Better Splittable Pseudorandom Number Generators (and Almost As Fast)

ANONYMOUS AUTHOR(S)

We have tested and analyzed the SplitMix pseudorandom number generator algorithm presented by Steele, Lea, and Flood (2014), and have discovered two additional classes of gamma values that produce weak pseudorandom sequences. In this paper we present a modification to the SplitMix algorithm that avoids all three classes of problematic gamma values, and also a completely new algorithm for splittable pseudorandom number generators, which we call TwinLinear.

Like SplitMix, TwinLinear provides both a *generate* operation that returns one (64-bit) pseudorandom value and a *split* operation that produces a new generator instance that with very high probability behaves as if statistically independent of all other instances. Also like SplitMix, TwinLinear requires no locking or other synchronization (other than the usual memory fence after instance initialization), and is suitable for use with SIMD instruction sets because it has no branches or loops.

The TwinLinear algorithm is the result of a systematic exploration of a substantial space of nonlinear mixing functions that combine the output of two independent generators of (perhaps not very strong) pseudorandom number sequences. We discuss this design space and our strategy for exploring it. We used the PractRand test suite (which has provision for failing fast) to filter out poor candidates, then used TestU01 BigCrush to verify the quality of candidates that withstood PractRand.

We present results of analysis and extensive testing on TwinLinear (using both TestU01 and PractRand). Single instances of TwinLinear have no known weaknesses, and TwinLinear is significantly more robust than SplitMix against accidental correlation in a multithreaded setting. It is slightly more costly than SplitMix (10 or 11 64-bit arithmetic operations per 64 bits generated, rather than 9) but has a shorter critical path (5 or 6 operations rather than 8). We believe that TwinLinear is suitable for the same sorts of applications as SplitMix, that is, "everyday" scientific and machine-learning applications (but not cryptographic applications), especially when concurrent threads or distributed processes are involved.

CCS Concepts: •**Theory of computation → Pseudorandomness and derandomization;** •**Computing methodologies → Concurrent algorithms;**

Additional Key Words and Phrases: collections, Java, multithreading, object-oriented, parallel computing, PRNG, pseudorandom, random number generator, recursive splitting, RNG, spliterator, splittable data structures, streams

ACM Reference format:
Anonymous Author(s). 2017. Better Splittable Pseudorandom Number Generators (and Almost As Fast). *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 41 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Steele, Lea, and Flood (2014) describe the SplitMix algorithm for pseudorandom number generators (PRNGs). (We refer readers to Section 1 of their paper for a general description of PRNG algorithms and the challenges that must be addressed when using them in parallel computations.) A 64-bit version of the SplitMix algorithm was deployed as class `java.util.SplittableRandom` for the Java programming language in JDK8 (Oracle 2016). The basic structure of one instance (in the object-oriented sense) of this algorithm is an object with a mutable 64-bit integer field `seed` and an immutable odd 64-bit integer parameter (`final` field) `gamma`; to generate a new pseudorandomly chosen 64-bit integer, the object adds `gamma` to `seed`, then applies a *mixing function* to the new value of `seed`. Steele, Lea, and Flood comment on, and offer some evidence for, the notion that certain choices of the parameter `gamma` might produce pseudorandom sequences of less than ideal quality. They therefore

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

include code to test whether a proposed value for gamma had sufficient number of 01 and 10 pairs in its binary representation; if it does not, the candidate value is exclusive-OR'd with the constant 0xaaaaaaaaaaaaaaaa, producing a replacement value that is still odd but has a sufficient number of 01 and 10 pairs. Their only comment about this trick relative to the quality of the sequences produced is "Testing shows that this appears to be effective." Other parts of the paper test the entire 64-bit SplitMix algorithm, including this trick to avoid what we will call *weak gamma values*, using the DieHarder test suite (Brown et al. 2006).

We set out to verify the conjecture about weak gamma values and to test the quality of the SplitMix algorithm in other ways. In particular, we wanted to find out whether there were other classes of weak gamma values, and we wanted to test the algorithm with the well-known TestU01 BigCrush test suite (L'Ecuyer and Simard 2007; Simard 2009). For additional assurance, we decided also to use the PractRand test suite (Doty-Humphrey 2011), which is less well known than TestU01 but has the virtue of "failing early" as soon as it detects an undesirable amount of bias. We also hoped to identify an equally fast but more robust mixing function.

We present here a technique that can be used in SplitMix to defend against all three classes of weak gamma value. With this modification, we believe that the SplitMix algorithm used in JDK8 for class SplittableRandom is satisfactory for many purposes, but still has three drawbacks: (a) its overall state space (considering all possible instances) is 127 bits, which may be on the small side for large-scale applications; (b) the tt split operation takes more time; and (c) it is possible that there are yet other classes of weak gamma values. Still, it is the best published design we have seen so far for a reasonably fast and completely splittable pseudorandom number generator.

Our investigations led us to consider testing a broader class of candidate PRNG algorithms. While there are many papers in the literature on the subject of constructing a PRNG with a long period by combining two or more Linear Congruential Generators (LCGs), almost all such papers use some linear method of combination (presumably because this makes the subsequent mathematical analysis tractable). Typically each LCG is computed using arithmetic modulo some prime number, with a different prime number for each constituent LCG, and then the result are summed modulo some prime number. For example, the well-known MRG32k3a algorithm (Fischer et al. 1999) has exactly this structure. One drawback of this approach is that an algorithm of this form does not straightforwardly generate all 2^k possible values for a k-bit integer with equal probability; instead, one must typically either settle for an approximation to true uniformity (which is perfectly acceptable for some applications) or use some iterative method such as rejection sampling.

We present here a new PRNG algorithm framework, TwinLinear, in which all arithmetic is performed modulo a power of 2 (typically 2^{64}), and in which the outputs of two Linear Congruential Generators are combined using a *nonlinear* function. The result is a generator that is reasonably fast (almost as fast as SplitMix, possibly the same speed on some architectures; seems to be completely immune to the problem of weak gamma values; and has a much lower probability of other sorts of unwanted statistical accidents. Its state space (and therefore its memory footprint) is exactly twice that of SplitMix: two mutable integer fields and two immutable odd integer parameters per instance. Part of its speed is attributable to the fact that it does not treat its state as an array, and therefore performs no indexing operations. (This fact makes it easy for a compiler to keep the state in registers during execution of inner loops.)

We present test results to demonstrate that the TwinLinear algorithm has no apparent statistical weaknesses in the case of a single instance, and a significantly smaller probability than the SplitMix algorithm of accidental correlations among multiple instances.

In Section 2 we review the SplitMix algorithm and describe its weaknesses. In Section 4 we describe our test framework. In Section 5 we report the results of testing the SplitMix algorithm with a variety of mixing functions. In Section 6 we describe a new class of PRNG algorithms, TwinLinear, and one specific mixing function that appears to work especially well. In Section 7 we discuss potential weaknesses of the TwinLinear algorithm. In Section 8 we report the results of testing the SplitMix algorithm with a variety of mixing functions. In Section 10 we compare to related work, and in Section 11 we present some conclusions.

2 THE SPLITMIX ALGORITHM

In Figure 1 we reproduce the essential parts of the SplitMix algorithm (coded in the Java programming language) given by Steele, Lea, and Flood (2014) in their Figures 16 and 17. The public constructors ensure that the private constructor is always given an odd value for the parameter `gamma`. The private method `nextSeed` advances the state of the (additive) sequence generator and returns a generated 64-bit integer. The public method `nextLong` generates a pseudorandom 64-bit integer by giving the result of `nextSeed` to the method `mix64`, which implements the 64-bit MurmurHash3 mixing function (Appleby 2011). The public method `nextDouble` generates a pseudorandom 64-bit floating-point value by using the 53 high-order bits of an integer generated by `nextLong` as a fixed-point fraction, multiplying it by `DOUBLE_ULP` to produce a result in the range $[0.0, 1.0)$. The public method `split` returns a new instance of `SplittableRandom` created by choosing "at random" an initial seed value, produced by `nextLong`, and a new `gamma` parameter, produced by giving the result of `nextSeed` to `mixGamma`, which applies a different mixing function (`mix64variant13`, which implements the 13th mixing function described by Stafford (2011)) and then further modifies the value if it is determined to be potentially weak.

3 TWO NEW CLASSES OF WEAK GAMMA VALUE FOR SPLITMIX

We already knew the conjecture of Steele, Lea, and Flood that gamma values with few `01` and `10` pairs might generate sequences so weak that the mixing function could not compensate. Such gamma values tend to produce sequences in which consecutive values are the same in many bit positions. From this we inferred a more general conjecture: having one or more equally spaced subsequences, such that consecutive values within each subsequence are the same in many bit positions, might give the mixing function the same trouble. This suggested that gamma values of the form $\frac{2^{64}}{k}$ (for some small integer k) might also produce very weak sequences, because most of the high order bits within the overall generated sequence would have a repeating pattern with period k, and the mixing function might not be able to mask this correlation.[1]

Our other conjecture was that a weak gamma value might produce sequences such that consecutive values are transformed by some initial part of the mixing function computation into values that are the same in many bit positions. This suggested that gamma values of the form $m(2^s) + 1$, where s is the shift distance in the first shift-and-XOR step of the mixing function, might prove to be weak, because results of the operation `z ^ (z >>> s)` on successive values of the seed will tend to have the same low-order bits, and the remainder of the mixing function might not be able to mask this correlation.

We set out to confirm or disprove these two new conjectures.

4 OUR TESTING FRAMEWORK

We built a small testing framework to control thousands of test runs of multiple PRNG algorithms, using both the TestU01 BigCrush test suite and the PractRand test suite. Nearly all the tests[2] were performed on a cluster of 16 nodes, each with two sockets, each with an E5-2660 2.2Ghz Intel Xeon processor (each having eight cores collectively supporting 16 threads). Therefore 512 threads can execute simultaneously. We made no attempt to parallelize the TestU01 and BigCrush test suites; instead, we used `make` files to generate thousands of jobs at a time. Each `make` file describes one batch of test runs. Each `make` file includes code to find out which of the 16 nodes it is being run on, so that a different subset of the batch of test runs will be run on each node. The use of `make` files allowed a very simple form of crash recovery: simply a matter of re-issuing the `make` command.

---

[1]Indeed, we now realize that it suggests that there may be weak gamma values of the form $\frac{w}{k}$ where w is itself a weak gamma value and k is a small integer. We have not yet thoroughly tested this even more general idea.

