Marsaglia proposed xorshift generators are a class of very fast, good-quality pseudorandom number generators. Subsequent analysis by Panneton and L’Ecuyer has lowered the expectations raised by Marsaglia’s article, showing several weaknesses of such generators. Nonetheless, many of the weaknesses of xorshift generators fade away if their result is scrambled by a nonlinear operation (as originally suggested by Marsaglia). In this article we explore the space of possible generators obtained by multiplying the result of a xorshift generator by a suitable constant. We sample generators at 100 points of their state space and obtain detailed statistics that lead us to choices of parameters that improve on the current ones. We then explore for the first time the space of high-dimensional xorshift generators, following another suggestion in Marsaglia’s article, finding choices of parameters providing periods of length $2^{1024} - 1$ and $2^{4096} - 1$. The resulting generators are of extremely high quality, faster than current similar alternatives, and generate long-period sequences passing strong statistical tests using only eight logical operations, one addition, and one multiplication by a constant.

Categories and Subject Descriptors: G.3 [Probability and Statistics]: Random Number Generation; G.3 [Probability and Statistics]: Experimental Design

General Terms: Algorithms, Experimentation, Measurement

Additional Key Words and Phrases: Pseudorandom number generators

ACM Reference Format:
Sebastiano Vigna. 2016. An experimental exploration of Marsaglia’s xorshift generators, scrambled. ACM Trans. Math. Softw. 42, 4, Article 30 (June 2016), 23 pages.
DOI: http://dx.doi.org/10.1145/2845077

1. INTRODUCTION
xorshift generators are a simple class of pseudorandom number generators introduced by Marsaglia [2003]. In Marsaglia’s view, their main feature is speed: In particular, a xorshift generator with a 64-bit state generates a new 64-bit value using just three 64-bit shifts and three 64-bit xors (i.e., exclusive ors), thus making it possible to generate hundreds of millions of values per second.

Subsequent analysis by Brent [2004] showed that the bits generated by xorshift generators are equivalent to certain linear feedback shift registers. Panneton and L’Ecuyer [2005] analyzed in detail the theoretical properties of the generators and found empirical weaknesses using the TestU01 suite [L’Ecuyer and Simard 2007]. They proposed an increase in the number of shifts, or combination with another generator, to improve quality.
In the first part of this article, as a warm-up, we explore experimentally the space of xorshift generators with 64 bits of state using statistical test suites. We sample generators at 100 points of their state space to easily identify spurious failures. Marsaglia proposes some choice of parameters, that, as we will see, and as already reported by Panneton and L’Ecuyer [2005], are not particularly good. We report results that are actually worse than those of Panneton and L’Ecuyer as we use the entire 64-bit output of the generators. While we can suggest some good parameter choices, the result remains poor.

Thus, we turn to the idea of scrambling the result of a xorshift generator using a multiplication, as it is typical, for instance, in the construction of practical hash functions due to the resulting avalanching behavior (bits of the result depend on several bits of the input). This method is actually suggested in passing in Marsaglia’s article. The third edition of the classic “Numerical Recipes” [Press et al. 2007], indeed, proposes this construction for a basic, all-purpose generator. From the wealth of data so obtained, we derive generators with better statistical properties than those suggested in “Numerical Recipes.”

In the last part of the article, we follow the suggestion about high-dimensional generators contained in Marsaglia’s article and compute several choices of parameters that provide full-period xorshift generators with a state of 1024 and 4096 bits. Once again, we propose generators that use a multiplication to scramble the result.

At the end of the article, we apply the same methodology to a number of popular non-cryptographic generators, and we discover that our high-dimensional generators are actually faster and of higher or equivalent statistical quality, as assessed by statistical test suites, than the alternatives.

The software used to perform the experiments described in this article is distributed by the author under the GNU General Public License. Moreover, all files generated during the experiments are available from the author. They contain a large amount of data that could be further analyzed (e.g., by studying the distribution of $p$-values over the seeds). We leave this issue open for further work.

2. AN INTRODUCTION TO xorshift GENERATORS

The basic idea of xorshift generators is that their state is modified by applying repeatedly a shift and an exclusive-or (xor) operation. In this article, we consider 64-bit shifts and states made of $2^n$ bits, with $n \geq 6$. We usually append $n$ to the name of a family of generators when we need to restrict the discussion to a specific state size.

For xorshift64 generators Marsaglia suggests a number of possible combination of shifts, shown in Figure 1. Not all choices of parameters give a full $(2^{64} - 1)$ period: There are 275 suitable choices of $a$, $b$, and $c$ and eight variants, totaling 2,200 generators.

In linear-algebra terms, if $L$ is the $64 \times 64$ matrix on $\mathbb{Z}/2\mathbb{Z}$ that effects a left shift of one position on a binary row vector (i.e., $L$ is all zeroes except for ones on the principal subdiagonal) and if $R$ is the right-shift matrix (the transpose of $L$), then each left/right shift and xor can be described as a linear multiplication by $(I + L^s)$ or $(I + R^s)$, respectively, where $s$ is the amount of shifting.\footnote{A more detailed study of the linear algebra behind xorshift generators can be found in Marsaglia [2003] and Panneton and L’Ecuyer [2005].} For instance, algorithm $A_0$ of Figure 1 is equivalent to the $\mathbb{Z}/2\mathbb{Z}$-linear transformation

$$X_1 = (I + L^a)(I + R^b)(I + L^c).$$

It is useful to associate with a linear transformation $M$ its characteristic polynomial $P(x) = \det(M - xI)$.\footnote{A more detailed study of the linear algebra behind xorshift generators can be found in Marsaglia [2003] and Panneton and L’Ecuyer [2005].}
An Experimental Exploration of Marsaglia’s xorshift Generators, Scrambled

The eight possible xorshift64 algorithms. The list is actually derived from Panneton and L’Ecuyer [2005], as they correctly remarked that two of the eight algorithms proposed by Marsaglia were redundant, whereas two (A6 and A7) were missing. On the right side we report the name of the linear transformation associated to the algorithm as denoted by Panneton and L’Ecuyer [2005]. With our numbering, algorithms A2i and A2i+1 are conjugate by reversal. Note that contiguous shifts in the same direction can be exchanged without affecting the resulting algorithm. We normalized such contiguous shifts so their letters are lexicographically sorted.

The associated generator has maximum-length period if and only if $P(x)$ is primitive over $\mathbb{Z}/2\mathbb{Z}$. This happens if $P(x)$ is irreducible and if $x$ has maximum period in the ring of polynomial over $\mathbb{Z}/2\mathbb{Z}$ modulo $P(x)$, that is, if the powers $x, x^2, \ldots, x^{2^n-1}$ are distinct modulo $P(x)$. Finally, to check the latter condition is sufficient to check that $x^{(2^n-1)/p} \neq 1 \mod P(x)$ for every prime $p$ dividing $2^n - 1$ [Lidl and Niederreiter 1994].

The weight of $P(x)$ is the number of terms in $P(x)$, that is, the number of nonzero coefficients. It is considered a good property for generators of this kind that the weight is close to $n/2$, that is, that the polynomial is neither too sparse nor too dense [Compagner 1991].

Note that the family of algorithms of Figure 1 is intended to generate 64-bit values. This means that the entire output of the algorithm should be used when performing tests. We will see that this has not always been the case in previous literature.

3. SETTING UP THE EXPERIMENTS

In this article we want to explore experimentally the space of a number of xorshift-based generators. Our purpose is to identify variants with full period that have particularly good statistical properties and test whether claims about good parameters made in the previous literature are confirmed.

