Scrambled Linear Pseudorandom Number Generators[*]

DAVID BLACKMAN, Independent researcher, Australia
SEBASTIANO VIGNA, Università degli Studi di Milano, Italy

Linear pseudorandom number generators are very popular due to their high speed, to the ease with which generators with a sizable state space can be created, and to their provable theoretical properties. However, they suffer from linear artifacts which show as failures in linearity-related statistical tests such as the binary-rank and the linear-complexity test. In this paper, we give three new contributions. First, we introduce two new linear transformations that have been handcrafted to have good statistical properties and at the same time to be programmable very efficiently on superscalar processors, or even directly in hardware. Then, we describe a new test for Hamming-weight dependencies that is able to discover subtle, previously unknown biases in existing generators (in particular, in linear ones). Finally, we describe a number of *scramblers*, that is, nonlinear functions applied to the state array that reduce or delete the linear artifacts, and propose combinations of linear transformations and scramblers that give extremely fast pseudorandom generators of high quality. A novelty in our approach is that we use ideas from the theory of filtered linear-feedback shift register to prove some properties of our scramblers, rather than relying purely on heuristics. In the end, we provide simple, extremely fast generators that use a few hundred bits of memory, have provable properties and pass very strong statistical tests.

CCS Concepts: • **Mathematics of computing → Random number generation**;

Additional Key Words and Phrases: Pseudorandom number generators

1 INTRODUCTION

In the last twenty years, in particular since the introduction of the Mersennne Twister [28], linear pseudorandom generators have been very popular: indeed, they are often the stock generator provided by several programming languages. Linear generators have several advantages: they are fast, it is easy to create full-period generators with large state spaces, and thanks to their connection with *linear-feedback shift registers* (LFSRs) [15] many of their properties, such as full period, are mathematically provable. Moreover, if suitably designed, they are rather easy to implement using simple xor and shift operations. In particular, Marsaglia [25] introduced the family of `xorshift` generators, which have a very simple structure.

The linear structure of such generators, however, is detectable by some randomness tests: in particular, the binary-rank test [26] and the linear-complexity test [4, 8] are failed by all linear generators. These tests were indeed devised to "catch" linear generators, and they are not considered problematic by the community working on such generators, as the advantage of being able to prove precise mathematical properties is perceived as outweighing the failure of such tests.

Nonetheless, one might find it desirable to mitigate or eliminate such *linear artifacts* by *scrambling* a linear generator, that is, applying a nonlinear function to its state array to produce the actual output. In this direction, two simple approaches are multiplication by a constant or adding two components of the state array. However, while empirical tests usually do not show linear artifacts anymore, it is easy to prove that the two lower bits are unchanged or just slightly modified by such

---

Authors' addresses: David Blackman, Independent researcher, Australia; Sebastiano Vigna, vigna@acm.org, Università degli Studi di Milano, Italy.

operations. Thus, those bits in isolation (or combined with a sufficiently small number of good bits) fail linearity tests.

In this paper, we try to find a middle ground by proposing very fast scrambled generators with provable properties. By combining in different ways an underlying linear engine and a scrambler we can provide different tradeoffs in terms of speed, space usage and statistical quality. For example, `xoshiro256**` is a 64-bit generator with 256 bits of state that emits a value in 0.90 ns; it passes all statistical tests we are aware of, and it is 4-dimensionally equidistributed, which is the best possible: it is our all-purpose generator.

However, if the user is interested in the generation of floating-point numbers only, we provide a `xoshiro256+` generator that generates a value in 0.79 ns (the value must then be converted to float); it is just 3-dimensionally equidistributed, and its lowest bits have low linear complexity, but since one needs just the upper 53 bits, the resulting floating-point values have no linear bias. If space is an issue, a `xoroshiro128**` or a `xoroshiro128+` generator provide similar timings and properties in less space. We also describe higher-dimensional generators, albeit mainly for theoretical reasons, and 32-bit generators with similar properties that are useful for embedded devices and GPUs. Our approach can even provide fast, reasonable 16-bit generators.

We also introduce and analyze in detail a new test based on the dependency of Hamming weights, that is, the number of ones in each output word. The test examines the stream produced by a generator in search for bias, and can be run for an arbitrarily long stretch: we usually stop testing after 1 PB (10^{15}) bytes, or when we detect a p-value below 10^{-20}. While all generators above pass the BigCrush test suite from TestU01 [21], we can show that `xoroshiro128+` is slightly weaker than the alternatives, as it fails our new test after about 8 TB of data. Our new test is strong enough to find bias in the Tiny Mersenne Twister [35] after almost 600 TB of data: it is indeed the first test in the literature to find bias in a generator of the Mersenne Twister [28] family that is not related to binary rank or linear complexity. It can also find easily bias in low-dimensional Mersenne Twisters, and in generators of the `xorshift` family scrambled with a sum, as in the case of `xorshift+` [43].

Finally, we develop some theory related to our linear engines and scramblers using results from the theory of noncommutative determinants and from the theory of filtered LFSRs.

The C code for the generators described in this paper is available from the authors and it is public domain.[1] The test code is distributed under the GNU General Public License version 2 or later.

2 ORGANIZATION OF THE PAPER

In this paper we consider words of size w, w-bit operations and generators with kw bits of state, $k \geq 2$. We aim mainly at 64-bit generators (i.e., $w = 64$), but we also provide 32-bit combinations. Some theoretical data are computed also for the 16-bit case.

The paper is organized in such a way to make immediately available code and basic information for our new generators as quickly as possible: all theoretical considerations and analysis are postponed to the second part of the paper, albeit sometimes this approach forces us to point at subsequent material.

Our generators consist of a *linear engine*[2] and a *scrambler*. The linear engine is a linear transformation on $\mathbf{Z}/2\mathbf{Z}$, represented by a matrix, and it is used to advance the internal state of the generator. The scrambler is an arbitrary function on the internal state which computes the actual output of the generator. We will usually apply the scrambler to the *current state*, to make it easy for the CPU to parallelize internally the operations of the linear engine and of the scrambler. Such a combination

---

[1]http://prng.di.unimi.it/

[2]We use consistently "engine" throughout the paper instead of "generator" when discussing combinations with scramblers to avoid confusion between the overall generator and underlying linear generator, but the two terms are otherwise equivalent.

is quite natural: for example, it was advocated by Marsaglia for xorshift generators [25] and it has been used in the design of XSAdd [37] and of the Tiny Mersenne Twister [38]. An alternative approach is that of combining a linear generator and a nonlinear generator [19].

In Section 3 we introduce our linear engines. We describe the matrices representing the transformation, and the associated code. In Section 4 we describe the scramblers we will be using, and their elementary properties. Finally, in Section 5 we describe generators given by a number of combinations between scramblers and linear engines, their speed and their results in statistical tests. Section 5.3 contains a guide to the choice of an appropriate generator.

Next we start to discuss the theoretical material. Section 6 describes in detail the new test we have used in addition to BigCrush, and some interesting implementation issues. In Section 7 and 8 we discuss mathematical properties of our linear engines: in particular, we introduce the idea of *word polynomials*, polynomials on $w \times w$ matrices associated with a linear engine. The word polynomial makes it easy to compute the *characteristic polynomial*, which is the basic tool to establish full period, and can be used as a heuristic to evaluate the resistance of the generator to our new test. We then provide *equidistribution* results.

In the last part of the paper, starting with Section 10, we apply ideas and techniques from the theory of filtered LFSRs to the problem of analyzing the behavior of our scramblers. We provide some exact results, and discuss a few heuristics based on extensive symbolic computation. Our discussion gives a somewhat more rigorous foundation to the choices made in Section 5, and opens several interesting problems.

3 LINEAR ENGINES

In this section we introduce our two linear engines xoshiro (xor/shift/rotate) and xoroshiro (xor/rotate/shift/rotate). They are built using not only shifts and xors, as it is customary in several traditional linear generators [25, 28, 34, 42, 43], but also rotations. All modern C/C++ compilers can compile a simulated rotation into a single CPU instruction, and Java provides intrinsified rotation static methods to the same purpose. As a result, rotations are no more expensive than a shift, and they provide better state diffusion, as no bit of the operand is discarded.[3]

We denote with S the $w \times w$ matrix on $\mathbf{Z}/2\mathbf{Z}$ that effects a left shift of one position on a binary row vector (i.e., S is all zeroes except for ones on the principal subdiagonal) and with R the $w \times w$ matrix on $\mathbf{Z}/2\mathbf{Z}$ that effects a left rotation of one position (i.e., R is all zeroes except for ones on the principal subdiagonal and a one in the upper right corner). We will use $\rho_r(-)$ to denote left rotation by r of a w-bit vector in formulae.

3.1 xoroshiro

The xoroshiro linear transformation is closer to traditional linear generators in the sense that it updates cyclically two words of a larger state array. The update rule is designed so that it is formed by two almost independent computation paths, leading to good parallelizability inside superscalar CPUs.

The base xoroshiro linear transformation is obtained combining a rotation, a shift and again a rotation (hence the name), and it is defined by the following $2w \times 2w$ matrix:

$$\mathscr{X}_{2w} = \begin{pmatrix} R^a + S^b + I & R^c \\ S^b + I & R^c \end{pmatrix}.$$

---

[3]Note that at least one shift is necessary, as rotations and xors map the set of words x satisying $xR^s = x$ for a fixed s into itself, so there are no full-period generators using only rotations.

```
static inline uint64_t rotl(const uint64_t x, int k) {
    return (x << k) | (x >> (64 - k));
}
```

Fig. 1. A sample C implementation of a left-rotation operator.

```
const uint64_t s0 = s[0];
uint64_t s1 = s[1];

const uint64_t result_plus = s0 + s1;
const uint64_t result_star = s0 * S;
const uint64_t result_starstar = rotl(s0 * S, R) * T;

s1 ^= s0;
s[0] = rotl(s0, A) ^ s1 ^ (s1 << B);
s[1] = rotl(s1, C);
```

Fig. 2. The C code for a xoroshiro128+/xoroshiro128*/xoroshiro128** generator. The array s contains two 64-bit unsigned integers, not all zeros.

The general $kw \times kw$ form is given instead by

$$
\mathcal{X}_{kw} = \begin{pmatrix}
0 & 0 & \cdots & 0 & R^a + S^b + I & R^c \\
I & 0 & \cdots & 0 & 0 & 0 \\
0 & I & \cdots & 0 & 0 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & \cdots & I & 0 & 0 \\
0 & 0 & \cdots & 0 & S^b + I & R^c
\end{pmatrix}
\tag{1}
$$

Note that the general form applies the basic form to the first and last words of state, and uses the result to replace the last and next-to-last words. The remaining words are shifted by one position.

The structure of the transformation may appear repetitive, but it has been so designed because this implies a very simple and efficient computation path. Indeed, in Figure 2 we show the C code implementing the xoroshiro transformation for $w = 64$ with 128 bits of state.[4] The constants prefixed with "result" are outputs computed using different scramblers, which will be discussed in Section 4.

The reader should note that after the first xor, which represents the only data dependency between the two words of the state array, the computation of the two new words can continue in parallel. The general case is better implemented using a form of cyclic update, as shown in Figure 3.

3.2 xoshiro

The xoshiro linear transformation uses only a shift and a rotation. It radically departs from traditional linear generators in that it updates *all* of the state at each iteration. As such, it is sensible

---

[4]In the code, and in the rest of the paper, we use rotl() to denote the left-rotation function in Figure 1.

```
    const int q = p;
    const uint64_t s0 = s[p = (p + 1) & 15];
    uint64_t s15 = s[q];

    const uint64_t result_plus = s0 + s15;
    const uint64_t result_star = s0 * S;
    const uint64_t result_starstar = rotl(s0 * S, R) * T;

    s15 ^= s0;
    s[q] = rotl(s0, A) ^ s15 ^ (s15 << B);
    s[p] = rotl(s15, C);
```

Fig. 3. The C code for a xoroshiro1024+/xoroshiro1024*/xoroshiro1024** generator. The state array s contains sixteen 64-bit unsigned integers, not all zeros, and the integer variable p holds a number in the interval $[0 \dots 16)$.

only for moderate state sizes. We will discuss the $4w \times 4w$ transformation

$$
\mathscr{S}_{4w} = \begin{pmatrix} I & I & I & 0 \\ I & I & S^a & R^b \\ 0 & I & I & 0 \\ I & 0 & 0 & R^b \end{pmatrix}
$$

and the $8w \times 8w$ transformation

$$
\mathscr{S}_{8w} = \begin{pmatrix} I & I & I & 0 & 0 & 0 & 0 & 0 \\ 0 & I & 0 & 0 & I & I & S^a & 0 \\ 0 & I & I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I & 0 & 0 & I & R^b \\ 0 & 0 & 0 & I & I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I & I & 0 & 0 \\ I & 0 & 0 & 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & I & R^b \end{pmatrix}.
$$

The layout of the two matrices above might seem arbitrary, but it is just derived from the implementation. In Figure 4 and 5 is it easy to see the algorithmic structure of a xoshiro transformation: the second word of the state array is shifted and stored; then, in order all words of the state array are xor'd with a different word; finally, the shifted part is xor'd into the next-to-last word of the state array, and the last word is rotated. The shape of the matrix depends on the order chosen for the all-words xor sequence. Due to the all-word state mix, xoshiro has better statistical properties than xoroshiro, but at the same time it becomes very slow in higher dimensions.

4 SCRAMBLERS

Scramblers are nonlinear mappings from the state of the linear engine to a w-bit value, which will be the output of the generator. The purpose of a scrambler is to improve the quality of the raw output of the linear engine: since in general linear transformations have several useful provable properties, this is a practical approach to obtain a fast, high-quality generator.

```
    const uint64_t result_plus = s[0] + s[3];
    const uint64_t result_starstar = rotl(s[1] * S, R) * T;

    const uint64_t t = s[1] << A;
    s[2] ^= s[0];
    s[3] ^= s[1];
    s[1] ^= s[2];
    s[0] ^= s[3];
    s[2] ^= t;
    s[3] = rotl(s[3], B);
```

Fig. 4. The C code for a xoshiro256+/xoshiro256** generator. The state array s contains four 64-bit unsigned integers, not all zeros.

```
    const uint64_t result_plus = s[0] + s[2];
    const uint64_t result_starstar = rotl(s[1] * S, R) * T;

    const uint64_t t = s[1] << A;
    s[2] ^= s[0];
    s[5] ^= s[1];
    s[1] ^= s[2];
    s[7] ^= s[3];
    s[3] ^= s[4];
    s[4] ^= s[5];
    s[0] ^= s[6];
    s[6] ^= s[7];
    s[6] ^= t;
    s[7] = rotl(s[7], B);
```

Fig. 5. The C code for a xoshiro512+/xoshiro512** generator. The state array s contains eight 64-bit unsigned integers, not all zeros.

4.1 Sum

The + scrambler simply adds two words of the state array. The choice of words is relevant to the quality of the resulting generator, and we performed a number of tests to choose the best pair depending on the underlying engine. The idea appeared in Saito and Matsumoto's XSadd generator [37], and was subsequently used by the xorshift+ family [43]. In Figure 2-5 the result_plus output is computed using the + scrambler.

Note that the lowest bit output by the + scrambler is just a xor of bits following the same linear recurrence, and thus follows in turn the same linear recurrence. For this reason, we consider + a *weak scrambler*. As we consider higher bits, there is still a linear recurrence describing their behavior, but it becomes quickly of such a high linear complexity to become undetectable. We will discuss in more detail this issue in Section 10.

