + 1)`). This introduces a small modulo bias when the range does not divide evenly into the bit-space. The primary API (`bbresRNG`) is not affected - it uses rejection sampling. The bias affects only the Fisher-Yates shuffle bootstrapper in G2, and its practical impact on final output quality is considered negligible given the subsequent entropy mixing stages.

## 7. Build & Run

---

### 7.1 Prerequisites

- Java Development Kit (JDK) 8 or higher
- Any Java IDE (IntelliJ IDEA, Eclipse, VS Code with Java Extension) or terminal

### 7.2 Build Instructions

```
# Clone the repository
git clone https://github.com/singhmehul7783/Bit-Based-Randomized-Entropy-System-
Scheduler-Based-RNG.git
cd Bit-Based-Randomized-Entropy-System-Scheduler-Based-RNG

# Compile all source files
javac src/bbresv1/*.java src/bbresRNG/*.java src/Main.java -d out/

# Run the demo
java -cp out Main
```

### 7.3 Expected Output

```
Random number between 0 and 10: 7
Random number between 5 and 15: 11
Random number between 1 and 100 with n=50: 43
Random number between 1 and 100 with invalid n=1: [uses default n=71]
Random number smaller than 10000: 8821
Random number smaller than 10000 using 2nd method with n=35: 3147
```

## 8. License

---

BBRES-RNG is released under a custom restrictive license. The terms are summarized as follows:

Category	Terms
Permitted	Academic research, education, personal study - with proper attribution to the author.
Prohibited (without written permission)	Commercial use, modification, redistribution, sublicensing, or creation of derivative works.
Attribution requirement	Any permitted use must credit Mehul Singh and reference the original repository.
Contact for permissions	<a href="https://github.com/singhmehul7783">github.com/singhmehul7783</a>   <a href="mailto:hello@mehul.engineer">hello@mehul.engineer</a>

---

BBRES-RNG • Research Prototype • Validated against 45 statistical tests over 10.9M+ bits