The basic idea is that of sampling the generators by executing a battery of tests starting with 100 different seeds that are equispaced in the state space. More precisely, if the state is made of $n$ bits, then we use the seeds $1 + i \lfloor 2^n/100 \rfloor$, $0 \leq i < 100$. The tests produce a number of statistics, and we decided to use as score the number of failed tests. A higher score, thus, means lower quality. Running multiple tests makes it easy to rule out spurious failures, as suggested also by Rukhin et al. [2001] in the context of cryptographic applications.\(^2\)

We use two tools to perform our tests. The first and most important is TestU01, a test suite developed by L’Ecuyer and Simard [2007] that contains several tests oriented

---

\(^2\)We remark that, arguably, a more principled choice would be choosing seeds that are equispaced in the sequence of states traversed by the generator. Unfortunately, this is possible only for generators with “jump-ahead” primitives, and we want our methodology to be universal. We checked that all sequences of states used in our tests on generators with 64 bits of state do not overlap. The chance that this happens with more than 128 bits of state is negligible.
towards the generation of uniform real numbers in $[0..1)$.\footnote{We use the double-dot notation for intervals introduced by C. A. R. Hoare and Lyle Ramshaw [Graham et al. 1994].} We also perform tests using Dieharder, a suite of tests developed by Brown [2013], both as a sanity check and to compare the power of the two suites. Dieharder contains all original tests from Marsaglia’s Diehard, plus many more additional tests. We refer frequently to the specific type of tests failed: The reader can refer to the TestU01 and Dieharder documentation for more information.

We consider a test failed if its $p$-value is outside of the interval $[0.001..0.999]$. This is the interval outside which TestU01 reports a test by default. Sometimes a much stricter threshold is used (for instance, L’Ecuyer and Simard [2007] use $[10^{-10}..1 - 10^{-10}]$ when applying TestU01 to a variety of generators), and weaker $p$-values are called suspicious values, but since we are going to repeat the test 100 times, we can use relatively weak $p$-values: Spurious failures will appear rarely, and we can catch borderline cases (e.g., tests failing on 50% of the seeds) that give us useful information.

We call systematic a failure that happens for all seeds. For all such failures in our tests, $p$-values are smaller than $10^{-15}$. Thus, all conclusions drawn in this article based on systematic failures would not change even if we lowered significantly the failure threshold. More generally, 90% of the $p$-values of failed tests are actually smaller than $10^{-6}$.

We remark that our choice (counting the number of failures) is somewhat rough; for example, we consider the same failure a $p$-value very close to 0 and a $p$-value just below 0.001. Indeed, other, more sophisticated methods might be used to aggregate the result of our samples: combining $p$-values, for instance, or computing a $p$-value of $p$-values [Rukhin et al. 2001]. However, our choice is very easy to interpret, and multiple samples partially compensate this problem (spurious failures will appear in few samples).

Of course, the number of experiments is very large—in fact, our experiments were carried out using hundreds of cores in parallel and, overall, they add up to more than a century of computational time. Our strategy is to apply a very fast test to all generators and seeds, in the hope of isolating a small group of generators that behave significantly better with respect to these tests. Stronger tests can then be applied to this subset. The same strategy has been followed by Panneton [2004] in the experimental study of xorshift generators contained in his Ph.D. thesis.

TestU01 offers three different predefined batteries of tests (SmallCrush, Crush, and BigCrush) with increasing computational cost and increased difficulty. Unfortunately, Dieharder does not provide such a segmentation.

Note that Dieharder has a concept of “weak” success and a concept of “failure,” depending on the $p$-value of the test, and we used command-line options to align its behavior with that of TestU01: A $p$-value outside of the range $[0.001..0.999]$ is a failure. Moreover, we disabled the initial timing tests so exactly the same stream of 64-bit numbers is fed to the two test suites.

In both cases we implemented our own xorshift generator. Some care is needed in this phase, as both TestU01 and Dieharder are inherently 32-bit test suites: Since we want to test xorshift as a 64-bit generator, it is important that all bits produced are actually fed into the test. For this reason, we implemented the generation of a uniform real value in $[0..1)$ by dividing the output of the generator by $2^{64}$, but we implemented the generation of uniform 32-bit integer values by returning first the lower and then the upper 32 bits of each 64-bit generated value.\footnote{If a real value is generated when the upper 32 bits of the last value are available, then they are simply discarded.} A possible downside of this approach
is that we might fail to detect some failure in the high bits (of the 64-bit, full output) due to the interleaving process: However, the fact that in our tests xorshift generators generate many more failures than those reported previously [Panneton and L’Ecuyer 2005] suggests that the approach is well founded.

An important consequence of this choice is that some of the bits are actually not used at all. When analyzing pseudorandom real numbers in the unit interval, there is an unavoidable bias towards high bits, as they are more significant. The very lowest bits have lesser importance and will in any case be perturbed by numerical errors. For this reason, it is a good practice to run tests both on a generator and on its reverse [Press et al. 2007]. In our case, this is even more necessary, as the lowest eleven bits returned by the generator are not used at all due to the fact that the mantissa of a 64-bit floating-point number is formed by 53 bits only.

A recent example shows the importance of testing the reverse generator. Saito and Matsumoto [2014] propose a different way to eliminate linear artifacts: Instead of multiplying the output of an underlying xorshift generator (with 128 bits of state and based on 32-bit shifts) by a constant, they add it (in \(\mathbb{Z}/2^{32}\mathbb{Z}\)) with the previous output. Since the sum in \(\mathbb{Z}/2^{32}\mathbb{Z}\) is not linear over \(\mathbb{Z}/2\mathbb{Z}\), the result should be free of linear artifacts. However, while their generator passes BigCrush, its reverse fails systematically the LinearComp, MatrixRank, MaxOft, and Permutation test of BigCrush, which highlights a significant weakness in its lower bits.

We remark that in this article we do not pursue the search for equidistribution—the property that all tuples of consecutive values, seen as vectors in the unit cube, are evenly distributed, as done, for instance, by Panneton and L’Ecuyer [2005]. Brent [2010] has already argued in detail that for long-period generators equidistribution is not particularly desirable, as it is a property of the whole sequence produced by the generator, and in the case of a long-period generator only a minuscule fraction of the sequence can be actually used. Moreover, equidistribution is currently impossible to evaluate exactly for long-period nonlinear generators, and in the formulation commonly used in the literature it is known to be biased towards the high bits [L’Ecuyer and Panneton 2005]: For instance, the WELL1024a generator has been designed to be maximally equidistributed [Panneton et al. 2006], and indeed it has measure of equidistribution \(\Delta_1 = 0\), but the generator obtained by reversing its bits has \(\Delta_1 = 366\), a quite counterintuitive result, as in general we expect all bits to be equally important.

Another problem with equidistribution is that it is intrinsically unstable, unless we restrict its usage to the class of linear generators, only. Indeed, if we take a maximally equidistributed sequence, no matter how long, and we flip the most significant bit of a single element of the sequence, the new sequence will have the worst possible \(\Delta_1\). For instance, by flipping the most significant bit of a single chosen value out of the output of WELL1024a, we can turn its equidistribution measure to \(\Delta_1 = 4143\). But for any statistical or practical purpose, the two sequences are indistinguishable—we are modifying 1 bit of \(2^5(2^{1024} - 1)\). However, in general this paradoxical behaviour is not a big issue, because the modified sequence can no longer be emitted by a linear generator.

We note that since multiplication by an invertible constant induces a permutation of the space of 64-bit values (and thus of \(t\)-tuples of such values), it preserves some of the equidistribution properties of the underlying generator (this is true of any bijective scrambling function); more details will be given in the rest of the article.

---

5That is, on the generator obtained by reversing the order of the 64 bits returned.
Fig. 2. Score-rank plot of the distribution of SmallCrush scores for the 2,200 possible full-period xorshift64 generators.