4.2 Multiplication

The * scrambler multiplies by a constant a chosen word of the state array, and since we are updating more than a word at a time, the choice of the word is relevant. Its only parameter is the multiplier.

The multiplier must be odd; moreover, if the second-lowest bit set is in position b, the first b bits of the output are unmodified, and the following bit is a xor of bit 0 and bit b, so the same consideration of the + scrambler apply to the lowest $b + 1$ bits. For this reason, we consider also * a weak scrambler.

We will use multipliers close to $\varphi 2^w$, where φ is the golden ratio, as φ is an excellent multiplier for multiplicative hashing [16]. To minimize the number of unmodified bits, however, we will adjust the lower bits in such a way that bit 1 is set. In Figure 2 and 3 the `result_star` output is computed using the * scrambler.

4.3 Multiplication, rotation, and again multiplication

The ** scrambler is given by a multiply-rotate-multiply sequence applied to a chosen word of the state array (once again, since we are updating more than a word at a time, the choice of the word is relevant). It thus requires three parameters: the first multiplier, the amount of left rotation, and the second multiplier. Since the rotation moves the highest bits obtained after the first multiplication to the lowest bits, it is easy to set up the parameters so that there are no linear bits (of low linear complexity) in the output. For this reason, we consider ** a *strong scrambler*.

We will mostly use multipliers of the form $2^s + 1$, which are usually computed very quickly, and which have the advantage of being alternatively implementable with a left shift by s and a sum (the compiler should make the right choice, but one can also benchmark both implementations). If Figure 2-5 the `result_starstar` output is computed using the ** scrambler.

5 COMBINING LINEAR ENGINES AND SCRAMBLERS

In this section we discuss a number of interesting combinations of linear engines and scramblers, both for the 64-bit and the 32-bit case, and report results of empirical tests. We remark that all our generators, being based on linear engines, have *jump functions* that make it possible to move ahead quickly by any number of next-state steps. Please refer to [10] for a simple explanation, and for a fast algorithm.[5]

Part of our experiments use the BigCrush test suite from the well-known framework TestU01 [21]. We follow the protocol described in in [42], which we briefly recall. We *sample* generators by executing BigCrush starting from a number of different seeds.[6] We consider a test failed if its p-value is outside of the interval $[0.001 . . 0.999]$. We call *systematic* a failure that happens for all seeds, and report systematic failures (a more detailed discussion of this choice can be found in [42]). Note that we run our tests both on a generator and on its reverse, that is, on the generator obtained by reversing the order of the 64 bits returned.

Moreover, we run a new test aimed at detecting Hamming-weight dependencies, that is, dependencies in the number of zeros and ones in each output word. The test will be described in full in Section 6. We run the test until we examine a petabyte (10^{15} bytes) of data, or if we obtain a p-value smaller than 10^{-20}, in which case we report the amount of data at which we stop. The amount gives an idea of how much the generator is resilient to the test, and makes it possible to discuss the seriousness of the failure. Note that, as discussed in Section 6, the test is very strong: it is able to find bias in the Tiny Mersenne Twister, for which no bias was previously known, except for linear artifacts (i.e., binary rank and linear complexity).

---

[5]We note that computation of jump functions can be enormously sped up by exploiting the built-in *carry-less multiplication* instructions available on modern processors; such instructions, in practice, implement multiplication of polynomials on $\mathbf{Z}/2\mathbf{Z}$.

[6]TestU01 is a 32-bit test suite. As in [42], 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, so the entire output of the generator is examined.

We warn immediately the reader that we do *not* consider all the generators we discuss useful from a practical viewpoint, but discussing several combinations and their test failures brings to the light the limitation of each component in a much clearer way. If the main interest is a practical choice, we suggest to skip to Section 5.3.

5.1 The 64-bit case

We consider engines xoroshiro128, xoshiro256, xoshiro512 and xoroshiro1024; parameters are provided in Table 2. The reason for the switch between the two engines at different sizes is that xoshiro is not definable for a state of $2w$ bits, and it is too slow for a state of $16w$ bits. However, xoshiro has better statistical properties, as clearly shown by the first eight lines of Table 1. All linear engines have obvious linear artifacts, but the xoroshiro engines requires an order of magnitude less data to fail our Hamming-weight dependency test. Note that this is not only a matter of size, but also of structure: compare xoshiro512 and xoroshiro1024. Analogously, the + scrambler deletes all detectable bias from the xoshiro generators, but it just improves the resilience of xoroshiro+ by almost three orders of magnitudes.

We then present data on the xoroshiro generators combined with the * scrambler: as the reader can notice, the * scrambler does a much better job at deleting Hamming-weight dependencies, but a worse job at deleting linear dependencies, as xoroshiro128* still fails MatrixRank when reversed. In Section 10 we will present some theory explaining in detail why this happens. Once we switch to the ** scrambler, we are not able to detect any bias. The parameters for all scramblers are provided in Table 3.

Our speed tests have been performed on an Intel® Core™ i7-7770 CPU @3.60GHz (Kaby Lake) using gcc 6.4.1. We used suitable options to keep the compiler from unrolling loops, or extracting loop invariants, or vectorizing the computation under the hood.

5.2 The 32-bit case

We consider engines xoroshiro64 and xoshiro128. Most of the considerations of the previous section are valid, but in this case for xoroshiro we suggest * as a weak scrambler: the + scrambler, albeit faster, in this case is actually too weak. As in the previous case, the lowest bits of the generators using a weak scrambler are linear: however, since the output is just 32 bits, BigCrush detects this linearity (see failures in the reverse test).[7]

Again, once we switch to the ** scrambler, we are not able to detect any bias. The parameters for all scramblers are provided in Table 6.

---

[7]We remark that testing *subsets* of bits of the output in the 64-bit case can lead to analogous results: as long as the subset contains the lowest bits in the most significant positions, BigCrush will be able to detect their linearity. This happens, for example, if one rotates right by one or more positions (or reverses the output) and then tests just the upper bits, or if one tests the lowest 32 bits, reversed. Subsets not containing the lowest bits of the generators will exhibit no systematic failures of MatrixRank or LinearComp. In principle, the linearity artifacts of the lowest bits might be detected also simply by modifying the parameters of the TestU01 tests. We will discuss in detail the linear complexity of the lowest bits in Section 10.

Table 1. Results of tests for 64-bit generators. The columns "S" and "R" report systematic failures in BigCrush (MR=MatrixRank, i.e., binary rank, LC=LinearComp, i.e., linear complexity). The column "HWD" reports the number of bytes generating a p-value smaller than 10^{-20} in the test described in Section 6; no value means that the test was passed after 10^{15} bytes. The time to emit a 64-bit integer and the number of clock cycles per byte were computed on an Intel® Core™ i7-7770 CPU @3.60GHz (Kaby Lake).

Generator	Failures			ns/64 b	cycles/B
	S	R	HWD		
xoroshiro128	MR, LC	MR, LC	1×10^{10}	0.79	0.35
xoshiro256	MR, LC	MR, LC	6×10^{13}	0.79	0.36
xoshiro512	MR, LC	MR, LC	—	0.90	0.40
xoroshiro1024	MR, LC	MR, LC	5×10^{12}	1.13	0.51
xoroshiro128+	—	—	8×10^{12}	0.79	0.36
xoshiro256+	—	—	—	0.79	0.36
xoshiro512+	—	—	—	0.96	0.43
xoroshiro1024+	—	—	4×10^{13}	1.14	0.51
xoroshiro128*	—	MR	—	0.97	0.44
xoroshiro1024*	—	—	—	1.20	0.54
xoroshiro128**	—	—	—	0.95	0.43
xoshiro256**	—	—	—	0.90	0.41
xoshiro512**	—	—	—	1.07	0.48
xoroshiro1024**	—	—	—	1.27	0.57

Table 2. Parameters suggested for the 64-bit linear engines used in Table 1.

Engine	A	B	C	Weight
xoroshiro128	24	16	37	53
xoshiro256	17	45	—	115
xoshiro512	11	21	—	251
xoroshiro1024	25	27	36	439

Table 3. Parameters suggested for the 64-bit scramblers used in Table 1.

Scrambler	S	R	T
*	0x9e3779b97f4a7c13	—	—
**		5	7	9

Table 4. Results of tests for 32-bit generators. The column labels are the same as Table 1.

Generator	Failures		
	S	R	HWD
xoroshiro64	MR, LC	MR, LC	5×10^8
xoshiro128	MR, LC	MR, LC	3.5×10^{13}
xoroshiro64*	—	MR, LC	—
xoshiro128+	—	MR, LC	—
xoroshiro64**	—	—	—
xoshiro128**	—	—	—

Table 5. Parameters suggested for the 32-bit linear engines used in Table 1.

Engine	A	B	C	Weight
xoroshiro64	26	9	13	31
xoshiro128	9	11	—	55

Table 6. Parameters suggested for the 32-bit scramblers used in Table 4.

Generator	S	R	T
xoroshiro64*	0x9E3779BB	—	—
xoroshiro64**	0x9E3779BB	5	5
xoshiro128**		5	7	9

5.3 Choosing a generator

Our main 64−bit proposal for an all-purpose, rock-solid generator is xoshiro256**. It has excellent speed, a state space that is large enough for any parallel application,[8] and passes all tests we are aware of. While specified using multiplications, it can be implemented using only a few shifts, xors and sums (rotations can be implemented with shifts, when not directly available).

If, however, one has to generate only 64-bit floating-point numbers (by extracting the upper 53 bits), or if the mild linear artifacts in its lowest 2 bits are not considered problematic, xoshiro256+ is a faster generator with analogous statistical properties.[9]

There are however some cases in which 256 bits of state are considered too much, for instance when throwing a very large number of lightweight threads, or in embedded hardware. In this case a similar discussion applies to xoroshiro128** and xoroshiro128+, with the *caveat* that the latter has mild problems with the Hamming-weight dependency test that will be discussed in Section 6: however, bias can be detected only after 8 TB of data, which makes it unlikely to affect in any way applications.

---

[8]With 256 bit of state, 2^{64} sequences starting at 2^{64} random points in the state space have an overlap probability of $\approx 2^{-64}$, which is entirely negligible.

[9]On our hardware, generating a floating-point number with 53 significant bits takes 1.15 ns. This datum can be compared, for example, with the dSFMT [36], which using extended SSE instructions provides a double with 52 significant bits only in 0.95 ns.

Finally, there might be cases that we cannot foresee in which more bits of state are necessary: `xoshiro512**` and `xoshiro512+` should be the first choice, switching to `xoroshiro1024**` or `xoroshiro1024*` if even more bits are necessary. In particular, if rotations are available `xoroshiro1024*` is an obvious better replacement for `xorshift1024*` [42]. As previously discussed, however, it is very difficult to motivate from a theoretical viewpoint a generator with more than 256 bits of state.[10]

Turning to 32-bit generators, `xoshiro128**` and `xoshiro128+` have a role corresponding to `xoshiro256**` and `xoshiro256+` in the 64-bit case: `xoshiro128**` is our first choice, while `xoshiro128+` is excellent for 32-bit floating-point generation. For `xoroshiro64` we suggest however a `*` scrambler, as the `+` scrambler turns out to be too weak for this simple engine.

We suggest to use a SplitMix [40] generator to initialize the state of our generators starting from a 64-bit seed, as research has shown that initialization must be performed with a generator radically different in nature from the one initialized to avoid correlation on similar seeds [30]. Since SplitMix is an equidistributed generator, the resulting initialized state will never be the all-zero state.

6 TESTING HAMMING-WEIGHT DEPENDENCIES

In this section we describe in detail the new statistical test we used beyond BigCrush for our experimental evaluation of Section 5 (Table 1 and 4). Similarly to the binary-rank and linear-complexity tests of BigCrush, our test does not depend on the *numerical values* of the numbers output by the generator: indeed, sorting the bits of each examined block (e.g., first all zeroes and then all ones) would not modify the results of the test (albeit the values would now be very small).

Hamming-weight dependency tests try to discover some statistical bias in the Hamming weight (i.e., the number of ones in binary representation, denoted by #) of the w-bit words emitted by a generator. Since the number of ones in a random w-bit word has a binomial distribution with w trials and probability of success $1/2$, in the most trivial instance one examines m consecutive outputs $x_0, x_1, \ldots, x_{m-1}$ and checks that the average of the their Hamming weights has the correct distribution (which will be quickly approximated very well by a normal distribution as m grows). More sophisticated tests try to detect *dependencies*: for example, one can consider (overlapping) *pairs* of consecutive outputs, and check that the associated pairs of Hamming weights have the expected distribution [20].

Dependency in Hamming weights is a known problem for linear generators: for example, the Mersenne Twister [28], when initialized in a state containing a single bit set to one, requires about a hundred thousand iterations before the Hamming weight of the output has the correct distribution. This happens because linear transformations (in particular, those with sparse matrices) map bit vectors of low weight into bit vectors of low weight. Matsumoto and Nashimura [29] have introduced a theoretical figure of merit that is able to predict after how many samples a linear generator will fail a specific Hamming-weight test.

During the design and testing of our generators we used extensively a test for detecting multidimensional Hamming-weight dependencies that was initially developed by the first author as part of the `gjrand` framework for testing PRNGs. In its current incarnation, the test is very powerful: for example, it finds in a matter of seconds bias in `xorshift128+` or `xorshift1024+` [43], in a matter of hours in `xoroshiro128+`, and in a matter of days in low-dimensional Mersenne Twisters and in the Tiny Mersenne Twister [38], for which no bias was previously known beyond linearity tests. Similarly to linearity tests, a failure in our test is an indication of lesser randomness, but in general the impact on applications will be negligible if we can detect bias only after, say, a gigabyte of data.

---

[10]We remark that, as discussed in Section 7, it is possible to create `xoroshiro` generators with even more bits of state.

6.1 Details

The test considers sequences of w-bit values (w even and not too small, say ≥ 16) extracted from the output of the generator. Usually w will be the entire output of the generator, but it is possible to run the test on a subset of bits, break the generator output in smaller pieces fed sequentially to the test, or glue (a subset of) the generator output into larger pieces.

We fix a parameter k and consider overlapping k-tuples of consecutive w-bit values (ideally, the number of bits of generator state should be less than kw). We map each w-bit value x into $[\#x \geq w/2 - \ell] + [\#x > w/2 + \ell]$: in other words, we count the number of ones in x and categorize them, as it is typical, in three classes: left tail (before the $2\ell + 1$ most frequent values), central (the $2\ell + 1$ central, most frequent values), right tail (after the $2\ell + 1$ most frequent values). The standard choice for ℓ is the largest integer such that the overall probability of the most frequent values does not exceed $1/2$.

We thus get from the k-tuple a *signature* of k trits. Now, given a sequence of m values, for each signature s we compute the average number of ones in the word appearing after a k-tuple with signature s in the sequence. This bookkeeping can be easily performed using 3^k integers while streaming the generator output. For a large m, this procedure yields 3^k values v_i with approximately normal distribution, which we normalize to a standard normal distribution.[11][12]

In principle, since the v_i's should be normally distributed one might compute the associated p-values and return a global p-value. But we can make the test much more powerful by applying a Walsh–Hadamard-like transform. The transform is given by the iterated Kronecker product of the base matrix

$$\begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & 0 & -\frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \end{pmatrix}.$$

Assuming that the resulting matrix is indexed using sequences of trits as numerals in base 3, the transform can be implemented recursively in the same vein as the fast Walsh-Hadamard transform. Figure 6 shows C code to compute the transform in place: it should be called with `sig` equal to 3^{k-1}. We shall write v_i' for the transformed values.

