Journal of Computational and Applied Mathematics 350 (2019) 139–142

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Short Communication

Xorshift1024*, xorshift1024+, xorshift128+ and xoroshiro128+ fail statistical tests for linearity

Daniel Lemire [a,*], Melissa E. O'Neill [b]

[a] Université du Québec (TELUQ), 5800 Saint-Denis, Montreal, QC, H2S 3L5, Canada
[b] Harvey Mudd College, 301 Platt Boulevard, Claremont, CA 91711, USA

HIGHLIGHTS

- We expose previously unknown faults in widely used random generators proposed by Vigna.
- These faults are systematic and decisive.
- They are generated by reversing the order of the bits, an approach stressed by Vigna himself.
- We provide open source code and scripts to make reproduction easy.
- Our work illustrates the need to test 64-bit generators more carefully.

ARTICLE INFO

Article history:
Received 19 December 2017
Received in revised form 12 October 2018

Keywords:
Random number generator
Statistical tests
Xorshift

ABSTRACT

L'Ecuyer & Simard's *Big Crush* statistical test suite has revealed statistical flaws in many popular random number generators including Marsaglia's Xorshift generators. Vigna recently proposed some 64-bit variations on the Xorshift scheme that are further *scrambled* (i.e., xorshift1024*, xorshift1024+, xorshift128+, xoroshiro128+). Unlike their unscrambled counterparts, they pass Big Crush when interleaving blocks of 32 bits for each 64-bit word (most significant, least significant, most significant, least significant, etc.). We report that these scrambled generators systematically fail Big Crush – specifically the linear-complexity and matrix-rank tests that detect linearity – when taking the 32 lowest-order bits in reverse order from each 64-bit word.

1. Introduction

Pseudorandom number generators (PRNGs) are useful in simulations, games, testing, artificial intelligence, probabilistic algorithms, and so forth. To make it easier to thoroughly test random number generators, L'Ecuyer and Simard published the TestU01 software library [1]. This freely available and widely used library supports several batteries of tests, the most thorough being Big Crush.

As an instance of particularly fast random number generators, Marsaglia proposed the Xorshift family [2]. In C, it can be implemented as a sequence of shift and xor operations (e.g., `x^=(x<<13); x^=(x>>7); x^=(x<<17)`) repeatedly applied on an integer value initialized from a user-provided seed. At each step, the returned random value is the state variable (e.g., `x`). Though the Xorshift generators are fast, Panneton and L'Ecuyer showed that they are statistically unreliable [3]. In particular, many Xorshift generators fail matrix-rank tests, which generate random $L \times L$ binary matrices, and compare their

* Corresponding author.
E-mail addresses: lemire@gmail.com (D. Lemire), oneill@cs.hmc.edu (M.E. O'Neill).

https://doi.org/10.1016/j.cam.2018.10.019
0377-0427/

```c
uint64_t s[16];
int p;
uint64_t mult = 0x106689D45497FDB5;
uint64_t xorshift1024star(void) {
  uint64_t s0 = s[p];
  uint64_t s1 = s[p = (p + 1) & 15];
  s1 ^= s1 << 31;
  s[p] = s1 ^ s0 ^ (s1 >> 11)
       ^ (s0 >> 30);
  return s[p] * mult;
}
```

(a) xorshift1024*

```c
uint64_t s[16];
int p;
uint64_t xorshift1024plus(void) {
  uint64_t s0 = s[p];
  uint64_t s1 = s[p = (p + 1) & 15];
  uint64_t result = s0 + s1;
  s1 ^= s1 << 31;
  s[p] = s1 ^ s0
       ^ (s1 >> 11)
       ^ (s0 >> 30);
  return result;
}
```

(b) xorshift1024+

```c
uint64_t s[2];
uint64_t xorshift128plus(void) {
  uint64_t s1 = s[0];
  uint64_t s0 = s[1];
  uint64_t result = s0 + s1;
  s[0] = s0;
  s1 ^= s1 << 23;
  s[1] = s1 ^ s0
    ^ (s1 >> 18) ^ (s0 >> 5);
  return result;
}
```

(c) xorshift128+

```c
uint64_t s[2];
uint64_t rotl(uint64_t x, int k) {
 return (x << k) | (x >> (64 - k));
}
uint64_t xoroshiro128(void) {
  uint64_t s0 = s[0];
  uint64_t s1 = s[1];
  uint64_t result = s0 + s1;
  s1 ^= s0;
  s[0] = rotl(s0, 55) ^ s1 ^ (s1 << 14);
  s[1] = rotl(s1, 36);
  return result;
}
```

(d) xoroshiro128

Fig. 1. C functions defining the various scrambled Xorshift generators.

ranks against the expected theoretical distribution. Following Marsaglia himself, Panneton and L'Ecuyer proposed that the Xorshift generators could be improved by combining them with other operations.