Table I. Best Four xorshift64 Generators Following SmallCrush

| Algorithm   | Failures | Conjugate | Failures | Overall | W   |
|-------------|----------|-----------|----------|---------|-----|
| $A_0(11, 31, 18)$ | 111      | $A_0(11, 31, 18)$ | 120      | 231     | 25  |
| $A_0(8, 29, 19)$  | 155      | $A_0(8, 29, 19)$ | 115      | 270     | 35  |
| $A_0(8, 29, 19)$  | 159      | $A_1(8, 29, 19)$ | 112      | 271     | 35  |
| $A_0(11, 31, 18)$ | 130      | $A_1(11, 31, 18)$ | 150      | 280     | 25  |
| $A_0(13, 7, 17)$  | 276      | $A_1(13, 7, 17)$ | 802      | 1078    | 25  |

4. RESULTS FOR xorshift64 GENERATORS

First, all generators fail at all seeds the MatrixRank test from TestU01’s SmallCrush suite. A score-rank plot of the SmallCrush scores for all generators is shown in Figure 2. The plot associates with abscissa $x$ the number of generators with $x$ or more failures. We observe immediately that there is a wide range of quality among the generators examined. The “bumps” in the plot correspond to new tests failed systematically.

A closer inspection would confirm that there is just a weak correlation between scores of algorithms conjugate by reversal, because of the bias of TestU01 towards high bits. We thus report in Table I the best four generators by combined scores (i.e., adding the scores of conjugate generators), which are the only ones failing systematically just the MatrixRank test. The table reports also results for the generator $A_0(13, 7, 17)$ suggested by Marsaglia in his original article, claiming that it “will provide an excellent period $2^{64} − 1$ RNG, […] but any of the above 2200 choices is likely to do as well.” Clearly, this is not the case: $A_0(13, 7, 17)/A_1(13, 7, 17)$ ranks 655 in the combined SmallCrush ranking and fails systematically several tests.

Sanity check 1. Is the result of our experiments dependent on our seed choice? To answer this question, we repeated our experiments on xorshift64 generators with SmallCrush on a different set of seeds, namely the integers in the interval $[1..100]$. Kendall’s

---

6Panneton and L’Ecuyer [2005] reports that half of the generators fail this test, but the authors have chosen to use only 32 of the 64 generated bits as output bits, in practice applying a kind of decimation to the output of the generator.

7Score-rank plots are the numerosity-based discrete analogous of the complementary cumulative distribution function of scores. They give a much clearer picture than frequency dot plots when the data points are scattered and highly variable.
An Experimental Exploration of Marsaglia's xorshift Generators, Scrambled

Table II. The Generators of Table I Tested with BigCrush

| Algorithm     | Failures | Conjugate | Failures | Overall |
|---------------|----------|-----------|----------|---------|
| $A_2(8, 29, 19)$ | 747      | $A_3(8, 29, 19)$ | 884      | 1633    |
| $A_0(11, 31, 18)$ | 748      | $A_1(11, 31, 18)$ | 926      | 1674    |
| $A_2(4, 35, 21)$ | 961      | $A_3(4, 35, 21)$ | 1444     | 2405    |
| $A_0(13, 7, 17)$ | 1049     | $A_1(13, 7, 17)$ | 5454     | 6503    |

Table III. The Generators of Table I Tested with Dieharder

| Algorithm     | Failures | Conjugate | Failures | Overall |
|---------------|----------|-----------|----------|---------|
| $A_2(8, 29, 19)$ | 179      | $A_3(8, 29, 19)$ | 182      | 358     |
| $A_0(11, 31, 18)$ | 181      | $A_1(11, 31, 18)$ | 186      | 367     |
| $A_2(4, 35, 21)$ | 189      | $A_3(4, 35, 21)$ | 187      | 376     |
| $A_0(13, 7, 17)$ | 183      | $A_1(13, 7, 17)$ | 1352     | 1535    |

$\tau$ [Kendall 1938, 1945] between the two rankings is 0.98, which makes it clear that the dependence on the seed is negligible. In particular, the four best conjugate pairs in Table I are the same with both seeds.

To gather more information, we ran the full BigCrush suite and Dieharder on our four best generators, on Marsaglia’s choice and on the best choice from “Numerical Recipes”: The results are given in Tables II and III. Even the four best generators fail now systematically the BirthdaySpacings, MatrixRank, and LinearComp tests. The first two generators, however, turn out to perform slightly better than other two. We also notice that BigCrush draws a much thicker line between our four best generators and the other ones, which now fail several more tests. Not surprisingly, Dieharder cannot really separate our four best generators from $A_2(4, 35, 21)/A_3(4, 35, 21)$.

4.1. Equidistribution

It is interesting to compare the ranking provided by equidistribution properties and that provided by statistical tests. Note that a xorshift64 generator is one-dimensionally equidistributed, that is, every 64-bit value appears exactly once except for zero. We refer to the already quoted article by Panneton and L’Ecuyer [2005] for a detailed description of the equidistribution statistics $\Delta_1$, the sum of dimension gaps: A lower value is better. A maximally distributed generator has $\Delta_1 = 0$, and we will refer to $\Delta_1$ as to the equidistribution score. We computed the equidistribution score for all generators using the implementation of Harase’s algorithm [Harase 2011] contained in the MTToolBox package from Saito [2013]. Similarly to SmallCrush scores, $\Delta_1$ has high-bits bias and a quite strong one [L’Ecuyer and Panneton 2005]. For a fair comparison, we thus combine the $\Delta_1$ score of a generator and of its reverse.

Figure 3 shows that there is some correlation ($\tau = 0.58$) between combined SmallCrush scores and combined equidistribution scores. Nonetheless, even if equidistribution is able to detect reliably generators with a very bad SmallCrush score, is not so good at detecting the generators with the best score, as is visible from the quite noisy lower left part of the plot. Indeed, when we restrict our attention to the best 30 generators (by combined SmallCrush scores) Kendall’s $\tau$ drops to 0.3. The first two generators by combined equidistribution score, $A_4(8, 29, 19)$ and $A_5(8, 29, 19)$, rank 20 (combined score 361) and 170 (score 596) in the combined SmallCrush test. When analyzed with the more powerful lens of BigCrush, they have combined scores 3441 and 4082, respectively, and fail systematically almost 20 additional tests with respect to
the top four generators of Table II. Definitely, choosing among xorshift64 generators by equidistribution score alone is not a good idea.

5. AN INTRODUCTION TO xorshift64* GENERATORS

Since a xorshift64 generator exhibits evident linearity artifacts, the next obvious step is to perturb its output using a nonlinear (in \(\mathbb{Z}/2\mathbb{Z}\) sense) transformation. A natural candidate is multiplication by a constant, also because such an operation is very fast in modern processors. Note that the current state of the generator is multiplied by a constant before returning it, but the state itself is not affected by the multiplication: Thus, the period is the same.

We call such a generator xorshift*. By choosing a constant invertible modulo \(2^{64}\) (i.e., odd), we can guarantee that the generator will output a permutation of the sequence output by the underlying xorshift generator.

This approach was noted in passing in Marsaglia’s article, and it is also proposed in a more systematic way in the third edition of “Numerical Recipes” [Press et al. 2007] to create a very fast, good-quality pseudorandom number generator. However, in the latter case, the authors first compute allegedly good triples for xorshift using Diehard (with results that markedly differ from ours, and in strident contrast with TestU01’s results, as discussed in Section 4) and then choose a multiplier. There is no reason why the best triples for a xorshift64 generator (which are computed empirically) should continue to be such in a xorshift64* generator, and, indeed, we will see that this is not the case.

We thus repeated the experiments of the previous section on xorshift64* generators. To choose scrambling constants, we followed the heuristic considerations of Press et al. [2007]. We consider primitive (e.g., full-period) elements of the multiplicative group of \(\mathbb{Z}/2^{64}\mathbb{Z}\): These elements have no fixed point except for zero, which is a very desirable property for a scrambling function. Moreover, we choose from L’Ecuyer [1999] primitive elements that have good qualities as multiplicative congruential linear generators, as we expect that multiplication by such elements will combine bits in a nontrivial way.

We use a standard theoretical measure of quality, the figure of merit, which is a normalized best distance between the hyperplanes of families covering tuples of length \(t\) given by successive outputs of the generators (see L’Ecuyer [1999] for details). Since \(t\) is an additional parameter, to further understand the dependency on the multiplier we
used three different multipliers, shown in Table IV, which have good figures of merit for different $t$’s. The first multiplier, $M_{32}$ (the one used in Press et al. [2007]) and the second, $M_{8}$, have been taken from L’Ecuyer [1999]. The third, $M_{2}$, was kindly provided by Richard Simard.

