LXM: Better Splittable Pseudorandom Number Generators (and Almost as Fast)

GUY L. STEELE JR., Oracle Labs, USA
SEBASTIANO VIGNA, Università degli Studi di Milano, Italy

In 2014, Steele, Lea, and Flood presented SplitMix, an object-oriented pseudorandom number generator (prng) that is quite fast (9 64-bit arithmetic/logical operations per 64 bits generated) and also *splittable*. A conventional prng object provides a *generate* method that returns one pseudorandom value and updates the state of the prng; a splittable prng object also has a second operation, *split*, that replaces the original prng object with two (seemingly) independent prng objects, by creating and returning a new such object and updating the state of the original object. Splittable prng objects make it easy to organize the use of pseudorandom numbers in multithreaded programs structured using fork-join parallelism. This overall strategy still appears to be sound, but the specific arithmetic calculation used for *generate* in the SplitMix algorithm has some detectable weaknesses, and the period of any one generator is limited to 2^{64}.

Here we present the LXM *family* of prng algorithms. The idea is an old one: combine the outputs of two independent prng algorithms, then (optionally) feed the result to a mixing function. An LXM algorithm uses a linear congruential subgenerator and an $\mathbf{F}_2$-linear subgenerator; the examples studied in this paper use a linear congruential generator (LCG) of period 2^{16}, 2^{32}, 2^{64}, or 2^{128} with one of the multipliers recommended by L'Ecuyer or by Steele and Vigna, and an $\mathbf{F}_2$-linear xor-based generator (XBG) of the `xoshiro` family or `xoroshiro` family as described by Blackman and Vigna. For mixing functions we study the MurmurHash3 finalizer function; variants by David Stafford, Doug Lea, and `degski`; and the null (identity) mixing function.

Like SplitMix, LXM provides both a *generate* operation and a *split* operation. Also like SplitMix, LXM requires no locking or other synchronization (other than the usual memory fence after instance initialization), and is suitable for use with simd instruction sets because it has no branches or loops.

We analyze the period and equidistribution properties of LXM generators, and present the results of thorough testing of specific members of this family, using the TestU01 and PractRand test suites, not only on single instances of the algorithm but also for collections of instances, used in parallel, ranging in size from 2 to 2^{24}. Single instances of LXM that include a strong mixing function appear to have no major weaknesses, and LXM is significantly more robust than SplitMix against accidental correlation in a multithreaded setting. We believe that LXM, like SplitMix, is suitable for "everyday" scientific and machine-learning applications (but not cryptographic applications), especially when concurrent threads or distributed processes are involved.

CCS Concepts: • **Mathematics of computing → Random number generation**; • **Computing methodologies → Parallel algorithms**.

Additional Key Words and Phrases: random number generator, pseudorandom, compound generator, mixing function, splittable, parallel, concurrent, DotMix, SplitMix, LXM, RNG, PRNG

ACM Reference Format:
Guy L. Steele Jr. and Sebastiano Vigna. 2021. LXM: Better Splittable Pseudorandom Number Generators (and Almost as Fast). *Proc. ACM Program. Lang.* 5, OOPSLA, Article 148 (October 2021), 31 pages. https://doi.org/10.1145/3485525

Authors' addresses: Guy L. Steele Jr., Oracle Labs, 35 Network Drive UBUR02-313, Burlington, Massachusetts, 01803, USA, guy.steele@oracle.com; Sebastiano Vigna, Università degli Studi di Milano, Italy, sebastiano.vigna@unimi.it.

2475-1421/2021/10-ART148
https://doi.org/10.1145/3485525

1 INTRODUCTION

Many algorithms used for physical simulations and machine learning, ranging from Monte Carlo methods to stochastic gradient descent, consume sequences of values that are either chosen "truly at random" (by some sufficiently unbiased physical process) or generated by a deterministic computation chosen in hopes that the computed values will "appear to be random," at least for the purposes of the specific application. Such computed sequences of values are called *pseudorandom.*

> *Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number—there are only methods to produce random numbers, and a strict arithmetic procedure is of course not such a method.* [von Neumann 1951]

Von Neumann made the preceding remark as part of his analysis of his *middle-square* method:

> *In obtaining y as the middle ten [decimal] digits in the square of a ten-digit number x, we are really mapping x onto y by a certain sawtoothed discontinuous curve $y = f(x)$, for $0 \leq x \leq 1$, $0 \leq y \leq 1$. When we take $x_{i+1} = f(x_i)$ for $i = 1, 2, 3, \ldots$, this curve will gradually scramble the digits of x_1 and produce something fairly pseudo-random.* [von Neumann 1951]

How can we know whether computed sequences "appear to be random"? By testing them:

> *A pseudo-random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians and depending somewhat on the uses to which the sequence is to be put.* [Lehmer 1951]

But what tests should be used?

> *We are here dealing with mere "cooking recipes" for making digits; probably they cannot be justified, but should merely be judged by their results. Some statistical study of the digits generated by a given recipe should be made, but exhaustive tests are impractical. If the digits work well on one problem, they seem usually to be successful with others of the same type.* [von Neumann 1951]

Indeed, the very idea of testing a sequence to see whether it is "random" is somewhat paradoxical:

> *... we would like to point out that all tests have a subjective element and that no test or sequence of such will establish a sequence of digits as being random.... if digits were generated by a [truly] random process then any method that rejected any sequence would be faulty.* [Hammer 1951]

Nevertheless, Forsythe did apply statistical tests to three variations of von Neumann's middle-square method; two failed quickly, and the third was then subjected to more rigorous tests:

> *The null hypothesis was formulated that, insofar as ordinary statistical tests can detect, method C is a process for generating independent, equidistributed digits. To test the null hypothesis, four χ^2 tests devised by Kendall and Babington Smith were applied [to a sequence of 100,000 generated digits]:*
> *(a) frequency test: frequency of $0, 1, 2, \ldots, 9$ in first 50,000 digits;*
> *(b) serial test: frequency of $00, 01, \ldots, 99$ in first 50,000 digits;*
> *(c) gap test: frequency of lengths of gaps between successive zero digits in entire sample;*
> *(d) poker test: frequency of combinations aaaa, aaab, aabb, aabc, abcd in all 25,000 groups of four consecutive digits.* [Forsythe 1951]

But while "method C" passed three of the tests, it failed test (b):

> *In summary, method C is not recommended for the generation of random digits, because the distribution of pairs of digits appears to be variable and not uniform.* [Forsythe 1951]

The evolution of pseudorandom number generator (PRNG) algorithms over the last seven decades may be fairly characterized as a competition between generators and tests, driven in part by algorithmic improvements and in part by Moore's Law. Ideally, each new value should be *generated* in polynomial time, but the generator output should not be *predictable* in polynomial time [Blum and Micali 1984]; there are generators satisfying this definition, but they are computationally expensive and they depend presently on hardness assumptions [Blum et al. 1986]. In practice one often trades quality for speed; statistical testing helps to avoid trading away too much quality.

There are also indirect methods for judging the quality of a PRNG algorithm, involving mathematical analysis of the code rather than running the code and testing the output. One of the earliest and most successful is the *spectral test* [Coveyou and Macpherson 1967; see also Knuth 1998, §3.3.4], which is applicable to what we now call *linear congruential generators* [Lehmer 1951; Rotenberg 1960; Thomson 1958], in which the state of the generator is advanced by performing integer multiplication by a fixed constant and then optionally adding another fixed constant. Marsaglia [1968] observed that all purely multiplicative congruential PRNG algorithms have unexpected biases when used to generate n-tuples of values; the spectral test quantified this bias, allowing tabulation of multipliers that minimize this bias [L'Ecuyer 1999b; Steele and Vigna 2021]. These results generalize to all linear congruential generators.

Some generators, instead, treat their state as a vector of values in the field $\mathbf{F}_2$ (multiplication is matrix-vector multiplication, often optimized by choosing the constant matrix so as to allow implementation using only a small number of SHIFT and XOR instructions), and these are called $\mathbf{F}_2$-linear generators. These have their own rich history, including Linear Feedback Shift Register techniques [Golomb 2006, 2017; Goresky and Klapper 2012; Wolfram 2016] and Marsaglia's "xorshift generators" [Brent 2004; Marsaglia 2003]. More recent examples are the `xoroshiro` and `xoshiro` algorithms [Blackman and Vigna 2018; Vigna 2014–2021], which we build upon in this paper. Briefly put, they provide a systematic way to derive good $\mathbf{F}_2$-linear functions that can be computed cheaply using only XOR, SHIFT, and ROTATE instructions; some versions use a "scrambler" (a fast output function, weaker than a full-blown hash function) to compensate for specific small flaws.

The appeal of linear generators of either kind is that they are much more amenable to mathematical analysis. This is a double-edged sword: it is easier to prove certain favorable properties but also easier to expose deficiencies.

Researchers began to explore *compound generators*. MacLaren and Marsaglia [1965] used one linear generator to "shuffle" the sequence produced by another. A more recent and widely used example is `MRG32k3a` [L'Ecuyer 1999a], which additively combined the outputs of two generators that have the same algorithmic structure but different moduli. There was also limited exploration of compounds whose component generators are of different kinds [L'Ecuyer and Granger-Piché 2003], for example one using integer multiplication and one using bit-matrix multiplication.

> *Congruential generators are liked, but not well-liked; . . . combination generators seem best—if the numbers are not random, they are at least higgledy piggledy.* [Marsaglia 1985]

In the meantime, extensive test suites were developed, starting in the 1990s: Diehard [Marsaglia 1995; Marsaglia and Tsang 2002], TestU01 [L'Ecuyer and Simard 2007, 2013; Simard 2009], Dieharder [Brown et al. 2003–2006], and the NIST Statistical Test Suite [Rukhin et al. 2001, 2010]. (L'Ecuyer has recently provided a detailed history of RNG algorithms and testing procedures [L'Ecuyer 2017].)

With the emergence of multiprocessor (and later, multicore) computer architectures, users needed PRNG algorithms for multithreaded computations. If you simply use one copy of a specific algorithm on each processor, there is a risk that the generated sequences may be statistically correlated in some way. One approach is to use an algorithm that can generate an extremely long sequence, then make sure that each processor uses a different subsequence of that long sequence. This led to the idea of a *jump function* that can quickly compute some far-distant future element of the sequence. (One strength of linear generators is that it is relatively easy to construct such jump functions.)

Another approach to parallel PRNG construction is to break the problem into two parts: first, make sure that the processors generate sequences that are *distinct* but not necessarily "random," then hash each generated value. An extreme variation (PHILOX) is to give every processor a counter, initialize the counters to widely spaced integers (the "jump function" is trivial), then apply a (perhaps cryptographically strong) hash function to the successive integers produced by each counter [Salmon et al. 2011]. Another approach (DotMix) is to maintain a "pedigree" vector that is guaranteed to be different for every pseudorandom value to be generated (the vector is extended every time a parallel computation is forked, and the last entry is incremented as necessary); pseudorandom values are produced by hashing the vector [Leiserson et al. 2012]. The SplitMix algorithm, which began as an attempt to optimize the performance of DotMix, uses Weyl generators (counters modulo 2^w, where each increment is an odd number and the increments are likely different for each processor) and then applies a hash function that is only moderately strong but much faster [Steele, Lea, and Flood 2014]; SplitMix had a known weakness that certain unfortunate choices for the increment might result in sequences that fail the test suites, but the designers described and implemented a technique for avoiding these poor choices in practice.[1] The SplitMix algorithm was implemented as class `SplittableRandom` [Oracle Corporation 2014b] in the library for the Java® programming language as part of Java Development Kit 8 (JDK8).

PHILOX was tested using TestU01; DotMix was tested using Dieharder; SplitMix was tested using both. In the last decade another test suite, PractRand [Doty-Humphrey 2011–2021], has emerged, that has largely superseded Dieharder and is especially useful as a complement to TestU01 because PractRand is extremely fast, has the ability to "fail early," and can be run as long (or as short) as desired, producing ever more precise results as it goes.

I think nobody who is practically concerned will want to use a sequence produced by any method without testing it statistically, and it has been the uniform experience with those sequences that it is more trouble to test them than to manufacture them. Hence the degree of complication of the method by which you make them is not terribly important; what is important is to carry out a relatively quick and efficient test. [von Neumann 1951]

It is now standard practice to use PractRand to test at least 4 terabytes (perhaps even 16 terabytes) of PRNG output. An algorithm that passes that test may then be given to TestU01 for final testing.

Computers are now fast enough, and have enough main memory, that it is feasible to perform extremely sensitive collision tests [Knuth 1998, §3.3.2.I] that use billions of bins.

In this paper, we return to the idea of constructing compound generators by adding outputs from two simple generators of different types: a linear congruential generator and a `xo(ro)shiro` generator. It turns out that PractRand can detect flaws in the low bits of these outputs, so we furthermore use a mixing function of the same kind used by SplitMix. The result is the LXM family of algorithms presented here, which we have tested using both PractRand and TestU01.

The LXM algorithm is a fairly simple idea that combines building blocks already in the literature in ways already studied in the literature—yet this precise combination seems not to have been

[1]The Weyl generator used in SplitMix is, in effect, a trivial linear congruential generator that "multiplies by 1" and then adds a constant γ; the generated sequence is $x_k = (k\gamma + x_0) \bmod 2^w$, which is not at all random. The quality of SplitMix output depends almost entirely on the quality of its mixing function, which can turn that linear sequence into a reasonably random-looking sequence *unless* γ is so poorly chosen that consecutive values x_j and x_{j+1} are the same in almost all bit positions—in that case, the mixing function isn't quite good enough to map consecutive Weyl-generator outputs to sufficiently different final values. This can happen in two different ways: (a) γ is almost all 0-bits or almost all 1-bits—then adding γ to x_j doesn't change very many bits; (b) the low k bits of γ and the low k bits of `γ >>> k` are very similar, where k is the shift distance used in the first step `x ^= (x >>> k);` of the mixing function—in that case, adding γ to x_k will tend to change two groups of bits in much the same way, and then the SHIFT-XOR step of the mixing function will tend to *cancel* that change in the k low-order bits. The remaining steps of the mixing function are not good enough to compensate for this effective "lessening of randomness" in the first step. A further wrinkle is that such weaknesses (of either type, (a) or (b)) can also occur when γ itself does not have either of these properties but some small integer multiple of γ does; for example, if 3γ is almost all 0-bits, then for every k, x_k and x_{k+3} will likely be so similar that the mixing function can't map them to sufficiently different final values. The designers of SplitMix anticipated case (a), but not case (b) or the further wrinkle about small multiples. It is possible to detect and avoid these other cases, but that would make the code more complicated.

previously studied systematically or put into widespread practice. The principal contributions of this paper are explaining why specific components were chosen and why they were combined in a specific way, analyzing certain properties of the combination, comparing this structure to prior work, and empirically probing for weaknesses through detailed quality tests and timing tests.

Section 2 describes the structure of the LXM algorithm in pragmatic terms and presents Java code for two versions. Section 3 explains how the *split* operation is performed for LXM. Section 4 defines special notations and terminology used in this paper. Section 5 presents a more mathematical description of the LXM algorithm, and Section 6 discusses properties of the algorithm, such as period and equidistribution. Section 7 presents results of testing for statistical quality; Section 8 presents timing tests for both LXM and SplitMix. Section 9 goes into more detail about how to split and jump LXM generators. Section 10 provides advice on choosing a PRNG algorithm for a specific application. Related work is cited in Section 11; conclusions are in Section 12.