[2]A very small fraction of the tests were run on a Macintosh Pro with two 2.8 GHz quad-core Xeon processors. This was done to validate the testing software before reserving time on the big cluster. The results of these initial runs constituted valid measurements and were retained.

```
package java.util;
import java.util.concurrent.atomic.AtomicLong;
public final class SplittableRandom {
  private long seed; private final long gamma;          // ‘gamma’ must be an odd integer
  private SplittableRandom(long seed, long gamma) {     // The argument ‘gamma’ must be odd
    this.seed = seed; this.gamma = gamma; }
  private long nextSeed() { return (seed += gamma); }
  private static long mix64(long z) {
    z = (z ^ (z >>> 33)) * 0xff51afd7ed558ccdL;
    z = (z ^ (z >>> 33)) * 0xc4ceb9fe1a85ec53L;
    return z ^ (z >>> 33); }
  private static long mix64variant13(long z) {
    z = (z ^ (z >>> 30)) * 0xbf58476d1ce4e5b9L;
    z = (z ^ (z >>> 27)) * 0x94d049bb173111ebL;
    return z ^ (z >>> 31); }
  private static long mixGamma(long z) {
    z = mix64variant13(z) | 1L;
    int n = Long.bitCount(z ^ (z >>> 1));
    if (n >= 24) z ^= 0xaaaaaaaaaaaaaaaaL;
    return z; }    // This result is always odd
  private static final double DOUBLE_ULP = 1.0 / (1L << 53);
  private static final long GOLDEN_GAMMA = 0x9e3779b97f4a7c15L;  // Note: this value is odd
  private static final AtomicLong defaultGen = new AtomicLong(initialSeed());
  public SplittableRandom(long seed) { this(seed, GOLDEN_GAMMA); }
  public SplittableRandom() {
    long s = defaultGen.getAndAdd(2 * GOLDEN_GAMMA);
    this.seed = mix64(s); this.gamma = mixGamma(s + GOLDEN_GAMMA); }
  public long nextLong() { return mix64(nextSeed()); }
  public double nextDouble() { return (nextLong() >>> 11) * DOUBLE_ULP; }
  public SplittableRandom split() {
    return new SplittableRandom(mix64(nextSeed()), mixGamma(nextSeed())); }
}
```

Fig. 1. Pertinent portions of class `SplittableRandom`

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

- There is a simple way to code new PRNG algorithms in C (or C++) and link them into the test suite. (This strategy means there is no I/O overhead for piping the PRNG output stream into the test suite.)
- Each reports results by printing text to "standard output". This report includes statistical information and also an indication of the total amount of CPU time (user execution time) consumed by the test.
- Each has a command-line interface that allows specification of which PRNG algorithm to test.
- The same command-line interface does not allow a complete specification of the initial state of the PRNG, but does allow specification of a 64-bit *seed* from which the initial state can be constructed, and the construction code can be user-specified and bundled with the code for the PRNG algorithm itself.

For our purposes, this last point meant that we had to design a series of specifications for using a single 64-bit integer to construct a wide variety of initial states. These specifications are presented in later sections.

The printed output of each individual test run is captured to a separate file. This file contains a date/time stamp at the start and the end (as produced by the Unix `date` command), with the printed output of the test suite in between. Once a batch of test runs has been executed and their printed reports captured to individual files, a custom "distillation" program (coded in Java) goes through the results and reduces each file to a data structure that can be summarized by a small row of numbers. (There are actually two such distillation programs, fairly similar in structure, one for BigCrush and one for PractRand.) The distillation program then writes a master summary file, a set of chart-producing data files, and a script file. The master summary file contains a description of the precise set of test runs in that batch, all of the summary results, and the total amount of user CPU time consumed by all the test runs. Each chart-producing data file is a `csv` (comma-separated values) file containing the data needed to produce one chart. The script file is a Unix shell script that uses the Macintosh application DataGraph to construct one `pdf` file from each of the chart-producing data files.

The overall workflow for a batch of test runs therefore looks something like this:

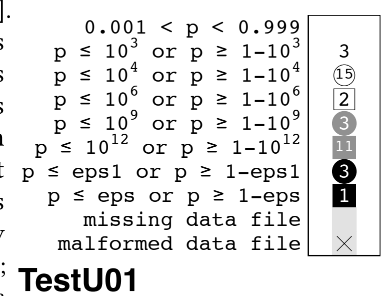
- Execute the `make` file (using the command "`make -j 32`") on each node of the cluster.
- Copy the directory containing all the resulting report files to the Macintosh Pro.
- Run the distillation program (either "`java DistillTestU01`" or "`java DistillPractRand`"), giving it a directory name (such as "`twotwin`") as an argument.
- Use "`chmod +x`" to cause the script file produced by the distillation program to be executable.
- Run the script file (a typical invocation would be "`./PractRand_twotwin_makegraphs`").

and the result is a set of `pdf` files, each containing one chart. Examples appear later in this paper.

Each chart is a two-dimensional grid of size up to 60 rows by 40 columns, where each grid entry is a square, roughly $\frac{1}{8}'' \times \frac{1}{8}''$, representing the result of one test run. Thus each chart can report the results of up to 2400 test runs. Different charts are organized in different ways, depending on the specific batch of test runs performed, but for each of the two test suites each individual entry is presented in the same way. The report from a single test run includes the results of many individual statistical tests (typically many dozens); these results are distilled into a single symbol and/or number to produce a chart entry.

4.1 Distilling TestU01 BigCrush Reports

The TestU01 BigCrush test suite runs 106 individual tests (L'Ecuyer and Simard 2013, function `bbattery_BigCrush`, pp. 148–152), computing 160 test statistics and p-values (L'Ecuyer and Simard 2007). A single test run typically prints about 110 kilobytes of information; at the end is either the message "`All tests were passed`" or a list of *anomalies*, that is, tests whose p-values were outside the range $[0.001, 0.999]$.

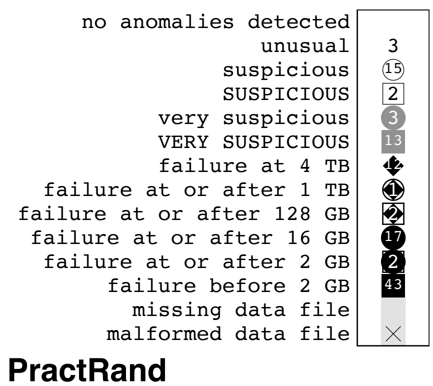


The distillation software for BigCrush test runs distills the list of anomalies for each test run into a pair of integers (f, c) (a *failure level* and a *count*) in this manner: If a test run file is missing, then $(f, c) = (-1, 0)$. If a test run file is present but is incomplete or malformed, then $(f, c) = (-2, 0)$ (this can happen if a test run was terminated before completion). If a test run file is present and all tests were passed, then $(f, c) = (0, 0)$. Otherwise, the test run file was present and well-formed but reported one or more anomalies. Each anomaly is categorized according to p-value into one of five failure levels, 1 through 5; then f is the highest failure level among all anomalies for the test run, and c is the number of anomalies having that highest failure level. For charting purposes, the pair of integers (f, c) is then reduced to a symbol and/or number. The idea is that the worse the results, the more ink on the page. These symbols were designed to make it easy to eyeball a chart and quickly recognize patterns of success and failure.

4.2 Distilling PractRand Reports

The PractRand test suite runs for an indefinite amount of time, normally producing intermediate reports after processing 2^m bytes of generated pseudorandom values for all integer values of m starting with $m = 27$. (We chose to provide command-line arguments that cause additional reports to be produced after processing 0.375×2^{40}, 0.75×2^{40}, 1.25×2^{40}, 1.5×2^{40}, 1.75×2^{40}, 2.25×2^{40}, 2.5×2^{40}, 2.75×2^{40}, 3×2^{40}, 3.25×2^{40}, 3.5×2^{40}, and 3.75×2^{40} bytes. We also provide command-line arguments that terminate the test run either after the first report that prints "FAIL" or after testing 4 terabytes of data, whichever comes first.) For a report produced after processing 2^m bytes of generated pseudorandom values, PractRand computes $4m - 56$ separate statistics; thus the first report (for $m = 27$) reports 52 test results, and the report for $m = 42$ (4 terabytes) reports 112 test results.

A single test run that gets all the way to 4 terabytes typically prints about 5 kilobytes of information. For each anomaly reported, PractRand prints not only a p-value but also a word or phrase describing that p-value; in increasing order of severity, they are `unusual`, `suspicious`, `SUSPICIOUS`, `very suspicious`, `VERY SUSPICIOUS`, and `FAIL`. (It may further print a varying number of exclamation points after the word "`FAIL`" but we chose to ignore those: failure is failure.) We relied on these nonnumerical descriptions in distilling the reports.



The distillation software for PractRand test runs distills a set of anomalies into a pair of integers (f, c) (a *failure level* and a *count*) in a manner not too different from the strategy used for BigCrush; these are then similarly reduced to a symbol and/or number. Again, the idea is that the worse the results, the more ink on the page.

5 TESTING THE SPLITMIX ALGORITHM

We tested the SplitMix algorithm using both BigCrush and PractRand, and using a range of gamma values. We also tested variants of the SplitMix algorithm that use other mixing functions. Our three goals: (a) comparing the strengths and weaknesses of BigCrush and PractRand as test suites, (b) comparing the strengths and weaknesses of the various mixing functions, and (c) determining whether the three classes of gamma values are indeed weak.

The BigCrush test suite is oriented toward testing double-precision floating-point values rather than 64-bit integers. Therefore we always used BigCrush in three different modes, indicated by the letters **f**, **r**, and **u**:

- **f** The high 53 bits of each generated 64-bit integer value is used to produce one double value.
- **r** The reverse of the low 53 bits of each generated 64-bit integer value is used to produce one double value.
- **u** Each generated 64-bit integer value is used to produce two double values by using first the low 32 bits, and then the high 32 bits, of the 64-bit integer value, adding 21 low-order 0-bits to each to make 53.

As it turned out, results were similar across all three modes. In this paper, we present data for only the **u** mode.

The PractRand test suite is oriented toward testing 64-bit integer values, and includes tests specifically designed to probe weakness in the low-order bits, so we used this test suite directly on the generated 64-bit values.

We considered mixing functions of this form:

```
ulonglong SplittableRandom_name_Mixer(ulonglong z) {
  z = (z ^ (z >> s1)) * m1;
  <optional rotation step>
  z = (z ^ (z >> s2)) * m2;
  return z ^ (z >> s3); }
```

name	s1	s2	s3	m1	m2
murmurhash3	33	33	33	0xff51afd7ed558ccdull	0xc4ceb9fe1a85ec53ull
lecuyer32	32	32	32	0x106689d45497fdb5ull	0x3b91f78bdac4c89dull
anneal32a	32	32	32	0xd6b0153091232c93ull	0x2e4df9428a87832dull
anneal32b	32	32	32	0xd753249aa6fce2c7ull	0xd9a9f6e3314b8bb5ull
anneal32c	32	32	32	0x387310ae4936362full	0xf9a9476328e05711ull
lea	32	32	32	0xdaba0b6eb09322e3ull	0xdaba0b6eb09322e3ull
stafford01	31	27	33	0x7fb5d329728ea185ull	0x81dadef4bc2dd44dull
stafford02	33	31	31	0x64dd81482cbd31d7ull	0xe36aa5c613612997ull
stafford03	31	30	33	0x99bcf6822b23ca35ull	0x14020a57acced8b7ull
stafford04	33	28	32	0x62a9d9ed799705f5ull	0xcb24d0a5c88c35b3ull
stafford05	31	29	30	0x79c135c1674b9addull	0x54c77c86f6913e45ull
stafford06	31	27	30	0x69b0bc90bd9a8c49ull	0x3d5e661a2a77868dull
stafford07	30	26	32	0x16a6ac37883af045ull	0xcc9c31a4274686a5ull
stafford08	30	28	31	0x294aa62849912f0bull	0x0a9ba9c8a5b15117ull
stafford09	32	29	32	0x4cd6944c5cc20b6dull	0xfc12c5b19d3259e9ull
stafford10	30	32	33	0xe4c7e495f4c683f5ull	0xfda871baea35a293ull
stafford11	27	28	32	0x97d461a8b11570d9ull	0x02271eb7c6c4cd6bull
stafford12	29	26	33	0x3cd0eb9d47532dfbull	0x63660277528772bbull
stafford13	30	27	31	0xbf58476d1ce4e5b9ull	0x94d049bb133111ebull
stafford14	30	29	31	0x4be98134a5976fd3ull	0x3bc0993a5ad19a13ull

Fig. 2. Parameters for mixing functions used while testing the SplitMix algorithm

We considered twenty sets of values for the parameters s1, s2, s3, m1, and m2, as shown in Fig. 2. These include the 64-bit MurmurHash3 mixing function (Appleby 2011) and the 14 mixing functions described by Stafford (2011). For each set of values, we used either no rotation step, or a left-rotation step (adding "rotl" to the name):

```
ulonglong lh = z & 0x00000000FFFFFFFFull;
ulonglong hh = z & 0xFFFFFFFF00000000ull;
z = hh | ((((lh << 32) | lh) << ((z >> 32) & 0x1f)) >> 32);
```

or a right-rotation step (adding "rotr" to the name):

```
ulonglong lh = z & 0x00000000FFFFFFFFull;
ulonglong hh = z & 0xFFFFFFFF00000000ull;
z = hh | ((((lh << 32) | lh) >> ((z >> 32) & 0x1f)) & 0x00000000FFFFFFFFull);
```

where in each of these rotation steps, the low 32 bits of z are rotated by an amount determined by the low 5 bits of the high 32 bits of z. (We were inspired by O'Neill (2014) to try the rotation variants.)