Since the matrix is unitary, the v_i''s must appear still to be drawn from a standard normal distribution. Their meaning, however, has changed. Let us write the index $\bar{\imath}$ of a transformed value v_i' as a sequence of trits $t_0, t_1, \ldots, t_{k-1}$. If the trits are all zero, we are just computing the normalized sum of all values, which is of little interest.

On the contrary, if a single trit, say in position $\bar{\jmath}$, is equal to 1, v_i' is given by the sum of all v_i's in which the $\bar{\jmath}$-th trit of i is 0 minus the sum of all v_i's in which the $\bar{\jmath}$-th trit of i is 2: if the Hamming weight of the output depends on the Hamming weight of the output $\bar{\jmath}$ values before, the value of v_i' will be biased.

If instead a single trit in position $\bar{\jmath}$ is equal to 2 we will detect a kind of bias in which the Hamming weight of the current value depends on whether the Hamming weight of the output $\bar{\jmath}$ values before was extremal or average.

---

[11]Breaking the generator output in smaller pieces provides obviously a stronger test, but a test with parameters k and w "covers" kw bits of output: if we analyze instead values made of $w/2$ bits, to cover kw bits of output we need to increase the length of tuples to $2k$, with a quadratic increase of memory usage.

[12]There is also a *transitional* variant of the test: we see the sequence of w-bit values as a stream of bits, xor the stream with itself shifted forward by one bit, and run the test on the resulting w-bit values. In practice, we look for Hamming-weight dependencies between bit *transitions*.

```
void transform(double v[], int sig) {
    double *p1 = v + sig, *p2 = p1 + sig;
    double a, b, c;

    for (int i = 0; i < sig; i++) {
        a = v[i]; b = p1[i]; c = p2[i];
        v[i] = (a+b+c) / sqrt(3.0);
        p1[i] = (a-c) / sqrt(2.0);
        p2[i] = (2*b-a-c) / sqrt(6.0);
    }

    if (sig /= 3) {
        transform(v, sig);
        transform(p1, sig);
        transform(p2, sig);
    }
}
```

Fig. 6. The code for the (in-place) transform described in Section 6.1. It should be invoked with sig equal to 3^{k-1}.

More complex trit patterns detect more complex dependencies (see Table 7): the most interesting patterns, however, are those with few nonzero trits (except for the index 0, whose associated value v'_0 we discard). Thus, we divide indices in C categories using the number of nonzero trits contained in their ternary representation: the first category contains indices with a single nonzero trit, the second category contains indices with two nonzero trits, and so on, until the last category, which contains all indices containing at least C nonzero trits (of course, $C \leq k$; usually, $C = \lfloor k/2 \rfloor + 1$).

A p-value is computed for each v'_i in a category; for each category we take then the minimum p-value obtained, say, $\bar{p}$, and compensate for the fact that we are considering a minimum by computing $1 - (1 - \bar{p})^c$, where c is the cardinality of the category. Finally, we take the minimum compensated p-value over all categories, and compensate once again for the fact we are considering the minimum over C values. This arrangement puts indices with a higher chance of giving low p-values in small categories, making the test more sensitive.

6.2 Results

We ran tests with $w = 32$ or $w = 64$ and k ranging from 8 to 16, depending on the state size. We performed the test incrementally, that is, for increasingly larger values of m, and stopped after a petabyte (10^{15} bytes) of if we detected a p-value smaller than 10^{-20}. We remark that statistical testing usually involves examining data that is orders of magnitude smaller.

Table 7 reports some results of our test on failing generators. All generators considered other than xorshift pass BigCrush, except for linearity tests (MatrixRank and LinearComp). We report the faulty signature, that is, the pattern of dependencies that caused the low p-value: it provides interesting insights on the structure of the generator. In Section 7 we will find a possible explanation for the increasing resistance to the test, starting from xorshift and going towards xoshiro.

We remark that our test is able to find bias in some small generator of the Mersenne Twister family. First of all, we consider the 64-bit Tiny Mersenne Twister [38], which has 127 bits of state and a significantly more complex structure than the other generators in Table 7: indeed, it is four

Table 7. Detailed results of the test described in Section 6 for $w = 64$. We report the number of bytes generating a p-value smaller than 10^{-20}; generators from Table 1 and 4 passing the test after 10^{15} bytes of data are not listed. We report also the trit signature which caused the low p-value. The results should be compared with those of Table 1.

Generator	$p = 10^{-20}$ @	Faulty signature
xorshift128	8×10^8	00000021
xorshift128+	6×10^9	00000021 (trans.)
xorshift1024	6×10^8	2000000000000001
xorshift1024+	9×10^9	2000000000000001
xoroshiro128	1×10^{10}	00000012
xoroshiro128+	8×10^{12}	00000012
xoroshiro1024	5×10^{12}	1200000000000001
xoroshiro1024+	4×10^{13}	1200000000000001
xoshiro256	6×10^{13}	000120102001 ($w = 32$)
Tiny Mersenne Twister (127 bits)	6×10^{14}	10001021 (trans., $w = 32$)
Mersenne Twister (521 bits)	$4 \times 10^{10}-$	1000000100000000
		2000000100000000
Mersenne Twister (607 bits)	$4 \times 10^8 - 4 \times 10^{10}$	1000000001000000000
		2000000001000000000

to five times slower. Moreover, contrarily to other members of the Mersenne Twister family, the Tiny Mersenne Twister *is scrambled*, as the output is computed by adding the two components of its state space in $\mathbf{Z}/2^{64}\mathbf{Z}$ and applying some further bijective transformation. To find the bias, we had to resort to a slightly more detailed analysis, using $w = 32$ and breaking up the 64-bit output of the generator in two 32-bit words. The best results were given by the transitional variant of the test, but the standard version is only slightly less sensitive. The interesting fact is that *no previous tests were known to fail for this generator* (except for the usual binary rank for large matrices).

We also analyzed the classical Mersenne Twister [28] at 521 and 607 bits. We used Matsumoto and Nishimura's library for dynamic creation of Mersenne Twisters [27], and generated eight different instances of each generator: this is why we report in Table 7 a range of values and multiple signatures. The results are quite interesting.

Firstly, all these generators fail our test quite soon, and also for these generators no failure in tests except for linearity tests was known.[13] Secondly, the 607-bit version performs *much worse* than the 521-bit version. But, more importantly, we found *huge variability* in the test results depending on the parameter generated by the library: in some cases the 607-bit Mersenne Twister perform worse than a xorshift128 generator. In other words, our results suggest that the dynamic generation process described in [11] yields generators with wildly different statistical quality, and should be avoided.

---

[13]Some tests based on specific set of lags had already caused other failures, such as in [11], but these tests (and the lags) were chosen using deep knowledge of the generator.

All in all, we believe that our results show that dependencies on Hamming weights should be considered a systematic problem for linear generators, together with more classical linear artifacts (i.e., failures in binary-rank and linear-complexity tests). In particular, our test shows that such dependencies are more difficult to mask by scrambling than low linear complexity.

6.3 Implementation

To be able to perform our test in the petabyte range, we carefully engineered its implementation: in particular, the main loop enumerating the output of the generator and computing the values v_i must be as fast as possible. Counting the number of ones in a word can be performed using single-clock specialized instructions in modern CPUs. Moreover, one can keep track very easily of the current trit signature by using the update rule $s \leftarrow \lfloor s/3 \rfloor + t \cdot 3^{k-1}$, where t is the next trit. We can replace the division with the fixed-point computation $\lfloor (\lceil 2^{32}/3 \rceil s)/2^{32} \rfloor$ (this strategy works up to $k = 19$; for larger k, one needs to perform an actual division), so by precomputing $\lceil 2^{32}/3 \rceil$ and 3^{k-1} the costly operations in the update of s can be reduced to two independent multiplications.

The main implementation challenge, however, is that of reducing the counter update area so that it fits into the cache. In a naive implementation, we would need to use two "large" 64-bit values. Instead, we will use a single "small" 32-bit value to store both the number of appearances of signature s, and the sum of Hamming weights of the following words, with a fourfold space saving. In particular, we will use 13 bits for the counter and 19 bits for the summation. We fix a batch size, and update the small values blindly through the batch. At the end of the batch, we update the large counters using the current values of the small counters, and zero the latter ones. At the same time, we check that the sum of the small counters is equal to the batch size: if not, a counter overflowed. Otherwise, we continue with the next batch, possibly computing the transform and generating a p-value.

How large should a batch be? Of course we prefer larger batches, as access to large counters will be minimized, but too large a batch will overflow small counters. This question is actually interesting, as it is related to the *mean passage time distribution* of the Markov chain having all possible signatures as states, and probability of moving from signature s to signature $\lfloor s/3 \rfloor + t \cdot 3^{k-1}$ given be the probability of observing the trit t. Let this probability be p for the central values (trit 1), and $(1 - p)/2$ for the extremal values (trits 0 and 2). We are interested in the following question: given a k, p and a batch size B, what is the probability that a counter will overflow? This question can be reduced to the question: given k, p and a batch size B, what is the probability that the Markov chain after B steps passes through the all-one signature more than 2^{13} times?[14] We want to keep this probability very low (say, 10^{-100}) so to not interfere with the computation of the p-values from the test; moreover, in this way if we detect a counter overflow we can simply report that we witnessed an event that cannot happen with probability less greater than 10^{-100}, given that the source is random, that is, a p-value.

A first observation is that for our analysis we can *lump* all 3^k states into $k + 1$ superstates. Each superstate is defined by its maximal suffix of trits 1: the first superstate lumps all signatures (states) whose last trit is 0 or 2; the second superstate the signatures whose last trit is a 1, but the next-to-last a 0 or a 2; and so on, until the last superstate, which contains the only signature (state) made entirely by ones. Transition probabilities are obvious: from all superstates we have a probability of $(1 - p)$ to go to the first superstate, and probability p to move to the superstate with

---

[14]It is obvious that the the all-ones signature has the highest probability in the steady-state distribution, and that by bounding its probability of overflow we obtain a valid bound also for all other signatures.

a longer suffix, except for the all-ones state, which has a probability p of returning to itself:

$$\begin{pmatrix}
1-p & p & 0 & 0 & \ldots & 0 & 0 \\
1-p & 0 & p & 0 & \ldots & 0 & 0 \\
1-p & 0 & 0 & p & \ldots & 0 & 0 \\
 & & & \ldots & & & \\
1-p & 0 & 0 & 0 & \ldots & p & 0 \\
1-p & 0 & 0 & 0 & \ldots & 0 & p \\
1-p & 0 & 0 & 0 & \ldots & 0 & p
\end{pmatrix}.$$

As a first approximation, we could use general results about Markov chain [12, Theorem 7.4.2] which state that in the limit the number of passages is normally distributed with mean and variance related to those of the *recurrence time distribution*, which can in turn be computed symbolically using the Drazin inverse [13, 31].

Since, however, no explicit bound is known for the convergence speed of the limit above, we decided to compute exactly the mean passage time distribution for the all-ones signature. To do this, we model the problem as a further Markov chain with states $x_{c,s}$, where $0 \le c \le m$, m is the overflow bound, and $0 \le s < k$ is a superstate of the previous Markov chain, with the proviso that the value $s = k - 1$ lumps further together the superstates $k - 1$ and k (we can lump them together as receiving a one increases the passage count in both cases).

The idea is that we will define transitions so that after t steps the probability of being in state $x_{c,s}$ will be the probability of having the original Markov chain being in a state lumped into s, and having counted exactly c passages through the all-ones (super)state (m or more, when $c = m$). We use an initial probability distribution in which all states with $c \ne 0$ have probability zero, and all states with $c = 0$ have probability equal to the state-steady probability of s, which implies that we are implicitly starting the original chain in the steady state. However, as argued also in [12], the initial distribution is essentially irrelevant in this context.

We now define the transitions so that the probability distribution of the new chain evolves in parallel with the distribution of passage times of the original chain (with the probability for more than m passages lumped together):

- all states have a transition with probability $1 - p$ to $x_{c,0}$;
- all states $x_{c,s}$ with $s < k - 1$ have a transition with probability p to $x_{c,s+1}$;
- all states $x_{c,k-1}$ with $c < m$ have a transition with probability p to $x_{c+1,k-1}$;
- there is a loop with probability p on $x_{m,k-1}$.

It is easy to show that after t steps the sum of the probabilities associated with the states $x_{-,m}$ is exactly the probability of overflow of the counter associated with the all-one (super)state of the original chain. We thus iterate the Markov chain until, say at step T, we obtain a probability of, say, $3^{-k}\bar{p}$: we can then guarantee that, given that the source is random, running our test with blocks of size T we can observe overflow only with probability at most $\bar{p}$.

This approach becomes unfeasible when we need to iterate the Markov chain more than, say 10^7 times. However, at that point we can use a very close approximation: we apply a simple dynamic-programming scheme on the results for 10^6 steps to extend the results to larger number of steps, using a strategy analogous to exponentiation by quadratures. The approximation is due to the fact that doing so we are implicitly assuming that the Markov chain is reset to its steady-state distribution after each 10^6 steps, but experiments at smaller sizes show that this approximation is, as expected, negligible. We can also report that in our case the normal approximation is not very precise even after a billion steps.

In the end, we computed T for $1 \leq k \leq 19$ and included the result into the our code (for example, when $w = 64$ one has $T \approx 15 \times 10^3$ for $k = 1$, $T \approx 23 \times 10^6$ for $k = 8$ and $T \approx 10^9$ for $k = 16$). Combining all ideas described in this section, our test for Hamming-weight dependencies with parameters $w = 64$ and $k = 8$ can analyze a terabyte of output of a 64-bit generator in about 3 minutes on an Intel® Core™ i7-7770 CPU @3.60GHz (Kaby Lake). The $k = 16$ test is an order of magnitude slower due to the larger memory accessed.

7 POLYNOMIALS AND FULL-PERIOD GENERATORS

One of the fundamental tools in the investigation of linear transformations is the *characteristic polynomial*. If M is the matrix representing the transformation associated with a linear engine the characteristic polynomial is

$$P(x) = \det(M - xI).$$

The associated linear engine has *full period* (i.e., maximum-length period) if and only if $P(x)$ is *primitive* over $\mathbf{Z}/2\mathbf{Z}$ [23], that is, if $P(x)$ is irreducible and if x has maximum period in the ring of polynomials over $\mathbf{Z}/2\mathbf{Z}$ modulo $P(x)$. By enumerating all possible parameter choices and checking primitivity of the associated polynomials we can discover all full-period generators.

In particular, *every bit* of a linear engine is a LFSR with characteristic polynomial $P(x)$. Different bits emit different outputs because they return sequences from different starting points in the orbit of the LFSR.

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 linear engine of this kind[15] that the weight is close to $n/2$, that is, that the polynomial is neither too sparse nor too dense [6].