We remark that many other choices for scrambling the output of a generator are possible, like adding or xoring a fixed word, xoring the output with the output of another generator, or using a bijective function with strong avalanching behavior, such as those used in the construction of high-quality hash functions. The three factors we considered in our choice are speed, good results in statistical test suites, and preservation of some equidistribution properties (similarly to the approach taken in L’Ecuyer and Granger-Piché [2003]). For instance, xoring with an additive Weyl generator (another suggestion in Marsaglia’s article) makes it in general impossible to prove any equidistribution property—not even that all 64-bit value except for zero are output by the generator. Multiplication by a constant is a very fast operation in modern processors, and mixing linear operations on $\mathbb{Z}/2\mathbb{Z}$ with operations in the ring $\mathbb{Z}/2^{64}\mathbb{Z}$ is a standard technique to avoid visible artifacts from either type of algebraic structure. A drawback is that the lowest bit is, in fact, not scrambled, and thus it is identical to the lowest bit of the underlying xorshift generator.\(^8\)

6. RESULTS FOR xorshift64* GENERATORS

The scatter plot in Figure 4 shows that there is essentially no correlation between the scores assigned by SmallCrush to a generator and its reverse ($\tau = 0.15$).\(^9\) Another interesting observation on Figure 4 is that the lower right half is essentially empty. So bad generators have a bad reverse, but there are good generators with a very bad reverse. This suggests that the quality of a xorshift64* generator can vary wildly from the low to the high bits.

\(^8\)As remarked by one of the referees, since our multipliers are all equal to 1 modulo 4, this is true also of the second-lowest bit.

\(^9\)We report plots only for $M_{32}$, as the ones for the other multipliers are visually identical.
Fig. 5. Score-rank plot of the distribution of SmallCrush scores for the 2200 possible xorshift64* generators with multiplier $M_{32}$.

Fig. 6. Scatter plots for Crush (left) and Dieharder (right) scores on xorshift64* generators with multiplier $M_{32}$ and their reverse for the 152 best generators.

A score-rank plot of the SmallCrush scores for all generators shown in Figure 5 provides us with further interesting information: Almost all generators have no systematic failure, but only about half of the reverse generators have no systematic failure. Moreover, the distribution of standard generators degrades smoothly, whereas the distribution of reverse generators sports again the “bump” phenomenon we observed in Figure 2.

Since we need to reduce the number of candidates to apply stronger tests, in the case of $M_{32}$, we decided to restrict our choice to generators with three or fewer overall failed tests, which left us with 152 generators. Similar cutoff points were chosen for $M_8$ and $M_2$.

These generators were few enough so we could apply both Crush and Dieharder. Once again, we examine the correlation between the score of a generator and its reverse by means of the scatter plots in Figure 6, which confirm the high-bits bias, albeit less so in the Dieharder case.

In Figure 7 we compare instead the two scores (Crush and Dieharder) available. The most remarkable feature is there are no points in the upper left corner: There is no generator that is considered good by Crush but not by Dieharder. On the contrary, Crush heavily penalizes (in particular, because of the score on the reverse generator) a
An Experimental Exploration of Marsaglia's xorshift Generators, Scrambled

Fig. 7. A scatter plot of Crush and Diehard combined scores of the 152 SmallCrush-best xorshift64* generators. The plot is in log-log scale to accommodate some very high values returned by Crush on reverse generators. The lower-left “sweet spot” corner contains generators that never fail systematically (not even reversed) in both test suites.

Fig. 8. Score-rank plot of the distribution of Crush scores for the 152 SmallCrush-best xorshift64* generators using multiplier $M_{32}$.

large number of generators. The generators we will select in the end all belong to the small cloud in the lower left corner, where the two test suite agree.

The score-rank plot in Figure 8 shows that our strategy pays off: We started with 152 generators with fewer than three failures, but, by analyzing them with the more powerful lens provided by Crush, we get a much more fine-grained analysis: In particular, only 73 of them give no systematic failure, and they all belong to the “sweet spot” of Figure 7, that is, they do not give any systematic failure in Dieharder, too.

Finally, we selected for each multiplier the eight generators with the best Crush scores and applied the BigCrush suite: We obtained several generators failing systematically the MatrixRank test only and shown in Table V (which should be compared with Table II).

6.1. Equidistribution

Multiplication by an invertible element just permutes the elements of $\mathbb{Z}/2^{64}\mathbb{Z}$ leaving zero fixed, so a xorshift64* generator, like the underlying xorshift64 generator, is one-dimensionally equidistributed.
Table V. Results of BigCrush on the Best Eight xorshift64* Generators Found by SmallCrush and Crush in Sequence. The Generators Fail Systematically Only MatrixRank Failures

| Algorithm | S | R | + | W |
|-----------|---|---|---|---|
| $M_{32}$  |   |   |   |   |
| $A_7(11, 5, 45)$ | 226 | 128 | 354 | 23 |
| $A_7(17, 23, 52)$ | 232 | 130 | 362 | 25 |
| $A_7(12, 25, 27)$ | 230 | 133 | 363 | 31 |
| $A_7(17, 23, 29)$ | 229 | 137 | 366 | 21 |
| $A_7(14, 23, 33)$ | 238 | 132 | 370 | 32 |
| $A_7(17, 47, 29)$ | 231 | 141 | 372 | 24 |
| $A_7(16, 25, 43)$ | 238 | 138 | 376 | 31 |
| $A_7(23, 9, 57)$ | 242 | 134 | 376 | 19 |
| $M_8$      |   |   |   |   |
| $A_7(11, 5, 32)$ | 229 | 122 | 351 | 13 |
| $A_7(8, 31, 17)$ | 229 | 126 | 355 | 21 |
| $A_7(3, 21, 31)$ | 230 | 141 | 371 | 33 |
| $A_7(17, 45, 22)$ | 241 | 133 | 374 | 27 |
| $A_7(8, 37, 21)$ | 239 | 136 | 375 | 33 |
| $A_7(13, 47, 23)$ | 232 | 144 | 376 | 27 |
| $A_7(13, 35, 30)$ | 244 | 136 | 380 | 27 |
| $A_7(9, 37, 31)$ | 243 | 141 | 384 | 27 |
| $M_5$      |   |   |   |   |
| $A_7(13, 19, 28)$ | 228 | 128 | 356 | 23 |
| $A_7(9, 21, 40)$ | 228 | 132 | 360 | 35 |
| $A_7(14, 23, 33)$ | 234 | 142 | 376 | 29 |
| $A_7(19, 43, 27)$ | 239 | 137 | 376 | 23 |
| $A_7(17, 47, 28)$ | 240 | 137 | 377 | 25 |
| $A_7(16, 11, 27)$ | 234 | 144 | 378 | 25 |
| $A_7(4, 35, 15)$ | 230 | 149 | 379 | 35 |
| $A_7(13, 21, 18)$ | 238 | 144 | 382 | 31 |

7. HIGH DIMENSION

Marsaglia [2003] describes a strategy for xorshift generators in high dimension: The idea is to use always three low-dimensional shifts but locating them in the context of a larger $t \times t$ block matrix of the form

$$M = \begin{pmatrix}
0 & 0 & 0 & \ldots & 0 & (I + L^a)(I + R^b) \\
I & 0 & 0 & \ldots & 0 & 0 \\
0 & I & 0 & \ldots & 0 & 0 \\
0 & 0 & I & \ldots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \ldots & I & (I + R^c)
\end{pmatrix}. $$

Marsaglia notes that even in this restricted form there are matrices of full period (he provides examples for 32-bit shifts up to 160 bits). However, this route has not been explored for high-dimensional (say, more than 1024 bits of state) generators. The only similar approach is that proposed by Brent [2007] with his xor2gens generators, which, however, uses four shifts. The obvious question is thus as follows: Is the additional shift really necessary to pass a strong statistical test such as BigCrush? We are thus going to look for good, full-period generators with 1024 or 4096 bits of state using 64-bit basic shifts.\(^{10}\)

\(^{10}\)The reason why the number 4,096 is relevant here is that we know the factorization of Fermat’s numbers $2^{2^k} + 1$ only up to $k = 11$. When more Fermat numbers will be factorized, it will be possible to design
The output of such generators will be given by the last 64 bits of the state. It is well known [Brent 2004; Niederreiter 1992] that every bit of state satisfies a linear recurrence (defined by the characteristic polynomial) with full period, so a fortiori the last 64 bits have full period, too.