2 THE LXM GENERATION ALGORITHM

A member of the LXM family of algorithms for word size w (where w is any non-negative integer, but typically either 64 or 32) consists of four components:

- L: a linear congruential pseudorandom number generator (LCG) with a k-bit state s, $k \geq w$
- X: an $\mathbf{F}_2$-linear [L'Ecuyer and Panneton 2009] pseudorandom number generator (we use the term XBG, for "XOR-based generator") with an n-bit state x, $n \geq w$
- a simple combining operation on two w-bit operands that produces a w-bit result
- M: a bijective mixing function that maps a w-bit argument to a w-bit result

The combining operation should have the property that if either argument is held constant, the resulting one-argument function is bijective; typically it is either binary integer addition '+' or bitwise XOR '$\oplus$' on w-bit words. In most practical applications k and n are integer multiples of w.

The *generate* operation for an LXM generator is described by the following pseudocode, where global variable s is the LCG state (a non-negative integer), global variable t is the XBG state (a bit vector), multiplier m is an integer such that $(m \bmod 8) = 5$, additive constant a is an odd integer, and update matrix U is an $n \times n$ matrix of bits. Elements of the product of matrix U and a bit vector of length n are computed in the two-element field $\mathbf{F}_2$ (addition is XOR). In practice, U is chosen so that such matrix-vector products can be computed by using a small number of instructions such as XOR, SHIFT, and ROTATE operating on w-bit words.

```
$generate()$ :
    $z \leftarrow mix(combine(w \text{ high-order bits of } s, w \text{ bits of } t))$
    $s \leftarrow LCG\_update(s)$
    $t \leftarrow XBG\_update(t)$
    return $z$

$LCG\_update(s)$ : return $(ms + a) \bmod 2^k$

$XBG\_update(t)$ : return $Ut$
```

This pseudocode uses the standard trick of using the *old* state values of the subgenerators to compute the result to be returned; this allows the state updates for the two subgenerators to be overlapped or interleaved not only with each other but with the computation of the combining and mixing functions, which may be advantageous on processors that can execute multiple instructions concurrently.

Figure 1 shows a specific implementation in the Java programming language of the *generate* operation for $w = 64, k = 64, m = 128$. The period of the LCG is 2^{64}. The XBG is `xoroshiro128` version 1.0 [Blackman and Vigna 2018], which has a period of $2^{128} - 1$. The combining function is binary addition. The mixing function is a variant of the MurmurHash3 mixing function [Appleby

```
private static final long M = 0xd1342543de82ef95L;          // Fixed multiplier
private final long a;     // Per-instance additive parameter (must be odd)
private long s, x0, x1;   // Per-instance state (x0 and x1 are never both zero)

public long nextLong() {
    // Combining operation
    long z = s + x0;
    // Mixing function (lea64)
    z = (z ^ (z >>> 32)) * 0xdaba0b6eb09322e3L;
    z = (z ^ (z >>> 32)) * 0xdaba0b6eb09322e3L;
    z = (z ^ (z >>> 32));
    // Update the LCG subgenerator
    s = M * s + a;
    // Update the XBG subgenerator (xoroshiro128v1_0)
    long q0 = x0, q1 = x1;
    q1 ^= q0;
    q0 = Long.rotateLeft(q0, 24);
    q0 = q0 ^ q1 ^ (q1 << 16);
    q1 = Long.rotateLeft(q1, 37);
    x0 = q0; x1 = q1;
    // Return result
    return z;
}
```

Fig. 1. Java code for the *generate* operation of an LXM generator with period $2^{64}(2^{128}-1)$

2011, 2016] identified by Doug Lea. The additive parameter `a` may be initialized to any odd integer, and the state variables `s`, `x0`, and `x1` may be initialized to any values as long as `x0` and `x1` are not both zero. Because the periods of the subgenerators are relatively prime, the overall period of this LXM generator is $2^{64}(2^{128}-1) = 2^{192} - 2^{64}$.

Figure 2 shows a second specific implementation, this time for $w = 64, k = 128, m = 256$. It uses the same 64-bit mixing function but uses a different (256-bit) XBG, `xoshiro256` [Blackman and Vigna 2018]. It also illustrates some interesting engineering tradeoffs when implementing a 128-bit LCG using 64-bit arithmetic. Computing the (128-bit) low half of two 128-bit operands requires computing the 128-bit product of the (64-bit) low halves, plus the (64-bit) low halves of products of two pairs of 64-bit values, each consisting of the high half one of 128-bit operand and the low half of the other. But testing seems to show that there is little extra benefit of using a 128-bit multiplier over a 65-bit multiplier; on the other hand, theory tells us that a 64-bit multiplier will produce an LCG of lower quality [Steele and Vigna 2021]. Therefore we choose to use a multiplier of the form $2^{64}+m$ where $m < 2^{64}$ and of course $(m \bmod 8) = 5$; this eliminates one 64-bit multiplication in the implementation. On the other hand, there *is* a benefit to be gained by using a full 128-bit additive parameter rather than settling for 64 bits. The code uses two `long` values `ah` and `al` to represent the high and low halves of the additive parameter, and similarly uses two `long` values `sh` and `sl` to represent the high and low halves of the LCG state. (Because Java has not yet implemented the method `Math.unsignedMultiplyHigh`, code for this operation is included in Figure 2, using the technique described in *Hacker's Delight* [Warren 2012, §8.3, p. 175].)

These implementations, and some others, are scheduled to be incorporated into a new package `java.util.random` as part of JDK17. This package will also include a new API intended to better

```
private static final long ML = 0xd605bbb58c8abbfdL; // Low half of fixed multiplier
private final long ah, al;      // Per-instance additive parameter (al must be odd)
private long sh, sl, x0, x1, x2, x3; // Per-instance state (x0, x1, x2, x3 not all 0)

private long unsignedMultiplyHigh(long a, long b) {
    return Math.multiplyHigh(a, b) + ((a >> 63) & b) + ((b >> 63) & a);
}

public long nextLong() {
    // Combining operation
    long z = sh + x0;
    // Mixing function (lea64)
    z = (z ^ (z >>> 32)) * 0xdaba0b6eb09322e3L;
    z = (z ^ (z >>> 32)) * 0xdaba0b6eb09322e3L;
    z = (z ^ (z >>> 32));
    // Update the LCG subgenerator
    // The LCG is, in effect, "s = m * s + a" where m = ((1LL << 64) + ML)
    final long u = ML * sl;
    sh = (ML * sh) + unsignedMultiplyHigh(ML, sl) + sl + ah;    // High half
    sl = u + al;                                                // Low half
    if (Long.compareUnsigned(sl, u) < 0) ++sh;          // Carry propagation
    // Update the XBG subgenerator (xoshiro256 1.0)
    long q0 = x0, q1 = x1, q2 = x2, q3 = x3;
    long t = q1 << 17;
    q2 ^= q0; q3 ^= q1; q1 ^= q2; q0 ^= q3; q2 ^= t;
    q3 = Long.rotateLeft(q3, 45);
    x0 = q0; x1 = q1; x2 = q2; x3 = q3;
    // Return result
    return z;
}
```

Fig. 2. Java code for the *generate* operation of an LXM generator with period $2^{128}(2^{256} - 1)$

support interchangeable use of various PRNG algorithms within an application. The centerpiece is a new interface `RandomGenerator`, which provides default implementations for many standard methods such as `nextFloat()`, `nextDouble()`, `nextGaussian()`, `ints()`, and `longs()`, provided only that any class that implements the interface must provide a method `period` (for reporting the length of the state cycle) and either a `nextLong()` method (for generating a pseudorandomly chosen 64-bit integer) or a `nextInt()` method (for generating a pseudorandomly chosen 32-bit integer). Other new interfaces support the possibility that a specific PRNG algorithm may provide a `jump()` method (for advancing a large distance along the state cycle) or a `split()` method (for creating a new generator from an existing one, as described by Steele, Lea, and Flood [2014]).

3 LXM IMPLEMENTATION OF SPLITTING

The motivation for the `split` operation is that each distinct instance of a splittable PRNG, once created, accesses only its own internal state. Instances of LXM do not communicate and do not share any state; therefore no synchronization is required. Methods such as `longs` that can generate streams of values in parallel rely on the standard Java spliterator mechanism [Oracle 2014]. The point of a spliterator is that it can (try to) split itself into two substreams that may then be processed in

```
public SplittableGenerator split() {
    return new L64X128MixRandom(this.nextLong(), this.nextLong(),
                                this.nextLong(), this.nextLong());
}
```

Fig. 3. Java code for the *split* operation of an LXM generator with period $2^{64}(2^{128}-1)$

parallel. The spliterator for a stream that generates values from a splittable PRNG also automatically splits the PRNG so that each substream will have its own PRNG instance rather than trying to share the same one. It is this step that eliminates the need for any locking or other synchronization in the parallel execution of stream expressions that use a splittable PRNG. This strategy is exactly that already used for class `SplittableRandom` [Steele, Lea, and Flood 2014, §2.2]. Here we describe the LXM implementation of the `split` method and the new `splits` method.

3.1 The Split Operation

Creating a new instance of an LXM algorithm from an existing one is done in a straightforward way: the `nextLong()` or `nextInt()` method of the existing one is used to generate values for the state variables of the LCG and XBG subgenerators and for the additive parameter of the LCG. See Figure 3. The constructor then forces the additive parameter to be odd by setting its low-order bit to 1, but beyond that no additional vetting of the additive parameter (to reject "weak values" [Steele, Lea, and Flood 2014]) is necessary. In the unlikely circumstance that the state for the XBG subgenerator is entirely 0, it is necessary to force it to be nonzero; this could be done by making additional calls to `nextLong()` or `nextInt()`, but it is easier (and acceptable in practice, because it happens so rarely) to derive the new XBG state from the new LCG state.

3.2 The Splits Operation

Existing JDK PRNG implementations, such as classes `Random` and `SplittableRandom` [Oracle Corporation 2014a,b], provide methods such as `ints()`, `longs()`, and `doubles()` that produce *streams* of pseudorandomly chosen values. JDK17 introduces a new method `rngs()` that produces a stream of PRNG instances; one can then use the `map` method of the stream to execute a piece of code many times, perhaps in parallel, each with its own PRNG instance so that there is no competition for a shared resource (such as a single, shared PRNG). PRNG algorithms that have a `jump()` method may also provide a `jumps()` method that is then automatically used to implement the `rngs()` method by jumping along the state cycle multiple times. On the other hand, PRNG algorithms that have a `split()` method may also provide a `splits()` method that is then automatically used to implement the `rngs()` method by using the `split()` method multiple times—but with a bit of cleverness. The details of the technique are outside the scope of this paper, which focuses on how values are generated, and why; but we touch on it briefly in Section 9.

4 NOTATION AND TERMINOLOGY

We use the standard lambda notation $\lambda x.e$ to denote a function that takes one argument and returns the value produced by the expression e with the parameter x bound to the given argument. If the argument is expected to be a tuple, we use a "nested destructuring parameter binding" notation; for example, if the argument is expected to be a 2-tuple containing a number and a 3-tuple, we could use a notation such as $\lambda\big(n,(x,y,z)\big).e$. In this paper we usually choose to use Greek letters such as σ and τ to name parameters.

We work with vectors and matrices whose elements are taken from the two-element field $\mathbf{F}_2$ (also known as $\mathbf{GF}(2)$ and $\mathbf{Z}/2\mathbf{Z}$). We casually refer to the elements of such vectors and matrices as *bits*, and we use both the symbol $\oplus$ and the name XOR to refer to addition within this field. We refer to elements and subvectors of a bit vector v by using brackets with 0-origin indexing, for example $v[i]$ or $v[i \,..\, j]$; "$[i \,..\, j]$" (where $i \leq j$) denotes subscripting by a range of integers i to j, inclusive. Where necessary, we assume that any integer j in the range $[0 \,..\, 2^w)$ (inclusive of 0, exclusive of 2^w) may be implicitly treated as a bit vector v of length w, and vice versa, by satisfying the relationship $j = \sum_{i=0}^{w-1} v[i]2^i$ (where $v[i]$ is implicitly converted to an integer 0 or 1 before multiplying by 2^i).

Let S and T be finite sets of values; we will also refer to S and T as *types*, in the sense that the value of a variable of type S must be an element of S, and similarly for T.

For our purposes, a PRNG with states of type S and outputs of type T is a triple (s_0, f, g) where $s_0 \in S$ is the *initial state*, $f : S \longrightarrow S$ is a bijective function on states, and $g : S \longrightarrow T$ is a function from states to outputs. Such a generator produces a *sequence of states* $s_0, s_1, s_2, \ldots$ defined by the recurrence $s_i = f(s_{i-1})$ for all $i > 0$; it also produces a *sequence of outputs* $t_0, t_1, t_2, \ldots$ such that for all $i \geq 0$, $t_i = g(s_i)$. Thus for all $i \geq 0$, $t_i = g(f^i(s_0))$.

Because S is finite, these sequences are *periodic*; because f is bijective, the sequence does not have a nonempty initial subsequence before commencing the periodic behavior. The *period* of the generator is the smallest $P > 0$ for which $s_P = s_0$; it follows that for all nonnegative integers i and k, $s_{i+kP} = s_i$ (and therefore $t_{i+kP} = t_i$). We sometimes refer to the finite cyclic sequence $s_0, s_1, \ldots, s_{P-1}$ as the *state cycle* of the generator; the *length* of this cycle is the period P.

We use V to refer to the bag (multiset) of outputs generated during one period of the generator, that is, $V = \wr\, t_i \mid 0 \leq i < P \,\int$. We sometimes regard this multiset as a function $V : T \longrightarrow \mathbb{N}$ that maps each element of T to the number of times that value occurs in the multiset; in other words, it is the number of times that that value appears within any length-P subsequence of the sequence of outputs. The *size* of the multiset V, written $|V|$, is defined to be $\sum_{v \in T} V(v)$; it follows that $|V| = P$.

Sometimes a PRNG with outputs of type T is regarded as a PRNG with outputs of type T^j for some $j > 0$—that is, as generating tuples of length j, where each element of the tuple is of type T. If the underlying PRNG of type T is the triple (s_0, f, g), then the alternate view may be described by the derived triple $\big(((t_0, t_1, \ldots, t_{j-1}), s_{j-1}),\ \lambda\big((\tau_0, \tau_1, \ldots, \tau_{j-1}), \sigma_{j-1}\big).\big((\tau_1, \ldots, \tau_{j-1}, g(f(\sigma_{j-1}))), f(\sigma_{j-1})\big),$ $\lambda\big((\tau_0, \tau_1, \ldots, \tau_{j-1}), \sigma_{j-1}\big).(\tau_0, \tau_1, \ldots, \tau_{j-1})\big)$. In other words, the generated tuples are the (overlapping) length-j subsequences of the output sequence of the underlying PRNG. Note that the PRNG of tuples has the same period as the underlying PRNG.

In some prior literature, a PRNG with outputs of type T is described as "equidistributed" if the multiset V of values generated during each period has the property that for any two values x and y of type T, $|V(x) - V(y)| \leq 1$; that is, the generated values are distributed "as equally as possible" over the values of type T. More generally, a PRNG is described as "j-dimensionally equidistributed" if it is equidistributed when regarded as a generator of j-tuples as described above. Note that being 1-dimensionally equidistributed is the same as being equidistributed.

