Testing PRNGs for High-Quality Randomness

This version of the document is dated 2020-08-17.

Peter Occil

According to my document on **pseudorandom number generator (PRNG) recommendations**, a high-quality PRNG, among other requirements—

> generates bits that behave like independent uniform random bits (at least for nearly all practical purposes outside of information security)[,]

a requirement called the "independence requirement" in this short document.

To determine whether a PRNG meets the independence requirement, its output should be sent to the PractRand program by Chris Doty-Humphrey and show no failures ("FAILs") in the PractRand tests at 1 TiB (2^40 bytes) or greater. For more information, see "**How to Test with PractRand**" by M. E. O'Neill.

Random number streams. Many PRNGs use different strategies to produce nearby sequences (or *streams*) of pseudorandom numbers. But not every strategy produces *independent* streams. To determine whether nearby sequences of the PRNG meet the independence requirement, the output sent to PractRand should be formed by interleaving the outputs of those sequences (e.g., one output from the first sequence, one output from the second, another output from the first, another from the second, and so on).

There are several kinds of nearby sequences to test for this purpose:

- The original PRNG state, and the state produced by discarding a huge number of PRNG outputs in an efficient way ("**jump-ahead**"). That number can matter.
- Two or more PRNGs initialized with consecutive seeds.
- Two or more PRNGs initialized with seeds that differ from each other by one bit (see also "**The wrap-up on PCG generators**").

The *leapfrogging* technique involves assigning N processes each a PRNG that differs from the last by 1 step, then having each such PRNG advance N steps, where N is the number of PRNGs, each time it generates a random number. However, note that testing nearby sequences produced by leapfrogging is redundant with testing the regular PRNG sequence without streams.

Hash functions and counter-based PRNGs. In general, a *counter-based PRNG* produces pseudorandom numbers by transforming a seed and a *counter*; with each number, it increments the counter and leaves the seed unchanged (Salmon et al. 2011) **(1)**. The seed and counter can be transformed using block ciphers, other permutation functions, or hash functions. In general, counter-based PRNGs that use hash functions (such as MD5, SHA-1, MurmurHash, CityHash, xxHash) will meet the independence requirement if the following hash stream (for that hash function) doesn't fail the PractRand tests at 1 TiB or greater:

1. Write out the hash code of `seed || 0x5F || counter` (the `||` symbol means concatenation).
2. Write out the hash code of `(seed+1) || 0x5F || counter`.
3. Add 1 to `counter` and go to the first step.

In general, a hash function without PractRand failures is worthy of mention if it's noncryptographic and faster than hash functions designed for cryptography, such as MD5

and the SHA family.

Combined PRNGs. As G. Marsaglia (in KISS), D. Jones (in JKISS), and A. Fog (2015)[2] have recognized, combining two or more PRNGs of weaker quality often leads to a higher-quality PRNG. A PRNG that isn't high-quality could be converted to a high-quality PRNG in one of the following ways:

- If the PRNG has at least 128 bits of state and uses a *permutation function*[3] `P(x)` to transform that state, have the PRNG generate each number as follows instead:
  1. Add 1 (or another odd constant[4]) to the state (using wraparound addition).
  2. Output either `P(state)` or `S(P(state))`, where `S(x)` is one of the four *scramblers* defined in (Blackman and Vigna 2019)[5] (+, ++, *, **).
- If the PRNG admits 2^{63} or more seeds and outputs N-bit numbers, then each number it outputs can be *combined* with the next number from a sequence that cycles through at least 2^{128} numbers, to produce a new N-bit number. (These two numbers can be combined via XOR or wraparound addition if they have the same size, or via hashing.) This sequence can be one of the following:
  - A *Weyl sequence* (a sequence formed by wraparound addition of a constant odd number).
  - A *permutation function* of an incrementing counter that starts at 0.
  - A PRNG with a fixed seed and a single cycle of 2^{128} or more numbers, such as a linear congruential generator.
- If the PRNG admits 2^{63} or more seeds, has a minimum cycle length of 2^{128} or more, and outputs N-bit numbers, each number it outputs can be *combined* with the next number from another PRNG to produce a new N-bit number.
- If the PRNG has a single cycle of at least 2^{63}, admits that many seeds, and outputs N-bit numbers, each number it outputs can be *combined* with the next number from another PRNG to produce a new N-bit number.