Since we already know that some deficiencies of low-dimensional xorshift generators are well corrected by multiplication by a constant, we will follow the same approach, thus looking for good xorshift* generators of high dimension.\footnote{As in the xorshift64 case, different choices for the shifts are possible. We will not pursue them here.} Note that since multiplication by an integer invertible in $\mathbb{Z}/2^{64}\mathbb{Z}$ is a permutation of $\mathbb{Z}/2^{64}\mathbb{Z}$, a high-dimension xorshift* generator has the same period of the underlying xorshift generator.

We cannot, in principle, claim full period if we look at a single bit of the output of a xorshift* generator; but this property can be easily proved by purely combinatorial means:

**Proposition 7.1.** Let $x_0, x_1, \ldots, x_{2^n-2}$ be a list of $2^t$-bit values, $t < n$, such that every value appears $2^{n-t}$ times, except for 0, which appears $2^{n-t} - 1$ times. Then, for every fixed bit $k$ the associated sequence has period $2^n - 1$.

**Proof.** Suppose that there is a $k$ and a $p \mid 2^n - 1$ such that the $k$th bit of $x_0, x_1, \ldots, x_{2^n-2}$ has period $p$ (that is, the sequence of bits associated with the $k$th bit is made by $(2^n - 1)/p$ repetitions of the same sequence of $p$ bits). The $k$th bit runs through $2^n - 1$ zeroes and $2^n - 1$ ones (as there is a missing zero in the output sequence). This means that $(2^n - 1)/p \mid 2^n - 1$, too, as the same number of ones must appear in every repeating subsequence, and since $(2^n - 1)/p$ is odd this implies $p = 2^n - 1$. \hfill $\square$

**Corollary 7.2.** Every bit of the output of a full-period xorshift* generator has full period.

### 7.1. Finding Good Shifts

The first step is to identify values of $a, b$, and $c$ for which the generator has maximum period using the primitivity check on the characteristic polynomial. We performed these computations using the algebra package Fermat [Lewis 2013], with the restriction that $a + b \leq 64$ and that $a$ is coprime with $b$ (see Brent [2007] for the rationale behind this choices, which significantly reduces the search space). The resulting sets of values are those shown in Tables VI and VIII.

For a state of 1024 bits, we obtain 20 possible parameter choices, which we examined in combination with our three multipliers both through BigCrush and through Dieharder. The results, reported in Tables VI and VII, are excellent: with the exception of two pathological choices, no test is failed systematically. For a state of 4096 bits (Tables VIII and IX) there are 10 possible parameter choices, and no generator fails a test systematically.

### 7.2. Equidistribution

Looking at the shape of the matrix defining high-dimensional xorshift generators, it is clear that if the state is made of $n$ bits the last $n/64$ output values, concatenated, are equal to the current state. This implies that such generators are $n/64$-dimensionally equidistributed (i.e., every $n/64$-tuple of consecutive 64-bit values appears exactly once, except for a missing tuple of zeroes), so xorshift1024 generators are 16-dimensionally xorshift or xorgens generators with larger state space [Brent 2007]. Note that, however, in practice a period of $2^{1024} - 1$ is more than sufficient for any purpose. For example, even if $2^{100}$ computers were to generate sequences of $2^{100}$ numbers starting from random seeds using a generator with period $2^{1024}$, the chances that two sequences overlap would be less than $2^{-724}$.
Table VI. Results of BigCrush on the xorshift1024* Generators. The Last Two Generators Fail Systematically.

| Generators | MatrixRank | SumCollector | WeightDistrib |
|------------|------------|--------------|---------------|

| $M_{32}$ | Failures | $M_8$ | Failures | $M_2$ | Failures |
|----------|----------|-------|----------|-------|----------|
| $a$, $b$, $c$ | S | R | + | W | $a$, $b$, $c$ | S | R | + | W | $a$, $b$, $c$ | S | R | + | W |
| 27, 13, 46 | 25 | 31 | 56 | 275 | 1, 13, 7 | 28 | 19 | 47 | 113 | 3, 26, 35 | 29 | 24 | 53 | 89 |
| 31, 33, 37 | 28 | 32 | 60 | 79 | 3, 26, 35 | 29 | 22 | 51 | 89 | 27, 13, 46 | 41 | 20 | 61 | 275 |
| 22, 7, 48 | 37 | 24 | 61 | 223 | 40, 11, 31 | 24 | 33 | 57 | 77 | 25, 8, 15 | 38 | 24 | 62 | 281 |
| 7, 16, 55 | 37 | 26 | 63 | 65 | 15, 16, 19 | 30 | 32 | 62 | 255 | 31, 10, 27 | 36 | 31 | 67 | 233 |
| 9, 14, 41 | 23 | 40 | 63 | 167 | 22, 7, 48 | 29 | 33 | 62 | 223 | 9, 5, 60 | 24 | 43 | 67 | 227 |
| 41, 7, 29 | 28 | 37 | 65 | 265 | 9, 14, 41 | 32 | 30 | 62 | 167 | 1, 13, 7 | 28 | 42 | 70 | 113 |
| 1, 13, 7 | 34 | 34 | 68 | 113 | 41, 7, 29 | 25 | 38 | 63 | 265 | 15, 16, 19 | 36 | 34 | 70 | 255 |
| 10, 11, 61 | 32 | 36 | 68 | 155 | 31, 11, 30 | 33 | 32 | 65 | 363 | 2, 11, 61 | 40 | 30 | 70 | 81 |
| 9, 5, 60 | 44 | 28 | 72 | 227 | 2, 11, 61 | 25 | 41 | 66 | 81 | 41, 7, 29 | 36 | 34 | 70 | 265 |
| 16, 23, 30 | 37 | 36 | 73 | 59 | 10, 11, 61 | 42 | 25 | 67 | 155 | 9, 14, 41 | 33 | 37 | 70 | 167 |
| 3, 26, 35 | 45 | 29 | 74 | 89 | 7, 16, 55 | 32 | 35 | 67 | 65 | 22, 7, 48 | 37 | 35 | 72 | 223 |
| 25, 8, 15 | 42 | 34 | 76 | 281 | 16, 23, 30 | 35 | 34 | 69 | 59 | 31, 11, 30 | 45 | 27 | 72 | 363 |
| 31, 11, 30 | 35 | 43 | 78 | 363 | 25, 8, 15 | 25 | 45 | 70 | 281 | 7, 16, 55 | 36 | 39 | 75 | 65 |
| 40, 11, 31 | 38 | 40 | 78 | 77 | 27, 13, 46 | 39 | 32 | 71 | 275 | 31, 33, 37 | 37 | 39 | 76 | 79 |
| 31, 10, 27 | 34 | 45 | 79 | 233 | 31, 10, 27 | 40 | 32 | 72 | 233 | 10, 11, 61 | 41 | 37 | 78 | 155 |
| 2, 11, 61 | 43 | 40 | 83 | 81 | 9, 5, 60 | 40 | 36 | 76 | 227 | 16, 23, 30 | 44 | 37 | 81 | 59 |
| 15, 16, 19 | 45 | 39 | 84 | 255 | 31, 33, 37 | 39 | 39 | 78 | 79 | 40, 11, 31 | 38 | 48 | 86 | 77 |
| 10, 9, 63 | 39 | 51 | 90 | 69 | 10, 9, 63 | 31 | 49 | 80 | 69 | 10, 9, 63 | 48 | 48 | 96 | 69 |
| 51, 1, 46 | 31 | 890 | 921 | 111 | 51, 1, 46 | 60 | 896 | 956 | 111 | 51, 1, 46 | 31 | 799 | 830 | 111 |
| 47, 1, 41 | 50 | 902 | 952 | 99 | 47, 1, 41 | 67 | 907 | 974 | 99 | 47, 1, 41 | 47 | 799 | 846 | 99 |
Table VII. Results of Dieharder on xorshift1024* Generators. No Test Is Failed Systematically