We introduce here a somewhat more detailed terminology: we say that a PRNG that generates values of type T is *δ-distributed* if for any two values x and y of type T, $|V(x) - V(y)| \leq \delta \left\lceil \frac{|V|}{|T|} \right\rceil$. (Omitting the ceiling brackets would make this definition slightly tighter, but including them allows a more concise form for the δ values that is more convenient in practice for purposes of comparison.) Since smaller values of δ are better, we will normally in each case cite the smallest possible value of δ, and if $\delta = 0$, we will say that the PRNG is *exactly equidistributed.* More generally, we will say a PRNG is *j-dimensionally δ-distributed* if it is δ-distributed when regarded as a generator of j-tuples; but if $\delta = 0$, we will say that the PRNG is *exactly j-dimensionally equidistributed.*

5 THEORETICAL CONSTRUCTION OF THE LXM ALGORITHM

We define an LCG with state size k such that $k \geq 3$, multiplier m such that $(m \bmod 8) = 5$, additive parameter a such that $1 \leq a < 2^k$ and a is odd, initial state s_0 such that $0 \leq s_0 < 2^k$, and output size w such that $0 \leq w \leq k$, as the triple $L = \big(s_0, \lambda\sigma.(m\sigma + a) \bmod 2^k, \lambda\sigma.\left\lfloor \sigma/2^{k-w} \right\rfloor\big)$, and we write $t_0, t_1, t_2, \ldots$ to refer to its outputs.

We define an XBG with state size n, n-by-n bit matrix U, initial state x_0 where x_0 is an n-bit vector, and output size w such that $0 \leq w \leq n$ as the triple $X = (x_0, \lambda\tau.U\tau, \lambda\tau.\tau[0\,..\,w-1])$, and we write $y_0, y_1, y_2, \ldots$ to refer to its outputs, where $\tau[0\,..\,w-1]$ produces a w-bit vector containing the first w bits of τ. (We use the first w bits of τ without loss of generality, because one can create an equivalent XBG that delivers any desired size-w subset of the state bits, in any order, by using some single fixed permutation to reorder the bits of the initial state and also to reorder both the rows and columns of the matrix U.)

Given such an LCG and XBG, a binary combining operation on w-bit values $\circledast$ (which is typically either + or $\oplus$), and a bijective mixing function μ on w-bit values, an LXM generator is the triple $G = \big((s_0, x_0), \lambda(\sigma, \tau).((m\sigma + a) \bmod 2^k, U\tau), \lambda(\sigma, \tau).\mu(\left\lfloor \sigma/2^{k-w} \right\rfloor \circledast \tau[0\,..\,w-1])\big)$. It is easy to see that the set of possible states of the LXM is the cross product of the sets of states of the LCM and XBG; that the state-update function for the LXM simply pairs an update of the LCG with an update of the XBG; and that the output function combines an output of the LCG with a corresponding output of the XBG and then applies the mixing function.

The reader may wonder, given that the state update function of the LCG uses an affine transformation $m\sigma + a$, why the state update function of the XBG does not more generally use an affine transformation $U\tau \oplus v$. The answer has more to do with engineering than theory; we address it below in Sections 6.5.2 and 6.5.3.

6 PROPERTIES OF THE LXM ALGORITHM

In this section we discuss some properties of the LXM algorithm and how they derive from properties of its components. First we provide brief answers to some obvious questions; the subsections that follow elaborate on these answers.

Why use two subgenerators? The usual reasons: each is fairly small and fast, and they are chosen so that the period of the LXM generator will be the product of their individual periods.

Why use an XBG for one subgenerator? XGBs are fast; they are already widely used to produce pseudorandom sequences of fairly good quality; they have a well-understood theory, including for which k they are k-dimensionally equidistributed; and it is easy to scale their state size.

Why use an LCG? An LCG whose period is a power of 2 provides exact equidistribution, and preserves any k-dimensional equidistribution contributed by the XBG. An LCG is fairly fast, and uses hardware resources (multiply and add) that may be different from those needed by the XBG. The LCG provides an easy way to provide an additive parameter. Finally, mixing two generators based on different algebraic operations may improve the quality of a PRNG.

Why have an additive parameter? Additive parameters are an alternative to using jump functions to ensure non-overlap of multiple sequences, but are faster, easier to use, and easier to code.

Why use a nonlinear mixing function? The graph of every LCG with the same multiplier m has the same shape, even if they have different additive parameters. A similar remark is true of a generalized form of XBG. Changing the parameter just shifts (and perhaps flips) the graph. Therefore the graph of the combined LCG/XBG part of LXM also always has the same shape (more precisely, one of two shapes). A good mixing function reacts nonlinearly to the additive parameter (as well as to more subtle linear correlations within the subgenerators). Testing confirms that a good mixing function appears to make different streams relatively uncorrelated, but we don't have a theoretical proof.

6.1 Period

A well-known fact about LCGs of period 2^k is that for all $0 \le j < k$, the sequence of bits consisting of successive values of bit j of the overall state (where the least significant bit is bit 0 and the most significant bit is bit $k-1$) has period 2^{j+1}. Therefore the most significant bit has period 2^k. It follows trivially that the sequence of w-bit values consisting of successive values of bits $k-1$ through $k-w$ of the overall state has period 2^k.

The XBG subgenerator of an LXM algorithm is always chosen so that the sequence of w-bit values consisting of successive values of a specific set of w bits within the n bits of state has an odd period P. Because any odd number is relatively prime to any power of 2, the overall period of an LXM generator will be $2^k P$. Note that the various `xoroshiro` and `xoshiro` algorithms each have the maximum possible period, $2^n - 1$, so an LXM algorithm that uses one of these generators as its XBG subgenerator will have period $2^k(2^m - 1)$.

6.2 Scalability of Period

The parameters k (size of LCG state) and n (size of XBG state) may be varied independently. When k is made very large, the cost of the multiplication operation grows quadratically (there are subquadratic multiplication algorithms, but they are not cost-effective for values of k within the range of currently practical interest), so if a larger period is desired, it may be preferable to increase n rather than k. Fortunately the `xoroshiro` family of XBG generators easily grows to support state sizes $2w$, $4w$, $8w$, $16w$, and beyond without a significant increase in computational cost per value generated (though for the specific sizes $4w$ and $8w$, the `xoshiro` algorithm may be preferable). For $w = 64$ (the sweet spot for many of today's microprocessors), practical choices for k are 64 or 128 and for n include 128, 256, 512, and 1024, supporting periods ranging from $2^{192} - 2^{64}$ to $2^{1152} - 2^{128}$. For $w = 32$ (a sweet spot for smaller processors used in embedded applications), $k = 32$ and $w = 64$ may be a good choice (period $2^{96} - 2^{32}$).

6.3 Probability of Overlapping Sequences

Given a PRNG algorithm with a single state cycle of period P, suppose that we choose two distinct positions on the cycle literally uniformly at random, and then for each one consider the sequence of length ℓ consisting of the state at that position and the $\ell-1$ states following it. What is the probability that the two sequences will overlap? We care about this because long overlapping subsequences will produce highly correlated (indeed, identical) outputs that would not be characteristic of sequences of values chosen truly at random.

By symmetry, without loss of generality we may assign the first chosen position q_1 the index ℓ, and then choose the second position q_2 uniformly at random from the range of integers $[0 \,..\, P-1]$. Overlap occurs if and only if $1 \le q_2 \le 2\ell - 1$. The number of choices that allow overlap is $2\ell - 1$, so the probability of overlap is $(2\ell - 1)/P$.

Now suppose instead of one big state cycle of period P, we have A distinct state cycles of period P/A, and we do the following process twice: first choose a state cycle uniformly at random, then choose a position on that state cycle uniformly at random, then consider a state sequence of length ℓ starting at that position. The two sequences can overlap only if they lie on the same state cycle (probability $1/A$); if they do, the probability of overlap is $(2\ell - 1)/(P/A)$ as before, so the overall probability is $(2\ell - 1)/A(P/A) = (2\ell - 1)/P$. Thus this intuition: breaking the big state cycle up into equal-sized pieces does not affect the probability of overlap.

In LXM, the effect of having an additive parameter in the LCG is to select one of a number (typically 2^{w-1} or 2^{k-1}) of state cycles (though, as we discuss below in Section 6.5.1, these state cycles are not terribly different), each of period $2^k(2^n - 1)$. The point we wish to make here is that bits in the additive parameter are just as effective as bits in the LCG state or the XBG state in reducing the probability of overlap, except for the fact that the lowest bit of an additive parameter is "wasted" because it must be 1. As an example, let's compare an LXM algorithm $L_1$with $k = 64$ and $n = 128$ with a modified LXM algorithm L_2 with $k = 128$ and $n = 128$ but the additive parameter is 1 in every instance. Each instance of L_1 has 64 bits of LCG state, a 64-bit additive parameter, and 128 bits of XBG state. Each instance of L_2 has 128 bits of LCG state and 128 bits of XBG state, and it needs no per-instance storage for the constant additive parameter. So the per-instance storage for each of L_1 and L_2 is 256 bits. For L_2, the probability of overlap is $(2\ell-1)/(2^{128}(2^{128}-1)) \approx (2\ell-1)/2^{256}$; for L_1, the probability of overlap is $(2\ell-1)/(2^{63}2^{64}(2^{128}-1)) \approx (2\ell-1)/2^{255}$, which is the same except for that one wasted bit. If we let $\ell = 2^{50}$ and create 2^{32} instances of L_1, initializing their states and additive parameters truly at random, then the chances that two of them will have the same additive parameter are fairly high, thanks to the Birthday Paradox (choosing 2^{30} values with replacement from a set of 2^{63} items), but the probability of any pair of instances overlapping is roughly 2^{-172}, and the probability that some pair out of the 2^{32} instances will overlap is roughly 2^{-140} (because 2^{32} is quite small compared to 2^{172}, the effect of the Birthday Paradox can be neglected).

It follows that, *under the crucial assumption that initializing the state of newly created instances using the output of a* PRNG *is sufficiently close to truly random for this purpose*, we can be confident that instances produced by the `split()` operation described in Section 3.1 are highly likely to avoid unwanted correlation due to accidental sequence overlap, and we can increase our confidence either by increasing the size of the XBG state, increasing the size of the LCG state, and/or increasing the number of bits in the additive parameter (remembering that this last size cannot usefully exceed the size of the LCG state).

6.4 Equidistribution

A k-bit LCG of period 2^k produces each possible k-bit value exactly once during each cycle, so it is exactly equidistributed. The high-order w bits of the output are likewise exactly equidistributed; each of the 2^w distinct values is produced 2^{k-w} times during the cycle.

An n-bit XBG of period $2^n - 1$ produces each w-bit value 2^{n-w} times, except that there is one value, typically 0, that is produced only $2^{n-w} - 1$ times. Such a generator is $2^{-(n-w)}$-distributed. For example, for $w = 64$, the `xoroshiro128` algorithm ($n = 128$) is 2^{-64}-distributed, and the `xoshiro256` algorithm ($n = 256$) is 2^{-192}-distributed.

An LXM algorithm that combines two such subgenerators is *exactly* equidistributed, because each position in the period of the LCG "meets" (and is therefore combined with) each position in the period of the XBG exactly once during the period of the LXM generator, so for every position in the XBG cycle, the w-bit value in that position has added to it every possible w-bit value exactly 2^{k-w} times. (Applying a bijective mixing function leaves equidistribution qualities unaffected.)

If the n-bit XBG of period $2^n - 1$ is (n/w)-dimensionally equidistributed—that is, using groups of n/w successive outputs to form (n/w)-tuples results in generating every possible tuple except one (call it Z, because it is typically the all-0 tuple), exactly once—then an LXM generator for which $k = w$ is also (n/w)-dimensionally equidistributed; precisely put, every possible (n/w)-tuple of values is generated 2^k times, except that if D is any (n/w)-tuple that can be generated by the LCG itself, then $D + Z$ is generated by the LXM generator only $2^k - 1$ times. (This conclusion relies on the fact that $k = w$ guarantees that no two of the 2^k (n/w)-tuples generated by the LCG are equal, whereas this is generally not true when $k > w$.)

For example, `xoroshiro128` is 2-dimensionally equidistributed [Blackman and Vigna 2018]; using the terminology we define in Section 4, we can observe that `xoroshiro128` is 2-dimensionally 1-distributed, and it follows that LXM using a 64-bit LCG and `xoroshiro128` is 2-dimensionally

2^{-64}-distributed; so both xoroshiro128 and the LXM based on it can be said to be 2-dimensionally equidistributed, but the LXM has a much better δ value, reflecting the fact that it really can generate all possible 2-tuples, though a few of them are generated very slightly less often than the others, whereas for xoroshiro128 by itself there is one 2-tuple that is *never* generated.

Similarly, we can observe that because xoshiro256 is 4-dimensionally equidistributed (more precisely, 4-dimensionally 1-distributed), an LXM using a 64-bit LCG and xoshiro256 is 4-dimensionally 2^{-64}-distributed. Likewise, LXM using a 64-bit LCG and xoshiro512 is 8-dimensionally 2^{-64}-distributed, and LXM using a 64-bit LCG and xoroshiro1024 is 16-dimensionally 2^{-64}-distributed.

To summarize, the LXM algorithm can improve the equidistribution properties of its XBG component in two ways: (1) by making the sequence of w-bit outputs exactly equidistributed rather than approximately; and (2) when $k = w$ and the XBG is j-dimensionally δ-distributed for some $j > 1$, by reducing δ by a factor of 2^w.

(We also note that for an application that makes heavy use of, say, 2-tuples of 64-bit values, one could use a modified version of LXM for which $w = 128$ and $k = 128$ for the LCG, but $w = 64$ and $n \geq 128$ for the XBG, where for every generated 2-tuple of 64-bit values the LCG is advanced once and the XBG is advanced twice. The overall generator would then be exactly 2-dimensionally equidistributed. However, we have not yet studied or tested such a generator in any depth.)

6.5 Why We Need a Nontrivial Mixing Function

6.5.1 The Shape of LCG Graphs. Durst [1989] observed that, in some sense, every LCG on w-bit words whose period is 2^w that uses the same multiplier m produces "the same sequence"; if we imagine a two-dimensional plot of points (i, y_i), then changing the additive constant a has the effect of shifting the graph horizontally and vertically and possibly also flipping it top-to-bottom, but the overall "shape" of the graph is unchanged. (Coveyou had earlier remarked that the choice of addend for an LCG is "not of great importance" [Coveyou 1969, §12.1].)

To see this, choose any specific m, a, and a' such that $(m \bmod 8) = 5$, and a and a' are odd, and consider two LCGs $L = (s_0, \lambda\sigma.(m\sigma + a) \bmod 2^w, \lambda\sigma.\sigma)$ and $L' = (s'_0, \lambda\sigma.(m\sigma + a') \bmod 2^w, \lambda\sigma.\sigma)$.

There are then two cases.