7.1 Word polynomials

A matrix M of size $kw \times kw$ can be viewed as a $k \times k$ matrix on the ring of $w \times w$ matrices. At that point, some generalization of the determinant to noncommutative rings can be used to obtain a characteristic polynomial $P_w(x)$ for M (which will be a polynomial with $w \times w$ matrices as coefficients): if the determinant of $P_w(x)$ on $\mathbf{Z}/2\mathbf{Z}$ is equal to the characteristic polynomial of M, then $P_w(x)$ is a *word polynomial of size w for M*.

The main purpose of word polynomials is to make easier the computation of the characteristic polynomial of M (and thus of the determinant of M), in particular for large matrices. But we will see that under some further commutativity constraints word polynomials are just a different representation of M, as in the commutative case, and we will present some empirical evidence that in all cases they provide useful information about the dependency of an output from the previous outputs.

In all our examples w is the intended output size of the generator, but it some cases it might be necessary to use a smaller w, say, $w/2$, to satisfy commutativity conditions: one might speak, in that case, of the *semi-word polynomial*. For $w = 1$ the word polynomial is simply the characteristic polynomial; the commutation constraints are trivially satisfied.

7.2 The commutative case

If all $w \times w$ blocks of M commute, then a very well-known result from Bourbaki [1] shows that computing the determinant of M in the commutative ring of $w \times w$ matrices R generated by the

---

[15]Technically, the criterion applies *to the LFSR represented by the characteristic polynomial*. The behavior of a linear engine, however, depends also on the relationships among its w bits, so the degree criterion must always be weighed against other evidence.

$w \times w$ blocks of M one has

$$\det(\mathrm{Det}(M)) = \det(M), \tag{2}$$

where "det" denotes the determinant in the base ring (in our case, $\mathbf{Z}/2\mathbf{Z}$) whereas "Det" denotes the determinant in R. This equivalence provides a very handy way to compute easily the determinants of large matrices with a block structure containing several zero blocks and commuting non-zero blocks: one simply operates on the blocks of the matrix as if they were scalars.

However, if M is the matrix associated with a linear engine, $\mathrm{Det}(M)$ can be used also to characterize how the current state of the generator depends on its previous states. Indeed, since we are working in a commutative ring the Cayley–Hamilton theorem holds, and thus M satisfies its own characteristic polynomial: if we let $P(x) = \mathrm{Det}(M - xI)$, then $P(M) = 0$. More in detail, if

$$P(x) = x^k + A_{k-1}x^{k-1} + \cdots + A_1 x + A_0$$

then

$$M^k + M^{k-1}A_{k-1} + \cdots + MA_1 + A_0 = 0.$$

Thus, given a sequence of states of the generator $s_0, s_1 = s_0 M, s_2 = s_0 M^2, \ldots, s_r = s_0 M^r$ we have[16]

$$s_r = s_{r-1}A_{k-1} + \cdots + s_{r-k+1}A_1 + s_{r-k}A_0.$$

This recurrence makes it possible to compute the next state of the generator knowing its previous w states.

This consideration may seem trivial, as we already know how to compute the next state given the previous state—a multiplication by M is sufficient—but the recurrence is true of *every* w-bit block of the state array. Said otherwise, no matter which word of the state array we use as output, we can *predict* the next output using the equation above knowing the last w outputs of the generator. Another way of looking at the same statement is that we expressed the linear engine described by M using Niederreiter's *multiple-recursive matrix method* [33].

It is thus tempting to conjecture that a too simple combination of previous outputs might be easy to discover, and that the structure of $P(x)$ might help us in identifying generators whose outputs do not satisfy too simple equations.

7.3 The noncommutative case

There are three issues in generalizing the arguments we made about the commutative case: first, we need a notion of noncommutative determinant; second, it would be useful to know whether (2) generalizes to our case; and third, we need to know whether a generalization of the Cayley–Hamilton theorem applies to our noncommutative determinant.

The first two problems can be solved by recent results of Sothanaphan [39]. One starts by defining a (standard) notion of determinant for noncommutative rings by fixing the order of the products in Leibniz's formula. In particular, we denote with $\mathrm{Det}^{\mathrm{r}}$ the *row-determinant* of an $n \times n$ matrix M on a noncommutative base ring:

$$\mathrm{Det}^{\mathrm{r}}(M) = \sum_{\pi \in S_n} \mathrm{sgn}(\pi) M_{0,\pi(0)} M_{1,\pi(1)} \cdots M_{n-1,\pi(n-1)} \tag{3}$$

Note that the definition is based on Leibniz's formula, but the order of the products has been fixed. Then, Theorem 1.2 of [39] shows that

$$\det(\mathrm{Det}^{\mathrm{r}}(M)) = \det(M), \tag{4}$$

provided that blocks in different columns, but not in the first row, commute. In other words, one can compute the characteristic polynomial of M by first computing the "row" characteristic polynomial

---

[16]Note that in this formula the coefficients A_i are $w \times w$ matrices of the ring R generated by the blocks of M.

of M by blocks and then computing the determinant of the resulting matrix. By the definition we gave, in this case $\text{Det}^r(M - xI)$ is a word polynomial for M.

The row-determinant is (trivially) antisymmetric with respect to the permutation of columns.[17] Moreover, a basic property of row-determinants depends on a commutativity condition on the matrix entries (blocks): if M has *weakly column-symmetric commutators*, that is, if

$$M_{ij}M_{kl} - M_{kl}M_{ij} = M_{il}M_{kj} - M_{kj}M_{il} \quad \text{whenever } i \neq k \text{ and } j \neq l,$$

then the row-determinant is antisymmetric with respect to the permutation of *rows* [3].

Dually, we can define the *column-determinant*

$$\text{Det}^c(M) = \sum_{\pi \in S_n} \text{sgn}(\pi) M_{\pi(0),0} M_{\pi(1),1} \cdots M_{\pi(n-1),n-1}. \tag{5}$$

All recalled properties can be easily dualized to the case of the column-determinant: in particular,

$$\det(\text{Det}^c(M)) = \det(M), \tag{6}$$

provided that blocks in different rows, but not in the first column, commute; if M has *weakly row-symmetric commutators*, that is, if

$$M_{ij}M_{kl} - M_{kl}M_{ij} = M_{kj}M_{il} - M_{il}M_{kj} \quad \text{whenever } i \neq k \text{ and } j \neq l,$$

then the column-determinant is antisymmetric with respect to the permutation of *columns* [3].

Finally, if M is *weakly commutative* [3], that is, M_{ij} and M_{kl} commute whenever $i \neq k$ and $j \neq l$ (i.e., noncommuting blocks lie either on the same row or on the same column) the two determinants are the same, as all products in (3) and (5) can be rearranged arbitrarily.[18]

7.3.1 Warmup: `xorshift`. The generators of the `xorshift` family [25] with more than $2w$ bits of state are defined by the linear transformation

$$\mathscr{M}_{kw} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & (I + S^a)\left(I + \left(S^T\right)^b\right) \\ I & 0 & 0 & \cdots & 0 & 0 \\ 0 & I & 0 & \cdots & 0 & 0 \\ 0 & 0 & I & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & I & \left(I + \left(S^T\right)^c\right) \end{pmatrix},$$

where S^T is the right-shift matrix. The matrix $\mathscr{M}_{kw}$ is weakly commutative and satisfies trivially the conditions for (4), so we can compute the row-determinant (which is equal to the column-determinant) by performing Laplace expansion along the first line, noting that all resulting minors are made of commuting blocks:

$$\begin{aligned} \text{Det}^c(\mathscr{M}_{kw} - xI) &= \text{Det}^r(\mathscr{M}_{kw} - xI) \\ &= x^{k-1}\left(I + \left(S^T\right)^c + xI\right) + (I + S^a)\left(I + \left(S^T\right)^b\right) \\ &= x^k I + x^{k-1}\left(I + \left(S^T\right)^c\right) + (I + S^a)\left(I + \left(S^T\right)^b\right). \end{aligned}$$

We thus obtain our first word polynomial.

---

[17]In our case, that is, on the base field $\mathbf{Z}/2\mathbf{Z}$ there is no difference between "symmetric" and "antisymmetric" as sign change is the identity. For sake of generality, however, we will recall the properties we need in the general case.

[18]We remark that if the only aim is to compute easily the characteristic polynomial, one can rearrange columns and rows at will until (4) or (6) is true, because these operations cannot change the value of the determinant on $\mathbf{Z}/2\mathbf{Z}$.

Now, an interesting fact is that the transpose of $\mathscr{M}_{kw}$ is a *Manin matrix* [24], a special subclass of matrices on *noncommutative* rings related to quantum groups which enjoy the extension of a number of classical theorems from the theory of matrices on *commutative* rings. In particular, the transpose of a Manin matrix enjoys a version of the Cayley–Hamilton theorem that uses the row-determinant and *left substitution* [5, Theorem 14]:

$$\mathscr{M}_{kw}^k + \mathscr{M}_{kw}^{k-1}\left(I + \left(S^T\right)^c\right) + \left(I + S^a\right)\left(I + \left(S^T\right)^b\right) = 0.$$

Thus, we can reason as in the commutative case: given a sequence of states s_0, $s_1 = s_0\mathscr{M}_{kw}$, $s_2 = s_0\mathscr{M}_{kw}^2$, ..., $s_r = s_0\mathscr{M}_{kw}^r$ we have

$$s_r = s_{r-1}\left(I + \left(S^T\right)^c\right) + s_{r-k}\left(I + S^a\right)\left(I + \left(S^T\right)^b\right). \tag{7}$$

In this particular case, the previous formula is entirely obvious: it corresponds exactly to the cyclic update rule of a high-dimensional `xorshift` generator [42]. When the update matrix is more complex, however, as in our case, it will turn out to be a very useful tool to understand the dependencies among subsequent outputs. Even when we do not have the theoretical support of an extension of the Cayley–Hamilton theorem, we can still check empirically if at least some of the bits are predictable using a word polynomial: indeed, the fact that the determinant of the word polynomial is the determinant of the original linear transformation suggests that there must be relevant information.

We remark that if a certain word of state is equidistributed in the maximum dimension (see Section 8) it is *always* possible to predict the next output given k consecutive outputs: by equidistribution, there is a matrix that given a sequence of k outputs returns the state that emits exactly those outputs: at that point, by advancing the engine k times we obtain a description of each bit of the current state (and thus of the current output) in terms of the bits the previous k outputs. However, such a description is not amenable in general to a block-based description, which, as we will see in Section 7.4, is one of the most interesting features of word polynomials.

7.3.2 `xoroshiro`. This case is more difficult as there is no way to modify the matrix to satisfy the conditions for (4), which would be our first try, given the previous example. However, it is easy to check that $\mathscr{X}_{kw}$ has weakly row-symmetric commutators, so we can move its next-to-last column to the first one, and then the resulting matrix clearly falls into the conditions for (6). We thus obtain a word polynomial based on the column-determinant:

$$\mathrm{Det}^c\left(\mathscr{X}_{kw} - xI\right)$$

$$= \mathrm{Det}^c\begin{pmatrix} R^a + S^b + I & xI & 0 & \cdots & 0 & R^c \\ 0 & I & xI & \cdots & 0 & 0 \\ 0 & 0 & I & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ xI & 0 & 0 & \cdots & I & 0 \\ S^b + I & 0 & 0 & \cdots & 0 & R^c + xI \end{pmatrix}$$

$$= \left(R^a + S^b + I\right)\left(R^c + xI\right) + x\left(R^c + xI\right)x^{k-2} + \left(S^b + I\right)R^c$$

$$= x^k I + x^{k-1}R^c + x\left(R^a + S^b + I\right) + R^{a+c}.$$

Given the results we used about the extensions of the Cayley–Hamilton theorem for Manin matrices, it is unlikely that the polynomial above can satisfy an equation analogous to (7): and indeed it does

not. However, empirically 4/5 of the bits of each word of state can be predicted using the polynomial above when $k = 2$ (xoroshiro128), and the ratio becomes about 1/2 for $k = 16$ (xoroshiro1024).[19]

7.3.3 xoshiro. In this case we have to perform an *ad hoc* maneuver to move $\mathscr{S}_{4w}$ and $\mathscr{S}_{8w}$ in a form amenable to the computation of a word polynomial by row-determinant: we have to exchange the first two rows. It is very easy to see that this operation cannot modify the row-determinant because every element of the first row commutes with every element of the second row: thus, in the products of (3) the first two elements can always be swapped.

At that point, by (a quite tedious) Laplace expansion along the first row we get

$$\mathrm{Det^r}\big(\mathscr{S}_{4w} - xI\big) = x^4 I + x^3\big(R^b + I\big) + x^2\big(S^a + R^b\big) + x\big(S^a + I\big)\big(R^b + I\big) + \big(S^a + I\big)R^b$$

and

$$
\begin{aligned}
\mathrm{Det^r}&\big(\mathscr{S}_{w8} - xI\big) \\
&= x^8 I + x^7\big(R^b + I\big) + x^6\big(R^b + I\big) + x^5\big(S^a + R^b + I\big) + x^4\big(S^a + I\big)\big(R^b + I\big) \\
&\quad + x^3\big(S^a R^b + R^b + S^a\big) + x^2\big(S^a + I\big)\big(R^b + I\big) + x\big(S^a R^b + R^b + I\big) + R^b.
\end{aligned}
$$

We should have a better chance to be able to predict an output using a row-determinant, and this is indeed the case: for $\mathscr{S}_{4w}$ (xoshiro256), the *second* state word can be predicted exactly[20]; for the other words, about two thirds of the bits can be predicted. In the case of $\mathscr{S}_{8w}$ (xoshiro512), *all words except the last one* can be predicted exactly; for the last one, again about two thirds of the bits can be predicted.

7.4 Word polynomials and Hamming-Weight Dependencies

Beside the usefulness of the word polynomial for computing the characteristic polynomial, we want to highlight what appears to be an empirical connection with the test described in Section 6. In Figure 7 we show for a few generators the correlation between the number of nonzero coefficients of the word polynomial (excluding the leading term) and the amount of data required by our test to generate a p-value below 10^{-20}. It is rather evident that the more complex dependencies on previous outputs exhibited by complex word polynomials have a correlation with the amount of data that it is necessary to analyze before finding a bias. We used this fact as a heuristic in the design of our linear engines.

In fact, the connection has a clear explanation: every bit of output predictable by the word polynomial implies the existence of a *dependence* of the bit from a subset of the previous output bits. Dependencies of this kind are very well known in the study of linear generators [11, 29], where their *number of terms* has been suggested as a figure of merit connected to the *Hamming-weight distribution test* [29]. Harase [11] describe sophisticated linear techniques that can find dependencies beyond k (which is the obvious limit using a word polynomial).

