A New Test for Hamming–Weight Dependencies

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

We describe a new statistical test for pseudorandom number generators (PRNGs). Our test can find bias induced by dependencies among the Hamming weights of the outputs of a PRNG, even for PRNGs that pass state-of-the-art tests of the same kind from the literature, and in particular for generators based on $\mathbf{F}_2$-linear transformations such as the dSFMT [22], `xoroshiro1024+` [1], and WELL512 [19].

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

Additional Key Words and Phrases: Pseudorandom number generators

1 INTRODUCTION

Pseudorandom number generators (PRNGs) are algorithms that generate a seemingly random output using a deterministic algorithm. A w-bit PRNG is defined by a *state space* S, a *transition* (or *next-state*) computable function $\tau : S \longrightarrow S$, and a computable *output function* $\varphi : S \longrightarrow \{\,0, 1\,\}^w$ that maps the state space into w-bit words. One then considers an *initial state*, or *seed* $\sigma \in S$, and computes the sequence of w-bit outputs

$$\varphi(\sigma), \varphi(\tau(\sigma)), \varphi\bigl(\tau^2(\sigma)\bigr), \varphi\bigl(\tau^3(\sigma)\bigr), \ldots$$

The outputs can be used to generate reals in the unit interval, for example multiplying them by 2^{-w}. Knuth discusses PRNGs at length [8].[1]

A classic example is given by *multiplicative congruential generators*, which are defined by a prime *modulus* μ and a *multiplier* α. Then $S = \mathbf{Z}/\mu\mathbf{Z}$, $\tau : x \mapsto \alpha x$, and the output function is given by the binary representation of x (one tries to choose μ close to 2^w). Another well-known example is the class of $\mathbf{F}_2$*-linear generators* [10], in which S is a vector of $\mathbf{F}_2^n$ (i.e., n bits) and τ is an $\mathbf{F}_2$-linear transformation on S; however, usually, the transformation can be expressed by basic $\mathbf{F}_2$-linear operations on words, such as rotations, shift, and XORs, rather than in matrix form. The output function might pick a word of w bits from the state: for example, $n = kw$ for some k then the state can be represented by k w-bit words, and the output function can just choose one of those. In some generators, moreover, the output function is not $\mathbf{F}_2$-linear.

Several theoretical properties help in the design of PRNGs: however, once designed a PRNG is submitted to a set of *statistical tests*, which try to discover some statistical bias in the output of the generator. The tests compute, using the output of the generator, statistics whose distribution is known (at least approximately) under the assumption that the output of the generator is random. Then, by applying the (complementary) cumulative distribution function to the statistics one obtains a p-value, which should be neither too close to zero nor too close to one (see Knuth [8] for a complete introduction to the statistical testing of PRNGs).

The *Hamming weight* of a w-bit word x is the number of ones in its binary representation. Tests for Hamming-weight dependencies try to discover some statistical bias in the Hamming weight of the output of the generator. In particular, such tests do not depend on the numerical values of

[1] We are here slightly simplifying the presentation: in general, the codomain of the output function can be an arbitrary finite set; moreover, depending on the detailed definition, the output sequence might start with $\varphi(\tau(\sigma))$.

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

the outputs: 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).

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 their Hamming weights has the correct distribution (which will be quickly approximated very well by a normal distribution as m grows [5]). Tests may also 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 [11]. Matsumoto and Nishimura [17] have introduced a theoretical figure of merit that can predict after how many samples a $\mathbf{F}_2$-linear generator will fail a specific Hamming-weight test. The NIST statistical test suite for PRNGs [20] contains tests based on Hamming-weight dependencies, too.

In this paper, we introduce a new test for Hamming-weight dependencies that improves significantly over the state of the art. We find bias in some old and some new generators for which tests of this type from TestU01 [12], a well-known framework for statistical testing of PRNGs, were unable to find bias even using a large amount of output.

All the code used in this paper is available under the GNU General Public License.[2] Code for reproducing the results of this paper has been permanently stored on the Zenodo platform.[3]

2 MOTIVATION

It is known since the early days of $\mathbf{F}_2$-linear generators that sparse transition matrices induce some form of dependency on the Hamming weight of the state. Since the output is computed starting from the state, these dependencies might induce Hamming-weight dependencies in the output, too. For example, if the state has very low Hamming weight, that is, very few ones, one might need a few iterations (or more than a million, in the case of the Mersenne Twister with 19937 bits of state [16]) before the state contains ones and zeroes approximately in the same amount. This is however a minor problem because for generators with, say, at least 128 bits of state, the probability of passing through such states is negligible.