(i) If $(a - a') \bmod 4 = 0$, let r be a solution to the congruence $a' \equiv a - (m-1)r \pmod{2^w}$; it is unique because $m - 1$ and $a - a'$ are multiples of 4, so we can rewrite it as $\frac{m-1}{4}r \equiv \frac{a-a'}{4} \pmod{2^w}$; then, because $m - 1$ is an *odd* multiple of 4, $\frac{m-1}{4}$ has a multiplicative inverse modulo 2^w, therefore $r = \left(\frac{m-1}{4}\right)^{-1} \frac{a-a'}{4} \bmod 2^w$. Let i be the smallest nonnegative integer such that $s'_i = (s_0 + r) \bmod 2^w$. Now an inductive argument: assume that $s'_{i+j} = (s_j + r) \bmod 2^w$; then

$$\begin{aligned} s'_{i+j+1} &= (ms'_{i+j} + a') \bmod 2^w \\ &= (m(s_j + r) + a - (m-1)r) \bmod 2^w \\ &= (ms_j + mr + a - mr + r) \bmod 2^w \\ &= (ms_j + a + r) \bmod 2^w \\ &= (s_{j+1} + r) \bmod 2^w \end{aligned}$$

and we can conclude that $s'_{i+j} = (s_j + r) \bmod 2^w$ is true for all $j \geq 0$. In words, the graph of L' is the result of shifting the graph of L rightward by i and upward by r, where the upward shift is actually a rotation modulo 2^w.

(ii) If $(a - a') \bmod 4 = 2$, let r be a solution to the congruence $a' \equiv (-a) + (m-1)r \pmod{2^w}$; it is unique because both $m - 1$ and $a + a'$ are multiples of 4, so we can rewrite it as $\frac{m-1}{4}r \equiv \frac{a+a'}{4} \pmod{2^w}$; therefore $r = \left(\frac{m-1}{4}\right)^{-1} \frac{a+a'}{4} \bmod 2^w$. Let i be the smallest nonnegative integer such that $s'_i = -(s_0 + r) \bmod 2^w$. Now an inductive argument: assume that $s'_{i+j} = -(s_j + r) \bmod 2^w$; then

$$\begin{aligned}
s'_{i+j+1} &= (ms'_{i+j} + a') \bmod 2^w \\
&= (m(-(s_j + r)) + (-a) + (m-1)r) \bmod 2^w \\
&= (-ms_j - mr - a + mr - r) \bmod 2^w \\
&= (-ms_j - a - r) \bmod 2^w \\
&= -(s_{j+1} + r) \bmod 2^w
\end{aligned}$$

and we can conclude that $s'_{i+j} = -(s_j + r) \bmod 2^w$ is true for all $j \geq 0$. In words, the graph of L' is the result of shifting the graph of L rightward by i and downward by r (rotating modulo 2^w), then flipping the graph vertically by negation of the y-axis (again modulo 2^w).

Because the output function selects the high-order bits of the LCG state, the effect is to shrink the graph vertically (dividing by 2^{k-w}) and then to apply a floor function; thus if $k > w$, the shape still remains roughly the same, though there is some jitter. Choosing different additive parameters for an LCG is not, of itself, a good way to produce streams that will appear to be independent.

6.5.2 The Shape of XBG Graphs. A similar (and simpler) argument shows that every full-period XBG that uses the same matrix U produces “the same sequence”; to see this, choose an n-by-n bit matrix U whose characteristic polynomial is primitive (therefore U is invertible), and also choose two n-bit vectors v and v'; then consider the two XBGs $X = (x_0, \lambda\tau.(U\tau \oplus v), \lambda\tau.\tau[0\,..\,w-1])$ and $X' = (x'_0, \lambda\tau.(U\tau \oplus v'), \lambda\tau.\tau[0\,..\,w-1])$. By the Cayley–Hamilton theorem and primitivity of the characteristic polynomial, any polynomial in U of degree $n-1$ or less can be expressed as a positive power of U [Engelberg 2015, §9.7]; it follows that because U is invertible, $U \oplus I$ is invertible.

Now consider the equation $v' = v \oplus (U \oplus I)r$; because $(U \oplus I)$ is invertible, we can easily solve the equation to get the unique solution $r = (U \oplus I)^{-1}(v \oplus v')$. Let i be the smallest nonnegative integer such that $x'_i = x_0 \oplus r$. Now an inductive argument: assume that $x'_{i+j} = x_j \oplus r$; then

$$\begin{aligned}
x'_{i+j+1} &= Ux'_{i+j} \oplus v' \\
&= U(x_j \oplus r) \oplus v \oplus (U \oplus I)r \\
&= Ux_j \oplus Ur \oplus v \oplus Ur \oplus r \\
&= Ux_j \oplus v \oplus r \\
&= x_{j+1} \oplus r
\end{aligned}$$

and we can conclude that $x'_{i+j} = x_j \oplus r$ is true for all $j \geq 0$. In words, the graph of X' is the result of shifting the graph of X rightward by i and “xor-flipping” the vertical axis by r.

Thus an XBG with state update function $\lambda\tau.(U\tau \oplus v)$ and output function $\lambda\tau.\tau[0\,..\,w-1]$ is effectively equivalent to an XBG with state update function $\lambda\tau.(U\tau)$ and output function $(\lambda\tau.(\tau[0\,..\,w-1] \oplus \hat{v})$, where $\hat{v} = (((U \oplus I)^{-1})v)[0\,..\,w-1] = (((U \oplus I)^{-1})[0\,..\,w-1; 0\,..\,n-1])v$. The graphs of all XBGs that use matrix U have “the same shape” but “xor-shifted” by a constant.

An xor with a constant affects the bits of the XBG state independently, and the output function selects high-order bits of the XBG state without regard to the value of any state bit; therefore graphs of the output values will also have “the same shape.” Choosing different additive parameters for an XBG is not, of itself, a good way to produce streams that will appear to be independent.

6.5.3 The Purpose of the Additive Parameter. In the LXM algorithm, the real purpose of the additive parameter in the LCG is not to select one of many LCG streams in hopes that these many streams will appear to be independent, because they cannot. Similarly, an additive parameter in an XBG will not select one of many independent streams. What we have seen is that, in effect, one might as well use a fixed LCG and a fixed XBG, combine their outputs, *then* add (or xor) a parameter, then apply the mixing function.

Then why does the parameter appear in the LCG rather than later in the algorithm? It is purely an engineering tweak, a bit of optimization. From a theoretical point of view, we can equally well introduce a parameter in any of *three* places: in the LCG, *or* in the XBG (by using an $\mathbf{F}_2$-affine state update function $\lambda\tau.U\tau \oplus v$ rather than the purely $\mathbf{F}_2$-linear state update function $\lambda\tau.U\tau$), *or* by using a combining function such as $\lambda(p,q).p+q+a$. (We could even introduce parameters in two, or all three, of those places, but there seems to be little extra benefit.) We observe that introducing the parameter in the XBG or the combining function requires "extra work"—perhaps one additional instruction—on today's typical hardware architectures, but the LCG *needs* to add *some* odd value in order to have full period, and it's easy to make that odd value be a parameter rather than a constant. Moreover, in the style of coding where the LCG update and XBG update are potentially computed in parallel with the combining and mixing functions, and given that a good mixing function takes longer to compute than the LCG update, adding the parameter in the LCG rather than in the combining step moves that addition operation off the critical path.

The hope, then, is that the additive parameter, despite being implemented at part of the LCG, will, in effect, select one of many *mixing* operations. In order to achieve this result, the mixing function certainly needs to be nonlinear, and ideally its range will appear to be a random permutation of its domain. Beyond this point theory offers us little firm guidance, and so we turn to empirical testing.

7 TESTING

We consider the TestU01 BigCrush test suite [L'Ecuyer and Simard 2007; Simard 2009] to be the current gold standard for final testing of any PRNG algorithm before deployment. However, we found PractRand [Doty-Humphrey 2011–2021] to be an extremely useful additional tool for two purposes: experimental exploration (because it fails fast on poor PRNG algorithms) and evaluating relative degrees of weakness (because the length to which a tested sequence must grow before failure is reported appears to be a more sensitive and repeatable metric than the p-value calculated for a sequence of fixed length). An algorithm that passes PractRand at the 4 TB threshold is worthy of final testing with BigCrush.

In testing variations of the LXM algorithm, we have performed over 52,000 complete runs of PractRand and over 50,000 complete runs of TestU01 BigCrush. For reasons of space we cannot present all the results of these tests, but we do present and describe tables that summarize salient results from BigCrush, and we describe and summarize in prose form salient results from PractRand.

7.1 Test Framework

We built a small testing framework to control thousands of test runs of multiple PRNG algorithms, using both the BigCrush test suite and the PractRand test suite.

Nearly all the tests were performed on a cluster of 16 nodes, each with two sockets, each with an E5-2660 2.2GHz Intel Xeon processor (each having eight cores collectively supporting 16 threads). Therefore 512 threads can execute simultaneously. (A very small fraction of the tests were run on a Macintosh Pro with two 2.8 GHz quad-core Xeon processors. This was done to validate the testing software before reserving time on the big cluster. The results of these initial runs constituted valid measurements and were retained.)

We made no attempt to parallelize the PractRand and BigCrush test suites; instead, we used `make` files to generate thousands of jobs at a time. Each `make` file describes one batch of test runs. Each `make` file includes code to find out which of the compute nodes it is being run on, so that a different subset of the batch of test runs will be run on each node. The use of `make` files allowed a very simple form of crash recovery: simply a matter of re-issuing the `make` command.

Each individual run tested the behavior of one PRNG algorithm, starting it from one specific state and testing the statistical quality of its output stream. While BigCrush and PractRand differ in the kinds of statistical tests they employ and the way they report the results of their analysis, they are alike in four key ways:

- There is a simple way to code new PRNG algorithms in C (or C++) and link them into the test suite. (This avoids the I/O overhead for piping the PRNG output stream into the test suite.)
- Results are reported by printing text to "standard output"; each report includes statistical information and also an indication of the total amount of CPU time (user execution time) consumed by the test.
- Each has a command-line interface that allows specification of which PRNG algorithm to test.
- The command-line interface does not allow a complete specification of the initial state of the PRNG, but does allow specification of a 64-bit *seed* from which the initial state can be constructed, and the construction code can be user-specified and bundled with the code for the PRNG algorithm itself.

We designed a detailed encoding that would allow us to use the single 64-bit integer parameter in the command line to specify a wide variety of initial states.

7.1.1 Distilling BigCrush Reports. The BigCrush test suite runs 106 individual tests [L'Ecuyer and Simard 2013, function `bbattery_BigCrush`, pp. 148–152], computing 160 test statistics and p-values [L'Ecuyer and Simard 2007]. A single test run typically prints about 110 kilobytes of information; at the end is either the message "`All tests were passed`" or a list of *anomalies*, that is, tests whose p-values were outside the range $[0.001\,..\,0.999]$.

For every algorithm tested with TestU01, we ran the entire suite three times, once in each of three distinct modes, identified by the letters `f`, `g`, and `u`. The `f` mode generates `double` values by generating a 64-bit integer, then right-shifting it by 11 and dividing by 2^{53} to produce a value in the range $[0.0\,..\,1.0)$. The `g` mode generates `double` values by generating a 64-bit integer, reversing the order of its bits so that bit j becomes bit $63 - j$, then right-shifting it by 11 and dividing by 2^{53}. The `u` mode generates `double` values by generating a 64-bit integer, then dividing each half (first the low half, then the high half) by 2^{32} to produce two `double` values, one after the other. (Late in our testing process we added a fourth mode, `w`, which generates `double` values by generating a 64-bit integer, then reversing the bit order of each half and dividing by 2^{32}.) As it turned out, we observed in the measured results no obvious differences between testing modes.

The distillation software for BigCrush test runs distills the list of anomalies for each test run into a pair of integers (l, c) (a *warning level* and a *count*) in this manner: If a test run file is missing, then $(l, c) = (-1, 0)$. If a test run file is present but is incomplete or malformed, then $(l, c) = (-2, 0)$ (this can happen if a test run was terminated before completion). If a test run file is present and all tests were passed ($10^{-3} < p < 1 - 10^{-3}$), then $(l, c) = (0, 0)$. Otherwise, the test run file was present and well-formed but reported one or more anomalies. Each anomaly is categorized according to its reported p-value (or, if $p > 0.5$, by using $1 - p$) into one of seven warning levels: if p is `eps` then 7, else if p is `eps1` then 6, else if $p \leq 10^{-12}$ then 5, else if $p \leq 10^{-9}$ then 4, else if $p \leq 10^{-6}$ then 3, else if $p \leq 10^{-4}$ then 2, else if $p \leq 10^{-3}$ then 1; then l is the highest warning level among all anomalies for the test run, and c is the number of anomalies having that highest warning level. We regard a run as a complete failure if l is 6 or 7.

7.1.2 Distilling PractRand Reports. The PractRand test suite is oriented toward testing 64-bit integer values and includes tests specifically designed to probe weakness in the low-order bits, so we used PractRand directly on the generated 64-bit values and made no attempt to define multiple testing modes. We ran each test until it had either reported failure or generated 4 TB of test data.

A single test run that gets all the way to 4 terabytes typically prints about 5 kilobytes of information. For each anomaly reported, PractRand prints not only a p-value but also a word or phrase

Table 1. Some of the "magic constants" used in testing. Multiplier values m are presented in decimal form and are among those recommended by L'Ecuyer [1999b, Table 4, p. 258]; all others are presented in hexadecimal form and are random values originally obtained from HotBits [Walker 1996], with A values forced to be odd.

32 bits		
$m_2 = 2891336453$	$A_8 =$ `0x4E1FD53B`	$S_8 =$ `0x4C3CA493`
$m_4 = 29943829$	$A_{10} =$ `0x950F5BFF`	$S_{10} =$ `0x734B1FEF`
$m_6 = 32310901$	$A_{12} =$ `0xFB999853`	$S_{12} =$ `0x36BAE016`
64 bits		
$m_2 = 2862933555777941757$	$A_8 =$ `0x856FA2A9BC6917B7`	$S_8 =$ `0xCFEADA5EE4037657`
$m_4 = 3202034522624059733$	$A_{10} =$ `0x873C0F33448D2C35`	$S_{10} =$ `0x0D1729016D5CA71D`
$m_6 = 3935559000370003845$	$A_{12} =$ `0xD321702ECD7BDA75`	$S_{12} =$ `0xAF5AA696D8C097F6`

describing that p-value; in increasing order of severity, they are `unusual`, `suspicious`, `SUSPICIOUS`, `very suspicious`, `VERY SUSPICIOUS`, and `FAIL`. (PractRand may further print a varying number of exclamation points after the word "`FAIL`" but we chose to ignore those: failure is failure.) We relied on these nonnumerical descriptions in distilling the reports into a pair of integers (l, c) (a *warning level*, ranging from 1 for `unusual` to 6 for `FAIL`, and a *count*) in a manner similar to that used for BigCrush. In addition, for each warning, the amount of data processed is recorded.

7.2 Results of BigCrush Tests

Table 1 lists some constants that are referred to by name in later tables. Not shown for lack of space are similar constants X_8, X_{10}, and X_{12}; also not shown are similar 16-bit and 128-bit constants.

Table 2 and other tables after it present summarized BigCrush results; the LaTeX source for these tables was generated automatically by the distillation software described in Section 7.1. Each line of the table summarizes a set of tests that differ only in stream count (the number of instances whose outputs are used in round-robin fashion) and mode. The first line of the table's footer shows the total number of test runs and the total CPU-thread time expended; the second line shows the set of stream counts and set of modes used for every line in the table.