We also studied 18 other mixing functions that were produced by an alternate simulated annealing process. They are all similar in structure to the function shown above, but omit the shift-XOR step that uses s1. In addition, the five functions anneal1LLXX through anneal5LLXX add left-rotation steps both before and after the multiplication by m1; the six functions anneal1LXLX through anneal6LXLX add a left-rotation step before the multiplication by m1 and before the multiplication by m2; and the seven functions anneal1XLX through anneal7XLX add a left-rotation step only before the multiplication by m2. For every mixer with "anneal" in its name, the parameters s1 (if relevant), s2, s3, m1, and m2 were determined by a simulated-annealing search.

8000000000000001 = $(1/2)2^{64}$	1745D1745D1745D1 = $(1/11)2^{64}$	6DB6DB6DB6DB6DB7 = $(3/7)2^{64}$
5555555555555555 = $(1/3)2^{64}$	1555555555555555 = $(1/12)2^{64}$	9249249249249249 = $(4/7)2^{64}$
4000000000000001 = $(1/4)2^{64}$	1381381381381381 = $(1/13)2^{64}$	45D1745D1745D175 = $(3/11)2^{64}$
3333333333333333 = $(1/5)2^{64}$	1249249249249249 = $(1/14)2^{64}$	5D1745D1745D1745 = $(4/11)2^{64}$
2AAAAAAAAAAAAAAB = $(1/6)2^{64}$	1111111111111111 = $(1/15)2^{64}$	3B13B13B13B13B13 = $(3/13)2^{64}$
2492492492492493 = $(1/7)2^{64}$	1000000000000001 = $(1/16)2^{64}$	3813813813813813 = $(23/105)2^{64}$
2000000000000001 = $(1/8)2^{64}$	0F0F0F0F0F0F0F0F = $(1/17)2^{64}$	DB8DB8DB8DB8DB8D = $(3512/4095)2^{64}$
1C71C71C71C71C71 = $(1/9)2^{64}$	0E38E38E38E38E39 = $(1/18)2^{64}$	
1999999999999999 = $(1/10)2^{64}$	0D79435E50D79435 = $(1/19)2^{64}$	

Fig. 3. Gamma values for testing: Fractional group

0000000000000001	0000000400000001	0001001000000001	2000004000000001	5550000000000001
0000000000000003	0000000800000001	0001002000000001	2000008000000001	5555000000000001
0000000000000005	0000001000000001	0001004000000001	4000000100000001	5555500000000001
0000000000000009	0000002000000001	0001008000000001	4000000200000001	FFFF7F7FFFF7FFF7
0000000000FFFFFF	0000004000000001	0040000000000001	4000000400000001	FFFFFF0000000001
0000000008000001	0000008000000001	0040000008000001	4000000800000001	FFFFFF7FFFF7FFF7
0000000010000001	0000010000000001	2000000100000001	4000001000000001	FFFFFFFFFFF7FFF7
0000000020000001	0000FFFFFFFF0001	2000000200000001	4000002000000001	FFFFFFFFFFFFFFF7
0000000040000001	0001000100000001	2000000400000001	4000004000000001	FFFFFFFFFFFFFFFB
0000000080000001	0001000200000001	2000000800000001	4000008000000001	FFFFFFFFFFFFFFFD
0000000100000001	0001000400000001	2000001000000001	5000000000000001	FFFFFFFFFFFFFFFF
0000000200000001	0001000800000001	2000002000000001	5500000000000001	

Fig. 4. Gamma values for testing: Sparse group

5.1 The Main Tests

In the main tests (named `maintests`), we tested several different groups of gamma values. The first group (the *fractional group*, Fig. 3) contains values of the form $\frac{2^{64}}{k}$ or $\frac{j2^{64}}{k}$ for small integers k and j, with the low-order bit forced to be 1 (so that the gamma value will be odd, as required by the SplitMix algorithm). Note that all gamma values are given in hexadecimal. (The last two values, for which j and k are not very small after all, are control values included for testing purposes.)

The second group (the *sparse group*, Fig. 4) contains values that have relatively few `01` or `10` transitions.

The third group (the *shift group*, Fig. 5) contains values of the form $m(2^s + 1)$ for s ranging from 26 to 33.

The fourth group (the *control group*, Fig. 6) contains values thought to be "truly random"; they are 64-bit values obtained from HotBits (Walker 1996).

Sample results for BigCrush are shown in Figures 13 and 14. The X-axis lists mixing functions tested, and the Y-axis lists gamma values. In Figure 13 we see that indeed most of the mixers have trouble with the fractional group (except for the two control values $(23/105)2^{64}$ and $(3512/4095)2^{64}$, for which `murmurhash3` through `stafford14` are just fine). It is interesting that `stafford05`, `stafford06`, and `anneal1LXLX` through `anneal6LXLX` do well with all gamma values in the fractional group except those that are also sparse. All mixers have trouble with gamma values in the sparse group. As expected, the shift group does indeed cause problems for some of the mixers, including the principal one used in JDK8 (`murmurhash3`), but not for `stafford05`, `stafford06`, `stafford10`, `stafford11`, `stafford13`, `stafford14`, `anneal1LLXX` through `anneal5LLXX`, and `anneal1LXLX`

00000132D4004CB5	shift distance 26	0000132D40004CB5	shift distance 30
000003A09C00E827	shift distance 26	00003A09C000E827	shift distance 30
00000265A8004CB5	shift distance 27	0000265A80004CB5	shift distance 31
000007413800E827	shift distance 27	000074138000E827	shift distance 31
000004CB50004CB5	shift distance 28	00004CB500004CB5	shift distance 32
00000E827000E827	shift distance 28	0000E8270000E827	shift distance 32
00000996A0004CB5	shift distance 29	0000996A00004CB5	shift distance 33
00001D04E000E827	shift distance 29	0001D04E0000E827	shift distance 33

Fig. 5. Gamma values for testing: Shift group

0A18B8E7AC904503	9E13DEEA6A5D1D9B	BF56F43B89525AA1	63F304E7AA9C5BFD

Fig. 6. Gamma values for testing: Sparse group

5555555555555555 = $(1/3)2^{64}$	0F0F0F0F0F0F0F0F = $(1/17)2^{64}$	6DB6DB6DB6DB6DB7 = $(3/7)2^{64}$
3333333333333333 = $(1/5)2^{64}$	0D79435E50D79435 = $(1/19)2^{64}$	9249249249249249 = $(4/7)2^{64}$
2492492492492492 = $(1/7)2^{64}$	0C30C30C30C30C30 = $(1/21)2^{64}$	45D1745D1745D175 = $(3/11)2^{64}$
1C71C71C71C71C71 = $(1/9)2^{64}$	0B21642C8590B216 = $(1/23)2^{64}$	5D1745D1745D1745 = $(4/11)2^{64}$
1745D1745D1745D1 = $(1/11)2^{64}$	0A3D70A3D70A3D70 = $(1/25)2^{64}$	3B13B13B13B13B13 = $(3/13)2^{64}$
1381381381381381 = $(1/13)2^{64}$	097B425ED097B425 = $(1/27)2^{64}$	4EC4EC4EC4EC4EC4 = $(4/13)2^{64}$
1111111111111111 = $(1/15)2^{64}$	08D3DCB08D3DCB08 = $(1/29)2^{64}$	

Fig. 7. Gamma values for variant testing are $((g \pm 2^s) \mid 1)$ where g is a value in this table

through `anneal6LXLX`. Mixers `murmurhash3` through `stafford14` do perfectly well with the control group. Note that these two charts contains areas for which data files were missing or malformed; we chose not to spend machine time to fill in these blanks because we had already learned what we wanted to know: we had confirmed that all three conjectured classes of weak gamma values can cause `SplitMix` to fail the BigCrush test suite. (Partial results from PractRand provide further confirmation; see Figures **??** and **??** in Appendix A.)

5.2 The Variant Tests

The variant tests (named `variants`) probe the behavior for gamma values surrounding values of the form $\frac{2^{64}}{k}$ or $\frac{j2^{64}}{k}$ for small integers k and j. For each value g in Fig. 7, the variant tests include a set of gamma values of the form $((g \pm 2^s) \mid 1)$ for $1 \leq s \leq 8$, to test a conjecture that the weakness of gamma values declines as they become more distant from members of the fractional group.

Sample results for BigCrush are shown in Figure 19, and corresponding results for PractRand are shown in Figure 20, both in Appendix A. The predicted effect was indeed observed, and test results were worse for values surrounding fractions with very small denominators k. The PractRand results also suggest that while fractional gamma values and values near them can be problematic, they are not as bad as sparse gamma values: PractRand does not declare failure for fractional gamma values such as in Fig. 3 until over 128 GB of generated values have been tested, and does not declare failure for nearby variant values until 4 TB of generated values have been tested.

5.3 The Round-Robin Tests

There are two sets of *round-robin* tests (named `rr8d` and `rr8c`). In each set of tests, rather than using just one instance of the SplitMix algorithm, eight instances are used, and their results are used in round-robin fashion; that is, the eight output streams are interleaved to produce a single output stream, with each instance contributing every eighth result. The *distant* round-robin tests `rr8d` use eight instances whose gamma values are actually generated randomly (using the given putative "gamma value" as a seed) and therefore should be quite distant from each other in the sense of Hamming distance; that is, any two of them differ in many bit positions. The *close* round-robin tests `rr8c` use eight instances whose initial seed values are identical and whose gamma values are identical except for at most three bits; the eight gamma values are generated from the given gamma value by systematically replacing the three bits corresponding to the 1-bits in the mask `000000000000000E` with all eight possible patterns for those three bits.

Sample results for PractRand are shown in Figures 21 and 22, both in Appendix A. Perhaps unsurprisingly, no problems are found with the distant round-robin tests. The close round-robin tests fail in some cases—though not all—when the given gamma value is already weak, but never fail when the given gamma value is not weak.

5.4 Conclusions Regarding the SplitMix Algorithm

We have confirmed that there are three distinct classes of weak gamma values for the SplitMix algorithm (and, as we indicated in footnote refnote:weak, speculated that there is a fourth). They are easily defended against: if multiplying the proposed gamma value by any odd integer (including 1) that is smaller than (say) 32 causes it either to be sparse (having fewer than 24 `01` and `10` transitions) or to have the property that the first shift-and-xor step of the mixing function causes too many (say more than $\frac{3s}{4}$) of the s low-order bits to be zero, then reject that candidate and try again. Doing this may ensure that each individual instance of SplitMix produces a good pseudorandom sequence; but it does have some cost.

The round-robin tests give some confidence that a modest collection of SplitMix instances with randomly chosen gamma values is very likely to behave as if they were statistically independent. However, because there are "only" 2^{63} distinct choices of gamma value, if the size of the collection is very large, the likelihood of all their gamma values being distinct may not be comfortably small.