In this spirit, Vigna proposed a 64-bit alternative, xorshift1024* (see Fig. 1a): he states that it passes Big Crush, even after reversing the bit order [4]. It combines a Xorshift generator with a multiplication. In a related attempt to improve the statistical properties of Xorshift generators, Saito and Matsumoto proposed the xorshift-add generator where two consecutive 32-bit outputs of a Xorshift generator are added together and returned as the random value [5]. They report that xorshift-add passes Big Crush. However, Vigna observes that if the bit order of the 32-bit random values is reversed (e.g., `0xff444881` becomes `0x811222ff`) before passing the results to Big Crush, the xorshift-add generator systematically fails [4] both matrix-rank and linear-complexity tests.

In further work, Vigna proposed xorshift128+ and xorshift1024+ (see Figs. 1b and 1c), two new 64-bit generators that resemble Saito and Matsumoto's xorshift-add in that they return the addition of two state values [6]. He again states that they pass Big Crush. However, unlike xorshift-add but like xorshift1024*, he adds that they pass Big Crush even with their bits reversed. A version of Vigna's xorshift128+ generator has been adopted by the V8 JavaScript engine used by the popular Chrome browser. V8 uses code identical to the function recommended by Vigna (Fig. 1c), but where the constants 23, 18, 5 are replaced by the constants 23, 17, 26. In more recent work, Blackman and Vigna proposed xoroshiro128+ (see Fig. 1d), presented as the successor of xorshift128+ [7]. We refer to these generators (xorshift1024*, xorshift1024+, xorshift128+ and xoroshiro128+) as *scrambled Xorshift* generators, following Vigna's terminology.

These scrambled Xorshift generators output 64-bit integers. Ideally, a 64-bit PRNG should be tested with a 64-bit test suite; using a test suite designed for 32-bit PRNGs (or actually 31-bit PRNGs in the case of Big Crush) necessarily involves compromises. Vigna's approach was to interleave the least-significant 32 bits and most-significant 32 bits of each 64-bit output, thus each 64-bit output becomes two 32-bit outputs. This interleaving strategy may detect flaws in some 64-bit PRNGs, but it may also hide statistical weaknesses. Such an approach is useful but is *insufficient*. A more comprehensive strategy is to also test the most-significant and least-significant 32 bits separately. When we focus on the least-significant 32 bits, and, specifically, in the case where their bits are reversed, we find that the scrambled Xorshift generators systematically fail Big Crush.

2. Results

It is expected that even good generators could fail some tests, some of the time. We are only interested in decisive and systematic failures:

- We focus solely on tests failed with extreme p-values according to TestU01. By excluding mild failures (e.g., $p = 0.05$), we reduce the likelihood of false negatives.
- We only report failures that occur irrespective of the initial seed.

Table 1
Tests failed systematically by the least significant 32 bits in reversed bit order.

Scrambled 64-bit Xorshift	Failed 32-bit Big Crush tests
Xorshift1024*	LinearComp
Xorshift1024+	LinearComp
Xorshift128+	LinearComp, MatrixRank
Xorshift128+ (v8)	LinearComp, MatrixRank
Xoroshiro128	LinearComp, MatrixRank

Thus we test the scrambled xorshift generators with one hundred different initial seeds (denoted by the variable s in Fig. 1). The generators under consideration require relatively long seeds (at least 16 bytes). To generate a sufficiently long seed (128 or 1024 bits), we produce a random integer with the Bash shell's internal RANDOM function as an initial seed for to the 64-bit SplitMix generator [8]. The SplitMix generator is called twice for the generators requiring 128-bit seeds and sixteen times for the generators requiring 1024-bit seeds. According to our tests, the 64-bit SplitMix generator passes Big Crush, even with the bit order reversed.