For each line in the table, the first three columns show w, k, and n. The next two columns name the mixing function and initialization strategy. The next five columns give m, a, s_0, x_0, and the combining function ($+$ or $\oplus$); if a value is underlined, then every instance uses the indicated value; otherwise each instance uses a value generated by some other instance in a manner dictated by the particular initialization strategy. N is the total number of test runs for that line of the table. The next eight columns show the number of test runs whose highest warning level was $0, 1, 2, \ldots, 7$; recall that warning levels 6 and 7 indicate complete failure (see Section 7.1.1). The last two columns give the total number of warnings (Σ) and the smallest p-value (P_{worst}) seen during the N runs.

Figure 4 shows C code for mixing functions we tested: `murmur32` and `murmur64` [Appleby 2011]; `degski32` and `degski64` [degski 2018]; `lea64`, by Doug Lea [Lea 2013]; `starstar16` [Blackman and Vigna 2018] (a scrambler, never intended to be a mixing function); and `madeup16`, by one of us.

These are the five initialization strategies that appear in the tables (let κ be the stream count, and it is implicitly understood that as the non-underlined values for an instance are filled in, the underlined values are also filled in as specified in the table):

`same` uses the listed m, z, s_0, and x_0 values to create a single extra instance of the LXM, outputs of which are used to initialize non-underlined values for the κ instances to be tested.

`tree` b uses m, z, s_0, and x_0 to initialize instance 0, then for all $1 \le j < \kappa$ in ascending order, output from instance $\lfloor j/b \rfloor$ is used to initialize non-underlined values for instance j.

```
uint32_t murmur32(uint32_t z) {
  z ^= (z >> 16);
  z *= 0x85ebca6bul;
  z ^= (z >> 13);
  z *= 0xc2b2ae35ul;
  return z ^ (z >> 16); }

uint64_t murmur64(uint64_t z) {
  z ^= (z >> 33);
  z *= 0xff51afd7ed558ccdull;
  z ^= (z >> 33);
  z *= 0xc4ceb9fe1a85ec53ull;
  return z ^ (z >> 33); }

uint32_t degski32(uint32_t z) {
  z ^= (z >> 16);
  z *= 0x45d9f3bul;
  z ^= (z >> 16);
  z *= 0x45d9f3bul;
  return z ^ (z >> 16); }

uint64_t degski64(uint64_t z) {
  z ^= (z >> 32);
  z *= 0xd6e8feb86659fd93ull;
  z ^= (z >> 32);
  z *= 0xd6e8feb86659fd93ull;
  return z ^ (z >> 32); }

uint64_t lea64(uint64_t z) {
  z ^= (z >> 32);
  z *= 0xdaba0b6eb09322e3ull;
  z ^= (z >> 32);
  z *= 0xdaba0b6eb09322e3ull;
  return z ^ (z >> 32); }

uint16_t starstar16(uint16_t z) {
  z = z * 5;
  return ((z << 7) | (z >> 9)) * 9; }

uint16_t madeup16(uint16_t z) {
  z = (uint16_t)((z ^ (z >> 8)) * 0xca6b);
  z = (uint16_t)((z ^ (z >> 9)) * 0xae35);
  return (uint16_t)(z ^ (z >> 8)); }
```

Fig. 4. Mixing functions used during testing

`skip` uses m, z, s_0, and x_0 to initialize instance 0, then for all $1 \leq j < \kappa$ in ascending order, all non-underlined values for instance i are copied from those of instance $i - 1$ and then the state of the XBG is advanced one position.

`jump` is the same as `skip`, except that the XBG is advanced by $2^{n/2}$ positions.

`leap` is the same as `skip`, except that the XBG is advanced by $2^{3n/4}$ positions.

7.2.1 Scaling the Number of Streams. Table 2 shows results from LXM instances that use a 64-bit LCG, either `xoroshiro128` or `xoshiro256`, and either one of three mixers or none. The combining function is + (addition). They are tested for stream counts $1, 2, 4, 8, 16, \ldots, 2^{24}$ and also three other non-power-of-two stream counts, chosen arbitrarily. For each stream count κ, five different initialization procedures are tested: `same`, `tree` 2, `skip`, `jump`, and `leap`. We observe BigCrush fails only the cases that use no mixing function and use `skip`, `jump`, or `leap` initialization. All three mixing functions appear to be equally effective in this set of tests.

We ran similar tests using a 128-bit LCG (with a 64-bit multiplier and either a 64-bit or 128-bit additive parameter) and `xoroshiro128` for the XBG, using the same set of stream counts and the same five initialization procedures. The results were quite similar to those in Table 2.

7.2.2 Tree-shaped (Potentially Parallel) Initialization Strategies. Table 3 shows BigCrush results from LXM instances that use a 64-bit LCG, either `xoroshiro128` or `xoshiro256`, and no mixing function. The combining function is + (addition). They are tested for stream counts 2^8, 2^{12}, 2^{14}, 2^{17}, 2^{21}, and 2^{24}. For each stream count, six different branching factors for the tree are tested: 3, 4, 5, 16, 32, and 256 (the tests shown in Table 2 cover the case of branching factor 2). None of these tests fail. Out of 216 tests, just one has a warning level as high as 3.

7.2.3 Instances with Very Similar Additive Constants. Table 4 shows BigCrush results from LXM instances with $k = 32$ and $n = 64$, $k = 32$ and $n = 128$, $k = 64$ and $n = 128$, or $k = 64$ and

Table 2. Test measurements for scaling the number of streams (see Section 7.2.1)

w	k	n	mixer	init	m	a	s_0	x_0	$\circledast$	N	0	1	2	3	4	5	6	7	Σ	p_{worst}
64	64	128	murmur64	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	61	19	3	1					28	2.0E-7
64	64	128	murmur64	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	54	26	4						37	3.0E-5
64	64	128	murmur64	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	59	19	6						29	3.0E-6
64	64	128	murmur64	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	58	22	4						34	3.8E-5
64	64	128	murmur64	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	65	15	4						24	6.0E-5
64	64	128	degski64	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	56	19	9						36	3.7E-5
64	64	128	degski64	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	64	12	8						24	1.0E-5
64	64	128	degski64	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	58	22	4						31	1.6E-6
64	64	128	degski64	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	61	18	5						30	2.3E-5
64	64	128	degski64	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	57	20	7						32	3.0E-5
64	64	128	lea64	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	63	19	1	1					24	2.8E-7
64	64	128	lea64	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	60	20	4						30	4.4E-6
64	64	128	lea64	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	62	21	1						26	9.4E-5
64	64	128	lea64	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	52	26	6						40	2.8E-5
64	64	128	lea64	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	56	18	10						35	2.7E-6
64	64	128	none	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	50	29	5						38	4.6E-5
64	64	128	none	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	57	23	4						33	1.1E-5
64	64	128	none	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	2	0	1	0	0	0	0	81	6406	eps
64	64	128	none	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	41	7	2	0	0	0	1	33	103	eps
64	64	128	none	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	43	5	3	0	0	0	0	33	104	eps
64	64	256	murmur64	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	63	16	5						23	7.1E-6
64	64	256	murmur64	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	55	23	6						36	4.4E-6
64	64	256	murmur64	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	57	21	6						32	1.1E-5
64	64	256	murmur64	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	66	15	3						21	3.2E-5
64	64	256	murmur64	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	59	20	5						30	1.9E-5
64	64	256	degski64	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	60	17	7						30	1.8E-5
64	64	256	degski64	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	60	20	4						26	8.1E-5
64	64	256	degski64	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	51	27	6						39	1.6E-5
64	64	256	degski64	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	62	19	2	1					31	2.4E-7
64	64	256	degski64	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	58	22	4						30	1.1E-5
64	64	256	lea64	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	63	18	3						22	7.4E-6
64	64	256	lea64	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	53	24	7						36	1.9E-6
64	64	256	lea64	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	59	18	7						26	3.7E-6
64	64	256	lea64	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	63	18	3						26	3.0E-5
64	64	256	lea64	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	62	17	5						27	2.4E-5
64	64	256	none	same	$\underline{m}_2$	A_8	S_8	X_8	+	84	55	27	2						38	4.5E-5
64	64	256	none	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	84	52	30	2						41	3.2E-5
64	64	256	none	skip	$\underline{m}_2$	A_8	S_8	X_8	+	84	2	1	0	0	0	0	0	81	6318	eps
64	64	256	none	jump	$\underline{m}_2$	A_8	S_8	X_8	+	84	32	17	2	0	0	0	0	33	95	eps
64	64	256	none	leap	$\underline{m}_2$	A_8	S_8	X_8	+	84	38	13	0	0	0	0	0	33	86	eps

3360 complete runs of BigCrush Total CPU-thread time: 1433 days + 13:31:27

Stream counts used: $\{ 2^j \mid 0 \leq j \leq 24 \} \cup \{ 1900547, 5242880, 12582912 \}$ Modes used: u f g

Table 3. Test measurements for tree-shaped (potentially parallel) initialization strategies (see Section 7.2.2)

w	k	n	mixer	init	m	a	s_0	x_0	$\circledast$	N	0	1	2	3	4	5	6	7	Σ	p_{worst}
64	64	128	none	`tree 3`	$\underline{m}_2$	A_8	S_8	X_8	+	18	10	6	2						8	`1.8E-5`
64	64	128	none	`tree 4`	$\underline{m}_2$	A_8	S_8	X_8	+	18	15	3							3	`4.8E-4`
64	64	128	none	`tree 5`	$\underline{m}_2$	A_8	S_8	X_8	+	18	10	6	2						9	`3.3E-5`
64	64	128	none	`tree 16`	$\underline{m}_2$	A_8	S_8	X_8	+	18	9	9							11	`1.3E-4`
64	64	128	none	`tree 32`	$\underline{m}_2$	A_8	S_8	X_8	+	18	10	6	2						12	`1.6E-5`
64	64	128	none	`tree 256`	$\underline{m}_2$	A_8	S_8	X_8	+	18	13	4	1						5	`1.0E-4`
64	64	256	none	`tree 3`	$\underline{m}_2$	A_8	S_8	X_8	+	18	15	3							3	`1.1E-4`
64	64	256	none	`tree 4`	$\underline{m}_2$	A_8	S_8	X_8	+	18	9	6	3						9	`1.8E-5`
64	64	256	none	`tree 5`	$\underline{m}_2$	A_8	S_8	X_8	+	18	11	7							7	`1.5E-4`
64	64	256	none	`tree 16`	$\underline{m}_2$	A_8	S_8	X_8	+	18	10	6	1	1					9	`2.2E-7`
64	64	256	none	`tree 32`	$\underline{m}_2$	A_8	S_8	X_8	+	18	14	3	1						7	`4.2E-5`
64	64	256	none	`tree 256`	$\underline{m}_2$	A_8	S_8	X_8	+	18	9	6	3						12	`4.1E-5`

216 complete runs of BigCrush Total CPU-thread time: 96 days + 15:34:05
Stream counts used: { $2^8, 2^{12}, 2^{14}, 2^{17}, 2^{21}, 2^{24}$ } Modes used: `u f g`

$n = 256$. The combining function is + (addition). We tested all 200 combinations of 25 stream counts ($\{ 2^j \mid 0 \leq j \leq 24 \}$), two different multipliers m_4 and m_6 for the LCG, 2 mixing functions (none, or `murmur` of the appropriate word size), and two ways to choose the additive constants. The initialization strategy was `same` in all cases, except that the additive constants were chosen to be very similar: for stream count κ, for $0 \leq i < \kappa$, the additive parameter was either $1 + 32i$ or $A_8 + 32i$. All cases with no mixing function and a stream count below 1024 fail. All cases using a `murmur` mixer passed, and out of 2000 tests, just one has a warning level as high as 3.

On the other hand, certain contrived tests fail BigCrush spectacularly: if the initial states s_0 and x_0 of two instances are identical (a situation unlikely in practice) and on top of that their additive constants a differ only in the high-order bit (even less likely), then the values produced by the combining function will differ only in the high-order bit, and it's asking too much of a fast mixing function to produce apparently independent streams from such inputs.

We conclude that the mixing function may play a valuable defensive role when the additive constants of the LCGs are somewhat similar, but in very rare cases may fail to do the job; it's important to try to initialize multiple instances to very different states.

7.2.4 Instances That Use XOR *for the Combining Function.* Table 5, which may be compared with Table 4, shows BigCrush results from LXM instances with either $k = 32$ and $n = 64$, or $k = 64$ and $n = 256$. The combining function is $\oplus$ (XOR). As in Section 7.2.3, we tested all 200 combinations of 25 stream counts ($\{ 2^j \mid 0 \leq j \leq 24 \}$), two different multipliers m_4 and m_6 for the LCG, 2 mixing functions (none, or `murmur` of the appropriate word size), and two ways to choose the additive constants. The initialization strategy was the `same` in all cases, except that the additive constants were chosen to be very similar: for stream count κ, for $0 \leq i < \kappa$, the additive parameter was either $1 + 32i$ or $A_8 + 32i$. All cases with no mixing function and a stream count below 1024 fail. All cases using a `murmur` mixer passed, and out of 1000 tests, just two have a warning level as high as 3. (We also tested multiplier m_2; the results, not shown here for lack of space, were similar.)

We conclude that when a good mixing function is used, using XOR for the combining function appears to be no worse than using addition (but note that using XOR may be significantly worse than using addition if no mixing function is used [Marsaglia 1985]).

Table 4. Test measurements for instances with very similar additive constants (see Section 7.2.3)

w	k	n	mixer	init	m	a	s_0	x_0	$\circledast$	N	0	1	2	3	4	5	6	7	Σ	p_{worst}
32	32	64	none	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	55	18	4	0	0	0	37	11	105	eps
32	32	64	none	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	58	17	2	0	0	0	37	11	112	eps
32	32	64	none	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	58	17	2	0	0	0	37	11	97	eps
32	32	64	none	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	58	14	5	0	0	0	37	11	107	eps
32	32	64	murmur	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	87	32	6						44	3.5E-5
32	32	64	murmur	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	82	36	7						52	1.8E-6
32	32	64	murmur	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	93	27	5						35	4.1E-5
32	32	64	murmur	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	95	24	5	1					38	5.8E-7
32	32	128	none	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	64	15	5	0	0	0	35	6	86	eps
32	32	128	none	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	61	17	6	0	0	0	35	6	93	eps
32	32	128	none	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	64	11	9	0	0	0	35	6	101	eps
32	32	128	none	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	61	19	4	0	0	0	35	6	84	eps
32	32	128	murmur	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	90	29	6						41	2.4E-5
32	32	128	murmur	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	77	45	3						54	8.7E-6
32	32	128	murmur	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	82	34	9						58	1.6E-6
32	32	128	murmur	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	85	35	5						47	4.6E-5
64	64	128	none	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	77	28	9	0	0	0	9	2	59	eps
64	64	128	none	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	79	30	4	1	0	0	9	2	57	eps
64	64	128	none	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	78	28	7	1	0	0	9	2	63	eps
64	64	128	none	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	76	38	0	0	0	0	9	2	59	eps
64	64	128	murmur	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	89	31	5						46	1.1E-5
64	64	128	murmur	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	91	27	7						36	1.1E-5
64	64	128	murmur	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	87	31	7						45	8.0E-5
64	64	128	murmur	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	86	33	6						47	2.7E-5
64	64	256	none	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	89	19	5	2	0	0	9	1	43	eps
64	64	256	none	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	83	29	3	0	0	0	9	1	53	eps
64	64	256	none	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	80	28	7	0	0	0	9	1	56	eps
64	64	256	none	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	83	26	5	1	0	0	9	1	52	eps
64	64	256	murmur	same	$\underline{m_4}$	$\underline{1+32i}$	S_8	X_8	+	125	88	35	2						42	8.2E-6
64	64	256	murmur	same	$\underline{m_4}$	$\underline{A_8+32i}$	S_8	X_8	+	125	87	30	8						44	3.4E-5
64	64	256	murmur	same	$\underline{m_6}$	$\underline{1+32i}$	S_8	X_8	+	125	84	33	8						48	6.7E-6
64	64	256	murmur	same	$\underline{m_6}$	$\underline{A_8+32i}$	S_8	X_8	+	125	93	26	6						40	6.3E-6

4000 complete runs of BigCrush — Total CPU-thread time: 1845 days + 12:33:10

Stream counts used: $\{ 2^j \mid 0 \le j \le 24 \}$ — Modes used: u v w f g

7.2.5 Scaling the State Size. One way to see how a family of PRNGs behaves is to consider the behavior of very small members of the family. We tested three small variants: $w = 32$, $k = 32$, $n = 128$; $w = 32$, $k = 32$, $n = 64$; and $w = 16$, $k = 16$, $n = 32$. In each case the combining function was addition.