| $M_2$ | Failures | $M_3$ | Failures | $M_5$ | Failures |
|-------|-----------|-------|-----------|-------|-----------|
| $a, b, c$ | S  | R  | + | W | $a, b, c$ | S  | R  | + | W | $a, b, c$ | S  | R  | + | W |
| 31, 33, 37 | 57 | 67 | 124 | 79 | 25, 8, 15 | 67 | 56 | 123 | 281 | 22, 7, 48 | 56 | 76 | 132 | 223 |
| 31, 11, 30 | 65 | 61 | 126 | 363 | 16, 23, 30 | 77 | 54 | 131 | 59 | 15, 16, 19 | 66 | 67 | 133 | 255 |
| 16, 23, 30 | 74 | 56 | 130 | 59 | 7, 16, 55 | 66 | 66 | 132 | 65 | 10, 9, 63 | 70 | 71 | 141 | 69 |
| 41, 7, 29 | 71 | 61 | 132 | 265 | 3, 26, 35 | 60 | 75 | 135 | 89 | 51, 1, 46 | 65 | 78 | 143 | 111 |
| 9, 14, 41 | 74 | 64 | 138 | 167 | 10, 11, 61 | 63 | 74 | 137 | 155 | 1, 13, 7 | 80 | 64 | 144 | 113 |
| 10, 9, 63 | 74 | 66 | 140 | 69 | 31, 10, 27 | 74 | 69 | 143 | 233 | 40, 11, 31 | 80 | 67 | 147 | 77 |
| 22, 7, 48 | 66 | 75 | 141 | 223 | 31, 33, 37 | 86 | 58 | 144 | 79 | 2, 11, 61 | 85 | 65 | 150 | 81 |
| 51, 1, 46 | 78 | 63 | 141 | 111 | 47, 1, 41 | 82 | 62 | 144 | 99 | 31, 11, 30 | 75 | 75 | 150 | 363 |
| 27, 13, 46 | 63 | 79 | 142 | 275 | 27, 13, 46 | 78 | 69 | 147 | 275 | 25, 8, 15 | 74 | 77 | 151 | 281 |
| 25, 8, 15 | 80 | 64 | 144 | 281 | 31, 11, 30 | 85 | 62 | 147 | 363 | 10, 11, 61 | 79 | 76 | 155 | 155 |
| 3, 26, 35 | 81 | 66 | 147 | 89 | 10, 9, 63 | 65 | 86 | 151 | 69 | 47, 1, 41 | 70 | 86 | 156 | 99 |
| 2, 11, 61 | 79 | 71 | 150 | 81 | 41, 7, 29 | 84 | 68 | 152 | 265 | 9, 5, 60 | 70 | 86 | 156 | 227 |
| 40, 11, 31 | 74 | 76 | 150 | 77 | 2, 11, 61 | 88 | 65 | 153 | 81 | 16, 23, 30 | 81 | 76 | 157 | 59 |
| 31, 10, 27 | 82 | 71 | 153 | 233 | 9, 14, 41 | 77 | 80 | 157 | 167 | 27, 13, 46 | 78 | 80 | 158 | 275 |
| 47, 1, 41 | 74 | 79 | 153 | 99 | 40, 11, 31 | 82 | 78 | 160 | 77 | 7, 16, 55 | 92 | 70 | 162 | 65 |
| 9, 5, 60 | 81 | 75 | 156 | 227 | 15, 16, 19 | 85 | 76 | 161 | 255 | 9, 14, 41 | 87 | 80 | 167 | 167 |
| 10, 11, 61 | 75 | 84 | 159 | 155 | 51, 1, 46 | 92 | 74 | 166 | 111 | 41, 7, 29 | 87 | 81 | 168 | 265 |
| 15, 16, 19 | 72 | 88 | 160 | 255 | 22, 7, 48 | 90 | 82 | 172 | 223 | 31, 10, 27 | 82 | 87 | 169 | 233 |
| 7, 16, 55 | 94 | 68 | 162 | 65 | 1, 13, 7 | 79 | 95 | 174 | 113 | 3, 26, 35 | 92 | 79 | 171 | 89 |
| 1, 13, 7 | 87 | 76 | 163 | 113 | 9, 5, 60 | 97 | 89 | 186 | 227 | 31, 33, 37 | 98 | 88 | 186 | 79 |
Table VIII. Results of BigCrush on xorshift4096c Generators

| Algorithm  | S  | R  | + | W  | Algorithm  | S  | R  | + | W  | Algorithm  | S  | R  | + | W  |
|------------|----|----|---|----|------------|----|----|---|----|------------|----|----|---|----|
| 14, 41, 15 | 33 | 27 | 60| 241| 11, 9, 25  | 30 | 33 | 63| 567| 5, 27, 21  | 36 | 35 | 71| 187|
| 5, 22, 27  | 34 | 30 | 64| 45 | 5, 22, 27  | 34 | 35 | 69| 45 | 5, 27, 21  | 36 | 35 | 71| 187|
| 30, 29, 39 | 33 | 32 | 65| 177| 25, 3, 49  | 35 | 37 | 72| 441| 25, 3, 49  | 33 | 34 | 67| 441|
| 25, 3, 49  | 30 | 38 | 68| 441| 7, 12, 59  | 34 | 39 | 73| 103| 19, 34, 19| 39 | 36 | 75| 291|
| 7, 12, 59  | 43 | 25 | 68| 103| 11, 9, 25  | 40 | 34 | 74| 567| 23, 26, 29| 40 | 35 | 75| 49 |
| 19, 34, 19 | 34 | 36 | 70| 291| 12, 11, 61| 36 | 33 | 74| 241| 30, 29, 39| 38 | 37 | 75| 177|
| 12, 11, 61 | 32 | 39 | 71| 195| 19, 34, 19| 39 | 35 | 74| 291| 12, 11, 61| 40 | 37 | 77| 195|
| 5, 27, 21  | 34 | 41 | 75| 187| 14, 41, 15| 43 | 34 | 77| 241| 14, 41, 15| 36 | 42 | 78| 241|
| 23, 26, 29 | 36 | 42 | 78| 49 | 30, 29, 39| 42 | 37 | 79| 177| 7, 12, 59  | 38 | 44 | 82| 103|
| 11, 9, 25  | 35 | 44 | 79| 567| 23, 26, 29| 38 | 43 | 81| 49 | 5, 22, 27  | 38 | 50 | 88| 45 |
An Experimental Exploration of Marsaglia’s xorshift Generators, Scrambled