However, what we witness very clearly in the case of almost-all-zeros state might be true in general: states with few ones might lead to states with few ones, or due to XOR operations states with many ones might lead to states with few ones. This kind of dependency is more difficult to detect.

Here we consider as motivation a few generators: `xorshift128+` [25] is the stock generator of most Javascript implementations in common browsers; the SFMT (SIMD-Friendly Mersenne Twister) [21] is a recent improvement on the classic Mersenne Twister using SIMD instructions, and we will use the version with 607 bits of state; the dSFMT [22] is another version of the Mersenne Twister which natively generate doubles, and we will use the version with 521 bits of state; WELL is a family of generators with excellent equidistribution properties [19], and we will use the version with 512 bits of state; finally, we consider a new $\mathbf{F}_2$-linear transformation, `xoroshiro`, designed by the authors, and the associated generators `xoroshiro128+` and `xoroshiro1024+` [1].[4] All these generators have quite sparse transition matrices (WELL512 having the densest matrix), and one would expect some kind of Hamming-weight dependency to appear.

To check whether this is true, we can test for such dependencies using TestU01 [12], a well-known framework for testing generators, which implements tests related to Hamming weights from [11] and [20] (please refer to the TestU01 guide for a detailed description of the tests). Table 1 shows

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

[3] https://zenodo.org/badge/latestdoi/412112034

[4] The generators combine two words of state using a sum in $\mathbf{Z}/2^{64}\mathbf{Z}$.

Table 1. Parameters for statistical tests related to Hamming weights from TestU01 [12]. We consider the entire output of the generator (i.e., TestU01 parameters $r = 0$, $s = 32$). The parameter d of HammingIndep has been set to zero. The parameter k varies among 30, 300, and 1200 for HI and 30, 300, 500 for HC, as in BigCrush. Moreover, in both cases we tested k equal to 128, 256, 512, and 1024 following a suggestion from a referee.

Label	Test parameters
HW0	HammingWeight2 ($N = 1$, $L = 10^6$)
HW1	HammingWeight2 ($N = 1$, $L = 10^7$)
HW2	HammingWeight2 ($N = 1$, $L = 10^8$)
HIk	HammingIndep ($N = 1$, $L = k$)
HCk	HammingCorr ($N = 1$, $L = k$)

Table 2. Results for the TestU01 [12] statistical tests related to Hamming weights. The n parameter gives the number of 32-bit outputs examined. The tests have been run on the lower 32 bits of the output ("L"), the upper 32 bits ("U"), and interleaving the upper and lower bits ("I"). We report the first test failed by a generator, where failure is a p-value outside of the range $[0.01 \mathrel{..} 0.99]$.

n	$n = 10^9$	$n = 10^{10}$	$n = 10^{11}$	$n = 10^{12}$	$n = 10^{13}$
SFMT (607 bits)	HI512 (I)				
`xorshift128+`	—	HI128 (U)			
dSFMT (521 bits)	—	—	—	—	—
WELL512	—	—	—	—	—
`xoroshiro128+`	—	—	—	—	HI128 (I)
`xoroshiro1024+`	—	—	—	—	—

the basic parameters of the nine tests we performed. The parameters were inspired by the author choices in the BigCrush test suite [12], but instead of analyzing 10^9 or fewer outputs, as happens in BigCrush, we analyze up to 10^{13} 32-bit values (e.g., 40 TB of data), hoping that examining a much larger amount of data might help in finding bias in the generators. Besides the parameter inspired by BigCrush, following a suggestion from a referee we also tried a number of power-of-two values for the L parameter of HammingIndep and HammingCorr.

Some of the generators have a 64-bit output, but TestU01 expects generators to return 32-bit integers, so we tested both the lower 32 bits, the upper 32 bits, and the upper and lower bits interleaved. (We do not discard any bits, as it is possible in TestU01.) In the case of the dSFMT, there is a specific call to generate a 32-bit value.

The disappointing results are reported in Table 2: despite the very sparse nature of the transition matrices of these generators, and the very large amount of data, only problems with the SFMT (already using 10^9 values), `xorshift128+` (using 10^{10} values), and `xoroshiro128+` (using 10^{13} values) are reported.[5] All other p-values at 10^{13} are within the range $[0.01 \mathrel{..} 0.99]$.[6]

[5] The latter failure emerged only after testing with additional values of the parameter L suggested by one of the referees.

