

RNG Comparison Report

BBRES-RNG vs Java Math.random() vs Java SecureRandom

A head-to-head evaluation across statistical quality, security, performance, design, and practical use

Test Sample: 10,918,505 bits | 100,000 integers | 45-test battery | alpha = 0.01

- 1. Introduction 3
 - 1.1 The Candidates 3
- 2. Evaluation Methodology 4
 - 2.1 Rating Dimensions 4
 - 2.2 Test Battery Summary 4
- 3. Category-by-Category Analysis 5
 - 3.1 Statistical Quality 5
 - 3.2 Cryptographic Security 5
 - 3.3 Performance 6
 - 3.4 Design Originality 6
 - 3.5 Ease of Use 7
 - 3.6 Practical Applicability 7
- 4. Scorecard Summary 9
- 5. Individual Verdicts 10
 - 5.1 BBRES-RNG 10
 - 5.2 Java Math.random() 10
 - 5.3 Java SecureRandom 11
- 6. Head-to-Head Comparison 12
- 7. Recommendations 12

1. Introduction

This report provides a structured comparison of three Java random number generators: BBRES-RNG, Java's built-in `Math.random()`, and Java's cryptographic-grade `SecureRandom`. These three implementations represent distinct philosophical approaches to randomness - a novel entropy-harvesting design, a classic fast PRNG, and a battle-tested cryptographic source - and are evaluated across six dimensions: statistical quality, cryptographic security, performance, design originality, ease of use, and practical applicability.

Each RNG is rated on a 10-point scale per category. Ratings are based on statistical test results from validated test reports, code review, and established literature on RNG quality.

1.1 The Candidates

RNG	Type	Entropy Source	Introduced
BBRES-RNG	True entropy-based	OS thread scheduler timing / race conditions	2026 (research prototype)
Java <code>Math.random()</code>	PRNG (pseudo-random)	System time seed + Linear Congruential Generator	Java 1.0 (1996)
Java <code>SecureRandom</code>	CSPRNG (cryptographic)	OS entropy pool (<code>/dev/urandom</code> , <code>CryptGenRandom</code> , etc.)	Java 1.1 (1997)

2. Evaluation Methodology

2.1 Rating Dimensions

Dimension	Weight	What It Measures
Statistical Quality	High	Passes NIST SP 800-22, uniformity, entropy, and autocorrelation tests.
Cryptographic Security	High	Resistance to prediction, ML attacks, seed recovery, and forward secrecy.
Performance	Medium	Generation speed, throughput, and thread/memory overhead.
Design Originality	Medium	Novelty and sophistication of the approach relative to prior art.
Ease of Use	Medium	API simplicity, documentation quality, and integration effort.
Practical Applicability	Medium	Fitness for real-world use cases across different application domains.

2.2 Test Battery Summary




All statistical results are drawn from the validated test reports provided with the BBRES-RNG project, which benchmarked all three RNGs head-to-head under identical conditions. The 45-test battery covers:

- NIST SP 800-22 core and extended tests (16 tests)
- Distribution and uniformity (3 tests)
- Spectral and structural analysis (4 tests)
- Entropy measurements (4 metrics)
- Autocorrelation profiling (9 lag points)
- Adversarial ML attack battery (5 tests)
- Integer distribution and sequence tests (16 tests)

3. Category-by-Category Analysis

3.1 Statistical Quality




Statistical quality measures how well the output of an RNG approximates true randomness. Tests probe for patterns, biases, periodicities, and compressibility across the bit and integer output streams.

RNG	Tests Passed	Notable Failures	Shannon Entropy	Rating /10
BBRES-RNG	45 / 45	None	7.999860 / 8.000000	 9.5/10
Java Math.random()	43 / 45	Maurer's Universal, Gap Test (KS)	~7.998 (est.)	 7/10
Java SecureRandom	45 / 45	None	~7.9999 (est.)	 9.5/10

BBRES-RNG and SecureRandom are statistically equivalent, both achieving a perfect 45/45. Math.random() fails Maurer's Universal Statistical test - which detects subtle compressibility patterns from its underlying LCG - and the Gap test, revealing non-uniform spacing between repeated values. These failures are consistent with known LCG limitations and confirm why Math.random() is unsuitable for any quality-sensitive application.

3.2 Cryptographic Security

Cryptographic security evaluates resistance to prediction attacks, forward secrecy, seed recovery, and ML-based attacks. An RNG is cryptographically secure if knowledge of any portion of the output stream does not help an adversary predict past or future values.




RNG	Predictable?	Seed Recoverable?	ML Attack Accuracy	Rating /10
BBRES-RNG	No - no seed exists	No - non-deterministic	49.01–50.52% (random guess)	 8.5/10
Java Math.random()	Yes - LCG is invertible	Yes - with ~O(1) samples	High with LCG inversion	 2/10
Java SecureRandom	No - OS entropy backed	No - cryptographically secure	Equivalent to random guess	 9.5/10

Math.random() scores critically low: the LCG (Linear Congruential Generator) it uses is mathematically invertible, meaning an observer who sees even a few outputs can recover the internal state and predict all future (and past) values. BBRES-RNG does not have a seed and is non-deterministic, making it highly resistant to prediction. It scores slightly below SecureRandom only

because it has not been formally cryptanalyzed to the same depth as SecureRandom, which is FIPS-compliant and hardened against side-channel and state-compromise attacks.