6 THE TWINLINEAR ALGORITHM

At this point in our investigations we reasoned that perhaps there would be less burden on the mixing function if a better initial sequence generator were used. We took inspiration from two sources: from the MRG32k3a algorithm (Fischer et al. 1999) we took the idea of using multiple linear congruential generators as initial sources, and from pcg (O'Neill 2014) we borrowed the idea of using rotate instructions as a part of nonlinear mixing.

The idea behind the TwinLinear framework is to use just two linear congruential generators and mix their outputs. It is well known that the low-order bits of lcg output are not very random, but the high-order bits are fairly good. Therefore we first mix the two outputs by using bitwise xor to combine them *after* rotating one of them so as to swap its halves. The second step is to rotate this result by a "random" distance determined by the highest-order bits of one lcg output. The third step is to further mix the bits of this rotated result by using a multiply step and a shift-and-xor step.

The specific case that we have tested thoroughly uses 64-bit arithmetic. It has 254 bits of internal state in the form of four 64-bit integers, two of which are required to be odd. We will call the four 64-bit integers s1, s2, g1, and g2; g1 and g2 must be odd. Once values have been chosen for s1 and s2 and g1 and g2 for any instance of the prng, s1 and s2 represent mutable state that may be altered whenever a *generate* or *split* operation is performed, but g1 and g2, once chosen, are unchanging for that instance. Thus, as with SplitMix, one may regard instances of a TwinLinear class as members of a prng *family*, distinguished by parameters g1 and g2.

```java
class TwinLinear {
  private long s1, s2; private final long g1, g2;             // ‘g1’ and ‘g2’ must be odd
  public TwinLinear(long s1, long s1, long g1, long g2) {
    this.s1 = s1; this.s2 = s2; this.g1 = g1 | 1; this.g2 = g2 | 1; }
  public long nextLong() {
    long r = Long.rotateLeft(s1, 32) ^ s2; long t = s1 >>> 58;
    r = Long.rotateLeft(r, (int)t); r *= 2685821657736338717L;
    s1 = s1 * 3202034522624059733L + g1; s2 = s2 * 3935559000370003845L + g2;
    return (r ^ (r >>> 32)); }
  private static final double DOUBLE_ULP = 1.0 / (1L << 53);
  public double nextDouble() { return (nextLong() >>> 11) * DOUBLE_ULP; }
  public TwinLinear split() {
    return new TwinLinear(nextLong(), nextLong(), nextLong(), nextLong()); }
}
```

Fig. 8. Java code for the TwinLinear algorithm (with the RXRRMX mixer)

In addition, we make use of three 64-bit fixed integer constants a1, a2, and a3 that are identical for all instances. For these we have chosen (guided by the tables provided by L’Ecuyer (1999)) the values

a1 = 3202034522624059733   a2 = 3935559000370003845   a3 = 2685821657736338717

The TwinLinear algorithm uses two distinct linear congruential generators, one whose state is s1 with parameter g1, and one whose state is s2 with parameter g2. For every *generate* step of the overall TwinLinear algorithm, each of the two linear congruential generators is used to generate a 64-bit value, and then the two 64-bit values are mixed to produce a single 64-bit result.

The overall technique for performing a *generate* operation is described by this pseudocode:

⟨1⟩ r := (s1 ROTATELEFT 32) XOR s2
⟨2⟩ r := r ROTATELEFT (s1 SHIFTRIGHT 58)
⟨3⟩ r := r × a3
⟨4⟩ s1 := (a1 × s1 + g1) mod 2^{64}
⟨5⟩ s2 := (a2 × s2 + g2) mod 2^{64}
⟨6⟩ return (r XOR (r SHIFTRIGHT 32))

Line ⟨4⟩ advances the state of the first linear congruential generator; line ⟨5⟩ likewise advances the state of the second linear congruential generator. These are placed *after* the uses of s1 and s2 so that their execution may be overlapped or interleaved with other computation. Line ⟨1⟩ combines s1 and s2 *nonlinearly* by first permuting the bits of s1 (using a ROTATELEFT operation) and then using a bitwise XOR operation on the result of the ROTATELEFT operation and s2, to produce a new value r. Line ⟨2⟩ performs a further nonlinear combination step by using the high 6 bits of s1 (obtained by using a SHIFTRIGHT operation on s1 for a distance of 58 bit positions) to determine a number of bit positions by which r is to be rotated by a ROTATELEFT operation. Lines ⟨3⟩ and ⟨6⟩ accomplish a final mixing on the value of r by first multiplying it by a3 and then performing a standard *xorshift* step. Note that line ⟨6⟩ contains two operations XOR and SHIFTRIGHT, but on architectures that allow direct addressing of the two halves of a 64-bit register as if they were two 32-bit registers, it may be possible to implement the computation in line ⟨6⟩ as a single 32-bit XOR instruction.

Java code for the TwinLinear algorithm appears in Figure 8, which may be compared with Figure 1. The `split` operation simply performs four `nextLong` operations to obtain four 64-bit integers, then calls the constructor (which forces the last two integers to be odd by using a bitwise OR operation with the integer constant 1).

twoRXR	swap halves of x, XOR x into y, rotate y by x
twoRXRR	XOR swapped halves of x into y, shift x right by 58, rotate y by x
twoXRXR	XOR y into x, swap halves of x, XORx into y, rotate y by x
twoXRXRR	XOR y into x, XOR swapped halves of x into y, shift x right by 58, rotate y by x
twoRARR	add swapped halves of x into y, shift x right by 58, rotate y by x
twoXRARR	XOR y into x, add swapped halves of x into y, shift x right by 58, rotate y by x
twoARXRR	add y into x, XOR swapped halves of x into y, shift x right by 58, rotate y by x
twoARARR	add y into x, add swapped halves of x into y, shift x right by 58, rotate y by x
twoRXRRM	XOR swapped halves of x into y, shift x right by 58, rotate y by x
twoXRXRRM	XOR y into x, XOR swapped halves of x into y, shift x right by 58, rotate y by x, multiply y by a3
twoRARRM	add swapped halves of x into y, shift x right by 58, rotate y by x, multiply y by a3
twoXRARRM	XOR y into x, add swapped halves of x into y, shift x right by 58, rotate y by x, multiply y by a3
twoARXRRM	add y into x, XOR swapped halves of x into y, shift x right by 58, rotate y by x, multiply y by a3
twoARARRM	add y into x, add swapped halves of x into y, shift x right by 58, rotate y by x, multiply y by a3
two6XR	rotate x left by 6, XOR x into y, rotate y by x
two6AR	rotate x left by 6, add x into y, rotate y by x
twoVXR	reverse bits of x, XOR x into y, rotate y by x
twoVAR	reverse bits of x, add x into y, rotate y by x
twoVXF	reverse bits of x, XOR x into y, bitflip y by x
twoVAF	reverse bits of x, add x into y, bitflip y by x
twoRX6F	XOR swapped halves of x into y, shift x right by 58, bitflip y by x
twoRA6F	add swapped halves of x into y, shift x right by 58, bitflip y by x
twoSXRR	XOR high half of x into low half of y, shift x right by 58, rotate y by x
twoSIRR	copy high half of x into low half of y, shift x right by 58, rotate y by x
twoSZRR	XOR high half of y into low half of y, shift x right by 58, rotate y by x
twoSXRRM	XOR high half of x into low half of y, shift x right by 58, rotate y by x, multiply y by a3
twoRXRRMX	XOR swapped halves of x into y, shift x right by 58, rotate y by x, multiply y by a3, XOR high half of y into low half of y
twoSXRRMX	XOR high half of x into low half of y, shift x right by 58, rotate y by x, multiply y by a3, XOR high half of y into low half of y
twoRRMX	shift x right by 58, rotate y by x, multiply y by a3, XOR high half of y into low half of y
twoRRNX	shift x right by 58, rotate y by x, multiply y by a4, XOR high half of y into low half of y
twoR4XRR	XOR (x rotated left by 4) into y, shift x right by 58, rotate y by x
twoR8XRR	XOR (x rotated left by 8) into y, shift x right by 58, rotate y by x
twoR12XRR	XOR (x rotated left by 12) into y, shift x right by 58, rotate y by x
⋮	⋮
twoR24XRR	XOR (x rotated left by 24) into y, shift x right by 58, rotate y by x
twoR28XRR	XOR (x rotated left by 28) into y, shift x right by 58, rotate y by x
twoR36XRR	XOR (x rotated left by 36) into y, shift x right by 58, rotate y by x
twoR40XRR	XOR (x rotated left by 40) into y, shift x right by 58, rotate y by x
⋮	⋮
twoR56XRR	XOR (x rotated left by 56) into y, shift x right by 58, rotate y by x
twoR60XRR	XOR (x rotated left by 60) into y, shift x right by 58, rotate y by x

Fig. 9. Mixing functions used while testing the TwinLinear algorithm

The `twopair` tests use seed `0xPPQNRSXXXXXYYYYY` in a complex way to initialize two TwinLinear instances:
Low bit of `YYYYY` indicates which of two initial state values to use for both generators in both instances.
`XXXXX | 1` initializes g1 for the first LCG of each instance.
`YYYYY | 1` initializes g2 for the second LCG of each instance.
If high bit of `Q` is `1`, multiply both these values by `1181783497276652981`, just to get nonzero bits up high.
Low bits of `Q` specify perturbation of one or both gamma values, or of a state value, for second instance only:

- `Q = 0` perturb first gamma value (rotation distance is `2R+1`)
- `Q = 1` perturb second gamma value (rotation distance is `2R+1`)
- `Q = 2` perturb both gamma values in same way (rotation distance is `2R+1`)
- `Q = 3` perturb both gamma values but in different ways (rotation distances are `2R+1` and `2R+9`)
- `Q = 4` perturb first gamma value (rotation distance is `PP`)
- `Q = 5` perturb second gamma value (rotation distance is `PP`)
- `Q = 6` perturb first state value (rotation distance is `PP`)
- `Q = 7` perturb second state value (rotation distance is `PP`)

(To "perturb" is to invert `N` bits, namely those at positions ($i \times \langle$rotation distance$\rangle$) mod 64 for $1 \le i \le$ `N`.)
`S` indicates how to jump forward the state values of the generators of the second instance only:

- `S = 0` jump first state value by `2**PP` steps
- `S = 1` jump second state value by `2**PP` steps
- `S = 2` jump both state values by `2**PP` steps
- `S = 3` jump both state values but in different ways (first by `2**PP` steps, second by `2**(PP+2)` steps)
- `S = 4` do not jump either state value
- `S = 5` jump first state value by `2**PP + R` steps
- `S = 6` jump second state value by `2**PP + R` steps
- `S = 7` jump both state values by `2**PP + R` steps
- `S = 8` jump first state value by `PP` steps
- `S = 9` jump second state value by `PP` steps
- `S = A` jump both state values by `PP` steps
- `S = B` jump both state values but in different ways (first by `PP` steps, second by `PP+2` steps)
- `S = C` unused
- `S = D` jump first state value by `R` steps
- `S = E` jump second state value by `R` steps
- `S = F` jump both state values by `R` steps

Typically $1 \le$ `N` ≤ 15 and $0 \le$ `P` ≤ 19; all combinations of `P` and `Q` and `R` thus produce 1200 tests.

Fig. 10. How states of two instances of `TwinLinear` are initialized for `twopair` tests

7 POSSIBLE WEAKNESSES OF THE TWINLINEAR ALGORITHM

As for the SplitMix algorithm, we tried to predict analytically possible weaknesses of the TwinLinear algorithm so that we could focus additional testing on such cases.