[6] In two cases we found p-values slightly outside this range, but a test using 1.2×10^{13} values showed that they were statistical flukes.

3 TESTING HAMMING-WEIGHT DEPENDENCIES

In this section, we introduce a new test for Hamming-weight dependencies that will find bias in all generators from Table 2. For the generators whose bias was detected by TestU01, the test will be able to obtain similar or better p-values using an order of magnitude fewer data.

Let us denote with νx [9] the Hamming weight of a w-bit word x, that is, the number of ones in its binary representation. We would like to examine the output of a generator and find *medium-range* dependencies in the Hamming weights of its outputs.

For example, the current output might have an average weight (close to $w/2$) with higher-than-expected probability if three outputs ago we have seen a word with average weight; or, the current output might have a non-average weight (high or low) with higher-than-expected probability depending on whether four outputs ago we have seen average weight *and* five outputs ago we have seen non-average weight.

We will consider 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 into smaller pieces fed sequentially to the test, or glue (a subset of) the generator output into larger pieces.

The basic idea of the test is that of generating a vector whose coordinates should appear to be drawn from independent random variables with a standard normal distribution, given that the original sequence was random; apply a unitary transformation, obtaining a transformed vector; and derive a p-value using the fact that the coordinates of the transformed vector should still appear to be drawn from independent random variables with a standard normal distribution [24], given that the original sequence was random. The transform will be designed in such a way to make dependencies as those we described emerge more clearly.

First of all, we first must define a *window* we will be working on: thus, we fix a parameter k and consider overlapping k-tuples of consecutive w-bit values (ideally, the number of bits of state should be less than kw). We will write $\langle x_0, x_1, \ldots, x_{k-1}\rangle$ for such a generic k-tuple.

Now we need to classify outputs as “average” or “extremal” with respect to their Hamming weight. We thus consider an integer parameter $\ell \leq w/2$, and the map

$$x \overset{d}{\mapsto} \begin{cases} 0 & \text{if } \nu x < w/2 - \ell; \\ 1 & \text{if } w/2 - \ell \leq \nu x \leq w/2 + \ell; \\ 2 & \text{if } \nu x > w/2 + \ell. \end{cases}$$

In other words, we compute the Hamming weight of x and categorize x in three classes: left tail (before the $2\ell + 1$ most frequent weights), central (the $2\ell + 1$ central, most frequent weights), right tail (after the $2\ell + 1$ most frequent weights). The standard choice for ℓ is the integer such that the overall probability of the $2\ell + 1$ most frequent weights is closest to $1/2$. For example, for $w = 32$ we have $\ell = 1$, whereas for $w = 64$ we have $\ell = 2$.

We thus get from the k-tuple $\langle x_0, x_1, \ldots, x_{k-1}\rangle$ a *signature* $\langle d(x_0), d(x_1), \ldots, d(x_{k-1})\rangle$ of k *trits* (base-3 digits), which we will identify with its value as a number in base 3:

$$\sum_{i=0}^{k-1} d(x_i) 3^{k-1-i}.$$

Now, given a sequence of m w-bit 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. More precisely, a subsequence of the form $\langle x_0, x_1, \ldots, x_k\rangle$ contributes νx_k to the average associated with the signature $\langle d(x_0), d(x_1), \ldots, d(x_{k-1})\rangle$.

This bookkeeping can be easily performed using 3^k integer variables while streaming the generator output. For a large m, this procedure yields 3^k values with approximately normal distribution [5],[7] which we normalize to a standard normal distribution; we denote the resulting row vector with $\boldsymbol{v} = \langle v_0, v_1, \ldots, v_{3^k-1}\rangle$.[8][9]

We now apply to $\boldsymbol{v}$ a Walsh–Hadamard-like transform, multiplying $\boldsymbol{v}$ by the k-th Kronecker power[10] T_k of the unitary base matrix