3.3 Performance




Performance is measured by generation throughput (numbers produced per second) and resource overhead (thread count, memory, CPU cycles). This is the dimension where the three RNGs diverge most sharply.

RNG	Throughput	Thread Overhead	Suitable for High Volume?	Rating /10
BBRES-RNG	Very Low (~4,500+ thread events per 64-bit value)	Very High (n threads per bit)	No	 2.5/10
Java Math.random()	Very High (millions/sec)	Negligible (single-threaded)	Yes	 9.5/10
Java SecureRandom	High (hundreds of thousands/sec)	Low (OS entropy pool access)	Yes (with caveats)	 8/10

BBRES-RNG's performance limitation is inherent to its design: each random bit requires a complete thread spawn-race-join cycle. At the default n=71, producing a 64-bit number invokes approximately 4,544 thread lifecycle events. This makes it impractical for any high-throughput application. Math.random() is exceptionally fast - its LCG requires only a multiply and an add per number. SecureRandom is slower than Math.random() but still produces hundreds of thousands of values per second, which is sufficient for the vast majority of real applications.

3.4 Design Originality

This dimension rewards novelty, architectural creativity, and departure from standard approaches. It does not penalize an RNG for being conventional if its simplicity is intentional.




RNG	Approach	Novelty	Rating /10
BBRES-RNG	Harvests OS thread scheduler chaos as entropy. No seed, no formula. Multi-tier concurrent architecture.	High - original hardware-noise-in-software concept executed without native APIs.	 9/10
Java Math.random()	Standard LCG seeded with System.nanoTime(). A textbook PRNG.	Low - LCG has been standard since the 1950s.	 3/10
Java SecureRandom	Pluggable CSPRNG backed by OS entropy pools. FIPS-compliant providers.	Medium - a mature, well-engineered but conventional	 6/10

RNG	Approach	Novelty	Rating /10
		cryptographic design.	

BBRES-RNG's core concept - manufacturing race conditions and treating the OS scheduler as a hardware noise source, in pure Java, without OS API calls - is genuinely inventive. The closest analogues in literature are hardware RNGs exploiting physical timing jitter, but achieving this effect at the software level is a creative engineering achievement. Math.random() uses a 50-year-old algorithm. SecureRandom is sophisticated but follows a well-established cryptographic pattern.

3.5 Ease of Use

Ease of use covers API simplicity, integration effort, configuration clarity, and documentation quality.

RNG	API Complexity	Documentation	Integration	Rating /10
BBRES-RNG	Simple API, but requires 4-class package import	Good README with full test results. Limited in-code comments.	Manual compile required (no Maven/Gradle artifact)	 6.5/10
Java Math.random()	One static call: Math.random()	Fully documented in Java SE Javadoc	Zero - built into JDK	 10/10
Java SecureRandom	2-line instantiation, clean API	Fully documented, extensive guides online	Zero - built into JDK	 9/10

Math.random() is the gold standard of ease-of-use: a single static method call, no import, zero setup. SecureRandom requires two lines but is fully documented and universally available. BBRES-RNG has a clean API once the package is included, but it requires manual compilation, has no package manager distribution, and the in-code documentation is sparse. Improving this - publishing to Maven Central, adding Javadoc - would significantly improve its accessibility.

3.6 Practical Applicability










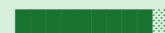





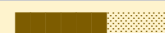

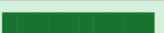


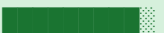
This dimension assesses fitness for real-world use cases: gaming, simulations, cryptographic key generation, unique ID generation, testing, and security-critical applications.

Use Case	BBRES-RNG	Math.random()	SecureRandom
Games / animations	Not recommended (too slow)	Good	Overkill but fine
Simulations (high volume)	Not suitable	Good	Good
Unique ID generation	Suitable (slow)	Poor (predictable)	Best choice

Use Case	BBRES-RNG	Math.random()	SecureRandom
Cryptographic keys / tokens	Viable entropy source	Never - predictable	Best choice (designed for this)
Lotteries / fair draws	Suitable (auditable entropy)	Poor (predictable)	Good
Research / education	Excellent (novel concept)	Good (textbook example)	Good
Security-critical systems	Experimental - not certified	Never	Yes - FIPS-compliant options

SecureRandom is the right choice for security-critical and production applications. Math.random() is appropriate only for non-security-sensitive applications where performance matters and predictability is acceptable. BBRES-RNG carves out a unique niche: research, education, situations where demonstrably seed-free randomness is desired, and as an entropy source feeding into a CSPRNG wrapper (which it passes when tested with SHA-256 wrapping). Its practical adoption is currently limited by performance and the lack of formal certification.