Qualitatively speaking, the SplitMix algorithm consists of an extremely weak (but very fast) sequence generator followed by a mixing function that needs to be strong enough to compensate. In contrast, the TwinLinear algorithm begins with two generators each of which is known to be fairly good—or at least "not completely terrible." Our particular choices of multipliers a1 and a2 produce two linear congruential generators with very good spectral properties, no matter what additive values g1 and g2 are used. There is good reason to believe that no matter what values of g1 and g2 are chosen and no matter where in the state cycle each of the two generators

The `twotwin` tests use seed `0xPQRSTUVVWWXXXYYY` in a complex way to initialize two TwinLinear instances:
`XXX | 1` initializes `g1` for both instances, and `YYY | 1` initializes `g2` for both instances.
If low bit of `XXX` is `1`, multiply `g1` for both instances by `1181783497276652981` (make high bits nonzero).
If low bit of `YYY` is `1`, multiply `g2` for both instances by `2685821657736338717` (make high bits nonzero).
There is a fixed 16-entry "`PS` table"; the first entry is `0`, and the others are random 64-bit integers from Hotbits.
There is a fixed 16-entry "`QT` table"; the first entry is `0`, and the others are random 64-bit integers from Hotbits.
`P` selects one of 16 values for `s1` for both instances from the `PS` table.
`Q` selects one of 16 values for a temporary value `M` from the `QT` table.
Bit `VV` of `M` is set to `1`, and all bits of `M` below that point are cleared. (If `VV` ≥ 64, then `M` becomes 0.)
`S` selects one of 16 values for `s2` for both instances from the `PS` table.
`T` selects one of 16 values for a temporary value `N` from the `QT` table.
Bit `WW` of `N` is set to `1`, and all bits of `N` below that point are cleared. (If `WW` ≥ 64, then `N` becomes 0.)
For the second instance only, `s1` is jumped forward `M+R` steps, and `s2` is jumped forward `N+U` steps.

Fig. 11. How states of two instances of `TwinLinear` are initialized for `twotwin` tests

is started, if one regards only the 32 high-order bits of each generated value, the two generated sequences will appear to be fairly uncorrelated. But it's not quite good enough just to take the high-order 32 bits from each generator and concatenate them to form 64-bit values. Therefore we also use a mixing function. Testing as shown that the particular one we have chosen appears to be quite good, and we have found no weaknesses in sequences generated by a single instance of TwinLinear, no matter what the choice of parameter values g1 and g2.

But if we consider using multiple instances of the TwinLinear algorithm together—for example, using two such instances and interleaving their outputs—there are additional opportunities for correlation. It might be that two such generators have similar initial states; but if their g1 and g2 values differ, then their states will soon become quite dissimilar. But suppose that two such instances have the same value of g1, or the same value of g2, or both? Let the initial state values be s1a and s2a (for the first instance of TwinLinear) and s1b and s2b (for the second instance). If the distance n1 between s1a and s1b along the state cycle of the first LCG is of the form $(2m_1 + 1)2^k_1$, then the low-order k_1 bits of s1a and s1b will always be the same, and similarly for s2a and s2b. Therefore we conjectured that the mixing function, if it is "barely good enough" in most cases, might encounter trouble in the case of interleaving the output of two instances with the same g1 and g2, with that trouble increasing as either k_1 or k_2 increases. We set out to test this conjecture.

8 TESTING THE TWINLINEAR ALGORITHM

We tested the general TwinLinear framework using both BigCrush and PractRand, using a number of distinct mixing functions, and using a variety of techniques for initializing the two states and two gamma values from a variety of 64-bit seeds. Again we had three goals in mind: comparing the strengths and weaknesses of BigCrush and PractRand as test suites, comparing the strengths and weaknesses of the various mixing functions, and trying to predict poor behavior and perhaps uncover unexpected poor behavior.

Brief characterizations of the mixing functions we used in our tests of TwinLinear are shown in Fig. 9 (each mixes a value x into a second value y, which becomes the result). Initial tests showed that each proposed mixer was either "good enough" or "really bad" (an example chart is Fig. 23 in Appendix A). Further, more intensive testing (both single-instance and round-robin) led us to focus on mixer `twoRXRRMX`.

We set up a series of tests (called "`twopair`") that would use interleaved output from two instances of TwinLinear using the `twoRXRRMX` mixer, with the initial state of the two instances governed by a set of parameters that could be packed into a single 64-bit integer command-line argument (see Fig. 10). This complex encoding allowed us to explore the extent to which the test suites would detect biases in the output if the initial states and/or gamma parameters of the two instances were correlated in various ways. (An example chart is Fig. 24 in Appendix A.) We were able to get the PractRand tests to fail only for cases where the two instances had identical `gamma` values for their first LCG *and* identical `gamma` values for their second LCG.

It was at this point that we formulated the conjecture presented in the previous section. We set up another series of tests (called "`twotwin`") that would, again, use interleaved output from two instances of TwinLinear using the `twoRXRRMX` mixer, but exploring a different initial states, specifically to test the conjecture that if the gamma values are identical, then output quality depends on the relative phases of the corresponding LCG states. The initial state of the two instances is governed by a different set of parameters packed into a single 64-bit integer command-line argument (see Fig. 11). The results confirm the conjecture. See, for example, Fig. 15 and 16; their X-axes together span the complete range 63–0 of possible numbers `VV` of trailing zeroes in `n1`, and their Y-axes describe various initializations. The horizontal black bands reflect precisely those cases in which corresponding gamma values are identical, and it may be observed that the the failures reported by BigCrush become less severe as one progresses from left to right. (In Appendix A, similar results for `n2` are shown in Fig. 25 and 26. Corresponding results reported by PractRand for the same test cases appear in Fig. 27, 28, 29, and 30.)

9 MISCELLANEOUS OBSERVATIONS

In hopes of constructing a somewhat smaller and more efficient version of TwinLinear, we have also studied a variant which we call SmallBig; it makes use of one 32-bit LCG and one 64-bit LCG. However, we have not yet found a version of SmallBig that is sufficiently robust when tested. We conjecture that something special happens when going from 32-bit arithmetic to 64-bit arithmetic: when the game is to satisfy BigCrush and PractRand, perhaps a 32-bit LCG really isn't "random enough" to succeed with this approach, but a 64-bit LCG is.

We also studied a large set of mixing functions similar to those already described, but in which a rotation step whose distance is determined by some part of the state is replaced with a flip step. In a rotation of 2^n bits by distance s, bit i is moved to position $(i + s) \bmod 2^n$; in a "flip" of a group of 2^n bits by parameter s, bit i is moved to position i XOR s. What is the value of a rotation step? O'Neill (2014, §6.3.3) suggests that a random rotation "ensures that all the bits are full period"; a simpler intuition is that a random rotation gives every result bit an equal chance of being drawn from every bit position of the input. By there criteria, a flip should be every bit as good as a rotation, and yet our testing shows that mixing functions that use flips are in every case much worse than otherwise identical mixing functions that use rotations. We hope that future work may illuminate why.

We conducted yet another group of tests (called "`twomulti`") capable of interleaving the outputs of up to 256 instances of TwinLinear, and did a large number of runs that interleaved 16 instances. We omit details for lack of space, but remark that the results were consistent with the model that instances behave as if statistically independent if they differ in at least one of the two `gamma` parameters.

We expended just over a thread-century of computing time running TestU01 and PractRand. A breakdown appears in Fig. 12. Over a quarter of the time was spent on the `twotwin` tests that, on the one hand, characterize correlations between TwinLinear instances whose corresponding `gamma` parameters are identical and, on the other hand, appear to confirm that differing in at least one `gamma` parameter confers statistical independence.

10 OTHER RELATED WORK

We refer the reader to the Related Work section provided by Steele, Lea, and Flood (2014) for an overview of related work prior to theirs.

O'Neill (2014) describes the PCG (Permuted Congruential Generator) family of PRNG algorithms. The basic idea is to start with a (single) Linear Congruential Generator, then apply a mixing function, typically consisting of two or three steps, each step being either shift-and-XOR, multiplication by a carefully chosen odd constant, or a bit-rotation of some part of the state by a distance specified by some other part of the state.

TestU01	`maintests`	10.45 yr	PractRand	`maintests`	6.33 yr	PractRand	`twopair`	17.06 yr
TestU01	`variants`	15.70 yr	PractRand	`variants`	2.80 yr	PractRand	`twopair_alt`	0.22 yr
TestU01	`twotests`	9.21 yr	PractRand	`twotests`	7.88 yr	PractRand	`twotwin`	27.71 yr
TestU01	`smallbig`	1.04 yr	PractRand	`smallbig`	0.16 yr	PractRand	`twomulti`	5.79 yr

Grand total CPU time used: 104.35 years

Fig. 12. CPU time (in years) used for PRNG testing (summed over all threads, typically 384 to 512 running simultaneously)

Vigna (2016b) provides a comparison of the quality and speed of a number of recent PRNG algorithms, including (64-bit) SplitMix, PCG, and three xorshift-style algorithms that he has developed, building on the work of Marsaglia (2003): `xorshift1024*` (Vigna 2016a), `xorshift128+` (Vigna 2017), and `xoroshiro128+`, all of which are very fast (but, like all xorshift generators, have a period that is 1 less than a power of 2, and therefore the value 0 is delivered with slightly less probability than any other value). All three do quite well when tested with BigCrush. The `xorshift1024*` algorithm has period 2^{1024}; like most PRNG algorithms with that much state, it uses array indexing to access the state. The `xorshift128+` algorithm uses just 128 bits of state; its source code is written in terms of array indexing, which might prevent a C compiler from keeping the state in registers, but because all indices used are constants (`0` or `1`), it could easily be rewritten to avoid use of array indexing in the same manner as `TwinLinear`. The `xoroshiro128+` algorithm is similar to `xorshift128+`, but uses even fewer operations, and two of them are rotates rather than shifts. All three are provided with jump functions that can quickly advance the state by approximately the square root of the period (2^{512} steps for `xorshift1024*`, 2^{64} steps for `xorshift128+` or `xoroshiro128+`), making it easy to create many instances that traverse different parts of the cycle for use by multiple threads; however, no `split` operation is provided.

11 CONCLUSIONS AND FUTURE WORK

At the end of their paper, Steele, Lea, and Flood (2014) commented: "It would be a delightful outcome if, in the end, the best way to split off a new PRNG is indeed simply to 'pick one at random.'" Perhaps we have now achieved that: our testing suggests that if the four arguments to the `TwinLinear` constructor are themselves chosen uniformly at random—with no need to filter out any "weak values"—then the interleaved outputs of two generators constructed in this way will pass the TestU01 BigCrush test suite (L'Ecuyer and Simard 2007; Simard 2009) and also the PractRand test suite (Doty-Humphrey 2011) with probability exceeding $1 - 2^{126}$. (This is well in excess of the sensitivity of the test suites themselves.) More specifically, we conjecture from testing (but this has not been proven mathematically) that the interleaved output of two such generators ought *always* pass the test suites if either their g1 values differ or their g2 values differ (or both).

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

The SplitMix algorithm used in JDK8 has 127 bits of state (of which 64 are updated per 64 bits generated) and uses 9 arithmetic operations per 64 bits generated, but the TwinLinear algorithm uses 254 bits of state (of which 128 are updated per 64 bits generated) and uses 11 arithmetic operations (or possibly 10, on some architectures) per 64 bits generated. For applications in which it is desired to have a significantly smaller probability of statistical correlations among multiple generators being used by parallel tasks, especially when it is desirable to create new generator instances on the fly (for example, when forking new threads), TwinLinear may be very attractive.