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

Assuming that T_k 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 (or any transform based on Kronecker powers), as if we write $\boldsymbol{v} = \begin{bmatrix}\boldsymbol{v}^0 & \boldsymbol{v}^1 & \boldsymbol{v}^2\end{bmatrix}$, where $\boldsymbol{v}^0$, $\boldsymbol{v}^1$, and $\boldsymbol{v}^2$ are the three subvectors indexed by signatures starting with 0, 1, and 2, respectively, we have by definition $T_0 = 1$ and

$$\begin{aligned} \boldsymbol{v}T_k &= \begin{bmatrix}\boldsymbol{v}^0 & \boldsymbol{v}^1 & \boldsymbol{v}^2\end{bmatrix} \begin{pmatrix} \frac{1}{\sqrt{3}}T_{k-1} & \frac{1}{\sqrt{2}}T_{k-1} & \frac{1}{\sqrt{6}}T_{k-1} \\ \frac{1}{\sqrt{3}}T_{k-1} & 0 & -\frac{2}{\sqrt{6}}T_{k-1} \\ \frac{1}{\sqrt{3}}T_{k-1} & -\frac{1}{\sqrt{2}}T_{k-1} & \frac{1}{\sqrt{6}}T_{k-1} \end{pmatrix} \\ &= \begin{bmatrix} \frac{1}{\sqrt{3}}\left(\boldsymbol{v}^0 + \boldsymbol{v}^1 + \boldsymbol{v}^2\right) T_{k-1} & \frac{1}{\sqrt{2}}\left(\boldsymbol{v}^0 - \boldsymbol{v}^2\right) T_{k-1} & \frac{1}{\sqrt{6}}\left(\boldsymbol{v}^0 - 2\boldsymbol{v}^1 + \boldsymbol{v}^2\right) T_{k-1} \end{bmatrix}. \end{aligned} \tag{2}$$

A detailed C implementation of T_k will be described in Section 4.3.

We will denote the transformed vector by $\boldsymbol{v}' = \boldsymbol{v}T_k$, and we shall write v_i' for the transformed values. Since T_k is unitary, the v_i''s must appear still to be drawn from a standard normal distribution, and we can thus compute p-values for each of them. We combine p-values by dividing the indices of the vector $\boldsymbol{v}'$ in C categories $\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_C$ using the number of nonzero trits contained in their base-3 representation, that is, the number of nonzero trits in the associated signature: $\mathscr{C}_j$, $1 \leq j < C$, contains indices with j nonzero trits, whereas $\mathscr{C}_C$ contains all remaining indices, whose base-3 representation has at least C nonzero trits ($C \leq k$; usually, $C = \lfloor k/2 \rfloor + 1$). We discard v_0'.

Given a category, say of cardinality c, we have thus a p-value p_i for each v_i' in the category; we then consider the minimum of the p-values, say, $\bar{p}$, and compute a final category p-value by composing with the cumulative distribution function of the minimum of c independent uniform random variables in the unit interval [3, Eq. (2.2.2)], obtaining the category p-value $1 - (1 - \bar{p})^c$. Finally, we take the minimum category p-value over all categories, and apply again the same cumulative distribution function with parameter C, since we are taking the minimum over C

[7] In practice, one must size m depending on the number of signatures, so that each signature has a sufficiently large number of associated samples. Implementations can provide quickly the user with a preview output using the value zero for random variables associated with non-appearing signatures, warning the user that final p-values too close to one might be artifacts.

[8] Breaking the generator output in smaller pieces provides obviously a finer analysis of the distribution of each piece, 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.

[9] 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*.

[10] For a definition of the Kronecker product, see [26, Section 4.3].

categories: this yields the final p-value of the test. Formally,

$$p = 1 - \left(1 - \min_{1 \le j \le C} \left(1 - \left(1 - \min_{i \in \mathscr{C}_j} p_i\right)^{|\mathscr{C}_i|}\right)\right)^C.$$

The point of the transform T_k is that while v_i represents the (normalized) average number of ones after k previous outputs with density pattern described by the trit representation of i, v'_i represents a combination of the average number of ones after k previous outputs satisfying different constraints: in the end, a unitary transformation is just a change of coordinates.

As a simple example, let us write the index $\bar\imath$ of a transformed value $v'_{\bar\imath}$ as a sequence of trits t_1, t_2, …, t_k. If the trits are all zero, looking at (2) one can see that we are just computing the normalized sum of all values, which is of little interest: indeed, we discard v'_0.