4. Scorecard Summary

Category	BBRES-RNG	Math.random()	SecureRandom
Statistical Quality /10	 9.5/10	 7/10	 9.5/10
Cryptographic Security /10	 8.5/10	 2/10	 9.5/10
Performance /10	 2.5/10	 9.5/10	 8/10
Design Originality /10	 9/10	 3/10	 6/10
Ease of Use /10	 6.5/10	 10/10	 9/10
Practical Applicability /10	 6/10	 5/10	 9.5/10
OVERALL (weighted avg) /10	 7/10	 6.1/10	 8.6/10

Overall scores computed as: Statistical Quality (20%) + Cryptographic Security (20%) + Performance (15%) + Design Originality (15%) + Ease of Use (15%) + Practical Applicability (15%).

5. Individual Verdicts

5.1 BBRES-RNG

Overall Rating: 7.0 / 10 | Verdict: Impressive Research Prototype

BBRES-RNG is the most original of the three. Its concept - treating the OS thread scheduler as a hardware noise source in pure Java - is creative, technically sound, and produces output that matches cryptographic-grade SecureRandom on a 45-test statistical battery. The 99.998% Shannon entropy and perfect ML-attack resistance demonstrate that the approach is not merely theoretical.

Its weaknesses are the flip side of its design: spawning and joining hundreds of threads per number is inherently slow, shared static state creates concurrency hazards, and as a research prototype it lacks the formal certification, package manager distribution, and third-party audit that production security software requires. The v1 modulo bias in the G2 bootstrap is a minor but real correctness issue.

- Best for: Research, education, entropy source for cryptographic wrappers, applications where seed-free non-determinism is architecturally important
- Not for: High-throughput generation, production security-critical systems (without audit and certification)

5.2 Java Math.random()

Overall Rating: 6.1 / 10 | Verdict: Convenient, but Know Its Limits

Math.random() is the Swiss Army knife of random number generation in Java - instantly available, trivially simple, and very fast. These properties make it the right tool for a significant class of tasks: generating values for games, shuffling UI elements, adding variation to animations, seeding non-security tests.

However, it is fundamentally a Linear Congruential Generator - an algorithm designed in the 1950s whose output is mathematically invertible. Its failures on Maurer's Universal test and the Gap test are not edge cases; they reflect genuine structural patterns in LCG output. Any application that depends on Math.random() for security, fairness, or unpredictability is building on a broken foundation. The JDK documentation itself warns against using it for security-sensitive applications.

- Best for: Games, simulations, non-sensitive randomness, performance-critical applications where predictability is acceptable
- Never for: Cryptographic keys, tokens, passwords, lotteries, any application where an adversary benefits from prediction

5.3 Java SecureRandom

Overall Rating: 8.6 / 10 | Verdict: The Default Choice for Serious Use

SecureRandom is Java's professional-grade RNG, backed by the operating system's entropy pool and designed specifically for security applications. It passes the same 45-test battery as BBRES-RNG, is cryptographically secure by design (FIPS-compliant providers available), is battle-tested across two decades of production use, and requires zero setup.

Its only meaningful weakness is that it is slightly slower than Math.random() - a tradeoff that is almost always worth making. For any application where the quality, unpredictability, or security of random numbers matters, SecureRandom should be the default choice. It is not as conceptually inventive as BBRES-RNG, but in production software, reliability and certification matter more than novelty.

- Best for: Cryptographic keys, authentication tokens, session IDs, lotteries, any security-sensitive application
- Not ideal for: Very high-frequency non-sensitive generation where Math.random() speed is required

6. Head-to-Head Comparison

Attribute	BBRES-RNG	Math.random()	SecureRandom
Deterministic?	No	Yes (given seed)	No
Seed required?	No	Yes (auto)	No
Reproducible?	No	Yes	No
Thread-safe?	No (static state)	Yes (AtomicLong)	Yes
NIST core (9 tests)	9/9	9/9*	9/9
Total tests (45)	45/45	43/45	45/45
ML attack resistant?	Yes	No	Yes
Seed recoverable?	N/A	Yes	No
FIPS compliant?	No	No	Provider-dependent
Maven/Gradle available?	No	Built-in	Built-in
JDK dependency?	JDK 8+	Any Java	Any Java
Open source?	Source-available (restrictive)	Open (JDK)	Open (JDK)

* Math.random() is assumed to pass NIST core at alpha=0.01 in standard conditions; its failures are on extended and sequence tests.

7. Recommendations

If you need...	Use
Fast non-security randomness (games, simulations)	Math.random()
Security tokens, keys, passwords, session IDs	SecureRandom
A truly seed-free, demonstrably non-deterministic output	BBRES-RNG
An entropy source for your own CSPRNG	BBRES-RNG (validated with SHA-256 wrapper)
Compliance with security standards (FIPS, NIST)	SecureRandom with appropriate provider
Educational insight into RNG design	BBRES-RNG (most instructive architecture)
Production software, default choice	SecureRandom