Small PRNGs: Table 6 shows BigCrush results for $w = 32$, $k = 32$, and n either 64 or 128. The 64-bit XBG algorithm is `xoroshiro64` [Blackman and Vigna 2018], that is,

```
{ q1 ^= q0; q0 = (q0 << 26) | (q0 >> 6); q0 = q0 ^ q1 ^ (q1 << 9);
  q1 = (q1 << 13) | (q1 >> 19); }
```

Table 5. Test measurements for instances that use XOR for the combining function (see Section 7.2.4)

w	k	n	mixer	init	m	a	s_0	x_0	$\circledast$	N	0	1	2	3	4	5	6	7	Σ	p_{worst}
32	32	64	none	same	$\underline{m}_4$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	47	16	2	0	0	0	35	25	149	eps
32	32	64	none	same	$\underline{m}_4$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	49	13	1	2	0	0	35	25	151	eps
32	32	64	none	same	$\underline{m}_6$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	48	13	4	0	0	0	35	25	142	eps
32	32	64	none	same	$\underline{m}_6$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	48	12	5	0	0	0	35	25	171	eps
32	32	64	murmur	same	$\underline{m}_4$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	94	29	2						38	4.6E-6
32	32	64	murmur	same	$\underline{m}_4$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	97	25	3						33	3.7E-5
32	32	64	murmur	same	$\underline{m}_6$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	88	32	5						42	2.1E-5
32	32	64	murmur	same	$\underline{m}_6$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	86	34	5						46	2.2E-5
64	64	128	none	same	$\underline{m}_4$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	76	24	4	0	0	0	7	14	64	eps
64	64	128	none	same	$\underline{m}_4$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	78	19	7	0	0	0	7	14	64	eps
64	64	128	none	same	$\underline{m}_6$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	70	31	3	0	0	0	7	14	81	eps
64	64	128	none	same	$\underline{m}_6$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	68	33	3	0	0	0	7	14	83	eps
64	64	128	murmur	same	$\underline{m}_4$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	85	31	8	1					49	7.6E-7
64	64	128	murmur	same	$\underline{m}_4$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	82	35	7	1					57	2.0E-7
64	64	128	murmur	same	$\underline{m}_6$	$\underline{1+32i}$	S_8	X_8	$\oplus$	125	81	35	9						51	2.3E-5
64	64	128	murmur	same	$\underline{m}_6$	$\underline{A_8+32i}$	S_8	X_8	$\oplus$	125	91	28	6						40	1.9E-5

2000 complete runs of BigCrush — Total CPU-thread time: 832 days + 23:59:39

Stream counts used: $\{\, 2^j \mid 0 \le j \le 24 \,\}$ — Modes used: u v w f g

with output q0. The 128-bit XBG algorithm is xoshiro128 [Blackman and Vigna 2018], that is,

```
{ uint32_t t = q1 << 9; q2 ^= q0; q3 ^= q1; q1 ^= q2; q0 ^= q3;
  q2 ^= t; q3 = (q3 << 11) | (q3 >> 21); }
```

with output q1. We tested all 240 combinations of 25 stream counts ($\{\, 2^j \mid 0 \le j \le 24 \,\}$), 4 mixing functions (none, murmur32, degski32, and lea32), and 2 initialization strategies (same and tree 2). For $n = 64$, the version with no mixer always failed when the number of streams was less than 64; for $n = 128$, the version with no mixer always failed when the number of streams was less than 16. In all other cases, no warning level worse than 2 was observed, except for one case with $n = 64$ and stream count 256, which had warning level 3.

Very small PRNGs: Table 7 shows BigCrush results for $w = 16$, $k = 16$, $n = 32$; the 32-bit XBG algorithm is

```
{ q ^= (q << 13); q ^= (q >> 17); q ^= (q << 5); }
```

which uses one of the triples of shift constants recommended by Marsaglia [2003, §3]. We tested all 240 combinations of 40 stream counts ($\{\, 2^j \mid 0 \le j \le 24 \,\} \cup \{256 + 16j \mid 1 \le j \le 15\,\}$), 3 mixing functions (none, starstar16, and madeup16), and 2 initialization strategies (same and tree 2). The version with no mixer always failed when the number of streams was less than 336; no warning level worse than 2 was observed for stream counts above 367. The starstar16 mixer produced no warning level worse than 2. The madeup16 mixer (so called because its constants were chosen at whim, with no attempt to optimize avalanche statistics) also produced no warning level worse than 2. So even at this very small scale we see that, on the one hand, even a simple mixing function clearly improves the quality, and on the other hand, even a simple mixing function suffices to get adequate quality. Focusing on the single-stream case, we find it remarkable that a PRNG with just 48 bits of state is able to pass BigCrush, and that (with the madeup16 mixer) PractRand tests 1 TB of its output (2^{39} generated values) before failing it (see Section 7.3).

Table 6. Test measurements for scaling the state size: 32-bit generators (see Section 7.2.5)

w	k	n	mixer	init	m	a	s_0	x_0	$\circledast$	N	0	1	2	3	4	5	6	7	Σ	p_{worst}
32	32	64	none	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	34	19	3	1	0	0	15	3	49	eps
32	32	64	none	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	37	18	2	0	0	0	15	3	52	eps
32	32	64	murmur32	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	41	29	5						37	8.7E-6
32	32	64	murmur32	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	53	17	5						25	7.5E-6
32	32	64	degski32	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	49	24	2						27	2.3E-5
32	32	64	degski32	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	48	25	2						30	4.0E-5
32	32	64	lea32	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	48	22	5						33	3.9E-5
32	32	64	lea32	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	57	15	3						20	5.3E-5
32	32	128	none	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	42	19	2	0	0	0	12		36	eps1
32	32	128	none	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	44	17	2	0	0	0	12		39	eps1
32	32	128	murmur32	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	52	19	4						31	2.5E-5
32	32	128	murmur32	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	52	21	2						29	5.1E-6
32	32	128	degski32	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	60	11	4						17	1.1E-5
32	32	128	degski32	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	55	15	5						25	1.9E-5
32	32	128	lea32	same	$\underline{m}_2$	A_8	S_8	X_8	+	75	61	11	3						16	5.3E-5
32	32	128	lea32	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	75	56	17	2						24	3.5E-5

1200 complete runs of BigCrush — Total CPU-thread time: 599 days + 19:18:47

Stream counts used: $\{ 2^j \mid 0 \leq j \leq 24 \}$ — Modes used: u f g

Table 7. Test measurements for scaling the state size: 16-bit generators (see Section 7.2.5)

w	k	n	mixer	init	m	a	s_0	x_0	$\circledast$	N	0	1	2	3	4	5	6	7	Σ	p_{worst}
16	16	32	none	same	$\underline{m}_2$	A_8	S_8	X_8	+	120	38	28	10	2	0	0	22	20	203	eps
16	16	32	none	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	120	52	15	8	2	1	0	22	20	193	eps
16	16	32	madeup16	same	$\underline{m}_2$	A_8	S_8	X_8	+	120	85	30	5						48	5.6E-6
16	16	32	madeup16	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	120	77	38	5						47	1.6E-5
16	16	32	starstar16	same	$\underline{m}_2$	A_8	S_8	X_8	+	120	81	35	4						47	2.3E-5
16	16	32	starstar16	tree 2	$\underline{m}_2$	A_8	S_8	X_8	+	120	86	30	4						40	2.5E-5

720 complete runs of BigCrush — Total CPU-thread time: 489 days + 6:50:16

Stream counts used: $\{ 2^j \mid 0 \leq j \leq 24 \} \cup \{ 2^8{+}2^4k \mid 1 \leq k \leq 15 \}$ — Modes used: u f g

7.2.6 LCG Multipliers. Most of our testing has used multipliers recommended by L'Ecuyer [1999b, Table 4, p. 258]; we have also run tests using multipliers recently discovered by Steele and Vigna [2021, Table 5, p. 17]. We have not detected any significant difference in test results; if LXM quality strongly depends on LCG multiplier quality, more sensitive, more specialized tests may be required.

7.3 Results of PractRand Tests

Many PractRand tests suggested LXM would generally produce sequences of good quality; for example, one set of 216 runs tested 36 different LXM generators with a `tree` 2 initialization strategy for stream counts 1, 2, 32, 2^{11}, 2^{15}, and 2^{20}; they detected no problems except that (a) for stream count 2^{11}, 32-bit LXM generators with no mixing function failed after 128 GB; (b) 16-bit LXM generators using no mixing function generally failed; and (c) 16-bit LXM generators using `madeup16` failed for stream counts 1 (at 1 TB) and 2 (at 4 TB). Other tests picked up the (unsurprising) fact that `starstar` (never intended for this purpose) is a weaker mixer than MurmurHash3 and its variants.

Table 8. Comparative timings (all measurements in nanoseconds per word generated)

	size (in bits)			Haswell				ARM			
				gcc		clang		gcc		clang	
	m	a	out	inline	noinline	inline	noinline	inline	noinline	inline	noinline
`L32X64`	32	32	32	1.615	2.311	1.602	2.290	2.566	4.076	2.564	4.320
`L32XX64`	32	32	32	1.679	2.321	1.713	2.321	2.566	4.076	2.641	4.321
`L32X128`	32	32	32	1.547	2.503	1.549	2.464	2.885	4.444	2.690	4.859
SplitMix	—	64	64	1.201	1.688	0.967	1.681	2.401	3.512	2.401	3.381
`L64X128`	64	64	64	1.614	2.488	1.679	2.232	3.601	4.603	3.602	4.393
`L64XX128`	64	64	64	1.528	2.650	1.713	2.239	3.601	4.609	3.602	4.399
`L64X256`	64	64	64	1.680	2.734	1.560	2.431	3.601	4.901	3.601	5.146
`L128AX128`	64	64	64	1.859	2.915	1.837	3.122	7.602	8.006	6.402	7.350
`L128BX128`	64	128	64	1.859	2.714	1.831	2.717	7.602	9.231	6.402	7.396
`L128CX128`	128	64	64	2.566	2.889	1.923	2.881	7.602	10.306	7.602	9.039
`L128DX128`	128	128	64	2.563	3.232	1.933	2.856	7.602	10.481	7.602	9.133
`L128EX128`	65	64	64	2.466	3.005	1.919	2.881	7.602	8.027	7.602	7.363
`L128FX128`	65	128	64	2.465	2.780	1.929	2.772	7.602	8.430	7.602	7.378
`L128AX256`	64	64	64	1.813	3.212	1.720	2.881	7.602	7.967	6.402	7.477
`L128BX256`	64	128	64	1.812	2.883	1.773	2.881	7.602	8.027	6.402	7.373
`L128CX256`	128	64	64	2.571	3.129	1.919	2.933	7.602	9.165	7.602	9.193
`L128DX256`	128	128	64	2.568	3.627	1.931	3.122	7.602	9.333	7.602	9.200
`L128EX256`	65	64	64	2.542	3.120	1.920	2.881	7.602	7.377	7.602	7.401
`L128FX256`	65	128	64	2.537	3.231	1.930	2.929	7.602	7.683	7.602	7.396

8 COMPARATIVE TIMING TESTS

Table 8 shows timings for SplitMix and a selection of LXM generators representative of various configurations of generator state. We tested two architectures: an Intel® Core™ i7-8700B CPU @3.20 GHz (Haswell) and an AWS Graviton 2 processor based on 64-bit Arm Neoverse cores @2.5 GHz. We performed our tests using two different compilers, `gcc` 10 (11 on Haswell) and `clang` 10. In each case, we tested the next-state function in two ways: forcing inlining, or blocking inlining; in the second case, the compiler has to reload the constants involved at each call, and we also pay for the function call itself. The two timings gives a differential view of the cost of pure computation (without constant loading) versus global cost. We report the average of ten runs; the measurements are very stable, with relative standard error below 1%, and in almost all cases below 0.5%.

The results depend on both architecture and compilers; in general, `clang` faster code, especially with larger state and constants; the case of SplitMix is very evident, but note that the timing with `gcc` 10 on Haswell is similar to that of `clang`, so that specific gap appears to be a scheduling defect introduced in version 11. The main visible difference in timing is that between 64-bit and larger multipliers. We see no relevant difference between 65-bit and 128-bit multipliers, except for the no-inline case of the ARM architecture, where the additional time required to load the larger constant is very visible. For the same reason, the difference in timing between the inline and the no-inline benchmarks on ARM is sometimes very large. Always on ARM, `clang` generates better inline code for 64-bit multipliers with respect to larger sizes, contrarily to `gcc`.

The size of the additive constant a (64 or 128 bits) appears to have little impact; in some cases, paradoxically a larger constant generates a shorter timing. Quirks of this kind are unavoidable because the models used by compilers to optimize instruction choice and pipelining are not perfect.

9 MORE ABOUT JUMPING AND SPLITTING

The standard way to jump an XBG by j positions is use some precomputed representation of U^j, then apply that matrix to the XBG state. One common convention is that "jump" advances by $2^{n/2}$ positions and "leap" ("long jump") advances by $2^{3n/4}$ positions. This is advantageous for LXM: jumping by a power of 2 at least as large the period of the LCG leaves the LCG state unchanged. But the representation of U^j is typically not as efficient to apply as U.

Imagine instead that we wish to make an LXM jump *backwards* by $2^n - 1$ positions; that would leave the XBG state unchanged, and put the LCG in the same state as if we had advanced the LCG just one position. So advancing just the LCG is a simple way to get a cheaper LXM jump function. And leaping backward by, say, $2^{k/2}(2^n - 1)$ positions is equally easy, because one can precompute constants m' and a' such that $\lambda\sigma.(m'\sigma + a') \bmod 2^k$ will advance the LCG by $2^{k/2}$ positions.