On the contrary, if a single trit, say in position $\bar\jmath$, is equal to 1, $v'_{\bar\imath}$ is given by the sum of all v_i's in which the $\bar\jmath$-th trit of i is 0 ($\bar\jmath$ steps before we have seen few zeros) minus the sum of all v_i's in which the $\bar\jmath$-th trit of i is 2 ($\bar\jmath$ steps before we have seen many zeros): if the Hamming weight of the output depends on the Hamming weight of the output $\bar\jmath$ steps before, the value of $v'_{\bar\imath}$ will be biased. Intuitively, if the transition matrix is very sparse we expect vectors with low or high Hamming weight to be mapped to vectors with the same property.

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 $\bar\jmath$ steps before was extremal or average: more precisely, whether the value is larger when the Hamming weight of the output $\bar\jmath$ steps before was average and smaller when the Hamming weight of the output $\bar\jmath$ steps before was extremal (and *vice versa*). Intuitively, we expect that the shift/rotate-XOR of states with a very small or very large number of ones will have a small number of ones (in the first case, by sparsity, in the second case, by cancellation).

More complex trit patterns detect more complex dependencies: the most interesting patterns, however, usually are those with few nonzero trits, as a zero trit acts as a "don't care about that previous output": this property is immediate from (2), as the first 3^{k-1} output values, which correspond to a "don't care" value in the first position, are obtained by applying recursively T_{k-1} over the renormalized sum of $\boldsymbol{v}_0$, $\boldsymbol{v}_1$, and $\boldsymbol{v}_2$, thus combining the values associated with signatures identical but for the first trit.

This is also why we assemble p-values by categories: by putting indices with a higher chance of giving low p-values in small categories, the test becomes more sensitive.

3.1 Results

We ran tests with $w = 32$ or $w = 64$ and k ranging from 8 to 19, depending on the state size. We performed the tests incrementally, that is, for increasingly larger values of m, and stopped after a petabyte (10^{15} bytes) of data or if we detected a p-value smaller than 10^{-20}.

Table 3 reports some generators failing our test. All generators considered other than `xorshift` pass BigCrush, except for *linearity tests* [2, 4, 14] (called MatrixRank and LinearComp in TestU01). We report the faulty signature, that is, the pattern of dependencies that caused the low p-value: it provides interesting insights into the structure of the generator. Indeed, we can see that for generators that cycle through their state array, combining a small part of the state, the test can locate exactly the dependencies from those parts: for example, the Hamming-weight the output of `xorshift1024` depends, not surprisingly, from the Hamming weight of the first and last word of state.

First, we examine a `xorshift` generator [13] with 128 bits of state, and its variant `xorshift128+` that we discussed in Section 2. We can find bias in the latter using just 6 GB of data. Analogously, we find bias in a `xorshift` generator using 1024 bits of state, and in the SFMT [21] with 607 bits of state using just 400 MB of data. On these extremely simple generators, the performance of the test is thus in line with that of the tests in TestU01.

However, once we turn to the other generators in Table 2 the situation is different: we can find bias in all generators, sometimes using an order of magnitude less data than in Table 2.

Our test can also find Hamming-weight dependencies in some generators of the Mersenne Twister family with small-to-medium size. First of all, we consider the 64-bit Tiny Mersenne Twister [23], which has 127 bits of state and a significantly more complex structure than the other generators in Table 3. Moreover, contrarily to other members of the Mersenne Twister family, the output function of the Tiny Mersenne Twister contains a non-$\mathbf{F}_2$-linear operation—a sum in $\mathbf{Z}/2^{64}\mathbf{Z}$. 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 into two 32-bit words. We report a range of results because we tried a few parameters published by the authors.

We also analyzed the classic Mersenne Twister [16] at 521 and 607 bits. We used Matsumoto and Nishimura's library for the dynamic creation of Mersenne Twisters [15], and generated eight different instances of each generator: this is why we report in Table 3 a range of values and multiple signatures. The 607-bit version performs much worse than the 521-bit version (in fact, all instances we tested failed even the classical Gap test from BigCrush). 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 performs in our test similarly to a `xorshift128` generator, which has a simpler structure and a much smaller state.

Finally, we were able to find bias in WELL512 [19]. In this case, we noticed that the p-value was slowly drifting towards zero at about 1 PB of data, so we continued the test until it passed the threshold 10^{-20}.

A comparison between Table 2 and Table 3 shows clearly that our new test is significantly more powerful than the tests of the same kind available in TestU01, as it can detect bias on $\mathbf{F}_2$-linear generators for which no such bias was previously detectable. In fact, to the best of our knowledge this is the first time that Tiny Mersenne Twister, the dSFMT at 521 bits, and WELL512 fail a test that is not a linearity test.