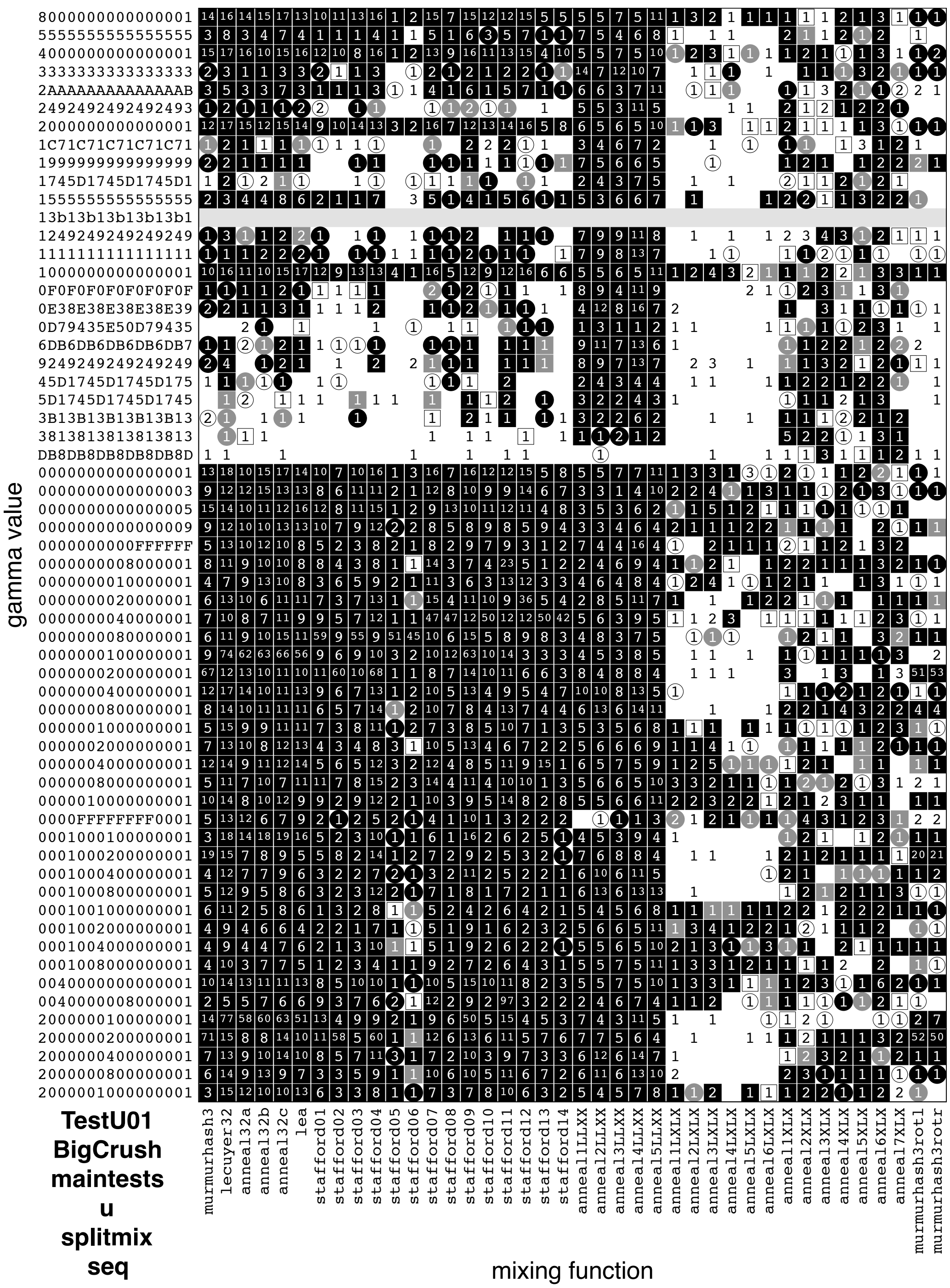


Fig. 13. TestU01_maintests_u_splitmix_Seq_graph_0

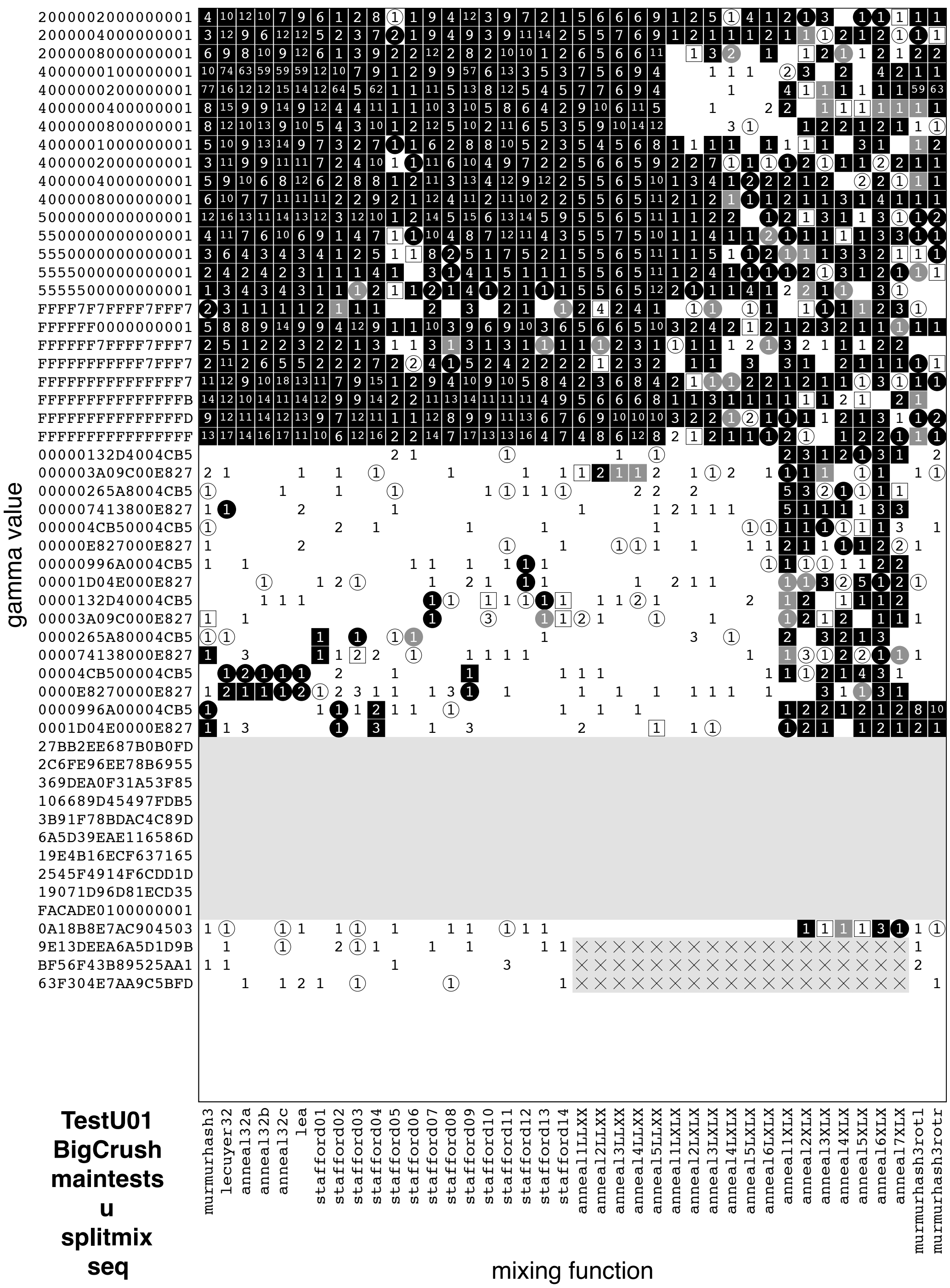


Fig. 14. `TestU01_maintests_u_splitmix_Seq_graph_1`

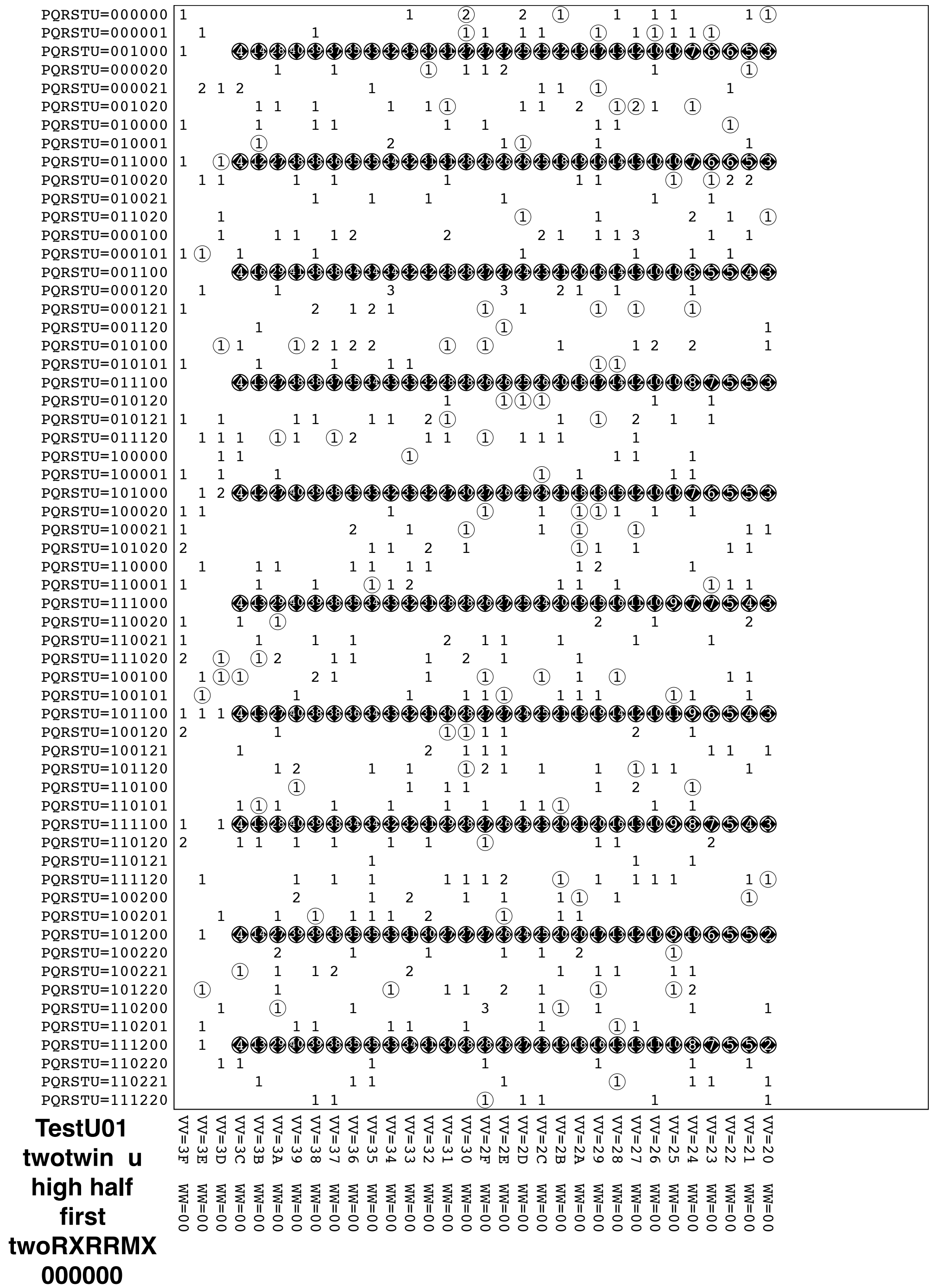


Fig. 15. TestU01_twotwin_u_twolcg_twoRXRRMX_0_twin__000000_graph_0

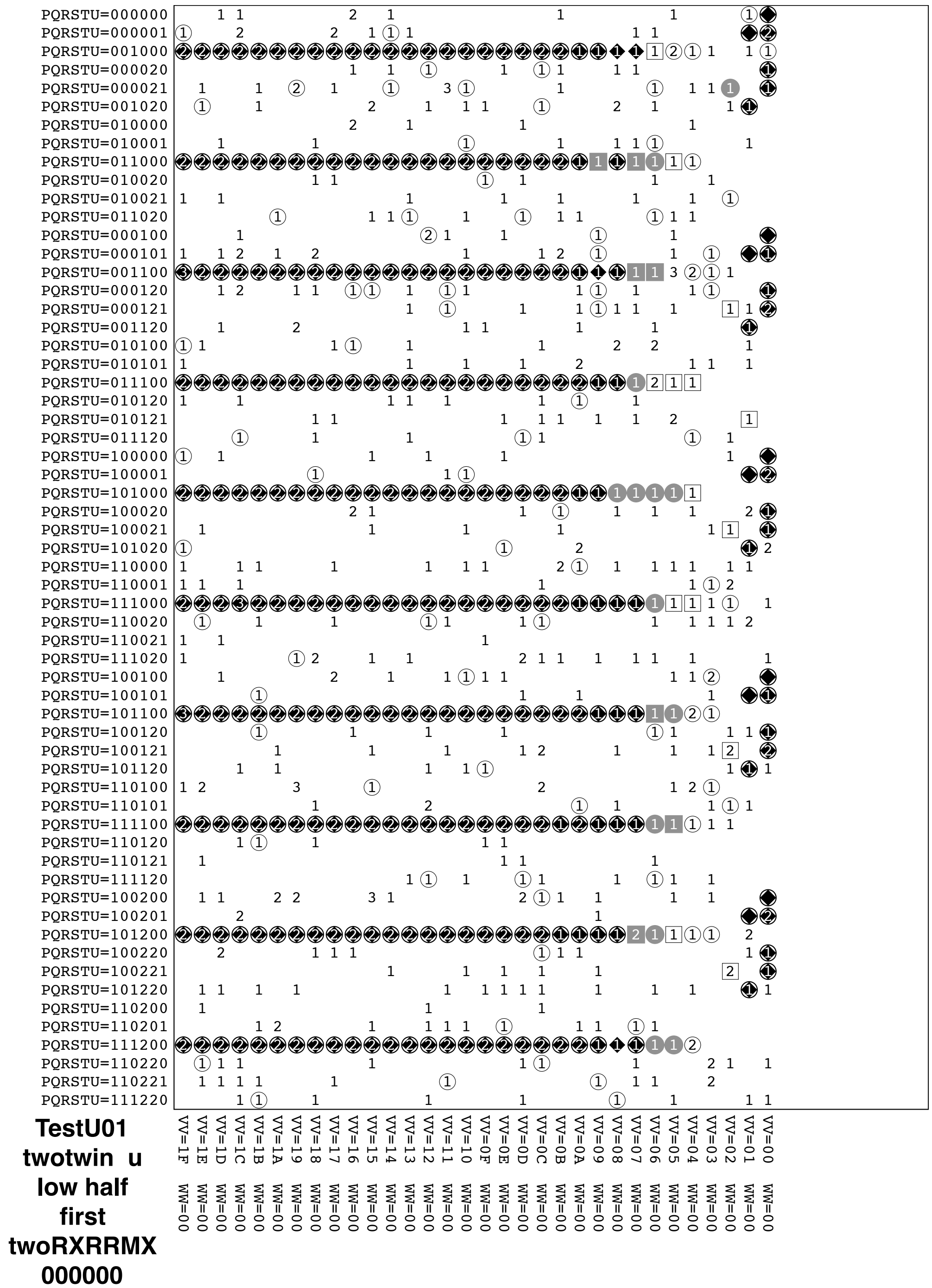


Fig. 16. TestU01_twotwin_u_twolcg_twoRXRRMX_0_twin__000000_graph_1

REFERENCES

Austin Appleby. 2011. MurmurHash3. (3 April 2011). Project wiki entry. http://code.google.com/p/smhasher/wiki/MurmurHash3 Accessed Sept. 10, 2013.

Robert G. Brown, Dirk Eddelbuettel, and David Bauer. 2003–2006. Dieharder: A Random Number Test Suite, version 3.31.1. (2003–2006). http://www.phy.duke.edu/~rgb/General/dieharder.php Accessed Sept. 10, 2013.

Chris Doty-Humphrey. 2011. PractRand version 0.92. (2011). Website at http://pracrand.sourceforge.net Accessed April 14, 2017. Undated; the year 2011 for its first appearance has been inferred from external sources. Yes, the software is called "PractRand" but the SourceForge project name is "pracrand".

Gregory W. Fischer, Ziv Carmon, Dan Ariely, Gal Zauberman, and Pierre L'Ecuyer. 1999. Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Operations Research* 47, 1 (Jan. 1999), 159–164. DOI:http://dx.doi.org/10.1287/opre.47.1.159

Guy L. Steele Jr., Doug Lea, and Christine H. Flood. 2014. Fast Splittable Pseudorandom Number Generators. In *OOPSLA '14: Proc. 2014 ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications.* ACM, New York, 453–472. DOI:http://dx.doi.org/10.1145/2660193.2660195

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

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

Pierre L'Ecuyer and Richard Simard. 2013. TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators: User's guide, compact version. (May 2013). http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf Accessed April 13, 2017.

George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14 (4 Jul. 2003), 1–6. http://www.jstatsoft.org/v08/i14

