Examples of High-Quality PRNGs

Peter Occil

This version of the document is dated 2024-12-24.

Besides cryptographic random number generators (RNGs), the following are examples of **high-quality pseudorandom number generators (PRNGs)**[1]. The "Fails PractRand Starting At" column in this and other tables in this page means the number of bytes (rounded up to the nearest power of two) at which PractRand detects a failure in the PRNG. (Note that high-quality PRNGs, as I define them, are not necessarily appropriate for information security.)

PRNG	Seeds Allowed	Cycle Length	Fails PractRand Starting At	Notes
xoshiro256**	2^256 - 1	2^256 - 1	??? TiB	
xoshiro256+	2^256 - 1	2^256 - 1	??? TiB	Lowest bits have low linear complexity (see (Blackman and Vigna 2019)[2] and see also "**Testing low bits in isolation**"); if the application or library cares, it can discard those bits before using this PRNG's output.
xoshiro256++	2^256 - 1	2^256 - 1	??? TiB	
xoshiro512**	2^512 - 1	2^512 - 1	??? TiB	
xoshiro512+	2^512 - 1	2^512 - 1	??? TiB	Lowest bits have low linear complexity
xoshiro512++	2^512 - 1	2^512 - 1	??? TiB	
xoroshiro128++	2^128 - 1	2^128 - 1	??? TiB	
xoroshiro128**	2^128 - 1	2^128 - 1	??? TiB	
SFC64 (C. Doty-Humphrey)	2^192	At least 2^64 per seed	??? TiB	256-bit state
Philox4×64-7	2^128	At least 2^256 per seed	??? TiB	384-bit state
Velox3b	2^64	At least 2^128 per seed	??? TiB	256-bit state
`gjrand` named after Geronimo Jones	2^128	At least 2^64 per seed	??? TiB	256-bit state

[1] https://peteroupc.github.io/random.html#High__Quality__RNGs__Requirements

[2] Blackman, D., Vigna, S. "Scrambled Linear Pseudorandom Number Generators", 2019 (xoroshiro and xoshiro families); S. Vigna, "**An experimental exploration of Marsaglia's `xorshift` generators, scrambled**", 2016 (scrambled xorshift family).

PRNG	Seeds Allowed	Cycle Length	Fails PractRand Starting At	Notes
MRG32k3a (L'Ecuyer 1999; L'Ecuyer et al. 2002)[3]	Near 2^192	2 cycles with length near 2^191	??? TiB	192-bit state
MRG31k3p (L'Ecuyer and Touzin 2000)[4]	Near 2^186	2 cycles with length near 2^185	??? TiB	192-bit state
JLKISS (Jones 2007/2010)[5]	2^64 * (2^64 - 1)^2	At least (2^128 - 2^64)	??? TiB	192-bit state
JLKISS64 (Jones 2007/2010)[6]	2^64 * (2^64 - 1)^3	At least (2^128 - 2^64)	??? TiB	256-bit state
A multiplicative **linear congruential generator**[7] (LCG) with prime modulus greater than 2^{63} described in Table 2 of (L'Ecuyer 1999)[8]	Modulus - 1	Modulus - 1	??? TiB	Memory used depends on modulus size
XorShift* 128/64	2^128 - 1	2^128 - 1	??? TiB	128-bit state. Described by M. O'Neill in "You don't have to use PCG!", 2017.[9]
XorShift* 64/32	2^64 - 1	2^64 - 1	??? TiB	64-bit state. Described by M. O'Neill in "You don't have to use PCG!", 2017.

0.1 PRNGs with Stream Support

Some PRNGs support multiple "streams" that behave like independent uniform random bit sequences. The test for independence involves interleaving two "streams"' outputs and sending the interleaved outputs to the PractRand tests.

The following lists high-quality PRNGs that support streams and their PractRand results for different strategies of forming pseudorandom number "streams".

[3]L'Ecuyer, P., "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators", *Operations Research* 47(1), 1999; L'Ecuyer, P., Simard, R., et al., "An Object-Oriented Random Number Package with Many Long Streams and Substreams", *Operations Research* 50(6), 2002.