It is worth noting that in the first submission of this paper `xoroshiro128+` did not present failures in Table 2. We were able to find a low p-value ($\approx 3 \times 10^{-11}$) only specifying the value 128 for the parameter L, as suggested by a referee. Larger values of L (e.g., 256, 300,...) do not yield a failure. This is in sharp contrast with our test, where testing with $k' > k$ will preserve the failures found in dimension k, because if s is a k-dimensional failing signature, then $0^{k'-k}s$ will be a failing k'-dimensional signature, with some small adjustments due to the different scaling to the standard normal distribution and the different size of categories. In other words, increasing the dimension of the test will not prevent the test from detecting bias that was previously detectable at a lower dimension: the same does not happen for the HammingIndep test of TestU01.

4 IMPLEMENTATION DETAILS

We will now discuss some implementation details. To be able to perform our test in the petabyte range, it must be engineered carefully: 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. The v_i's are stored in an array of 3^k elements indexed by the value of a signature as a numeral in base 3, as required by the recursive implementation of T_k (see Section 4.3). One can keep track very easily of the current trit signature value by using the update rule $s \leftarrow \lfloor s/3 \rfloor + t \cdot 3^{k-1}$, where t is the next trit.

Table 3. Detailed results of the test described in Section 3 for $w = 64$. We report the number of bytes generating a p-value smaller than 10^{-20}. We report also the trit signature which caused the low p-value. Ranges (represented using the arrow symbol ⟶) appear when we tried several variants: a missing right extreme means that some instances did not fail the test within the 1 PB limit.

Generator	$p = 10^{-20}$ @	Faulty signature
xorshift128	8×10^8	00000021
xorshift128+	6×10^9	00000012 (transitional)
xorshift1024	6×10^8	2000000000000001
xorshift1024+	9×10^9	2000000000000001 (transitional)
xoroshiro128	1×10^{10}	00000012
xoroshiro128+	5×10^{12}	00000012
xoroshiro1024	5×10^{12}	1100000000000001
xoroshiro1024+	4×10^{13}	1100000000000001 (transitional)
Tiny Mersenne Twister (127 bits)	$8 \times 10^{13} \longrightarrow$	00000202 ($w = 32$)
Mersenne Twister (521 bits)	$4 \times 10^{10} \longrightarrow$	1000000100000000 2000000100000000
Mersenne Twister (607 bits)	$4 \times 10^8 \longrightarrow 4 \times 10^{10}$	10000000001000000000 20000000001000000000
SFMT (607 bits)	4×10^8	001000001000
dSFMT (521 bits)	6×10^{12}	1001000100100010
WELL512	3×10^{15}	2001002200000000

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 $k = 19$ using 64-bit integers), 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.

4.1 Small counters

The main implementation challenge, however, is that of reducing the counter update area to improve the locality of access to the counters, and possibly making it fit into some level of the processor cache.[11] In a naive implementation, we would need to use two "large" 64-bit values to store the number of appearances of signature s, and the sum of Hamming weights of the following words. Instead, we will use a single "small" 32-bit value, with a fourfold space saving. In particular, we will use 13 bits for the counter and 19 bits for the summation. This is a good choice as the largest Hamming weight for $w = 32$ or $w = 64$ is 64, so if the counter does not overflow, the summation will not, either.[12]

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.

[11] When k is large, this is not possible, but we provide the option of improving memory access using large pages of the Translation Lookaside Buffer where available.

[12] With a similar argument, when $w = 16$ one can choose 14 bits for the counter and 18 bits for the summation.

4.2 Batch sizes

How large should a batch be? We prefer larger batches, as access to large counters will be minimized, but too large a batch will overflow small counters. This question is interesting, as it is related to the *mean passage time distribution* of the Markov chain having all possible signatures as states, and the probability of moving from signature s to signature $\lfloor s/3 \rfloor + t \cdot 3^{k-1}$ given by 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?[13] We want to keep this probability very low (say, 10^{-100}) as 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 greater than 10^{-100}, given that the source is random, that is, a p-value.

Note that, in principle, we could use general results about Markov chains [6, 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 [7, 18].

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 $0 \leq c \leq b$, b is a given overflow bound, and $0 \leq j < k$.