We use the latest version of TestU01 (version 1.2.3). We report the tests failed in Big Crush for each generator in Table 1; we only report tests that failed for all one hundred different seeds. All scrambled generators fail the linear-complexity tests (LinearComp). The 128-bit xorshift+ generators additionally fail the matrix-rank tests (MatrixRank) for both $L = 1000$ and $L = 5000$. All matrix-rank tests fail with a p-value smaller than 10^{-100}, all linear-complexity tests fail with a p-value greater than or equal to $1-10^{-15}$, except for one seeding of xorshift128+, where we observed a p-value of $1-4.3\times10^{-11}$. Getting such extreme p-values for one hundred different seeds indicates a systematic failure. There are many more minor, nonsystematic failures (e.g., 33 for xorshift128+) that might require further analysis [9]. To ease reproducibility, both our scripts and results are freely available online.[1]

3. Conclusion

The least significant 32 bits of the 64-bit scrambled xorshift generators (xorshift1024*, xorshift1024+, xorshift128+ and xoroshiro128+) systematically fail Big Crush. In particular, all fail the linear-complexity tests. These scrambled xorshift generators are derived from Marsaglia's xorshift generators which also systematically fail these tests [1].

Panneton et al. [10] note that PRNG schemes based on linear recurrences modulo 2 – which includes xorshift schemes with simple output functions – can expect to see detectable linear dependencies in tests such as the matrix-rank test [11] and the linear-complexity tests [12], so the failures we report are unsurprising. Panneton et al. also note that although these problems are an issue for some applications, for many applications they would go unnoticed. Thus readers should not assume that failing some statistical tests renders a PRNG scheme worthless. But it is useful to note that there are multiple PRNG schemes available that do not fail any currently available statistical tests and thus do not require weighing whether detectable linearity is a problem for a particular use case or not [1].

Mirroring a related recommendation by Press et al. [13], Vigna argued that TestU01 should always be applied on the reversed generators [4] to address the problem of linearity issues in the low-order bits of a PRNG going undetected. Moreover, our own results suggest that when assessing a 64-bit generator with TestU01, thorough testing requires us to test the least significant 32 bits separately from the most significant 32 bits, otherwise statistical issues may likewise go undetected.

Acknowledgment

The first author was supported by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN-2017-03910.

References

[1] P. L'Ecuyer, R. Simard, TestU01: A C library for empirical testing of random number generators, ACM Trans. Math. Software 33 (4) (2007) 22:1–22:40.
[2] G. Marsaglia, Xorshift RNGs, J. Stat. Softw. 8 (14) (2003) 1–6.
[3] F. Panneton, P. L'Ecuyer, On the Xorshift random number generators, ACM Trans. Model. Comput. Simul. 15 (4) (2005) 346–361.
[4] S. Vigna, An experimental exploration of Marsaglia's Xorshift generators, scrambled, ACM Trans. Math. Software 42 (4) (2016) 30:1–30:23.
[5] M. Saito, M. Matsumoto, Xorshift-add (XSadd): A variant of Xorshift, 2014, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/XSADD/ [last checked October 2017].
[6] S. Vigna, Further scramblings of Marsaglia's Xorshift generators, J. Comput. Appl. Math. 315 (C) (2017) 175–181.
[7] D. Blackman, S. Vigna, Xoroshiro128+, 2016, http://xoroshiro.di.unimi.it/xoroshiro128plus.c [last checked October 2017].
[8] G.L. Steele Jr., D. Lea, C.H. Flood, Fast splittable pseudorandom number generators, SIGPLAN Not. 49 (10) (2014) 453–472.
[9] H. Haramoto, Automation of statistical tests on randomness to obtain clearer conclusion, in: Monte Carlo and Quasi-Monte Carlo Methods 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 411–421.
[10] F. Panneton, P. L'Ecuyer, M. Matsumoto, Improved long-period generators based on linear recurrences modulo 2, ACM Trans. Math. Software 32 (1) (2006) 1–16.

[1] See https://github.com/lemire/testingRNG.

[11] G. Marsaglia, A current view of random number generators, in: L. Billard (Ed.), Computer Science and Statistics, Sixteenth Symposium on the Interface, Elsevier Science Publishers, North-Holland, Amsterdam, 1985, pp. 3–10.
[12] E.D. Erdmann, Empirical tests of binary keystreams, Master's Thesis, Department of Mathematics, Royal Holloway and Bedford New College, University of London, 1992.
[13] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes 3rd Edition: The Art of Scientific Computing, third ed., Cambridge University Press, New York, NY, USA, 2007.