[4]L'Ecuyer, P., Touzin, R., "Fast Combined Multiple Recursive Generators with Multipliers of the Form a = $\pm 2^{q} \pm 2^{r}$", *Proceedings of the 2000 Winter Simulation Conference*, 2000.

[5]Jones, D., "Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications", 2007/2010.

[6]Jones, D., "Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications", 2007/2010.

[7]https://en.wikipedia.org/wiki/Linear_congruential_generator

[8]P. L'Ecuyer, "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure", *Mathematics of Computation* 68(225), January 1999, with **errata**.

[9]This XorShift* generator is not to be confused with S. Vigna's *-scrambled PRNGs, which multiply the PRNG state differently than this one does.

PRNG	Fails PractRand Starting At	Notes
xoshiro256**	Jump-ahead by 2^64: ??? TiBJump-ahead by 2^128: ??? TiBJump-ahead by 2^256/ φ : ??? TiBConsecutive seeds: ??? TiB	
xoshiro256++	Jump-ahead by 2^64: ??? TiBJump-ahead by 2^128: ??? TiBJump-ahead by 2^256/ φ : ??? TiBConsecutive seeds: ??? TiB	
xoroshiro128**	Jump-ahead by 2^64: ??? TiBJump-ahead by 2^128/ φ : ??? TiBConsecutive seeds: ??? TiB	
xoroshiro128++	Jump-ahead by 2^64: ??? TiBJump-ahead by 2^128/ φ : ??? TiBConsecutive seeds: ??? TiB	
SFC64	Consecutive seeds: ??? TiBSeed increment by 2^64: ??? TiB	
Philox4×64-7	Consecutive seeds: ??? TiBSeed increment by 2^64: ??? TiB	
PCG64	Jump-ahead by period/ φ : ??? TiB	What PCG calls "streams" does not produce independent sequences.
???	Jump-ahead by period/ φ : ??? TiB	

0.2 Counter-Based PRNGs

Constructions for counter-based PRNGs (using the definition from (Salmon et al. 2011, section 2)[10] include:

1. A PRNG that outputs hash codes of a counter and the seed.
2. A PRNG that uses a block cipher with the seed as a key to give out encrypted counters.

More specifically, let C and S each be 64 or greater and divisible by 8. Then:

1. A C-bit counter is set to 0 and an S-bit seed is chosen. In each iteration, the PRNG outputs `H(seed || 0x5F || counter)` (where `H` is a hash function, `||` means concatenation, `0x5F` is the 8-bit block 0x5F, and `seed` and `counter` are little-endian encodings of the seed or counter, respectively), and adds 1 to the counter by wraparound addition. Or…
2. A C-bit counter is set to 0 and an S-bit seed is chosen. In each iteration, the PRNG gives out a block generated by a C-bit block cipher where the key is a little-endian encoding of the seed, and where the cleartext is a little-endian encoding of the counter, and adds 1 to the counter by wraparound addition.

The following lists hash functions and block ciphers that form high-quality counter-based PRNGs. It's possible that reduced-round versions of these and other functions will also produce high-quality counter-based PRNGs.

[10]Salmon, John K., Mark A. Moraes, Ron O. Dror, and David E. Shaw. "Parallel random numbers: as easy as 1, 2, 3." In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-12. 2011.

Function	Fails PractRand Starting At	Notes
Hash Functions: BEBB4185; BLAKE2S-256; BLAKE3; CityHash64; Falkhash; FarmHash128; FarmHash32; FarmHash64; Farsh32; Farsh64; Floppsyhash; GoodOAAT; Half-SipHash; Hasshe2; MD5 (low 32 bits); Metrohash128; Mirhash; Mirhashstrict (low 32 bits); MUM; MurmurHash64A for x64; MurmurHash3 (128-bit) for x64; Seahash; Seahash (low 32 bits); SHA-256; SHA-256 (low 64 bits); SHA3-256; SipHash; Spooky64; Fast Positive Hash (32-bit big-endian); TSip; xxHash v3 64-bit (both full and low 32 bits)	> 1 TiB	S = 64, C = 128. Failure figure applies to regular sequence; 2, 4, and 11 interleaved streams from consecutive seeds; 2, 4, and 11 interleaved streams from counters incremented by 2^{64}; 2, 4, and 11 interleaved streams from counters incremented by 2^{96}.
???	??? TiB (Consecutive seeds: ??? TiB)	
???	??? TiB (Consecutive seeds: ??? TiB)	
???	??? TiB (Consecutive seeds: ??? TiB)	

0.3 Combined PRNGs

The following lists high-quality combined PRNGs. See “**Testing PRNGs for High-Quality Randomness**[11]” for more information on combining PRNGs.