The idea is that we will define transitions so that after u steps the probability of being in state $x_{c,j}$ will be the probability that after examining u w-bit values our current trit signature has a maximal suffix of j trits equal to one, and that we have counted exactly c passages through the all-ones signature (b or more, when $c = b$), with the proviso that the value $j = k - 1$ represents both maximal suffixes of length $k - 1$ and of length k (we can lump them together as receiving a one increases the passage count in both cases). We use an initial probability distribution in which all states with $c \neq 0$ have probability zero, and all states with $c = 0$ have probability equal to the steady-state probability of j, which implies that we are implicitly starting the original chain in the steady state. However, as argued also in [6], 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 b passages lumped together):

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

It is easy to show that after u steps the sum of the probabilities associated with the states $x_{b,-}$ is exactly the probability of overflow of the counter associated with the all-one signature. We thus iterate the Markov chain (squaring the transition matrix is possible only for small k) until, say at step B, 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 batches of size B we can observe overflow only with probability at most $\bar{p}$.

[13] 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.

This approach becomes unfeasible when we need to iterate the Markov chain more than, say, 10^7 times. However, at that point we use a very good approximation: we apply a simple dynamic-programming scheme on the results for 10^6 steps to extend the results to a larger number of steps. The idea is that if you know the probability $q_{u,c}$ that the counter for the all-ones signature is c after u steps, then approximately

$$\begin{aligned} q_{u+v,c} &= \sum_{f+g=c} q_{u,f} \cdot q_{v,g} \qquad \text{for } 0 \leq c < b, \\ q_{u+v,b} &= \sum_{f+g\geq b} q_{u,f} \cdot q_{v,g}. \end{aligned}$$

The approximation is due to the fact that the equations above implicitly assume that the Markov chain is reset to its steady-state distribution after u steps, but experiments at smaller sizes show that the error caused by this approximation is, as expected, negligible for large u. We thus initialize $q_{u,c}$ for $u = 10^6$ with exact data, and then we iterate the process above to obtain the probabilities $q_{2^h u}, c$. These probabilities are then combined in the same way to approximate the probabilities associated with every multiple of u; at that point we can find the desired batch size by a binary search governed by the condition that the probability associated with the overflow bound b is below a suitable threshold (e.g., $10^{-100}/3^k$).[14]

In the end, we computed the ideal batch size as described above for $1 \leq k \leq 19$ and included the result into our code (for example, when $w = 64$ one obtains 15×10^3 for $k = 1$, 23×10^6 for $k = 8$, and 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 little more than 3 minutes on an Intel® Core™ i7-8700B CPU @3.20GHz. The $k = 16$ test is an order of magnitude slower due to the larger memory accessed.

4.3 Implementing the transform T_k

In figure 1 we show an in-place, recursive C implementation of the transform T_k defined in Section 3. The code is similar to analogous code for the Walsh–Hadamard transform or similar transforms based on Kronecker powers.

The code assumes that the 3^k-dimensional vector $\boldsymbol{v}$ is represented in the array `v`. The value associated with each signature is stored in a position equal to the signature (considered, as usual, as a base-3 numeral). In particular, the first 3^{k-1} values correspond to signatures of the form $0s$, the following 3^{k-1} values to signatures of the form $1s$, and the last 3^{k-1} values to signatures of the form $2s$. The function `transform()` must be invoked on `v` with the additional parameter `sig` set to 3^{k-1}.

We first note that if $k = 1$ the function will just execute once the body of the for loop, resulting in the in-place multiplication of the 3-dimensional vector $\boldsymbol{v}$ by the base matrix M, as expected.

In the general case, the code scans the three subarrays `v`, `p1`, and `p2`, of length 3^{k-1}, which as discussed above correspond to signatures starting with 0, 1, and 2, respectively. With the notation of (2), these subarrays correspond to the subvectors $\boldsymbol{v}^0$, $\boldsymbol{v}^1$, and $\boldsymbol{v}^2$, respectively, and it is immediate that the three subvectors appearing in the final result of (2) are computed in place by the for loop. After that computation, the recursion applies by induction the transform with one dimension less to each of the three subarrays in place. We conclude that `transform()` implements correctly in place the transform T_k.

[14] It is worth noting that, based on the computations above, the normal approximation [6] is not very accurate even after a billion steps.

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

    for (int i = 0; i < sig; i++) {
        const double 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. 1. The code for the (in-place) transform described in Section 3. It should be invoked with `sig` equal to 3^{k-1}.