Other combinations and transformations. There are other ways to combine two PRNGs, or to transform a single PRNG, but they are not preferred ways to build a *high-quality PRNG*. They include:

- Keeping some outputs and discarding others (as in RANLUX).
- The **Bays–Durham shuffle** (as in C++'s `shuffle_block_engine`).
- Transforming a PRNG's outputs with a permutation function (e.g., Mersenne Twister's "tempering").
- The "shrinking generator" technique, which takes each bit from one PRNG only if the corresponding bit from another PRNG is set (see (Cook 2019)[6]).
- "Self-shrinking" and von Neumann unbiasing, which each transform a PRNG based on pairs of output bits.

Splittable PRNGs. A *splittable PRNG* consists of two operations: a `split` operation to create multiple new internal states from one, and a `generate` operation to produce a pseudorandom number from a state (Schaathun 2015; Claessen et al., 2013)[7]. The Schaathun paper surveys several known constructions of splittable PRNGs. Some of the constructions can be used by any PRNG, but do not necessarily lead to high-quality splittable PRNGs.

The Schaathun paper suggests the following four random number sequences for testing purposes:

- Sequence suggested in section 5.5:
  1. Set `seed` and `g` to `split(seed)[0]` and `split(seed)[1]`, respectively.

    2. Set `t` to `split(split(g)[0])`, write out `generate(t[0])`, and write out `generate(t[1])`.
    3. Set `t` to `split(split(g)[1])`, write out `generate(t[0])`, and write out `generate(t[1])`.
    4. Go to the first step.
- Sequence SL (section 5.6): Set `seed` and `g` to `split(seed)[1]` and `split(seed)[0]`, respectively, and write out `generate(g)`. Go to the first step.
- Sequence SR (section 5.6): Set `seed` and `g` to `split(seed)[0]` and `split(seed)[1]`, respectively, and write out `generate(g)`. Go to the first step.
- Sequence SA (section 5.6):
    1. Set `seed` and `g` to `split(seed)[1]` and `split(seed)[0]`, respectively, and write out `generate(g)`.
    2. Set `seed` and `g` to `split(seed)[0]` and `split(seed)[1]`, respectively, and write out `generate(g)`.
    3. Go to the first step.

1 Notes

(1) Salmon, J.K.; Moraes, M.A.; et al., "Parallel Random Numbers: As Easy as 1, 2, 3", 2011.

(2) Agner Fog, "**Pseudo-Random Number Generators for Vector Processors and Multicore Processors**", *Journal of Modern Applied Statistical Methods* 14(1), article 23 (2015).

(3) A *permutation function* (or *bijection*) is a reversible mapping from N-bit integers to N-bit integers. Examples include: JSF64 by B. Jenkins; MIX and MIX-i (part of Tyche and Tyche-i); the Romu family by Mark Overton; block ciphers with a fixed key; mixing functions with reversible operations as described in "**Hash functions**" by B. Mulvey.

(4) As **P. Evensen shows**, the choice of constant can matter for a given permutation function.

(5) Blackman, D., Vigna, S., "Scrambled Linear Pseudorandom Number Generators", 2019.

(6) J. D. Cook, "Using one RNG to sample another", June 4, 2019.

(7) Schaathun, H.G. "Evaluation of Splittable Pseudo-Random Generators", 2015; Claessen, K., et al. "Splittable Pseudorandom Number Generators using Cryptographic Hashing", Proceedings of Haskell Symposium 2013, pp. 47-58.