But the point of jump functions is usually to create multiple generators in such a way that their generated sequences will not overlap. We believe (but admit that we have not yet proved) that the additive parameter provides a very simple way to do that if the mixing function is good: just ensure that each instance has a different additive parameter. Choosing the additive value at random, as the `split()` method does, may do that with high probability if 2^k is sufficiently larger than the number of instances. On the other hand, it is very easy for the `splits()` method to ensure that all the generators in a single generated stream have different additive parameters; this is even easier than the cheap strategy for jumping. Testing seems to confirm that this strategy is effective, and splitting is easier to use than jumping in applications structured to use recursive fork-join parallelism.

10 CHOOSING AN LXM ALGORITHM

If we were presenting a single algorithm, our message would be simple: "use our new algorithm." But because we have described a large family of algorithms and tested several, we offer the following suggestions for how to choose an algorithm appropriate for a specific application.

If an application requires a random number generator algorithm that is cryptographically secure, then *no* member of the LXM family is appropriate; programs coded in the Java language should continue to use an instance of the class `java.security.SecureRandom`.

For applications with no special requirements, `L64X128MixRandom` has a good balance among speed, space, and period, and is suitable for both single-threaded and multi-threaded applications when used properly (a separate instance for each thread). But for a single-threaded application, `Xoroshiro128PlusPlus` is even smaller and faster, and certainly has a sufficiently long period.

For an application running in a 32-bit hardware environment and using only one thread or a small number of threads, `L32X64MixRandom` may be a good choice for reasons of speed and space.

For an application that uses many threads that are allocated in one batch at the start of the computation, one may prefer either a jumpable generator such as `Xoroshiro128PlusPlus` or `Xoshiro256PlusPlus`, or a splittable generator such as `L64X128MixRandom` or `L64X256MixRandom`.

For an application that creates many threads dynamically, perhaps through the use of Java spliterators, we recommend a splittable generator such as `L64X128MixRandom` or `L64X256MixRandom`. If the number of generators created dynamically may be very large (millions or more), then `L128X128MixRandom` or `L128X256MixRandom`, which use a 128-bit (rather than 64-bit) parameter for the LCG subgenerator, will make it much less likely that two instances use the same state cycle.

For an application that uses n-tuples of consecutively generated values, it may be desirable to use a generator that is k-equidistributed such that $k \geq n$. The generator `L64X256MixRandom` is provably 4-equidistributed, and `L64X1024MixRandom` is provably 16-equidistributed.

For applications that generate large permutations, it may be best to use a generator whose period is much larger than the total number of possible permutations; otherwise it will be impossible to generate some of the intended permutations. For example, if the goal is to shuffle a deck of 52 cards, the number of possible permutations is 52! (52 factorial), which is larger than 2^{225} (but smaller than 2^{226}), so it may be best to use a generator whose period at least 2^{256}, such as `L64X256MixRandom` or `L64X1024MixRandom` or `L128X256MixRandom` or `L128X1024MixRandom`. (It is of course also necessary to provide sufficiently many seed bits when the generator is initialized, or else it will still be impossible to generate some of the intended permutations.)

11 RELATED WORK

Schaathun [2015] has recently surveyed a number of techniques for splittable pseudorandom generators. He traces the origin of the ideas to the 1980s, and in particular to Warnock's work [Warnock 1983] in particle physics, where splitting occurs when a particle being simulated spawns new particles. A few years later several studies proposed to use different additive constants of LCGs to perform splitting, generating a *Lehmer tree*, until Durst [1989] proved that such sequences are strictly correlated, as we discuss in Section 6.5.1. Notably, Schaathun concludes that the cryptographic approach of Claessen and Pałka [2013], which uses cryptographic hashing on the splitting tree, is the safest and the only one providing some theoretical guarantees. Later, Steele, Lea, and Flood introduced SplitMix [2014, §7]; while they do not perform comparative measurements with Claessen and Pałka's approach, they conjecture that the latter should yield sequences with better statistical qualities than SplitMix, while SplitMix should be faster.

Also the combination of generators of different nature has a long history. A relatively recent video on YouTube [Losego 2016] has reverse-engineered the code used for random number generation by the well-known video game *Super Mario World* [Nintendo 1990], which was released on November 21, 1990. The code merits study as an example of excellent engineering within a severely resource-constrained computing environment (a Ricoh 5A22 CPU, closely related to the WDC 65C816), and it happens to be very closely related to the LXM algorithm. The generator produces two 8-bit bytes each time it is called; each byte is the result of one call to a subroutine. The subroutine implements two subgenerators, each with one 8-bit byte of state, and the output of the subroutine is the bitwise xor of the outputs s and t of the two subgenerators. One subgenerator is an LCG whose period is 256, and the other an XBG using an $\mathbf{F}_2$-affine state update function whose period is 217, so the overall period of the subgenerator (viewed as a generator of bytes) is 55552. (As far as we can tell, the principal advantage of using an $\mathbf{F}_2$-affine state update function rather than a purely $\mathbf{F}_2$-linear function—either would have been equally easy to implement—is that the state of the prng can be reset by zeroing both state bytes.) The overall period of the main generator (viewed as a generator of pairs of bytes) is therefore 27776. The update computation for the two subgenerators is

$$s \leftarrow 5 \times s + 1; t \leftarrow \bigl(t \ll 1\bigr) \oplus \bigl(\bigl(t \oplus (t \ll 3)\bigr) \ggg 7\bigr) \oplus 1$$

The spectral quality of the multiplier 5 is suboptimal, but on a microprocessor with no multiply instruction, 5 is the fastest nontrivial multiplier that provides full period (the entire LCG update is five instructions). The period 217 for the xor-based subgenerator is not the best possible, but updating a subgenerator of period 255 would take more instructions; 217 is the longest period possible among xor-based subgenerators that use relatively few instructions (the entire update is eight instructions) and have odd period. Computing the bitwise xor of the subgenerator outputs rather than the sum saves one instruction on a microprocessor that has no add instruction, only add-with-carry. The result is a prng that is small, fast, and adequate in quality for the application.

Generators in Marsaglia and Zaman's KISS family [Marsaglia and Zaman 1993; Rose 2018] combine three or four generators of different nature to improve the randomness of the output.

L'Ecuyer and Granger-Piché [2003] study combined generators with components from different families, focusing on combining one linear subgenerator with another subgenerator that may or may not be linear. They prove that, under appropriate conditions, combining an LFSR (which is one kind of XBG) with another generator will preserve equidistribution properties of the LFSR. They also test a number of combined generators and conclude that "combining two different types of linear generators, such as a LCG or MRG with a LFSR, seems to do as well as the linear-nonlinear combinations, at least from the empirical perspective."

The xorgens generator [Brent 2010] combines an $\mathbf{F}_2$-linear generator using four xorshift operations with a Weyl generator. The author furthermore suggests subjecting the output of the Weyl generator to a simple mixing function $\lambda\sigma.\sigma \oplus \mathit{rotate}(\sigma, \gamma)$ (for some constant $\gamma \approx w/2$) before, rather than after, adding it to the output of the xorshift generator.

Recently a number of interacting online blogs and projects have reported discovering improved mixing functions, as well as improved tools and techniques for discovering and testing them [Ettinger 2019; Evensen 2018, 2019, 2020; Mulvey 2016; Wellons 2018, 2019]; we speculate that such mixers might provide useful improvements when used in LXM algorithms. They observe that on 64-bit words, for any $0 < a < b < 64$, $\lambda\tau.\tau \oplus \mathit{rotate}(\tau, a) \oplus \mathit{rotate}(\tau, b)$ is a bijective transformation.

12 CONCLUSIONS AND FUTURE WORK

At the end of their paper, Steele, Lea, and Flood [2014] commented: "It would be a delightful outcome if, in the end, the best way to split off a new PRNG is indeed simply to 'pick one at random.' " Perhaps we have now achieved that: our testing suggests that if the arguments to the LXM constructor are themselves chosen uniformly at random—with no need to filter out any "weak values" other than ensuring that the additive parameter a is odd and that the initial state of the XBG subgenerator is nonzero—then the interleaved outputs of two or more generators constructed in this way will pass the BigCrush test suite [L'Ecuyer and Simard 2007; Simard 2009] and also the PractRand test suite [Doty-Humphrey 2011–2021] with extremely high probability.

The SplitMix algorithm used in JDK8 has 127 bits of state (of which 64 are updated per 64 bits generated) and uses 9 arithmetic operations per 64 bits generated [Steele, Lea, and Flood 2014]. The 64-bit LXM algorithm L64X128, which has a 64-bit LCG and xoroshiro128 as subgenerators, uses 255 bits of state (of which 192 are updated per 64 bits generated) and uses 17 arithmetic operations (or possibly 14, on architectures that allow operations on 32-bit halfwords of 64-bit registers) per 64 bits generated (see Figure 1). Our timing measurements confirm that on contemporary architectures and using popular compilers, the basic *generate* operation for L64X128 is somewhat slower than that for SplitMix, but never by more than a factor of 2. For applications in which it is desired to have a significantly smaller probability of statistical correlations among multiple generators being used by parallel tasks, especially when it is desirable to create new generator instances on the fly (for example, when forking new threads), L64X128 may be very attractive. This instance of LXM, and several others, will be provided in JDK17 later in 2021 as part of a new RandomGenerator API designed to make it easier for applications to use a variety of PRNG algorithms interchangeably.

Work yet to be done includes (1) exploration of even better mixing functions, (2) exploration of different congruential components, such as Marsaglia's multiply-with-carry generators, and (3) even more thorough testing of (a) LXM generator combinations and (b) a simplified generator that consists only of an additive constant (or a Weyl generator), an XBG generator, a combining function, and a mixing function.

ACKNOWLEDGMENTS

The first author thanks Melissa O'Neill of Harvey Mudd College for her critique of very early stages of this work, which has allowed us to greatly improve our presentation. We thank James Laskey, Joseph Darcy, and other members of the Oracle Java Development team for their assistance in testing and integrating specific instances of the LXM algorithm for deployment in JDK17.

REFERENCES

Austin Appleby. 2011. MurmurHash3 (project wiki page). 3 April 2011. https://github.com/aappleby/smhasher/wiki/MurmurHash3 (also at Internet Archive 10 March 2021 11:25:05).

Describes the MurmurHash3 hashing algorithm, and gives code for 32-bit and 64-bit finalizers (mixing functions). Formerly at http://code.google.com/p/smhasher/wiki/MurmurHash3 Retrieved 10 Sept. 2013.

Austin Appleby. 2016. SMHasher (GitHub project). 8 Jan. 2016. https://github.com/aappleby/smhasher (also at Internet Archive 6 April 2021 20:10:30).

The home for the MurmurHash family of hash functions along with the SMHasher test suite used to verify them.

David Blackman and Sebastiano Vigna. 2018. Scrambled Linear Pseudorandom Number Generators. 3 May 2018. 41 pages. https://arxiv.org/abs/1805.01407 To appear in *ACM Transactions on Mathematical Software*.

Lenore Blum, Manuel Blum, and Mike Shub. 1986. A simple unpredictable pseudo-random number generator. *SIAM Journal on computing* 15, 2, 364–383.

Manuel Blum and Silvio Micali. 1984. How to Generate Cryptographically Strong Sequences of Pseudorandom Bits. *SIAM J. Comput.* 13, 4, 850–864. https://doi.org/10.1137/0213053

Richard P. Brent. 2004. Note on Marsaglia's Xorshift Random Number Generators. *Journal of Statistical Software* 11, 5 (Aug.), 1–5. https://doi.org/10.18637/jss.v011.i05 Also at https://maths-people.anu.edu.au/~brent/pd/rpb218.pdf

Richard P. Brent. 2010. Some long-period random number generators using shifts and xors. 10 April 2010. 11 pages. https://arxiv.org/abs/1004.3115

Robert G. Brown, Dirk Eddelbuettel, and David Bauer. 2003–2006. Dieharder: A Random Number Test Suite, version 3.31.1. 2003–2006. https://webhome.phy.duke.edu/~rgb/General/dieharder.php (also at Internet Archive 3 July 2017 03:00:47).

Dieharder is a random number generator (rng) testing suite. It is intended to test generators, not files of possibly random numbers, as the latter is a fallacious view of what it means to be random.

Dieharder is a tool designed to permit one to push a weak generator to unambiguous failure (at the e.g. 0.0001% level), not leave one in the "limbo" of 1% or 5% maybe-failure. It also contains many tests and is extensible so that eventually it will contain many more tests than it already does.

Koen Claessen and Michał H. Pałka. 2013. Splittable Pseudorandom Number Generators Using Cryptographic Hashing. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) (*Haskell '13*). Association for Computing Machinery, New York, New York, USA (Sept.), 47–58. ISBN 9781450323833. https://doi.org/10.1145/2503778.2503784

R. R. Coveyou. 1969. Random Number Generation Is Too Important to Be Left to Chance. In *Studies in Applied Mathematics 3*, B. R. Agins and M. H. Kalos (Eds.). Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, USA, 70–111.

12.1. There is at present no RNG known to be superior to an SLCRNG [Simple Linear Congruential Random Number Generator $t_{k+1} \equiv \lambda t_k + \mu \mod P$] with a carefully chosen multiplier λ. The choice of addend μ is not of great importance.

12.2. With computers of adequate word length, the choice between multiplicative ($\mu = 0$) and other SLCRNG is of no great statistical importance. ...

12.4. The absence of the apologetic prefix "pseudo," usually affixed to RNG, is not accidental. I know of no useful definition of "stochastic process" which excludes these RNG. Nor is this a triviality; the identification of randomness with ignorance is fundamentally mistaken. Fully deterministic processes are stochastic processes. but if we know nothing about a process, one of the things we do not know is that it is a stochastic process. ...

12.6. No practical amount of empirical testings of RNG can suffice for *acceptance*. Of course, poor performance is standard in standard statistical tests as a basis for *rejection*. Hence reports of *good* performance of RNG, not supported by theory, must be deemed irrelevant.

A collection of papers presented by invitations at the Symposia on Applied Probability and Monte Carlo Methods and Modern Aspects of Dynamics sponsored by the Air Force Office of Scentific Research at the 1967 National Meetings of SIAM in Washington, D.C., June 11–15, 1967.

R. R. Coveyou and R. D. Macpherson. 1967. Fourier Analysis of Uniform Random Number Generators. *J. ACM* 14, 1 (Jan.), 100–119. https://doi.org/10.1145/321371.321379

degski. 2018. `invertible_hash_functions.hpp`. https://gist.github.com/degski/6e2069d6035ae04d5d6f64981c995ec2 (also at Internet Archive 23 March 2019 04:52:58). Code for four hash functions similar in structure to MurmurHash3.

Chris Doty-Humphrey. 2011–2021. PractRand. 2011–2021. http://pracrand.sourceforge.net/ (also at Internet Archive 12 Nov. 2020 03:13:23). Undated; the year 2011 for its first appearance has been inferred from external sources. The software is called "PractRand" but the SourceForge project name is "`pracrand`".

Mark J. Durst. 1989. Using Linear Congruential Generators for Parallel Random Number Generation. In *Proceedings of the 21st Conference on Winter Simulation* (Washington, D.C., USA) (*WSC '89*). Association for Computing Machinery, New York, New York, USA, 462–466. ISBN 0911801588. https://doi.org/10.1145/76738.76798

Shlomo Engelberg. 2015. *A Mathematical Introduction To Control Theory* (second ed.). Series in Electrical and Computer Engineering, Vol. 4. Imperial College Press, London, England. ISBN 9781860945700.