Melissa E. O'Neill. 2014. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. (2014). Harvey Mudd College. Undated, unpublished paper; the year 2014 for its first appearance has been inferred from external sources. http://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf Accessed April 11, 2017.

Oracle Corporation. 2016. Java Platform Standard Edition 8 Documentation: Class SplittableRandom. (2016). https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html Accessed April 13, 2017.

Richard Simard. 2009. TestU01 version 1.2.3. (Aug. 2009). Website at http://simul.iro.umontreal.ca/testu01/tu01.html Accessed April 13, 2017.

David Stafford. 2011. Better Bit Mixing: Improving on MurmurHash3's 64-bit Finalizer. (28 Sept. 2011). Blog "Twiddling the Bits." http://zimbry.blogspot.com/2011/09/better-bit-mixing-improving-on.html Accessed Sept. 10, 2013.

Sebastiano Vigna. 2016a. An Experimental Exploration of Marsaglia's Xorshift Generators, Scrambled. *ACM Trans. Math. Software* 42, 4, Article 30 (June 2016), 23 pages. DOI:http://dx.doi.org/10.1145/2845077

Sebastiano Vigna. 2016b. xoroshiro+ / xorshift* / xorshift+ generators and the PRNG shootout. (2016). Website at http://xoroshiro.di.unimi.it Accessed April 14, 2017.

Sebastiano Vigna. 2017. Further scramblings of Marsaglia's xorshift generators. *J. Comput. Appl. Math.* 315 (2017), 175–181. DOI:http://dx.doi.org/10.1016/j.cam.2016.11.006

John Walker. 1996. HotBits: Genuine random numbers, generated by radioactive decay. (May 1996). http://www.fourmilab.ch/hotbits/ Accessed April 13, 2017.

APPENDIX

In this Appendix we present additional measurement charts and details about processes used to distill reports for both TestU01 BigCrush and PractRand test runs.

A ADDITIONAL MEASUREMENT CHARTS

On the following pages we present 14 additional measurement charts. They will not all fit into the final conference paper, but we wish to allow reviewers to check our summary descriptions of what we have measured.