In our case, it is interesting to note that once the coefficients have been fully expanded, every rotation (included the identity) contributes a term in all dependencies, but shifts (even multiplied by a rotation) add a term or not depending on their parameters and on the output bit. In this sense, rotations are measurably more efficient in creating dependencies with more terms. For example, some bits of xorshift can depend on as few as three previous bits, but a bit of xoroshiro is guaranteed to depend on at least four, and sometimes five; the number grows with the xoshiro linear engines (we could indeed obtain a more precise version of Figure 7 counting exactly such

---

[19]The empirical observation about predicted bits are based on the parameters of Table 2 and 5: different parameters will generate different results.
[20]Incidentally, if we reverse multiplication order in the coefficients, the *first* word can be predicted exactly instead.
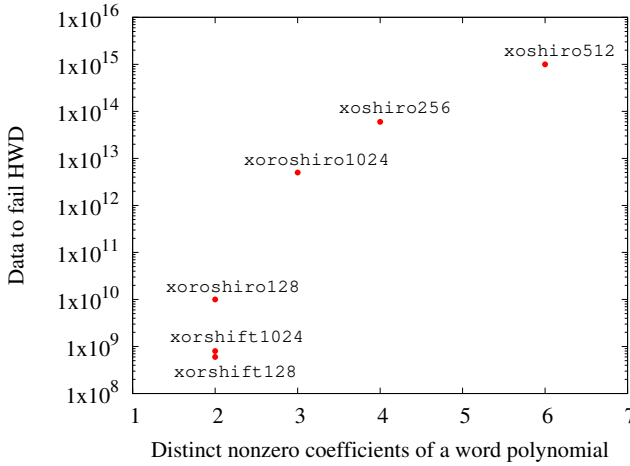
Fig. 7. Correlation between nonzero coefficients of a word polynomial (excluding the leading term) and amount of data required to generate a p-value of less than 10^{-20} using the test of Section 6 on some linear engines. Note that xoshiro512 does not actually fail the test, but we stopped testing after a petabyte of data.

Table 8. Number of xoroshiro primitive polynomials depending on word size and state size.

		State size in bit						
		64	128	256	512	1024	2048	4096
	16	26	21	7	3	1	0	0
w	32	250	149	59	41	16	5	6
	64		1000	491	261	129	42	25

terms, instead of the coefficients, which are a rough approximation). The faulty signatures for xorshift and xoroshiro in Table 7 reflect exactly the terms of the word polynomial with nonzero coefficients, that is, the terms of the induced dependency. We leave for future work a more detailed, formal study of the dependencies induced by word polynomials.

7.5 Full-period generators

Using the word polynomials just described we computed exhaustively all parameters providing full-period generators using the Fermat algebra system [22], stopping the search at 4096 bits of state.[21] Table 8 and 10 report the number of full-period generators, whereas Table 9 and 11 report the maximum weight of a primitive polynomial. As one can expect, with respect to xoshiro we find that xoroshiro has many more full generators at many more different state sizes, due to the additional parameter. We remark that we did not discuss 16-bit generators, but there is a xoshiro

Table 9. Maximum weight of a xoroshiro primitive polynomial depending on word size and state size.

		State size in bit						
		64	128	256	512	1024	2048	4096
	16	37	45	73	35	41		
w	32	39	67	115	201	187	195	143
	64		75	139	263	475	651	653

Table 10. Number of xoshiro primitive polynomials depending on word size and state size.

		State size in bit			
		64	128	256	512
	16	1	0		
w	32		1	0	
	64			4	4

Table 11. Maximum weight of a xoshiro primitive polynomial depending on word size and state size.

		State size in bit			
		64	128	256	512
	16	33			
w	32		55		
	64			131	251

and several xoroshiro choices available. For example, the Parallax Propeller 2 microcontroller contains a version of the 16-bit engine xoroshiro32 in every processor core.

8 EQUIDISTRIBUTION

Equidistribution is a uniformity property of pseudorandom number generators: a generator with kw bits of state and w output bits is d-dimensionally equidistributed if when we consider the sequences of the first d output values over all possible states of the generator, each sequence appears exactly the same number of times [18]. In practice, over the whole output every d-tuple of consecutive output values must appear $2^{w(k-d)}$ times, except for the zero d-tuple, which appears $2^{w(k-d)} - 1$ times, as the zero state is not used. In particular, 1-dimensionally equidistributed generators with w bits of state emit each w-bit value exactly one time, except for the value zero, which is never emitted. We will start by discussing the equidistribution of our linear engines (without scramblers).

---

[21]The reason why the number 4096 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 look for full-period generators with larger state.

Typically, linear generators (e.g., xorshift) update cyclically a position of their state array. In this case, the simple fact that the generator has full period guarantees that the generator is equidistributed in the maximum dimension, that is, k. However, since our linear engines update more than one position at a time, full period is not sufficient, and moreover different words of the state may display different equidistribution properties.

Testing for maximum equidistribution of a word of the state array is easy: if we consider the j-th word, the matrix obtained juxtaposing the j-th block columns of $M^0 = I$, $M^1 = M$, M^2, ..., M^{d-1} must be invertible (the inverted matrix returns, given an output sequence of d words, the state that will emit the sequence). It is straightforward to check that every word of xoroshiro (for every state size) and of xoshiro512 is equidistributed in the maximum dimension. The words of xoshiro256 are equidistributed in the maximum dimension with the exception of the third word, for which equidistribution depends on the parameters; we will not use it.

The $\ast$ and $\ast\ast$ scramblers cannot alter the equidistribution of a linear engine, as they just remap sequences bijectively. Thus, all our generators using such scramblers are k-dimensionally equidistributed (i.e., in the maximum dimension).

We are left with proving equidistribution results for our generators based on the $+$ scrambler. For d-dimensionally equidistributed generators which update cyclically a single position adding two consecutive outputs can be easily proven to provide a $(d-1)$-dimensionally equidistributed generator. However, our generators update more than one position, making *ad hoc* proofs necessary.

Throughout this section $-$ will denote, with a slight abuse of notation, difference in $\mathbf{Z}/2^w\mathbf{Z}$, whereas $+$ will continue to denote sum in $(\mathbf{Z}/2\mathbf{Z})^w$

PROPOSITION 8.1. *When adding the first and last state words a* xoroshiro+ *generator with w bits of output and kw bits of state is $(k-1)$-dimensionally equidistributed.*

PROOF. For the case $k = 2$ the full period of the underlying xoroshiro generator proves the statement.

Consider now a $(k-1)$-tuple $\langle v_0, v_1, v_2, \ldots, v_{k-2} \rangle$, $k > 2$. If we require that the sum in $\mathbf{Z}/2^w\mathbf{Z}$ of the first and last w-bit blocks of state enumerate the v_i's, we have the following initial constraints on the $k-1$ successive states, written as k blocks of w bits:

$$
\begin{matrix}
v_0 - x & ? & ? & ? & ? & \cdots & x \\
? & ? & ? & ? & ? & \cdots & ? \\
? & ? & ? & ? & ? & \cdots & ? \\
& & \cdots & & & & \\
? & ? & ? & ? & ? & \cdots & ?
\end{matrix}
$$

Here x is a free parameter. Because of the shape of $\mathscr{X}_{kw}$ in (1), if we consider the values x_1 and y_1 (their dependence on x is understood) given by

$$
[y_1 \; x_1] = [v_0 - x \; x] \begin{pmatrix} R^a + S^b + I & R^c \\ S^b + I & R^c \end{pmatrix}
$$

we have (propagating values using the identity submatrices) necessarily

$v_0 - x$	$v_1 - x_1$	?	?	?	$\cdots$	?	?	x
$v_1 - x_1$	?	?	?	?	$\cdots$	?	y_1	x_1
?	?	?	?	?	$\cdots$	y_1	?	?
			$\cdots$					
?	?	y_1	?	?	$\cdots$	?	?	?
?	y_1	?	?	?	$\cdots$	?	?	?

Iterating once more, assuming

$$[y_2\ x_2] = [v_1 - x_1\ x_1]\begin{pmatrix} R^a + S^b + I & R^c \\ S^b + I & R^c \end{pmatrix}$$

we have

$v_0 - x$	$v_1 - x_1$	$v_2 - x_2$	?	?	$\cdots$	?	?	x
$v_1 - x_1$	$v_2 - x_2$	?	?	?	$\cdots$	?	y_1	x_1
$v_2 - x_2$	?	?	?	?	$\cdots$	y_1	y_2	x_2
?	?	?	?	?	$\cdots$	y_2	?	?
			$\cdots$					
?	?	y_1	y_2	?	$\cdots$	?	?	?
?	y_1	y_2	?	?	$\cdots$	?	?	?

Eventually, we fill all blocks:

$v_0 - x$	$v_1 - x_1$	$v_2 - x_2$	$v_3 - x_3$	$v_4 - x_5$	$\cdots$	$v_{k-3} - x_{k-3}$	$v_{k-2} - x_{k-2}$	x
$v_1 - x_1$	$v_2 - x_2$	$v_3 - x_3$	$v_4 - x_4$	$v_5 - x_5$	$\cdots$	$v_{k-2} - x_{k-2}$	y_1	x_1
$v_2 - x_2$	$v_3 - x_3$	$v_4 - x_4$	$v_5 - x_5$	$v_6 - x_6$	$\cdots$	y_1	y_2	x_2
$v_3 - x_3$	$v_4 - x_4$	$v_5 - x_5$	$v_6 - x_6$	$v_7 - x_7$	$\cdots$	y_2	y_3	x_3
			$\cdots$					
$v_{k-3} - x_{k-3}$	$v_{k-2} - x_{k-2}$	y_1	y_2	y_3	$\cdots$	y_{k-4}	y_{k-3}	x_{k-3}
$v_{k-2} - x_{k-2}$	y_1	y_2	y_3	y_4	$\cdots$	y_{k-3}	y_{k-2}	x_{k-2}

In other words, for every choice of x we can compute the x_i's and y_i's, and starting from the state $\langle v_0 - x, v_1 - x_1, v_2 - x_2, \ldots, v_{k-3} - x_{k-3}, v_{k-2} - x_{k-2}, x\rangle$ the output is exactly $\langle v_0, v_1, v_2, \ldots, v_{k-2}\rangle$. □

Note that in general it is impossible to claim k-dimensional equidistribution. Consider the full-period 5-bit generator with 10 bits of state and parameters $a = 1$, $b = 3$ and $c = 1$. As a xoroshiro generator it is 2-dimensionally equidistributed, but it is easy to verify that the sequence of outputs of the associated xoroshiro+ generator is not 2-dimensionally equidistributed (it is, of course, 1-dimensionally equidistributed by Proposition 8.1).

PROPOSITION 8.2. *When adding the first and last state words a* xoshiro+ *generator with w bits of output and $4w$ bits of state is 3-dimensionally equidistributed.*

PROOF. Consider a triple of values $\langle v_0, v_1, v_2 \rangle$ and let us compute the first and last words of the two states following $\langle v_0 - z, x, y, z \rangle$, whose output is of course v_0. We obtain the following constraints:

$$(v_0 - z) + x + z = v_1 - (xR^a + zR^a)$$

$$xR^a + y + z(I + R^a) = v_2 - \big((v_0 - z)R^a + x(R^{2a} + R^a) + yR^a + zR^{2a}\big)$$

We now add a variable $u = x + z$, which gives us

$$u = x + z$$

$$z = v_0 - ((v_1 - uR^a) + u)$$

$$uR^a + y + z = v_2 - \big((v_0 - z)R^a + u(R^{2a} + R^a) + zR^a + yR^a\big)$$

Replacing the value of z into $(v_0 - z)$ in the second equation leads to some simplification:

$$u = x + z$$

$$z = v_0 - ((v_1 - uR^a) + u)$$

$$uR^a + y + z = v_2 - \big((v_1 - uR^a)R^a + uR^{2a} + zR^a + yR^a\big)$$

Finally, we add a variable $t = uR^a + z + y$:

$$t = uR^a + z + y$$

$$u = x + z$$

$$z = v_0 - ((v_1 - uR^a) + u)$$

$$t = v_2 - \big((v_1 - uR^a) + t\big)R^a$$

We are now in the position of making the dependence of u on t explicit in the last equation, as u appears just once:

$$u = \big(v_1 - \big((v_2 - t)R^{-a} + t\big)\big)R^{-a}.$$

If we choose a value for t we obtain a value for u and a value for z by the last two equations, and then the first two equations give the values of x and y. Thus, a triple $\langle v_0, v_1, v_2 \rangle$ can appear at most 2^w times, which implies that all triples appear exactly 2^w times. □

Once again, it is impossible to claim 4-dimensional equidistribution. The only possible 2-bit xoshiro generator with 8 bits of state has full period but it is easy to verify that the associated xoshiro+ generator is not 2-dimensionally equidistributed.

Finally, we prove an analogous equidistribution result for xoshiro512+.

PROPOSITION 8.3. *When adding the first and third state words a* xoshiro+ *generator with w bits of output and $8w$ bits of state is 7-dimensionally equidistributed.*

PROOF. A manual proof like that of Proposition 8.2 would be quite unmanageable in this case. We thus proceed to develop a more general technique: we consider a vector of variables $\boldsymbol{x} = \langle x_0, x_1, \ldots, x_7 \rangle$ representing the state of the generator. Then for each $0 \le i < d$, where d is the target equidistribution, we add variables t_i, u_i and equations $t_i = \big(\boldsymbol{x}\mathscr{S}_{8w}^i\big)_0$, $u_i = \big(\boldsymbol{x}\mathscr{S}_{8w}^i\big)_2$ (note that the subscripts depend on the of parts of the state we are adding). Given a target sequence of output values $\langle v_0, v_1, \ldots, v_{d-1} \rangle$, we would like to show that there are $2^{w(8-d)}$ possible values of $\boldsymbol{x}$ such that $t_i = v_i - u_i$. However, to avoid mixing operations in $\mathbf{Z}/2^w\mathbf{Z}$ and $(\mathbf{Z}/2\mathbf{Z})^w$ we first solve

the x_i's and the t_i's with respect to the u_i's. After some standard linear algebra, this gives for $d = 7$ the constraints

$$t_0 = u_0 + u_1$$
$$t_1 = u_1 + u_2$$
$$t_2 = u_2 + u_3$$
$$t_3 = u_3 + u_4$$
$$t_4 = u_4 + u_5$$
$$t_5 = u_5 + u_6$$

while the x_i's are expressible in terms of the t_i's and the u_i's. Choosing a value for u_0 (and thus $t_0 = v_0 - u_0$) gives by the first equation $u_1 = t_0 + u_0$ and thus $t_1 = v_1 - u_1$, by the second equation $u_2 = t_1 + u_1$ and $t_2 = v_2 - u_2$, and so on. Thus, each sequence $\langle v_0, v_1, \ldots, v_6 \rangle$ can appear at most 2^w times, which implies that all such sequences appear exactly 2^w times. $\square$

Note that it is impossible to claim 8-dimensional equidistribution: the `xoshiro+` generator associated with the only full period 5-bit `xoshiro` generator with 40 bits of state ($a = 2, b = 3$) is not 8-dimensionally equidistributed.

An immediate consequence of the last two propositions is that every individual bit of our generators has full period:

PROPOSITION 8.4. *Every bit of a generator in Table 1 and 4 with n bits of state has period $2^n - 1$.*

PROOF. By the results in this section all such generators are at least 1-dimensionally equidistributed, and we just have to apply Proposition 7.1 from [42]. $\square$

9 ESCAPING ZEROLAND

We show in Figure 8 the speed at which the generators hitherto examined "escape from zeroland" [34]: linear engines need some time to get from an initial state with a small number of ones to a state in which the ones are approximately half,[22] and while scrambling reduces this phenomenon, it is nonetheless detectable. The figure shows a measure of escape time given by the ratio of ones in a window of 4 consecutive 64-bit values sliding over the first 1000 generated values, averaged over all possible seeds with exactly one bit set (see [34] for a detailed description). Table 12 condenses Figure 8 into the mean and standard deviation of the displayed values. Comparing these values with those reported in [43], one can observe that `xoroshiro` escapes a little faster than `xorshift`.