Function	Fails PractRand Starting At	Notes
??? combined with Weyl sequence	??? TiB	
??? combined with 128-bit LCG	??? TiB	
JSF64 combined with ???	??? TiB	
JSF64 combined with ???	??? TiB	
Tyche combined with ???	??? TiB	
Tyche-i combined with ???	??? TiB	
??? combined with ???	??? TiB	

0.4 Splittable PRNGs

The following lists high-quality splittable PRNGs. See “**Testing PRNGs for High-Quality Randomness**[12]” for more information on testing splittable PRNGs, and see the appendix for splittable PRNG constructions.

[11] https://github.com/peteroupc/peteroupc.github.io/blob/master/randomtest.md
[12] https://github.com/peteroupc/peteroupc.github.io/blob/master/randomtest.md

Function	Fails PractRand Starting At	Notes
???	??? TiB	
???	??? TiB	
???	??? TiB	

0.5 PRNGs Not Preferred

Although the following are technically high-quality PRNGs, they are not preferred:

PRNG	Notes
C++'s **`std::ranlux48` engine**	Usually takes about 192 8-bit bytes of memory. Admits up to 2^577 - 2 seeds; seed's bits cannot be all zeros or all ones (Lüscher 1994)[13]. The maximum cycle length for `ranlux48`'s underlying generator is very close to 2^576.
A high-quality PRNG that is an LCG with nonprime modulus (or a PRNG based on one, such as PCG)	If the modulus is a power of 2, this PRNG can produce highly correlated pseudorandom number sequences from seeds that differ only in their high bits (see S. Vigna, "**The wrap-up on PCG generators**") and lowest bits have short cycles. (What PCG calls "streams" does not produce independent sequences.)

0.6 Not High-Quality PRNGs

The following are not considered high-quality PRNGs:

Algorithm	Notes
Sequential counter	Doesn't behave like independent random sequence
A linear congruential generator with modulus less than 2^{63} (such as `java.util.Random` and C++'s `std::minstd_rand` and `std::minstd_rand0` engines)	Admits fewer than 2^{63} seeds
Mersenne Twister (MT19937)	Shows a **systematic failure** in BigCrush's LinearComp test (part of L'Ecuyer and Simard's "TestU01"). (See also (Vigna 2019)[14].) Moreover, it usually takes about 2500 8-bit bytes of memory.
Marsaglia's `xorshift` family ("Xorshift RNGs", 2003)	Shows systematic failures in SmallCrush's MatrixRank test (Vigna 2016)[15]
`System.Random`, as implemented in the .NET Framework 4.7	Admits fewer than 2^{63} seeds
Ran2 (*Numerical Recipes*)	Minimum cycle length less than 2^{63}

[13] Lüscher, M., "A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations", arXiv:hep-lat/9309020 (1994). See also Conrads, C., "**Faster RANLUX Pseudo-Random Number Generators**". https://christoph-conrads.name/faster-ranlux-pseudo-random-number-generators/

[14] S. Vigna, "**It Is High Time We Let Go of the Mersenne Twister**", arXiv:1910.06437 [cs.DS], 2019. https://arxiv.org/abs/1910.06437

[15] S. Vigna, "**An experimental exploration of Marsaglia's `xorshift` generators, scrambled**", 2016.