Table IX. Results of Dieharder on xorshift4096* Generators

| Algorithm   | $M_{32}$ Failures | Algorithm   | $M_{8}$ Failures | Algorithm   | $M_{2}$ Failures |
|-------------|-------------------|-------------|-------------------|-------------|-----------------|
|             | $S$   | $R$   | $+$  | $W$  | $S$   | $R$   | $+$  | $W$  | $S$   | $R$   | $+$  | $W$  |
| 25, 3, 49   | 70    | 70    | 140  | 441 | 25, 3, 49 | 67    | 70    | 137  | 441 | 19, 34, 19 | 75    | 64    | 139  | 291 |
| 12, 11, 61  | 58    | 83    | 141  | 195 | 14, 41, 15 | 72    | 69    | 141  | 241 | 5, 22, 27 | 67    | 77    | 144  | 45  |
| 30, 29, 39  | 67    | 77    | 144  | 177 | 30, 29, 39 | 70    | 75    | 145  | 177 | 25, 3, 49 | 77    | 71    | 148  | 441 |
| 5, 22, 27   | 62    | 84    | 146  | 45  | 11, 9, 25  | 73    | 77    | 150  | 567 | 11, 9, 25  | 81    | 76    | 157  | 567 |
| 11, 9, 25   | 73    | 75    | 148  | 567 | 12, 11, 61 | 75    | 80    | 155  | 195 | 14, 41, 15 | 79    | 78    | 157  | 241 |
| 19, 34, 19  | 85    | 66    | 151  | 291 | 19, 34, 19 | 89    | 67    | 156  | 291 | 14, 41, 15 | 74    | 84    | 158  | 49  |
| 14, 41, 15  | 83    | 74    | 157  | 241 | 5, 22, 27  | 93    | 65    | 158  | 45  | 23, 26, 29 | 74    | 84    | 158  | 49  |
| 7, 12, 59   | 73    | 85    | 158  | 103 | 23, 26, 29 | 72    | 87    | 159  | 49  | 12, 11, 61 | 74    | 85    | 159  | 195 |
| 23, 26, 29  | 73    | 88    | 161  | 49  | 5, 27, 21  | 75    | 84    | 159  | 187 | 7, 12, 59  | 84    | 79    | 163  | 103 |
| 5, 27, 21   | 98    | 67    | 165  | 187 | 7, 12, 59  | 90    | 77    | 167  | 103 | 30, 29, 39 | 78    | 89    | 167  | 177 |
Table X. A Comparison of Generators Using BigCrush

| Algorithm               | Failures | Systematic               |
|-------------------------|----------|--------------------------|
| Algorithm               | S        | R           | +   | W/n        |
| A₁(12, 25, 27) · M₃₂  | 230      | 133         | 363 | 0.48       |
| A₂(4, 35, 21) · M₅₂  | 240      | 223         | 463 | 0.38       |
| xorshift1024*          | 33       | 32          | 65  | 0.35       |
| xorshift1024*          | 34       | 34          | 61  | —          |
| xorshift4096*          | 42       | 40          | 82  | 0.23       |
| xorshift4096*          | 258      | 258         | 516 | 0.34       |
| xorgens4096            | 441      | 441         | 882 | 0.40       |
| MT19937                | 235      | 233         | 468 | 0.43       |
| WELL1024a              | 441      | 441         | 882 | 0.40       |
| WELL19937a             | 235      | 233         | 468 | 0.43       |

equidistributed and xorshift4096 generators are 64-dimensionally equidistributed. Since multiplication by a constant just permutes the space of tuples, the same is true of the associated xorshift* generators.

8. COMPARISON

How do our best xorshift* generators score with respect to more complex generators in the literature? We decided to perform a comparison with the popular Mersenne Twister MT19937 [Matsumoto and Nishimura 1998],¹² with WELL1024a/WELL19937a, two generators introduced by Panneton et al. [2006] as an improvement over the Mersenne Twister; and with xorgens4096, a very recent 4096-bit generator introduced by Brent [2007] we mentioned in Section 7. All these generators are noncryptographic and aim at fast, high-quality generation. As usual, 100 tests are performed at 100 equispaced points of the state space.

We choose generators from the xorshift* family that perform well on both BigCrush and Dieharder and have a good weight score and enough large parameters (which provide faster state change spreading): more precisely, the xorshift64* generator \( A₁(12, 25, 27) \cdot M₃₂ \) (Figure 10); xorshift1024* with parameters 31, 11, 30 and multiplier \( M₈ \) (Figure 11); and xorshift4096* with parameters 25, 3, 49 and multiplier \( M₂ \).

8.1. Quality

Table X compares the BigCrush scores of the generators we discussed. The results are quite interesting. A simple 64-bit xorshift* generator has fewer linear artifacts than MT19937, WELL1024a, or WELL19937a and, thus, a significantly better score. High-dimension xorgens4096 and xorshift* generators perform significantly better, in spite of being extremely simple, and have no systematic failure. The 64-bit xorshift* generator suggested by “Numerical Recipes” fails systematically the BirthdaySpacings test, contrarily the one we have selected.¹³ We do not report the results of Dieharder, as at this level of quality the suite is unable to make any significant distinction among the generators.

8.2. Escaping Zeroland

We show in Figure 9 the speed at which a few of the generators of Table X “escape from zeroland” [Panneton et al. 2006]: purely linearly recurrent generators with a very large state space need a very long time to get from an initial state with a small number of ones to a state in which the ones are approximately half. The figure shows a measure

¹² More precisely, with its 64-bit version.
¹³ Note that we report the number of failed tests on our 100 seeds. L’Ecuyer and Simard [2007] report the number of types of failed tests (e.g., failing two distinct RandomWalk tests counts as one) on a single run, so some care must be taken when comparing the results we report and those reported by them.
of escape time given by the ratio of ones in a window of four consecutive 64-bit values sliding over the first 100,000 generated values, averaged over all possible seeds with exactly one bit set (see Panneton et al. [2006] for a detailed description).

As is known, MT19937 needs hundreds of thousands of iterations to start behaving correctly, xorshift4096* and xorgens4096 need a few thousand (but xorgens4096 oscillates always around 1/2) and WELL19937a and xorshift1024* a few hundred, whereas WELL1024a just a few dozens, and xorshift64* is almost unaffected.

Table XI condenses Figure 9 into the mean and standard deviation of the displayed values. Clearly, the multiplication step helps in reducing the correlation between the number of ones in the state and the number of ones in the output values. Also, the slowness in recovering from states with too many zeroes is directly correlated to the size of the state space—a very good argument against linear generators with too-large state spaces.

8.3. Speed

Finally, we benchmark the generators of Table X. Our tests were run on an Intel® Core™ i7-4770 CPU @3.40GHz (Haswell), and the results are shown in Table XII (variance is undetectable, as we generate $10^{10}$ values in each test). We also report as a strong baseline results about SFMT19937, the SIMD-Oriented Fast Mersenne Twister [Saito and Matsumoto 2008], a 128-bit version of the Mersenne Twister based on the SSE2
Table XII. Time to Emit a 64-Bit Integer on an Intel®
Core™ i7-4770 CPU @3.40GHz (Haswell)

| Algorithm                | Speed (ns/64 bits) |
|--------------------------|--------------------|
| xorshift64*              | 1.58               |
| xorshift1024*            | 1.36               |
| xorshift4096*            | 1.36               |
| xorgens4096              | 2.06               |
| MT19937 (64-bit version) | 3.10               |
| SFMT19937                | 1.60               |
| WELL1024a                | 10.56              |
| WELL19937a               | 8.23               |

`#include <stdint.h>`

```c
uint64_t x;

uint64_t next(void) {
    x ^= x >> 12; // a
    x ^= x << 25; // b
    x ^= x >> 27; // c
    return x * UINT64_C(2685821657736338717);
}
```

Fig. 10. The suggested xorshift64* generator in C99 code. The variable x should be initialized to a nonzero seed before calling `next()`.

`#include <stdint.h>`

```c
uint64_t s[16];
int p;

uint64_t next(void) {
    const uint64_t s0 = s[p];
    uint64_t s1 = s[p] = (p + 1) & 15;
    s1 ^= s1 << 31; // a
    s[p] = s1 ^ s0 ^ (s1 >> 11) ^ (s0 >> 30); // b, c
    return s[p] * UINT64_C(1181783497276652981);
}
```

Fig. 11. The suggested xorshift1024* generator in C99 code. The array s should be initialized to a nonzero seed before calling `next()`.

extended instruction set of Intel processors (and thus not usable, in principle, on other processors). We used suitable options to keep the compiler from unrolling loops or extracting loop invariants.

The highest speed is achieved by the high-dimensional xorshift* generators. SFMT19937 is a major improvement in speed over MT19937, albeit slightly slower than a high-dimensional xorshift* generator; it fails systematically, moreover, the same tests of MT19937.