10 A THEORETICAL ANALYSIS OF SCRAMBLERS

We conclude the paper by discussing our scramblers from a theoretical point of view. We cast our discussion in the same theoretical framework as that of *filtered linear-feedback shift registers*. A filtered LFSR is given by an underlying LFSR, and by a *Boolean function* that is applied to the state of the register. The final output is the output of the Boolean function. Since shift registers update one bit at a time, we can see the Boolean function as sliding on the sequence of bits generated by the LFSR, emitting a scrambled output. The purpose of filtering a LFSR is that of making it more difficult to guess its next bit: the analogy with linear engines and scramblers is evident, as every

---

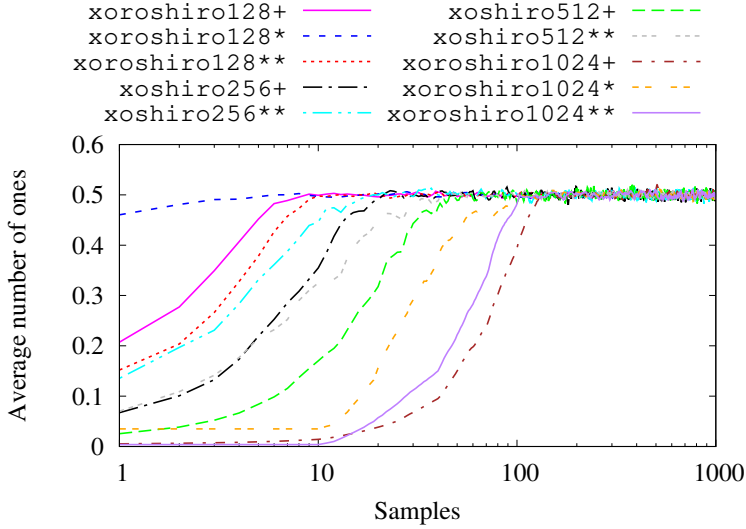[22]Famously, the Mersenne Twister requires about a hundred thousand iterations.

Fig. 8. Convergence to "half of the bits are ones in average" plot.

Table 12. Mean and standard deviation for the data shown in Figure 8.

Generator	Mean	Standard deviation
xoroshiro128+	0.498701	0.017392
xoroshiro128*	0.499723	0.003958
xoroshiro128**	0.498389	0.021876
xoshiro256+	0.495908	0.033354
xoshiro256**	0.497779	0.024809
xoshiro512+	0.491710	0.051387
xoshiro512**	0.495308	0.035350
xoroshiro1024+	0.464861	0.108212
xoroshiro1024*	0.483116	0.070485
xoroshiro1024**	0.471698	0.096509

scrambler can be seen as set of w boolean functions applied to the state of the linear engine. There are however a few differences:

- we use only primitive polynomials;
- we use *several* Boolean functions, and we are concerned with the behavior of their combined outputs;
- we do not apply a Boolean function to a sliding window of the same LFSR: rather, we have kw copies of the same LFSR whose state is different, and we apply our set of Boolean functions to their single-bit output concatenated;
- we are not free to design our favorite Boolean functions: we are restricted to the ones computable with few arithmetic and logical operations;

- we are not concerned with predictability in the cryptographic sense, but just in the elimination of linear artifacts, that is, failures in tests for binary rank, linear complexity and Hamming-weight dependencies.

Nonetheless, we will see that many basic techniques coming from the cryptographic analysis of filtered LFSR can be put to good use in our case. To make our discussion more concrete, we will bring along a very simple example: a `xorshift` generator with $w = 3$ and 6 bits of state [42]. Its parameters are $a = 1$, $b = 2$, $c = 1$ and its characteristic polynomial is $p(x) = x^6 + x^5 + x^3 + x^2 + 1$.

10.1 Representation by generating functions

We know already that all bits of a linear engine are LFSRs with the same characteristic polynomial, but we can be more precise: we can fix a nonzero initial state and compute easily for each bit the *generating function* associated with the bit output (see [15] for a detailed algorithm). Such functions have at the denominator the *reciprocal polynomial* $x^n p(1/x^n)$ (n here is the degree of p), whereas the numerator (a polynomial of degree less then n) represents the initial state. In our example, representing the first word using x_0 (lowest bit), x_1, x_2 and the second word using y_0, y_1 and y_2, we have

$$
F_{x_0}(z) = \frac{z^5 + z^2 + z + 1}{z^6 + z^4 + z^3 + z + 1} \qquad\qquad F_{y_0}(z) = \frac{z^5 + z^4 + z^3 + z^2 + z}{z^6 + z^4 + z^3 + z + 1}
$$

$$
F_{x_1}(z) = \frac{z^5 + z^2}{z^6 + z^4 + z^3 + z + 1} \qquad\qquad F_{y_1}(z) = \frac{z^4 + z}{z^6 + z^4 + z^3 + z + 1}
$$

$$
F_{x_2}(z) = \frac{z^5 + z^4}{z^6 + z^4 + z^3 + z + 1} \qquad\qquad F_{y_2}(z) = \frac{z^4 + z^3}{z^6 + z^4 + z^3 + z + 1}
$$

If we write the formal series associated with each function, the i-th coefficient will give exactly the i-th output of the corresponding bit of the generator. In particular, the first output for x_0 is one, whereas for all other bits it is zero.

The interest in the representation by generating function lies in the fact that now we can perform some operations on the bits. For example, if we want to study the associated `xorshift+` generator, the lowest bit will be just $x_0 + y_0$, and we can actually easily compute the associated function, as adding coefficients is the same as adding functions:

$$
\begin{aligned}
F_{x_0 + y_0}(z) &= F_{x_0}(z) + F_{y_0}(z) \\
&= \frac{z^5 + z^2 + z + 1}{z^6 + z^4 + z^3 + z + 1} + \frac{z^5 + z^4 + z^3 + z^2 + z}{z^6 + z^4 + z^3 + z + 1} = \frac{z^4 + z^3 + 1}{z^6 + z^4 + z^3 + z + 1}.
\end{aligned}
$$

However, we are now stuck, because to compute the functions associated with the `xorshift+` generator we need more than just xors. Indeed, it is very well known that if we consider sequences $x_0, x_1, \ldots, x_{w-1}$ and $y_0, y_1, \ldots, y_{w-1}$ representing the bits, from least significant to most significant, of two w-bit words, and we want to represent their arithmetic sum over $\mathbf{Z}/2^w\mathbf{Z}$, we can define the result bits s_i and the carry bits c_i using the recurrence

$$
s_i = x_i + y_i + c_{i-1} \tag{8}
$$

$$
c_i = (x_i + y_i)c_{i-1} + x_i y_i \tag{9}
$$

where $c_{-1} = 0$. This recurrence is fundamental because carries are the *only* source of nonlinearity in our scramblers (even multiplication by a constant can be turned into a series of shifts and sums). The expression for c_i is simply the standard definition of carry expressed using the base of xor/and (sum/product) that is typical of the arithmetic in $\mathbf{Z}/2\mathbf{Z}$. It is clear that to continue to the higher bits

we need to be able to *multiply* two sequences, but multiplying generating functions unfortunately corresponds to a *convolution* of coefficients.

10.2   Representation in the splitting field

We now start to use the fact that the characteristic polynomial $p(x)$ of our generator is primitive. Let $\mathbf{E}$ be the *splitting field* a polynomial $p(x)$ of degree n over $\mathbf{Z}/2\mathbf{Z}$ [23]. In particular, $\mathbf{E}$ can be represented by polynomials in α computed modulo $p(\alpha)$, and in that case by primitivity the zeroes of $p(x)$ are exactly the powers

$$\alpha, \alpha^2, \alpha^4, \alpha^8, \ldots, \alpha^{2^{n-1}},$$

that is, the powers having exponents in the *cyclotomic coset* $C = \left\{ 1, 2, 4, 8, \ldots, 2^{n-1} \right\}$. Note that $\alpha^{2^n} = \alpha$. Every rational function $f(z)$ representing a bit of the generator can then be expressed as a sum of partial fractions

$$f(z) = \sum_{c \in C} \frac{\beta_c}{1 - z\alpha^c}, \tag{10}$$

where $\beta_i \in \mathbf{E}$, $\beta_c \neq 0$. As a consequence [15], the j-th bit b_j of the sequence associated with $f(z)$ has an explicit description:

$$b_j = \sum_{c \in C} \beta_c \left(\alpha^c \right)^j. \tag{11}$$

This property makes it possible to compute easily in closed form the sum of two sequences and the (output-by-output) product of two sequences. We just need to compute the sum or the product of the representation (11). In the case of the sum of two sequences it is just a term-by-term sum, whereas in the case of a product we obtain a convolution.[23]

In both cases, we might experience *cancellation*—some of the β's might become zero. But, whichever operation we apply, we will obtain in the end for a suitable set $S \subseteq [2^n]$ a representation of the form

$$\sum_{c \in S} \beta_c \alpha^c. \tag{12}$$

with $\beta_c \neq 0$. The cardinality of S is now exactly the degree of the polynomial at the denominator of the rational function

$$g(z) = \sum_{c \in S} \frac{\beta_c}{1 - z\alpha^c}$$

associated with the new sequence, that is, its *linear complexity* [15].

In our example, the coefficients of the representation (11) of x_0 are

$$\beta_1 = \alpha^4 + \alpha^3$$
$$\beta_2 = \alpha^5 + \alpha^3 + \alpha^2 + \alpha$$
$$\beta_4 = \alpha^5 + \alpha^4 + \alpha^2 + 1$$
$$\beta_8 = \alpha^4 + \alpha^3 + \alpha^2 + \alpha$$
$$\beta_{16} = \alpha^5 + \alpha^4 + \alpha^3 + \alpha$$
$$\beta_{32} = \alpha^5 + \alpha^2 + a$$

and similar descriptions are available for the other bits, so we are finally in the position of computing exactly the values of the recurrence (8): we simply have to use the representation in the splitting field to obtain a representation of s_i, and then revert to functional form using (10).

---

[23]We used the Sage algebra system [41] to perform our computations

$$F_{s_0}(z) = \frac{z^4 + z^3 + 1}{z^6 + z^4 + z^3 + z + 1}$$

$$F_{s_1}(z) = \frac{z^{13} + z^9 + z^8 + z}{z^{15} + z^{14} + z^{11} + z^7 + z^4 + z^3 + 1}$$

$$F_{s_2}(z) = \frac{z^{37} + z^{35} + z^{32} + z^{30} + z^{27} + z^{25} + z^{24} + z^{20} + z^{19} + z^{17} + z^{14} + z^{11} + z^8 + z^2}{z^{41} + z^{39} + z^{34} + z^{32} + z^{30} + z^{28} + z^{27} + z^{26} + z^{24} + z^{23} + z^{17} + z^{16} + z^{15} + z^{14} + z^{13} + z^{11} + z^9 + z^8 + z^7 + z^6 + z^5 + z^3 + 1}$$

Fig. 9. The generating functions of the three bits of the xorshift+ generator.

The result is shown in Figure 9: as it is easy to see, we can still express the bits of xorshift+ as LFSRs, but their linear complexity rises quickly (remember that every generator with n bits of state is a linear generator of degree 2^n with characteristic polynomial $x^{2^n} + 1$, so "linear" should always mean "linear of low degree").

In fact, the generating function is irrelevant for our purposes: the only relevant fact is that the representation in the splitting field of the first bit has 6 coefficients, that of the second bit 15 and that of the third bit 41, because, as we have already observed, these numbers are equal to the linear complexity of the bits.

Unfortunately, this approach can be applied only to state arrays of less than a dozen bits: as the linear complexity increases due to the influence of carries, the number of terms in the representation (12) grows quickly, up to being unmanageable. Thus, this approach is limited to the analysis of small examples or to the construction of counterexamples.

10.3 Representing scramblers by polynomials

A less exact but more practical approach to the analysis of the scrambled output of a generator is that of studying the scrambler in isolation. To do so, we are going to follow the practice of the theory of filtered LFSRs: we will represent the scramblers as a sum of *Zhegalkin polynomials*, that is, *squarefree polynomials* over $\mathbf{Z}/2\mathbf{Z}$. Due to the peculiarity of the field, no coefficients or exponents are necessary. If we are able to describe the function as a sum of *distinct* polynomials, we will say that the function is in *algebraic normal form* (ANF). For example, the 3-bit scrambler of our example generator can be described by expanding recurrence (8) into the following three functions in ANF:

$$S_0(x_0, x_1, x_2, y_0, y_1, y_2) = x_0 + y_0$$
$$S_1(x_0, x_1, x_2, y_0, y_1, y_2) = x_1 + y_1 + x_0 y_0$$
$$S_2(x_0, x_1, x_2, y_0, y_1, y_2) = x_2 + y_2 + x_1 y_1 + x_0 y_0 x_1 + x_0 y_0 y_1$$

There is indeed a connection between the *polynomial degree* of a Boolean function, that is, the maximum degree of a polynomial in its ANF, the linear complexity of the bits of a linear engine and the linear complexity of the bit returned by the Boolean function applied to the engine state.

LEMMA 10.1. *Let* $\mathbf{E}$ *be the splitting field of a primitive polynomial* $p(x)$ *of degree* n, *represented by polynomials in* α *computed modulo* $p(\alpha)$. *Then, there is a tuple* $\langle t_0, t_1, \ldots t_{k-1} \rangle \in [n]^k$ *such that*

$$\prod_{i \in [k]} \alpha^{2^{t_i}} = \alpha^c$$

iff there is an $S \subseteq [n]$ *with* $0 < |S| \le k$ *and*

$$c = \sum_{s \in S} 2^s.$$

PROOF. First we show that the all c's are of the form above. When all the t_i's are distinct, we have trivially $S = \{ t_i \mid 0 \le i < k \}$. If $t_i = t_j$

$$\alpha^{2^{t_i}} \alpha^{2^{t_j}} = \alpha^{2 \cdot 2^{t_i}} = \alpha^{2^{t_i+1}},$$

remembering that computations of exponents of α are to be made modulo $2^n - 1$. Thus, the problem is now reduced to a smaller tuple, and we can argue by induction that the result will be true for some $S \subseteq [k-1] \subseteq [k]$.

Now we show that for every S as in the statement there exists a corresponding tuple. If $|S| = k$, this is obvious. Otherwise, let $|S| = j$ and $s_0, s_1, \ldots, s_{j-1}$ be an enumeration of the elements of S. Then, the k-tuple

$$s_0, s_1, \ldots, s_{j-2}, s_{j-1} - 1, s_{j-1} - 2, \ldots, s_{j-1} - k + j + 1, s_{j-1} - k + j, s_{j-1} - k + j,$$

where the operations above are modulo $2^n - 1$, gives rise exactly to the set S, as

$$\alpha^{2^{s_{j-1}-1}} \alpha^{2^{s_{j-1}-2}} \cdots \alpha^{2^{s_{j-1}-k+j+1}} \alpha^{2^{s_{j-1}-k+j}} \alpha^{2^{s_{j-1}-k+j}} = \alpha^{2^{s_j}}.$$

$\square$

An immediate consequence of the previous lemma is that there is a bound on the increase of linear complexity that a Boolean function, and thus a scrambler, can induce:

PROPOSITION 10.2. *If f is a Boolean function of n variables with polynomial degree d in ANF and $x_i, 0 \le i < n$, are the bits of a linear engine with n bits of state, then the rational function representing $f(x_0, x_1, \ldots, x_{n-1})$ has linear complexity at most*

$$U(n, d) = \sum_{j=1}^{d} \binom{n}{j}. \tag{13}$$

The result is obvious, as the number of possible nonzero coefficients of the splitting-field representation of $f(x_0, x_1 \ldots, x_{n-1})$ are bounded by $U(n, d)$ by Lemma 10.1. Indeed, $U(n, d)$ is very well known in the theory of filtered LFSR: it is the standard bound on the linear complexity of a filtered LFSR. Our case is slightly different, as we are applying Boolean functions to bits coming from different instances of the same LFSR, but the mathematics goes along essentially in the same way.