Algorithm	Notes
`msws` (Widynski 2017)[16]	Admits fewer than 2^{63} seeds (about $2^{54.1}$ valid seeds)
JSF32 (B. Jenkins's "A small noncryptographic PRNG")	Admits fewer than 2^{63} seeds; proven minimum cycle length is only 2^{20} or more
JSF64 (B. Jenkins's "A small noncryptographic PRNG")	No proven minimum cycle of at least 2^{63} values
Middle square	No proven minimum cycle of at least 2^{63} values
Many cellular-automaton PRNGs (especially if they are neither reversible nor maximal-length[17])	No proven minimum cycle of at least 2^{63} values
Tyche/Tyche-i (Neves and Araujo 2011)[18]	No proven minimum cycle of at least 2^{63} values
ISAAC ("ISAAC and RC4" by B. Jenkins)	Proven minimum cycle length is only 2^{40} or more

1 Notes

2 Appendix

2.1 Implementation Notes: Splittable PRNGs

Here are implementation notes on splittable PRNGs. The **pseudocode conventions**[19] apply to this section. In addition, the following notation is used:

- The `||` symbol means concatenation.
- `TOBYTES(x, n)` converts an integer to a sequence of `n` 8-bit bytes, in "little-endian" encoding: the first byte is the 8 least significant bits, the second byte is the next 8 bits, and so on. No more than `n` times 8 bits are encoded, and unused bytes become zeros.
- `BLOCKLEN` is the hash function's block size in bits. For noncryptographic hash functions, this can be the function's output size in bits instead. `BLOCKLEN` is rounded up to the closest multiple of 8.
- `TOBLOCK(x)` is the same as `TOBYTES(x, BLOCKLEN / 8)`.

The splittable PRNG designs described here use *keyed hash functions*, which hash a message with a given key and give out a hash code. An unkeyed hash function can become a keyed hash function by hashing the following data: `key || TOBYTES(0x5F, 1) || message`.

The Claessen–Pałka splittable PRNG (Claessen and Pałka 2013)[20] can be described as follows:

- A PRNG state has two components: a seed and a path (a vector of bits). A new state's seed is `TOBLOCK(seed)` and its path is an empty bit vector.
- `split` creates two new states from the old one; the first (or second) is a copy of the old state, except a 0 (or 1, respectively) is appended to the path. If a new state's path reaches `BLOCKLEN` bits this way, the state's seed is set to the result of hashing `BitsToBytes(path)` with the seed as the key, and the state's path is set to an empty bit vector.
- `generate` creates a random number by hashing `BitsToBytes(path)` with the seed as the key.

[16] Widynski, B., "**Middle Square Weyl Sequence RNG**", arXiv:1704.00358 [cs.CR], 2017. https://arxiv.org/abs/1704.00358

[17] Bhattacharjee, K., "**Cellular Automata: Reversibility, Semi-reversibility and Randomness**", arXiv:1911.03609 [cs.FL], 2019. https://arxiv.org/abs/1911.03609

[18] Neves, S., and Araujo, F., "Fast and Small Nonlinear Pseudorandom Number Generators for Computer Simulation", 2011.

[19] https://peteroupc.github.io/pseudocode.html

[20] Claessen, K, Pałka, M., "Splittable Pseudorandom Number Generators using Cryptographic Hashing", *ACM SIGPLAN Notices* 48(12), December 2013.

(The Claessen paper, section 5, also shows how a sequence of numbers can be generated from a state, essentially by hashing the path with the seed as the key, and in turn hashing a counter with that hash code as the key, but uses a rather complicated encoding to achieve this.)

The following helper method, in pseudocode, is used in the description above:

```
METHOD BitsToBytes(bits)
   // Unfortunately, the Claessen paper sec. 3.3 pads
   // blocks with zeros, creating a risk that different paths
   // encode to the same byte sequence (for example, <1100> vs.
   // <11000> or <0011> vs. <00011>). Even without this padding,
   // this risk exists unless the underlying hash function hashes
   // bit sequences (not just byte sequences), which is rare.
   // Therefore, encode the bits to a sequence of bytes
   // rather than using the encoding given in sec. 3.3.
   v = []
   for i in 0...size(bits): v = v || TOBYTES(bits[i], 1)
   return v
END METHOD
```

The splittable PRNG described in "**JAX PRNG Design**[21]" can be described as follows:

- A PRNG state is generated from a seed with `TOBLOCK(seed)`.
- `split` creates two new states from the old one; the first (or second) is a hash of `TOBLOCK(0)` (or `TOBLOCK(1)`, respectively) with the old state as the key.
- `generate` creates one or more random numbers by hashing `TOBLOCK(n)` with the state as the key, where `n` starts at 1 and increases by 1 for each new random number.

[21]https://github.com/google/jax/blob/master/design_notes/prng.md