A xorshift64* generator is actually slower than its high-dimensional counterparts. This is not surprising, as the three shift/xors in a xorshift64* generator form a dependency chain and must be executed in sequence, whereas two of the shifts of a higher-dimensional generator are independent and can be internally parallelized by the CPU. WELL1024a and WELL19937a are heavily penalized by their 32-bit structure.
9. CONCLUSIONS

After our careful experimental analysis, we reach the following conclusions:

A **xorshift1024** generator is an excellent choice for a general-purpose, high-speed generator. The statistical quality of the generator is very high (it has, actually, the best results in BigCrush), and its period is so large that the probability of overlapping sequences is practically zero, even in the largest parallel simulation. Nonetheless, the state space is reasonably small, so seeding it with high-quality bits is not too expensive, and recovery from states with a large number of zeroes happens quickly. The generator is also blazingly fast (it is actually the fastest generator we tested). The reasonable state space makes it also easier, in case a large number of generators is used at the same time, to fit their state into the cache. In any case, with respect to other generators, the state is accessed in a more localized way, as read and write operations happen at two consecutive locations and thus will generate at most one cache miss.

In case memory is an issue, or array access is expensive, a very good general-purpose generator is a **xorshift64** generator. While the generator $A_1(12, 25, 27) \cdot M_{32}$ fails systematically the MatrixRank test, it has less linear artifacts than MT19937, WELL1024a, or WELL19937a, which fail systematically even more tests. It is a very good choice if memory footprint is an issue and a very large number of generators is necessary. It can also be used, for instance, to generate the initial state of another generator with a larger state space using a 64-bit seed. We remark that a **xorshift64** generator can also actually be faster than a **xorshift1024** generator if the underlying language incurs significant costs when accessing an array: For instance, in Java a **xorshift64** generator emits a value in 1.62 ns, whereas a **xorshift1024** generator needs 2.06 ns.

Linear generators with an excessively long period have a number of problems that are not compensated by higher statistical quality. WELL19937a is almost four slower than **xorshift1024** and has a worse performance in BigCrush; moreover, recovery from states with many zeroes, albeit enormously improved with respect to MT19937, is still very slow, and seeding properly the generator requires almost 20,000 random bits. In the end, it is in general difficult to motivate state spaces larger than $2^{1024}$. Similar considerations are made by Press et al. [2007] and L'Ecuyer and Panneton [2005].

Surprisingly simple and fast generators can produce sequences that pass strong statistical tests. The code in Figure 11 is extremely shorter and simpler than that of MT19937, WELL1024a, or WELL19937a. Yet, it performs significantly better on BigCrush. It is a tribute to Marsaglia's cleverness that just eight logical operations, one addition, and one multiplication by a constant can produce sequences of such high quality. xorgens generators are similar with this respect but use several more operations due to the additional shift and to combination with a Weyl generator to hide linear artifacts [Brent 2007].

The $t$ for which the multiplier has a good figure of merit has no detectable effect on the quality of the generator. In our tests, we could not find any significant difference between the behavior of generators based on $M_{32}$, $M_8$, or $M_2$. It could be interesting to experiment with multipliers having very bad figures of merit or, more generally, with multipliers chosen using different heuristics.

Equidistribution is more useful as a design feature than as an evaluation feature. While designing generators around equidistribution might be a good idea, as it leads in general to good generators, evaluation by equidistribution is a more delicate
matter because of high-bits bias, instability issues, and failure to detect the generators having the best scores in statistical suites.

**TestU01 has significantly more resolution than Dieharder as a test suite.** In particular in the high-dimension case, TestU01 is able to provide useful information, whereas Dieharder scores flatten down. However, TestU01 (as any other test suite with high-bits bias) must always be applied to the reverse generator, too.

**REFERENCES**

Richard P. Brent. 2004. Note on Marsaglia’s Xorshift random number generators. *J. Stat. Softw.* 11, 5 (2004), 1–5.

Richard P. Brent. 2007. Some long-period random number generators using shifts and xors. In *Proceedings of the 13th Biennial Computational Techniques and Applications Conference, CTAC-2006 (ANZIAM J.),* Wayne Read and A. J. Roberts (Eds.), Vol. 48. C188–C202.

Richard P. Brent. 2010. The myth of equidistribution for high-dimensional simulation. *CoRR* abs/1005.1320 (2010).

Robert G. Brown. 2013. Dieharder: A Random Number Test Suite (Version 3.31). (2013). Retrieved January 8, 2014 from http://www.phy.duke.edu/~rgb/General/dieharder.php.

Aaldert Compagner. 1991. The hierarchy of correlations in random binary sequences. *J. Stat. Phys.* 63, 5–6 (1991), 883–896.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics* (2nd ed.). Addison–Wesley.

Shin Harase. 2011. An efficient lattice reduction method for $\mathbb{F}_2$-linear pseudorandom number generators using Mulders and Storjohann algorithm. *J. Comput. Appl. Math.* 236, 2 (2011), 141–149.

Maurice G. Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

Maurice G. Kendall. 1945. The treatment of ties in ranking problems. *Biometrika* 33, 3 (1945), 239–251.

Pierre L’Ecuyer. 1999. Tables of linear congruential generators of different sizes and good lattice structure. *Math. Comput* 68, 225 (1999), 249–260.

Pierre L’Ecuyer and Françoise Panneton. 2003. Combined generators with components from different families. *Math. Comput. Simul.* 62, 3 (2003), 395–404.

Pierre L’Ecuyer and François Panneton. 2005. Fast random number generators based on linear recurrences modulo 2: Overview and comparison. In *Proceedings of the 37th Winter Simulation Conference.* Winter Simulation Conference, 110–119.

Pierre L’Ecuyer and Richard Simard. 2007. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.* 33, Article 22 (August 2007). Issue 4.

Robert H. Lewis. 2013. Fermat: A Computer Algebra System for Polynomial and Matrix Computation (Version 5.1). (2013). Retrieved January 8, 2014 from http://home.bway.net/lewis/.

Rudolf Lidl and Harald Niederreiter. 1994. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, Cambridge.

George Marsaglia. 2003. Xorshift RNGs. *J. Stat. Softw.* 8, 14 (2003), 1–6.

Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (1998), 3–30.

Harald Niederreiter. 1992. *Random Number Generation and Quasi-Monte Carlo Methods.* CBMS-NSF regional conference series in Appl. Math., Vol. 63. SIAM.

François Panneton. 2004. *Construction d’ensembles de points basé sur une récurrence linéaire dans un corps fini de caractéristique 2 pour la simulation Monte Carlo et l’intégration quasi-Monte Carlo.* Ph.D. Dissertation. Université de Montréal.

François Panneton and Pierre L’Ecuyer. 2005. On the Xorshift random number generators. *ACM Trans. Model. Comput. Simul.* 15, 4 (2005), 346–361.

François Panneton, Pierre L’Ecuyer, and Makoto Matsumoto. 2006. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw.* 32, 1 (2006), 1–16.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge.

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.* National Institute for
An Experimental Exploration of Marsaglia’s xorshift Generators, Scrambled

Standards and Technology, pub-NIST:adr. NIST Special Publication 800-22, with revisions dated May 15, 2001.

Mutsuo Saito. 2013. MTToolBox (Version 0.2). (2013). Retrieved January 8, 2014 from http://msaito.github.io/MTToolBox/en/.

Mutsuo Saito and Makoto Matsumoto. 2008. SIMD-oriented fast mersenne twister: A 128-bit pseudorandom number generator. In Monte Carlo and Quasi-Monte Carlo Methods 2006, Alexander Keller, Stefan Heinrich, and Harald Niederreiter (Eds.). Springer, 607–622.

Mutsuo Saito and Makoto Matsumoto. 2014. XSadd (Version 1.1). (25 March 2014). http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/XSADD/.

Received January 2014; revised August 2015; accepted November 2015