Getting back to our example, the bounds for linear complexity of the bits of our xorshift+ generator are respectively $\binom{6}{1} = 6$, $\binom{6}{1} + \binom{6}{2} = 21$ and $\binom{6}{1} + \binom{6}{2} + \binom{6}{3} = 41$. Thus, from Figure 9 we see that the less significant and the most significant bit attain the upper bound (13), whereas the intermediate bit does not. However, Lemma 10.1 implies that in principle *every* subset of S might be associated with a nonzero coefficient. If this does not happen, as in the case of the intermediate bit, it must be the case that all the contribution for that subset of S canceled out.

The actual amount of cancellation happening for a specific combination of linear engine and scrambler can in principle be computed exactly using the splitting-field representation, but as we have discussed this approach does not lend itself to computations beyond very small generators. However, we gather some empirical evidence by computing the polynomial degree of the Boolean function associated with a bit using (8) and then by measuring directly the linear complexity using the Berlekamp–Massey algorithm [15]: with a careful implementation, this technique can be applied much beyond where the splitting-field representation can get. The algorithm needs an upper bound on the linear complexity to return a reliable result, but we have (13), so this is not a problem.

We ran extensive tests on a number of generators, the largest being 12-bit generators with 24 bits of state. The results are quite uniform: unless the state array of the linear engine is tiny, *cancellation is an extremely rare event*. Table 13 reports the measured linear complexity of a 12-bit xorshift

generator with 24 bits of state after applying a + scrambler. As a comparison, we also show the large loss with respect to the upper bound (13) in the case of a generator based on an irreducible, but not primitive polynomial. Similar figures are reported in Table 14 for the ** scrambler.

These empirical results suggest that it is a good idea to independently study scramblers as Boolean functions, and in particular estimating or computing their polynomial degree. Then, given a class of generator, one should gather some empirical evidence that cancellation is rare, and at that point use the upper bound (13) as an estimate of linear complexity.

We remark however that a simply high polynomial degree is not sufficient to guarantee to pass all tests related to linearity. The problem is that such tests depends on the joint output of the Boolean functions we are considering. Moreover, there is a great difference between having high polynomial degree and passing a linear-complexity or binary-rank test.

For example, consider the following pathological Boolean function that could be part of a scrambler:

$$x_{w-1} + \prod_{i \in [w-1]} x_i. \tag{14}$$

Clearly, this function has very high polynomial degree, and thus a likely high linear complexity. The problem is that if, say, $w = 64$ from a practical viewpoint it is indistinguishable from x_{w-1}, as the "correction" that raises its linear complexity almost never happens. If the state array is small, this bit will fail all linearity tests. A single high-degree monomial is not sufficient in isolation, in spite of Lemma 10.1.

As a last counterexample, and cautionary tale, we consider the scrambler given by change of sign, that is, multiplication by the all-ones word. It is trivial to write this scrambler using negated variables, but when we expand it in ANF we get

$$\bar{x}_{w-1} + \prod_{k \in [w-1]} \bar{x}_k = 1 + x_{w-1} + \prod_{k \in [w-1]} \left(1 + x_k\right) = 1 + x_{w-1} + \prod_{S \subseteq [w-1]} \prod_{k \in S} x_k. \tag{15}$$

In other words, the ANF contains *all* monomials formed with all other bits, but the Boolean function is still as pathological as (14), as there is no technical difference between x_i and $\bar{x}_i$.

10.4   The + scrambler

We conclude this part of the paper with a detailed discussion of each scrambler, starting from +, introduced in Section 4.1. Recurrence (8) can be easily unfolded to a closed form for the scrambled bit s_b:

$$s_b = x_b + y_b + \sum_{i=1}^{b} x_{i-1} y_{i-1} \sum_{S \subseteq [b-i]} \prod_{j \in S} x_{i+j} \prod_{j \in [b-i] \setminus S} y_{i+j}$$

$$= x_b + y_b + \sum_{i=1}^{b} x_{i-1} y_{i-1} \prod_{j \in [b-i]} \left(x_{i+j} + y_{i+j}\right) \tag{16}$$

If the x_i's and the y_i's are distinct, the expressions above are in ANF: there are exactly $2^b + 1$ monomials with maximum degree $b + 1$. Thus, if the underlying linear engine has n bits of state the linear-degree bound for bit b will be $U(n, b + 1)$.

An important observation is that no monomial appears in two instances of the formula for different values of b. This implies that any linear combination of bits output by the + scrambler have the same linear complexity of the bit of highest degree, and at least as many monomials: we

Polynomial degree	$a = 1, b = 7, c = 5$ (primitive)		$a = 2, b = 1, c = 11$ (nonprimitive)	
	Linear degree	Loss	Linear degree	Loss
1	24	0	24	0
2	300	0	300	0
3	2324	0	2316	8
4	12950	0	12814	136
5	55454	0	53374	2080
6	190050	0	170714	19336
7	536150	4	413812	122342
8	1271625	0	729739	541886
9	2579121	8	927962	1651167
10	4540385	0	980861	3559524
11	7036529	0	986654	6049875
12	9740685	0	986888	8753797

Table 13. Polynomial degree versus linear complexity for two 12-bit xorshift generators with 24 bits of state using the + scrambler. The first polynomial is primitive, whereas the second polynomial is just irreducible. The second column shows the loss in linear complexity with respect to the upper bound (13). In the first case the loss is due to cancellation only, in the second case also to the lack of primitivity.

$5\rho_5(3x)$			$9\rho_7(5x)$			$17\rho_9(9x)$		
Pol. deg.	Lin. deg.	Loss	Pol. deg.	Lin. deg.	Loss	Pol. deg.	Lin. deg.	Loss
7	536154	0	4	12590	0	1	24	0
8	1271625	0	5	55454	0	2	300	0
9	2579129	0	6	190050	0	3	2324	0
10	4540385	0	7	536154	0	4	12590	0
11	7036529	0	8	1271622	3	5	55454	0
11	7036529	0	9	2579126	3	6	190050	0
11	7036525	4	10	4540385	0	7	536154	0
11	7036529	0	11	7036526	0	9	2579125	4
11	7036529	0	11	7036529	0	10	4540377	8
11	7036526	3	11	7036529	0	11	7036529	0
11	7036529	0	11	7036529	0	11	7036526	3
11	7036529	0	11	7036529	0	11	7036529	0

Table 14. Polynomial degree versus linear complexity for a 12-bit xorshift generator with 24 bits of state and parameters $a = 1$, $b = 7$ and $c = 5$ using a few ** scramblers. The second column shows the loss in linear complexity with respect to the upper bound (13) due to cancellation.

xoroshiro128+	xoshiro256+	xoshiro512+	xoroshiro1024+	xoshiro64+	xoshiro128+
128	256	512	1024	64	128
8256	32896	131328	524800	2080	8256
349632	2796416	22370048	178957824	43744	349632
11017632	177589056	2852247168	45723987200	679120	11017632
275584032	8987138112	290367762560	9336909979904	8303632	275584032

Table 15. Estimated linear complexity of the five lowest bit of generators (first line is bit 0) using the + scrambler.

say in this case that there is no *polynomial degree loss*. Thus, with the exception of the very lowest bits, no linearity will be detectable: in Table 15 we report the linear complexity of the lowest bits of some generators; the degree has been estimated using (13). The lowest values have also been verified using the Berlekamp–Massey algorithm: as expected, we could not detect any linear-degree loss. While an accurate linear-complexity test might catch the fourth lowest bit of xoroshiro128+, the degree raises quickly to the point the linearity is undetectable.

The situation for Hamming-weight dependencies is not so good, however, as empirically (Table 1) we have already observed that xoroshiro engines still fail our test (albeit using three order of magnitudes more data). We believe that this is due to the excessively regular structure of the monomials: the simplest successful scrambler against the test is indeed multiplication by a constant, which provides a very irregular monomial structure.

If there is some overlap it is necessary to rewrite (16) replacing suitably x_i and y_i and try to obtain an algebraic normal form, as we will do in the next section.

10.5 The ∗ scrambler

We start by discussing the ∗ scrambler, introduced in Section 4.2, in the case of a multiplicative constant of the form $2^s + 1$. This case is particularly interesting because it is very fast on recent hardware; in particular, $(2^s+1) \cdot x = x + (x \ll s)$, which provides a multiplication-free implementation. Moreover, as we will see, the analysis of the $2^s + 1$ case sheds light on the general case, too.

Let $z = (2^s + 1)x$. Specializing (16), we have that $z_b = x_b$ when $b < s$; otherwise, $b = c + s \geq s$ and

$$z_b = z_{c+s} = x_{c+s} + x_c + \sum_{i=1}^{c} x_{i-1+s}\, x_{i-1} \sum_{S \subseteq [c-i]} \prod_{j \in S} x_{i+j+s} \prod_{j \in [c-i] \setminus S} x_{i+j}$$

$$= x_{c+s} + x_c + \sum_{i=1}^{c} x_{i-1+s}\, x_{i-1} \prod_{k \in [c-i]} \left(x_{i+k+s} + x_{i+k} \right). \quad (17)$$

However, contrarily to (16) the expressions above do not denote an ANF, as the same variable may appear many times in the same monomial.

We note that the monomial $x_s x_0 x_{s+1} \cdots x_{s+c-1}$, which is of degree $c + 1$, appears only and always in the function associated with y_b, $b > s$. Thus, bits with $b \leq s$ have degree one, whereas bits b with $b > s$ have degree $b - s + 1$. In particular, as in the case of +, there is no polynomial degree loss when combining different bits.

In case of a generic (odd) constant m, one has to modify recurrence (8) so as to start including shifted bits at the right stage, which creates a very complex monomial structure. Note, however, that

bits after the second-lowest bit set in m cannot modify the polynomial degree. Thus, the decrease of Hamming-weight dependencies we observe in Table 1 even for xoroshiro* is not due to higher polynomial degree with respect to + (indeed, the opposite is true), but to a richer structure of the monomials. The degree reported for the + scrambler in Table 15 can indeed be adapted to the present case: one has just to copy the first line as many times as the index of the second-lowest bit set in m.

To get some intuition about the monomial structure, it is instructive to get back to the simpler case $m = 2^s + 1$. From (17) it is evident that monomials associated with different values of i cannot be equal, as the minimum variable appearing in a monomial is x_{i-1}. Once we fix i with $1 \le i \le c$, the number of monomials is equal to the number of sets of the form

$$S + s \;\cup\; [c - i] \setminus S \;\cup\; \{s - 1\} \qquad S \subseteq [c - i] \tag{18}$$

that can be expressed by an *odd* number of values of S (if you can express the set in an even number of ways, they cancel out). But such sets are in bijection with the values $(v \ll s) \vee \neg s \vee (1 \ll s - 1)$ as v varies among the words of $c - i$ bits. In a picture, we are looking at the logical or by columns of the following diagram, where the b_j's are the bits of v, for convenience numbered from the most significant:

					$\neg b_0$	$\neg b_1$	$\cdots$	$\neg b_{c-i-s-1}$	1	$\neg b_{c-i-s+1}$	$\cdots$	$\neg b_{c-i-1}$
b_0	b_1	$\cdots$	b_{s-2}	b_{s-1}	b_s	b_{s+1}	$\ldots$	b_{c-i-1}				

A first obvious observation is that if $s > c - i$ the two rows are nonoverlapping, and they are not influenced by the one in position $s - 1$. In this case, we obtain all possible 2^{c-i} monomials. More generally, such sets are all distinct iff $s \ge (c - i + 1)/2$, as in that case the value must differ either in the first s or in the last $s - 1$ bits: consequently, the number of monomials in this case is again 2^{c-i}. Minimizing i and maximizing c we obtain $s \ge (w - s - 1)/2$, whence $s \ge (w - 1)/3$. In this case, the monomials of z_b are exactly $2^{b-s} + 1$ when $b \ge s$.

As s moves down from $(w - 1)/3$, we observe empirically more and more reduction in the number of monomials with respect to the maximum possible 2^{b-s}. When we reach $s = 1$, however, a radical change happens: the number of monomials grows as $2^{b/2}$.

THEOREM 10.3. *The number of monomials of the boolean function representing bit b of $3x$ is*

$$(2 + [b \text{ odd}]) \cdot 2^{\lfloor b/2 \rfloor} - 1.$$

We remark a surprising combinatorial connection: this is the *number of binary palindromes smaller than 2^b*, that is, A052955 in the "On-Line Encyclopedia of Integer Sequences" [14].

PROOF. When $s = 1$, the different subsets in (18) obtained when S varies are in bijection with the values $v \vee \neg(v \gg 1)$ as v varies among the words of $c - i$ bits. Again, we are looking at the logical or by columns of the following diagram, where the b_j's are the bits of v numbered from the most significant:

	$\neg b_0$	$\neg b_1$	$\neg b_2$	$\cdots$	$\neg b_{c-i-2}$
b_0	b_1	b_2	b_3	$\cdots$	b_{c-i-1}

Note that if there is a b_j whose value is irrelevant, flipping will generate two monomials which will cancel each other.

Let us consider now a successive assignment of values to the b_j's, starting from b_0. We remark that as long as we assign ones, no assigned bit is irrelevant. As soon as we assign a zero, however, say to b_j, we have that the value of b_{j+1} will no longer be relevant. To make b_{j+1} relevant, we need to set $b_{j+2} = 0$. The argument continues until the end of the word, so we can actually choose the value of $(c - i - j - 1)/2$ bits, and only if $c - i - j - 1$ is even (otherwise, b_{c-i-1} has no influence).

We now note that if we flip a bit b_k that we were forced to set to zero, there are two possibilities: either we chose $b_{k-1} = 1$, in which case we obtain a different word, or we chose $b_{k-1} = 0$, in which case b_k is irrelevant, but by flipping also b_{k+1} we obtain once again the same word, so the two copies cancel each other.

Said otherwise, words with an odd number of occurrences are generated either when all bits of v are set to one, or when there is a string of ones followed by a suffix of odd length in which every other bit (starting from the first one) is zero. All in all, we have

$$1 + \sum_{k=0}^{\left\lfloor \frac{c-i-1}{2} \right\rfloor} 2^k = 2^{\left\lceil \frac{c-i-1}{2} \right\rceil}$$

possible monomials, where $2k + 1$ is the length of the suffix. Adding up over all i's, and adding the two degree-one monomials we have that the number of monomials of y_b for $b = c + 1 > 0$ is

$$2 + \sum_{i=1}^{b-1} 2^{\left\lceil \frac{b-i-2}{2} \right\rceil} = 1 + 1 + \sum_{i=1}^{b-1} 2^{\left\lceil \frac{b-i-2}{2} \right\rceil} = (2 + [b \text{ odd}]) \cdot 2^{\lfloor b/2 \rfloor} - 1.$$

The correctness for the case $b = 0$ can be checked directly. □

We remark that in empirical tests the $3x$ scrambler performs very badly, which seems to confirm the intuition that the cancellation of monomials is excessive in that case.