Tommy Ettinger. 2019. PelicanRNG. 16 July 2019. https://github.com/tommyettinger/sarong/blob/master/src/main/java/sarong/PelicanRNG.java (also at Internet Archive 14 April 2021 03:10:13). GitHub project; accessed April 2, 2021.

Pelle Evensen. 2018. On the mixing functions in "Fast Splittable Pseudorandom Number Generators", MurmurHash3 and David Stafford's improved variants on the MurmurHash3 finalizer (blog post). 13 July 2018. http://mostlymangling.blogspot.com/2018/07/on-mixing-functions-in-fast-splittable.html (also at Internet Archive 18 Jan. 2021 16:26:29).

Pelle Evensen. 2019. Better, stronger mixer and a test procedure (blog post). 24 Jan. 2019. http://mostlymangling.blogspot.com/2019/01/better-stronger-mixer-and-test-procedure.html (also at Internet Archive 1 Dec. 2020 06:10:00).

Pelle Evensen. 2020. NASAM: Not Another Strange Acronym Mixer! (blog post). 3 Jan. 2020. http://mostlymangling.blogspot.com/2020/01/nasam-not-another-strange-acronym-mixer.html (also at Internet Archive 7 Feb. 2021 12:33:45).

George E. Forsythe. 1951. Generation and Testing of Random Digits at the National Bureau of Standards, Los Angeles. In *Monte Carlo Method*, A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.). National Bureau of Standards Applied Mathematics Series, Vol. 12. United States Government Printing Office, Washington, DC, USA (11 June), Chapter 12, 34–35. https://babel.hathitrust.org/cgi/pt?id=osu.32435030295547 (entire proceedings). Proceedings of a symposium held June 29, 30, and July 1, 1949, in Los Angeles, California, under the sponsorship of the RAND Corporation, and the National Bureau of Standards, with the cooperation of the Oak Ridge National Laboratory.

Solomon W. Golomb. 2006. Shift Register Sequences: A Retrospective Account. In *Sequences and Their Applications – SETA 2006*, Guang Gong, Tor Helleseth, Hong-Yeop Song, and Kyeongcheol Yang (Eds.). Springer, Berlin and Heidelberg, Germany, 1–4. ISBN 978-3-540-44524-1.

Solomon W Golomb. 2017. *Shift Register Sequences* (3rd revised ed.). World Scientific, Hackensack, New Jersey, USA. ISBN 978-9814632003. https://doi.org/10.1142/9361 See also first edition, 1967, Holden-Day, San Francisco, California, USA; second edition, 1982, Aegean Park Press, Laguna Hills, California, USA, ISBN 978-0894120480.

Mark Goresky and Andrew Klapper. 2012. *Algebraic Shift Register Sequences.* Cambridge University Press, Cambridge, UK. 514 pages. ISBN 9781107014992.

Preston C. Hammer. 1951. The Mid-Square Method of Generating Digits. In *Monte Carlo Method*, A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.). National Bureau of Standards Applied Mathematics Series, Vol. 12. United States Government Printing Office, Washington, DC, USA (11 June), Chapter 11, 33. https://babel.hathitrust.org/cgi/pt?id=osu.32435030295547 (entire proceedings). Proceedings of a symposium held June 29, 30, and July 1, 1949, in Los Angeles, California, under the sponsorship of the RAND Corporation, and the National Bureau of Standards, with the cooperation of the Oak Ridge National Laboratory.

Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third ed.). Addison-Wesley, Reading, Massachusetts, USA. ISBN 9780201896848.

Doug Lea. 2013. SplittableRandom progress (private communication). 15 Nov. 2013–. Series of three email messages on the subject of MurmurHash3-like mixing functions.

Pierre L'Ecuyer. 1999a. Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Operations Research* 47, 1 (Jan.), 159–164. https://doi.org/10.1287/opre.47.1.159

Pierre L'Ecuyer. 1999b. Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure. *Math. Comp.* 68, 225 (Jan.), 249–260. https://doi.org/10.1090/S0025-5718-99-00996-5

Pierre L'Ecuyer. 2017. History of Uniform Random Number Generation. In *Proceedings of the 2017 Winter Simulation Conference (WSC)* (Las Vegas, Nevada, USA), W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page (Eds.). IEEE (Dec.), 202–230. https://doi.org/10.1109/WSC.2017.8247790 Also at https://hal.inria.fr/hal-01561551

Pierre L'Ecuyer and Jacinthe Granger-Piché. 2003. Combined generators with components from different families. *Mathematics and Computers in Simulation* 62, 3 (3 March), 395–404. https://doi.org/10.1016/S0378-4754(02)00234-3 Preprint at https://www.iro.umontreal.ca/~lecuyer/myftp/papers/linnlin.pdf Also at https://www.researchgate.net/publication/222418141 3rd IMACS Seminar on Monte Carlo Methods.

Pierre L'Ecuyer and François Panneton. 2009. $\mathbb{F}_2$-Linear Random Number Generators. In *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, Christos Alexopoulos, David Goldsman, and James R. Wilson (Eds.). International Series in Operations Research & Management Science, Vol. 133. Springer Science and Business Media, New York, New York, USA, 169–193. ISBN 9781441908162. https://doi.org/10.1007/b110059_9 Also at https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.483.8737&rep=rep1&type=pdf (retrieved 14 April 2021) Also at https://www.researchgate.net/profile/Pierre-Lecuyer/publication/225226425_F2-Linear_Random_Number_Generators/links/09e415108471274e7f000000/F2-Linear-Random-Number-Generators.pdf (retrieved 14 April 2021)

Pierre L'Ecuyer and Richard Simard. 2007. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Trans. Math. Software* 33, 4 (Aug.), Article 22, 40 pages. https://doi.org/10.1145/1268776.1268777

Pierre L'Ecuyer and Richard Simard. 2013. TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators: User's guide, compact version. 16 May 2013. http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf (also at Internet Archive 17 Feb. 2021 14:37:38).

D. H. Lehmer. 1951. Mathematical Methods in Large-Scale Computing Units. In *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery: Jointly Sponsored by The Navy Department Bureauof Ordnance and Harvard Universtiry at The Computation Laboratory, 13–16 September 1949*. The Annals of the Computation Laboratory of Harvard University, Vol. XXVI. Harvard University Press, Cambridge, Massachusetts, USA, 141–146. http://www.bitsavers.org/pdf/harvard/Proceedings_of_a_Second_Symposium_on_Large-Scale_Digital_Calculating_Machinery_Sep49.pdf (entire proceedings; also at Internet Archive 5 May 2010 13:13:16).

Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. 2012. Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) (*PPoPP '12*). Association for Computing Machinery, New York, New York, USA, 193–204. ISBN 9781450311601. https://doi.org/10.1145/2145816.2145841

Alex Losego. 2016. Super Mario World—Random Number Generation (video). YouTube. 5 Oct. 2016. Duration 14:04. https://youtu.be/q15yNrJHOak

Presents pseudocode and reverse-engineered assembly code for the PRNG used in the well-known Nintendo game *Super Mario World*. Also documents all the ways the PRNG is used in the game, from choosing awards for player achievements to making torches in dungeon scenes flicker.

How does random number generation work in *Super Mario World*? It's all explained right here.

M. Donald MacLaren and George Marsaglia. 1965. Uniform Random Number Generators. *J. ACM* 12, 1 (Jan.), 83–89. https://doi.org/10.1145/321250.321257

George Marsaglia. 1968. Random Numbers Fall Mainly in the Planes. *Proceedings of the National Academy of Sciences of the United States of America* 61, 1, 25–28. https://doi.org/10.1073/pnas.61.1.25

George Marsaglia. 1985. A Current View Of Random Number Generators. In *Computer Science and Statistics: Proceedings of the 16th Symposium on the Interface* (Atlanta, Georgia, USA), Lynne Billard (Ed.). North-Holland/Elsevier Science Publishers BV, Amsterdam, The Netherlands, 3–10. ISBN 978-0444877253. Alternate version at http://www.evensen.org/marsaglia/keynote.ps (also at Internet Archive 19 July 2004 06:39:43). Keynote address.

George Marsaglia. 1995. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness (website). Archived at https://web.archive.org/web/20010515072724/http://www.stat.fsu.edu/pub/diehard/ Contents of a CD-ROM published in 1995.

George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14, 1–6. https://doi.org/10.18637/jss.v008.i14

George Marsaglia and Wai Wan Tsang. 2002. Some Difficult-to-Pass Tests of Randomness. *Journal of Statistical Software* 7, 3, 1–9. https://doi.org/10.18637/jss.v007.i03

George Marsaglia and Arif Zaman. 1993. *The KISS Generator*. Technical Report. Florida State University, Tallahassee, Florida, USA.

Bret Mulvey. 2016. Hash Functions (blog post). 2016. https://papa.bretmulvey.com/post/124027987928/hash-functions (also at Internet Archive 7 Nov. 2020 22:32:40).

Contains a list of reversible operations on bit vectors that are easily implemented in a few machine instructions, plus a discussion of how to compute first-order avalanche statistics.

Nintendo. 1990. Super Mario World (video game for Super Nintendo Entertainment System). 21 Nov. 1990. Sold in the form of a proprietary cartridge.

Oracle. 2014. Interface `Spliterator<T>`. https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html (also at Internet Archive 29 March 2014 01:54:31). Documentation for Java^{TM} Platform Standard Ed. 8.

Oracle Corporation. 2014a. Java Platform Standard Edition 8 Documentation: Class Random (online documentation). https://docs.oracle.com/javase/8/docs/api/java/util/Random.html (also at Internet Archive 31 March 2014 01:25:08).

Oracle Corporation. 2014b. Java Platform Standard Edition 8 Documentation: Class SplittableRandom (online documentation). https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html (also at Internet Archive 30 March 2014 23:59:53).

Gregory G. Rose. 2018. KISS: A Bit Too Simple. *Cryptography and Communications* 10, 123–137. https://doi.org/10.1007/s12095-017-0225-x Includes the original code for Marsaglia and Zaman's KISS generator.

A. Rotenberg. 1960. A New Pseudo-Random Number Generator. *J. ACM* 7, 1 (Jan.), 75–77. https://doi.org/10.1145/321008.321019

Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. 2001. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Technical Report NIST Special Publication 800-22. Booz Allen and Hamilton Inc., McLean, Virginia, USA. https://apps.dtic.mil/sti/pdfs/ADA393366.pdf (also at Internet Archive 13 Aug. 2021 01:48:31). With revisions dated May 15, 2001.

Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo, and Lawrence E. Bassham III. 2010. *A Statistical Test Suite for Random and*

Pseudorandom Number Generators for Cryptographic Applications. Technical Report NIST Special Publication 800-22, Revision 1a. National Institute of Standards and Technology, United States Department of Commerce, Gaithersburg, Maryland, USA (April). 131 pages. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf (also at Internet Archive 26 June 2016 11:16:52).

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. 2011. Parallel Random Numbers: As Easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington, USA) (*SC '11*). Association for Computing Machinery, New York, New York, USA, Article 16, 12 pages. ISBN 9781450307710. https://doi.org/10.1145/2063384.2063405

Hans Georg Schaathun. 2015. Evaluation of splittable pseudo-random generators. *Journal of Functional Programming* 25 (17 June), Article e6, 19 pages. https://doi.org/10.1017/S095679681500012X Preprint at http://www.hg.schaathun.net/research/Papers/hgs2015jfp.pdf

Richard Simard. 2009. TestU01 version 1.2.3 (website). Aug. 2009. http://simul.iro.umontreal.ca/testu01/tu01.html (also at Internet Archive 12 Nov. 2020 00:20:45).

Guy Steele and Sebastiano Vigna. 2021. Computationally Easy, Spectrally Good Multipliers for Congruential Pseudorandom Number Generators. 22 Jan. 2021. 23 pages. https://arxiv.org/abs/2001.05304 Revised version to appear in *Software: Practice and Experience.* https://doi.org/10.1002/spe.3030

Guy L. Steele Jr., Doug Lea, and Christine H. Flood. 2014. Fast Splittable Pseudorandom Number Generators. In *OOPSLA '14: Proceedings of the 2014 ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications* (Portland, Oregon, USA) (*OOPSLA '14*). ACM, New York, New York, USA, 453–472. ISBN 9781450325851. https://doi.org/10.1145/2660193.2660195

W. E. Thomson. 1958. A Modified Congruence Method of Generating Pseudo-random Numbers. *Comput. J.* 1, 2, 83, 86. https://doi.org/10.1093/comjnl/1.2.83 second page (page 86) is at the end of another article: https://doi.org/10.1093/comjnl/1.2.84

Sebastiano Vigna. 2014–2021. `xoshiro` / `xoroshiro` generators and the PRNG shootout (website). 2014–2021. https://prng.di.unimi.it/ (also at Internet Archive 6 April 2021 18:02:55).

This page describes some new pseudorandom number generators (PRNGs) that David Blackman and Sebastiano Vigna have been working on recently, and a shootout comparing them with other generators.

John von Neumann. 1951. Various Techniques Used in Connection with Random Digits. In *Monte Carlo Method*, A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.). National Bureau of Standards Applied Mathematics Series, Vol. 12. United States Government Printing Office, Washington, DC, USA (11 June), Chapter 13, 36–38. https://dornsifecms.usc.edu/assets/sites/520/docs/VonNeumann-ams12p36-38.pdf (also at Internet Archive 10 Sept. 2014 21:29:50). See also https://babel.hathitrust.org/cgi/pt?id=osu.32435030295547 (entire proceedings). Proceedings of a symposium held June 29, 30, and July 1, 1949, in Los Angeles, California, under the sponsorship of the RAND Corporation, and the National Bureau of Standards, with the cooperation of the Oak Ridge National Laboratory.

John Walker. 1996. HotBits: Genuine random numbers, generated by radioactive decay (data server). May 1996. http://www.fourmilab.ch/hotbits/ (also at Internet Archive 1 March 2021 15:29:15).

HotBits is an Internet resource that brings genuine random numbers, generated by a process fundamentally governed by the inherent uncertainty in the quantum mechanical laws of nature, directly to your computer in a variety of forms.

Tony T. Warnock. 1983. Synchronization of random number generators. *Congressus numerantium* 37, 135–144.

Henry S. Warren, Jr. 2012. *Hacker's Delight.* Pearson Education, Boston, Massachusetts, USA. ISBN 9780133085013.

Chris Wellons. 2018. Prospecting for Hash Functions (blog post). 31 July 2018. https://nullprogram.com/blog/2018/07/31/ (also at Internet Archive 25 Nov. 2020 19:01:27).

Description of software that searches for better mixing functions, using Bret Mulvey's list of invertible operations as primitive building blocks.

Christopher Wellons. 2019. Hash Function Prospector (GitHub project). March 2019. https://github.com/skeeto/hash-prospector (also at Internet Archive 12 Nov. 2020 01:46:36).

Software that searches for good mixing functions.

Stephen Wolfram. 2016. Solomon Golomb (1932–2016) (blog post). 25 May 2016. https://writings.stephenwolfram.com/2016/05/solomon-golomb-19322016/ (also at Internet Archive 26 Oct. 2019 12:39:30).

Read the specifications for 3G, LTE, Wi-Fi, Bluetooth, or for that matter GPS, and you'll find mentions of polynomials that determine the shift register sequences these systems use to encode the data they send. Solomon Golomb is the person who figured out how to construct all these polynomials.