5 CONCLUSIONS

We have described a new test for Hamming-weight dependencies based on a unitary transform. Properly implemented, the test is very powerful: for example, it finds in a matter of hours bias in the dSFMT with 521 bits of state and in `xoroshiro1024+`; it can even find bias in WELL512, even though its transition matrix is much denser, and in the Tiny Mersenne Twister. For these generators no bias was previously known beyond linearity tests. In particular, the Hamming-weight tests in TestU01 [12], a state-of-the-art testing framework, are unable to find any bias in several generators of Table 3, whereas all those generators fail our test.

Our test is very effective on $\mathbf{F}_2$-linear generators with relatively sparse transitions matrices, in particular when wk is not smaller than the number of bits of state of the generator. In practice, the best results are obtained on generators with less than a few thousand bits of state.

Similar to linearity tests, a failure in our test is an indication of lesser randomness, but in general the impact will depend on the application. We do not expect dependencies like those in `xorshift128+` or the SFMT to be pernicious, but they highlight a weakness of the associated $\mathbf{F}_2$-linear transformation.

ACKNOWLEDGMENTS

The authors would like to thank Jeffrey Hunter for useful pointers to the literature about mean passage times in Markov chains and Pierre L'Ecuyer for a number of suggestions that improved the quality of the presentation.

REFERENCES

[1] David Blackman and Sebastiano Vigna. 2021. Scrambled Linear Pseudorandom Number Generators. *ACM Trans. Math. Softw.* 47 (2021), 1–32. Issue 4.

[2] G. D. Carter. 1989. *Aspects of local linear complexity.* Ph. D. Dissertation. University of London.

[3] H.A. David and H.N. Nagaraja. 2004. *Order Statistics.* Wiley.

[4] E. D. Erdmann. 1992. Empirical tests of binary keystreams.

[5] Christian Hipp and Lutz Mattner. 2008. On the Normal Approximation to Symmetric Binomial Distributions. *Theory Probab. Appl.* 52, 3 (2008), 516–523.

[6] Jeffrey J. Hunter. 1983. *Mathematical Techniques of Applied Probability, Volume 2, Discrete-Time Models: Techniques and Applications.* Academic Press, New York.

[7] 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.

[8] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third ed.). Addison-Wesley, Reading, MA, USA.

[9] Donald E. Knuth. 2011. *The Art of Computer Programming: Volume 4, Combinatorial algorithms. Part 1.* Vol. 4A. Addison-Wesley. xv + 883 pages.

[10] Pierre L'Ecuyer and François Panneton. 2009. $\mathbf{F}_2$-Linear Random Number Generators. In *Advancing the Frontiers of Simulation*, Christos Alexopoulos, David Goldsman, and James R. Wilson (Eds.). International Series in Operations Research & Management Science, Vol. 133. Springer US, 169–193.

[11] 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.

[12] 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).

[13] George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14 (2003), 1–6.

[14] George Marsaglia and Liang-Huei Tsay. 1985. Matrices and the structure of random number sequences. *Linear Algebra Appl.* 67 (1985), 147–156.

[15] Makoto Matsumoto and Takuji Nishimura. 1998. Dynamic creation of pseudorandom number generators. *Monte Carlo and Quasi-Monte Carlo Methods* 2000 (1998), 56–69.

[16] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (1998), 3–30.

[17] Makoto Matsumoto and Takuji Nishimura. 2002. A Nonempirical Test on the Weight of Pseudorandom Number Generators. In *Monte Carlo and Quasi-Monte Carlo Methods 2000: Proceedings of a Conference held at Hong Kong Baptist University, Hong Kong SAR, China*, Kai-Tai Fang, Harald Niederreiter, and Fred J. Hickernell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–395.

[18] 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.

[19] 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.

[20] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. 2001. *A Statistical Test Suite For Random and Pseudorandom Number Generators for Cryptographic Applications.* National Institute for Standards and Technology, pub-NIST:adr. NIST Special Publication 800-22, with revisions dated May 15, 2001..

[21] Mutsuo Saito and Makoto Matsumoto. 2008. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, Alexander Keller, Stefan Heinrich, and Harald Niederreiter (Eds.). Springer, 607–622.

[22] 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. https://doi.org/10.1007/978-3-642-04107-5_38

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

[24] Y.L. Tong. 2012. *The Multivariate Normal Distribution.* Springer New York.

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

[26] F. Zhang. 2011. *Matrix Theory: Basic Results and Techniques.* Springer New York.