10.6 The ⋆⋆ scrambler

We will now examine the strong scrambler ⋆⋆ introduced in Section 4.3. We start by discussing the case with multiplicative constants of the form $2^s + 1$ and $2^t + 1$, which is particularly fast.

Let $z = \rho_r(x \cdot (2^s + 1)) \cdot (2^t + 1)$. Clearly, the $\min\{r, t\}$ lowest bits of z are the $\min\{r, t\}$ highest bits of $x \cdot (2^s + 1)$. To choose s, r and t we can leverage our previous knowledge of the scrambler ⋆. We start by imposing that $s < t$, as choosing $s = t$ generates several duplicates that reduce significantly the number of monomials in the ANF of the final Boolean functions, whereas $t < s$ provably yields a lower minimum degree for the same r (empirical computations show also a smaller number of monomials). We also have to impose $t < r$, for otherwise some bits or xor of pair of bits will have very low linear complexity (polynomial degree one). So we have to choose our parameters with the constraint $s < t < r$. Since the degree of the lowest bit is $\max(1, w - r - s + 1)$, choosing $r = t + 1$ maximizes the minimum degree across the bits. Moreover, we would like to keep s and t as small as possible, to increase the minimum linear complexity and also to make the scrambler faster.

Computing an ANF for the final Boolean functions appears to be a hard combinatorial problem: nonetheless, with this setup we know that the lowest bit will have (when $r + s \le w$) polynomial degree $w - r - s + 1$, and we expect that the following bits will have increasing degree up to saturation. Nonetheless, symbolic computations in low dimension show some polynomial degree loss caused by the second multiplication unless $r = 2t + 1$; moreover, for that value of r the polynomial degree loss when combining bits is almost absent.[24] Taking into consideration the bad behavior of the

---

[24]We remark that empirically the scrambler never goes beyond polynomial degree $w - 1$, the maximum possible being w. The reason is that when expanding the products in recurrence (8) the monomial $x_0 x_1 \cdots x_{w-1}$ appears always an even number of times, and cancels out. Why this happens is an interesting combinatorial open problem.

xoroshiro128**	xoshiro256**	xoshiro512**	xoshiro1024**	xoshiro64**	xoshiro128**
3×10^{37}	2×10^{57}	3×10^{75}	7×10^{91}	2×10^{18}	8×10^{25}
4×10^{37}	7×10^{57}	3×10^{76}	1×10^{93}	3×10^{18}	3×10^{26}
5×10^{37}	2×10^{58}	2×10^{77}	2×10^{94}	5×10^{18}	1×10^{27}
7×10^{37}	9×10^{58}	2×10^{78}	3×10^{95}	6×10^{18}	5×10^{27}

Table 16. Approximate estimated linear complexity of the four lowest bit of generators (first line is bit 0) using the ** scrambler with parameters from Table 3 and 6.

multiplier 3 highlighted by Theorem 10.3, we conclude that the best choice is $s = 2$, $t = 3$ and consequently $r = 7$. These are the parameters reported in Table 3. The linear complexity of the lowest bits is extremely large, as shown in Table 16.

At 32 bits, however, tests show that this scrambler is not sufficiently powerful for xoroshiro64, and Table 6 reports indeed different parameters: the first multiplier is the constant used for the ⋆ scrambler, and the second multiplier $2^t + 1$ has been chosen so that bit t is not set in the first constant. Again, $r = 2t + 1$, following the same heuristic of the previous case.[25]

10.7   The ++ scrambler

We conclude our discussion mentioning a strong scrambler that we ended up not using, but studied quite extensively. We choose two words x, y from the state of the linear engine and then $z = \rho_r(x + y) + x$. This scrambler is slightly faster than the **, and it has similar linearity properties: in fact, it can reach higher polynomial degree, as it uses $2w$ bits.[26] However, the very regular structure of the + scrambler makes ++ much less effective on Hamming-weight dependencies. While some specific combination of engine parameters and values of r can be made to work (i.e., pass all tests), we found no theoretical explanation or clear empirical evidence for a choice of r.

Nonetheless, the scrambler can be interesting in some contexts, like hardware design and small state sizes, as it increases significantly linear complexity in all bits with a very low cost. For example, it has been used by Parallax to embed in their Propeller 2 microcontroller multiple 16-bit xoroshiro32++ generators (parameters $a = 14$, $b = 2$, $c = 7$, $r = 5$). Note that if the underlying linear engine is d-dimensionally equidistributed, the generator will be in general at most $(d − 1)$-dimensionally equidistributed: establishing a precise dimension requires proofs and counterexamples in the style of Section 8.

11   CONCLUSIONS

The combination of xoroshiro/xoshiro and suitable scramblers provides a wide range of high-quality and fast solutions for pseudorandom number generation. Parallax has embedded xoroshiro128** and xoroshiro32++ in their recently designed Propeller 2 microcontroller; xoroshiro116+ is the stock generator of Erlang, and the next version of the popular embedded language Lua will sport xoshiro256** as stock generator.

---

[25]We remark that in this case in theory we would not strictly need to fix the first constant so that it has the second-lowest bit set, but we prefer to use the same constant of the ⋆ case for uniformity. The same consideration is true of the $r = 2t + 1$ heuristics.

[26]Once again, symbolic computation suggests that this scrambler can reach only polynomial degree $2w − 3$; proving this bound is an open problem.

We believe that a more complete study of scramblers can shed some further light on the behavior of such generators: the main open problem is that of devising a model explaining the elimination of Hamming-weight dependencies. While targeting a high polynomial degree (and several low-degree monomials) seem to be sufficient to obtain resilience to binary-rank and linear-complexity test, we have little intuition of what causes an improvement in the results in the test described in Section 6. The main difficulty is that analyzing the Boolean functions representing each scrambled bit in isolation is not sufficient, as Hamming-weight dependencies are generated by their collective behavior.

There are variants of the scramblers we discussed that do not use rotations: for example, in the ** and ++ scrambler the rotation can be replaced by xoring x with $x \gg r$, as also this operation will increase the linear complexity of the lower bits. For contexts in which rotations are not available or too expensive, one might explore the possibility of using xorshift generators scrambled with such variants.

On a more theoretical side, it would be interesting to find a closed form for the number of monomials of the scrambler * with constant $(2^s + 1)$ when $s > 1$, thus extending Theorem 10.3, or an ANF for the Boolean functions of the ** scrambler.

For more complex matrices, word polynomials might be based on more general approaches, such as *quasideterminants* [9]. In some desperate case, one might degenerate to the *Dieudonné determinant* [7], which basically forces commutativity of all blocks by remapping them in the Abelianization of the subring R they generate. In the case of a word polynomial predicting (almost) exactly the generator it might be interesting to tune the parameters of the generator using the figure of merit introduced in [32].

There is of course a vast literature on filtered LFSR that might be used to prove aspects we approached only with symbolic small-state computations in Section 10. For example, in [17] the authors prove a *lower* bound on the linear degree of a Boolean function made of single very specific monomial, something for which we just argued on the basis of measurement made using the Berlekamp–Massey algorithm. In [2] the authors try to provide closed forms or even ANFs when the argument of a Boolean function is multiplied or summed with a constant, which might be a starting point for a closed form for the ** scrambler. Finally, several concepts developed in the context of filtered LFSR, such as *algebraic immunity* and *Bent functions* might prove to be useful in choosing scrambler parameters, or defining new ones.

In general, it is an interesting open problem to correlate explicitly the monomial structure of a Boolean function in ANF with the resilience to linearity tests. Intuitively, recalling (15), one sees that besides large-degree monomials one needs small-degree monomials to make the tests "perceive" the increase in linear complexity at the right time.

REFERENCES

[1] Nicolas Bourbaki. 1989. *Algebra: Elements of Mathematics.* Springer-Verlag.

[2] An Braeken and Igor A. Semaev. 2005. The ANF of the Composition of Addition and Multiplication mod 2^n with a Boolean Function. In *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers (Lecture Notes in Computer Science)*, Henri Gilbert and Helena Handschuh (Eds.), Vol. 3557.

Springer, 112–125.

[3]   Sergio Caracciolo, Alan D. Sokal, and Andrea Sportiello. 2009. Noncommutative determinants, Cauchy–Binet for-
      mulae, and Capelli-type identities I. Generalizations of the Capelli and Turnbull identities. *The Electronic Journal of
      Combinatorics* 16, 1 (2009), 103.
[4]   G. D. Carter. 1989. *Aspects of local linear complexity*. Ph.D. Dissertation. University of London.
[5]   A. Chervov, G. Falqui, and V. Rubtsov. 2009. Algebraic properties of Manin matrices 1. *Advances in Applied Mathematics*
      43, 3 (2009), 239–315.
[6]   Aaldert Compagner. 1991. The hierarchy of correlations in random binary sequences. *Journal of Statistical Physics* 63,
      5-6 (1991), 883–896.
[7]   Jean Dieudonné. 1943. Les déterminants sur un corps non commutatif. *Bull. Soc. Math. France* 71, 171-180 (1943), 95.
[8]   E. D. Erdmann. 1992. Empirical tests of binary keystreams. (1992).
[9]   Israel M Gel'fand and Vladimir S Retakh. 1991. Determinants of matrices over noncommutative rings. *Functional
      Analysis and Its Applications* 25, 2 (1991), 91–102.
[10] Hiroshi Haramoto, Makoto Matsumoto, Takuji Nishimura, François Panneton, and Pierre L'Ecuyer. 2008. Efficient
      Jump Ahead for $\mathbb{F}_2$-Linear Random Number Generators. *INFORMS Journal on Computing* 20, 3 (2008), 385–390.
[11] Shin Harase. 2014. On the $\mathbb{F}_2$-linear relations of Mersenne Twister pseudorandom number generators. *Mathematics
      and Computers in Simulation* 100 (2014), 103–113.
[12] Jeffrey J. Hunter. 1983. *Mathematical Techniques of Applied Probability, Volume 2, Discrete-Time Models: Techniques and
      Applications*. Academic Press, New York.
[13] Jeffrey J. Hunter. 2008. Variances of first passage times in a Markov chain with applications to mixing times. *Linear
      Algebra Appl.* 429, 5 (2008), 1135–1162.
[14] OEIS Foundation Inc. 2017. The On-Line Encyclopedia of Integer Sequences. (2017). http://oeis.org/
[15] Andreas Klein. 2013. *Stream Ciphers*. Springer London, London.
[16] Donald E. Knuth. 1973. *The Art of Computer Programming*. Addison–Wesley.
[17] Nicholas Kolokotronis, Konstantinos Limniotis, and Nicholas Kalouptsidis. 2007. Improved Bounds on the Linear
      Complexity of Keystreams Obtained by Filter Generators. In *Inscrypt (Lecture Notes in Computer Science)*, Dingyi Pei,
      Moti Yung, Dongdai Lin, and Chuankun Wu (Eds.), Vol. 4990. Springer, 246–255.
[18] Pierre L'Ecuyer. 1996. Maximally Equidistributed Combined Tausworthe Generators. *Math. Comp.* 64, 213 (1996),
      203–213.
[19] Pierre L'Ecuyer and Jacinthe Granger-Piché. 2003. Combined generators with components from different families.
      Mathematics and Computers in Simulation 62, 3 (2003), 395–404. 3rd IMACS Seminar on Monte Carlo Methods.
[20] Pierre L'Ecuyer and Richard Simard. 1999. Beware of linear congruential generators with multipliers of the form
      $a = \pm 2^q \pm 2^r$. *ACM Trans. Math. Softw.* 25, 3 (1999), 367–374.
[21] Pierre L'Ecuyer and Richard Simard. 2007. TestU01: A C library for empirical testing of random number generators.
      ACM Trans. Math. Softw. 33, 4, Article 22 (2007).
[22] Robert H. Lewis. 2018. Fermat: A Computer Algebra System for Polynomial and Matrix Computation. (2018).
      http://home.bway.net/lewis/
[23] Rudolf Lidl and Harald Niederreiter. 1994. *Introduction to finite fields and their applications*. Cambridge University
      Press, Cambridge.
[24] Yuri Ivanovitch Manin. 1988. *Quantum groups and non-commutative geometry*. Centre de Recherches mathématiques,
      Université de Montréal, Montréal QC, Canada.
[25] George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14 (2003), 1–6.
[26] George Marsaglia and Liang-Huei Tsay. 1985. Matrices and the structure of random number sequences. *Linear Algebra
      Appl.* 67 (1985), 147–156.
[27] Makoto Matsumoto and Takuji Nishimura. 1998a. Dynamic creation of pseudorandom number generators. *Monte
      Carlo and Quasi-Monte Carlo Methods* 2000 (1998), 56–69.
[28] Makoto Matsumoto and Takuji Nishimura. 1998b. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform
      Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (1998), 3–30.
[29] Makoto Matsumoto and Takuji Nishimura. 2002. *A Nonempirical Test on the Weight of Pseudorandom Number Generators*.
      Springer Berlin Heidelberg, Berlin, Heidelberg, 381–395.
[30] Makoto Matsumoto, Isaku Wada, Ai Kuramoto, and Hyo Ashihara. 2007. Common Defects in Initialization of
      Pseudorandom Number Generators. *ACM Trans. Model. Comput. Simul.* 17, 4 (2007).
[31] Carl D. Jr. Meyer. 1975. The Role of the Group Generalized Inverse in the Theory of Finite Markov Chains. *SIAM Rev.*
      17, 3 (1975), 443–464.
[32] Harald Niederreiter. 1992. *Random number generation and quasi-Monte Carlo methods*. CBMS-NSF regional conference
      series in Appl. Math., Vol. 63. SIAM.
[33] Harald Niederreiter. 1995. The multiple-recursive matrix method for pseudorandom number generation. *Finite Fields*

and their Applications 1, 1 (1995), 3–30.

[34] 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.

[35] 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*. Springer, 607–622.

[36] Mutsuo Saito and Makoto Matsumoto. 2009. A PRNG Specialized in Double Precision Floating Point Numbers Using an Affine Transition. In *Monte Carlo and Quasi-Monte Carlo Methods 2008*, Pierre L'Ecuyer and Art B. Owen (Eds.). Springer Berlin Heidelberg, 589–602. DOI : http://dx.doi.org/10.1007/978-3-642-04107-5_38

[37] 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/

[38] Mutsuo Saito and Makoto Matsumoto. 2015. Tiny Mersenne Twister (Version 1.1). (24 March 2015). http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/

[39] Nat Sothanaphan. 2017. Determinants of block matrices with noncommuting blocks. *Linear Algebra Appl.* 512, Supplement C (2017), 202–218.

[40] Guy L. Steele, Jr., Doug Lea, and Christine H. Flood. 2014. Fast Splittable Pseudorandom Number Generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 453–472.

[41] The Sage Developers. 2018. *SageMath, the Sage Mathematics Software System (Version 8.0)*. http://www.sagemath.org

[42] Sebastiano Vigna. 2016a. An experimental exploration of Marsaglia's xorshift generators, scrambled. *ACM Trans. Math. Software* 42, 4 (2016). Article No. 30.

[43] Sebastiano Vigna. 2016b. Further scramblings of Marsaglia's xorshift generators. *J. Comput. Appl. Math.* 315 (2016), 175–181.