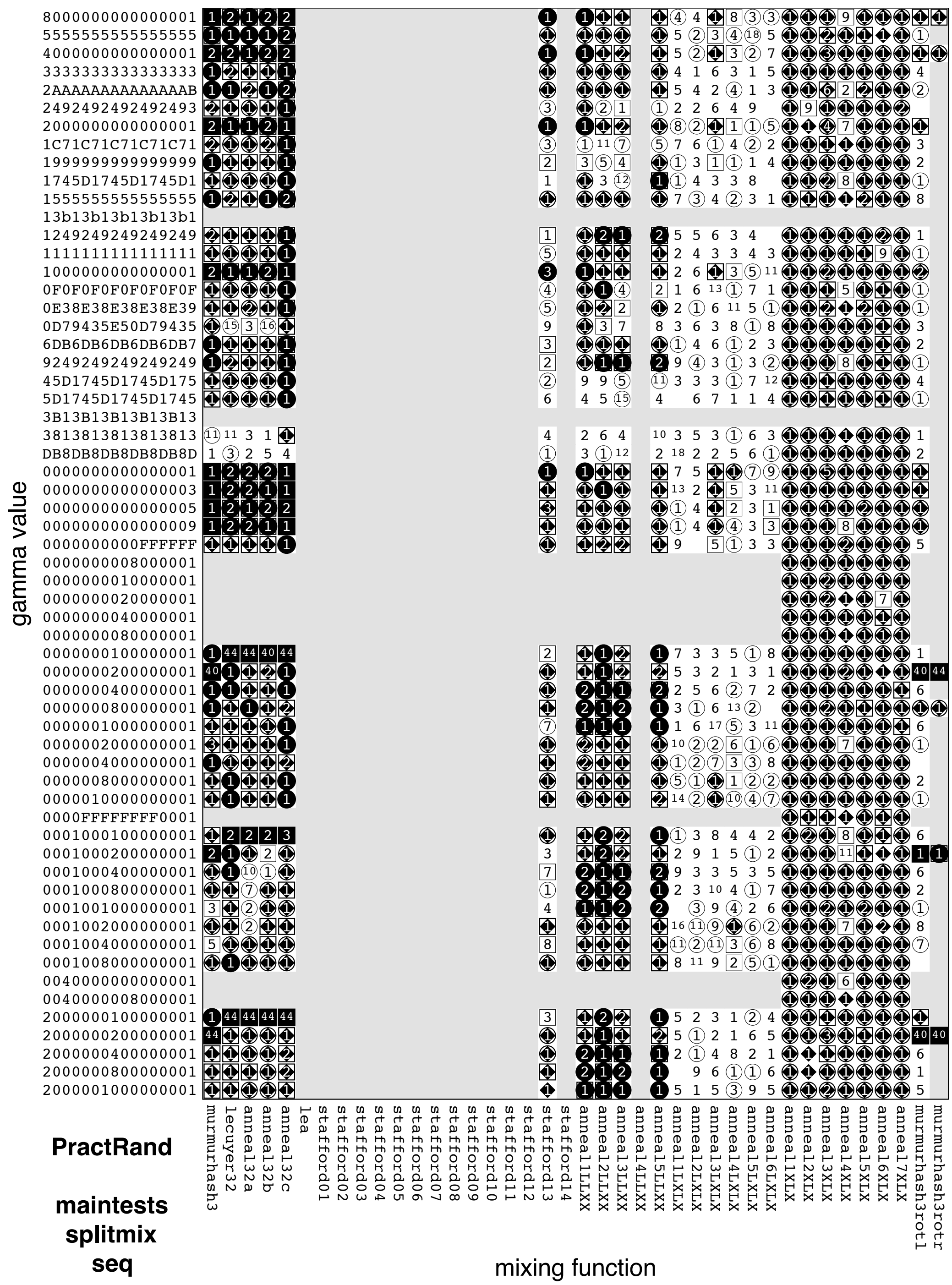


Fig. 17. PractRand_maintests_splitmix_Seq_graph_0.pdf

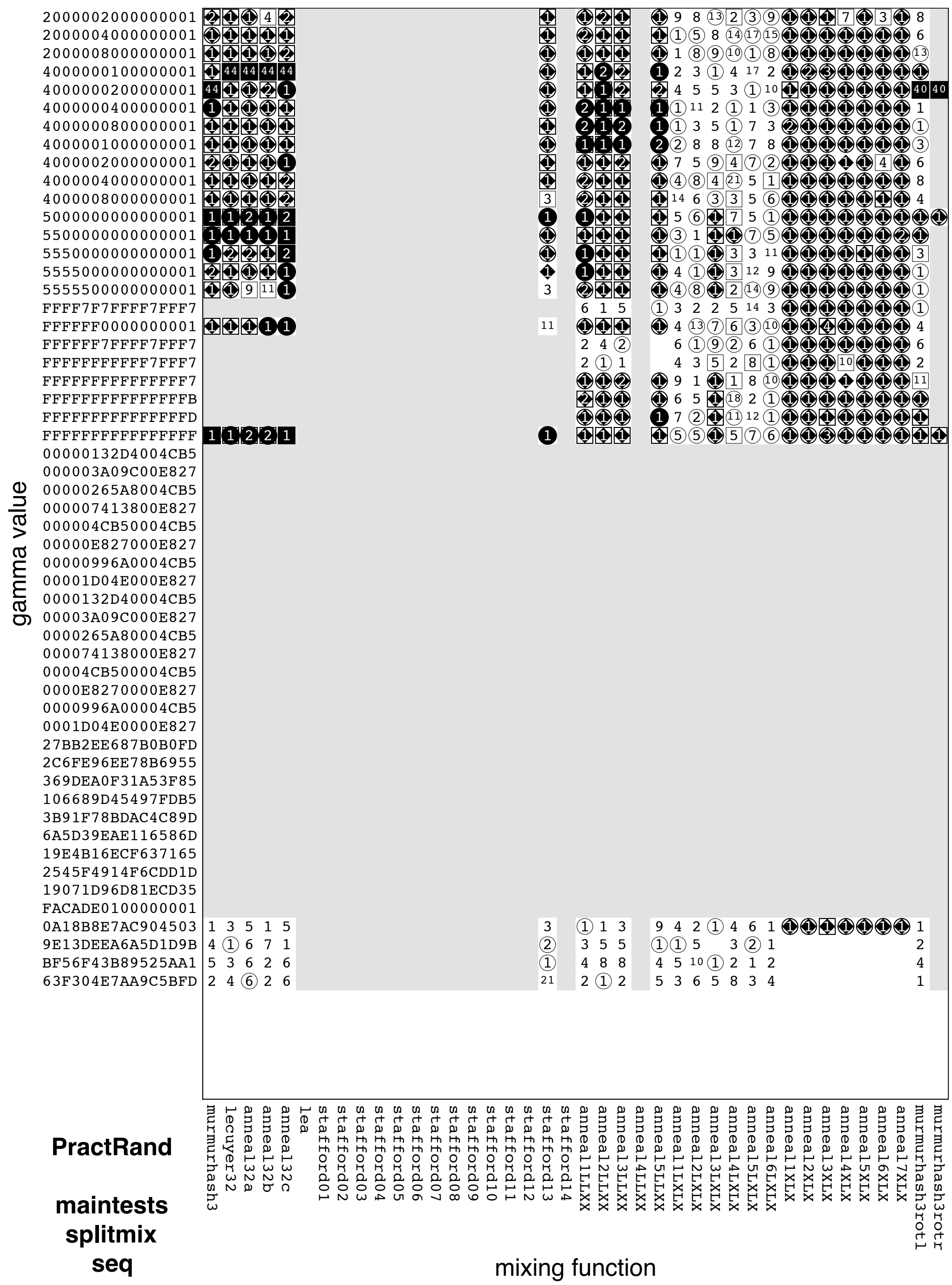


Fig. 18. PractRand_maintests_splitmix_Seq_graph_1.pdf

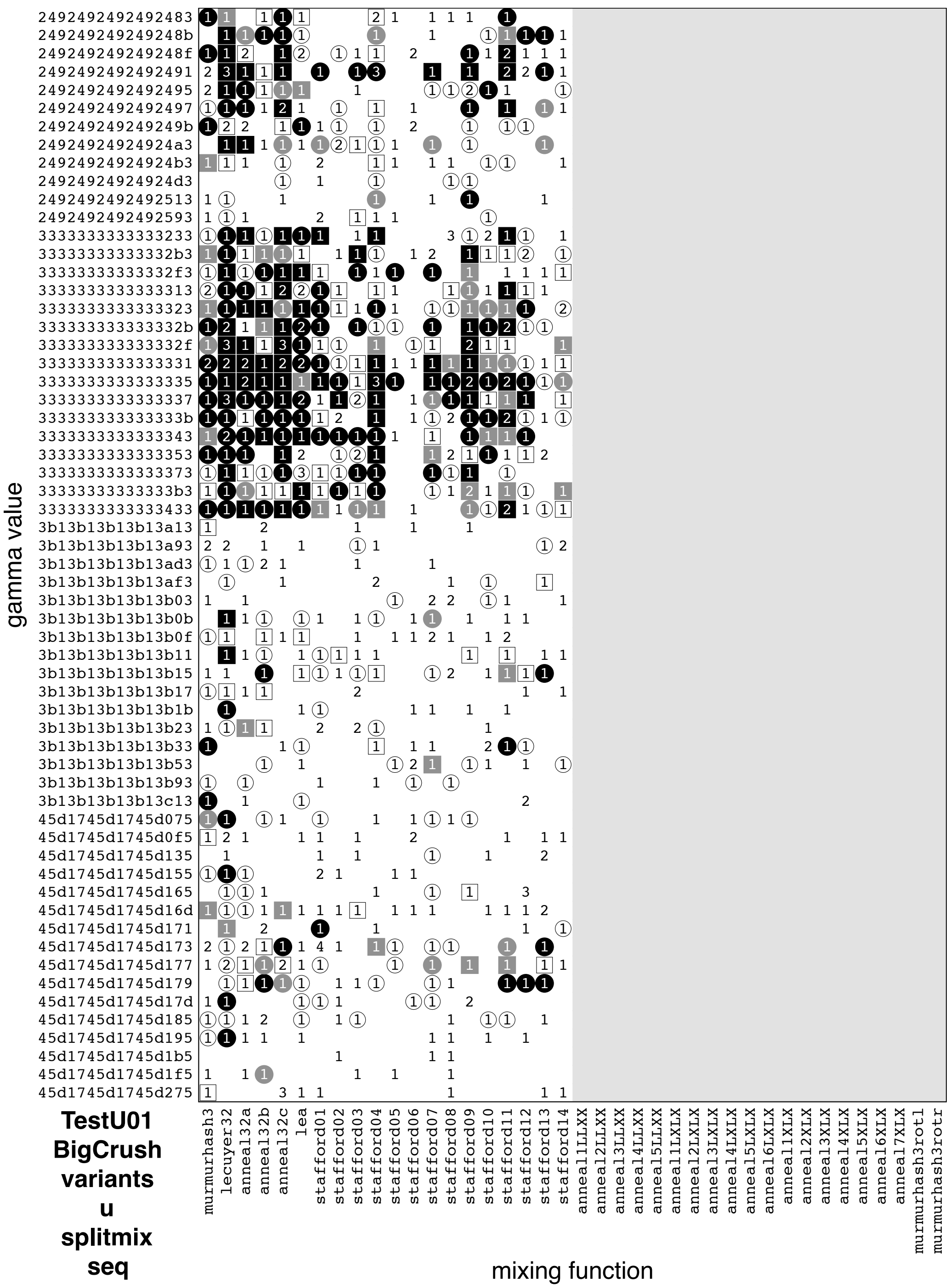


Fig. 19. TestU01_variants_u_splitmix_Seq_graph_3

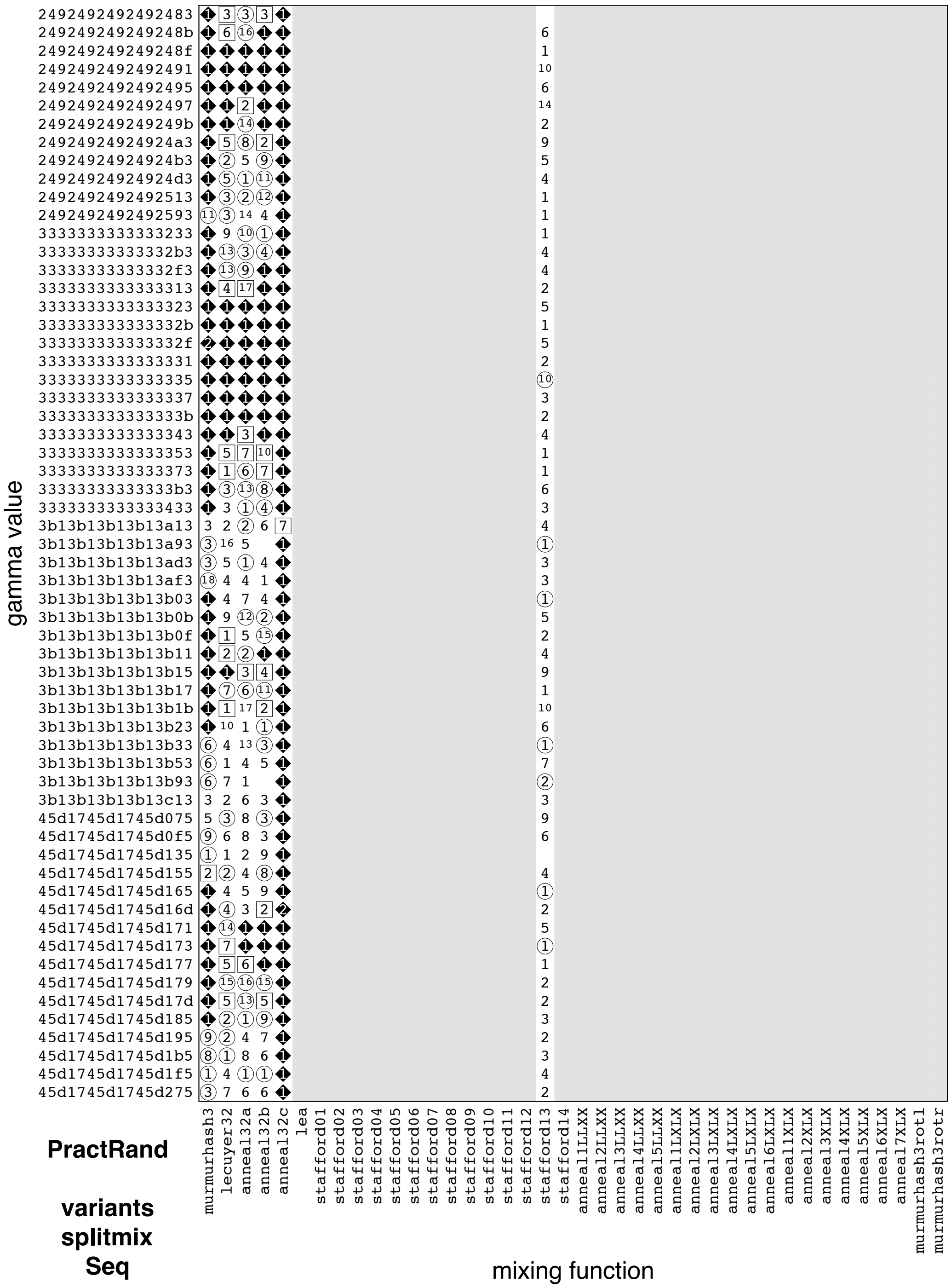


Fig. 20. PractRand_variants_splitmix_Seq_graph_3

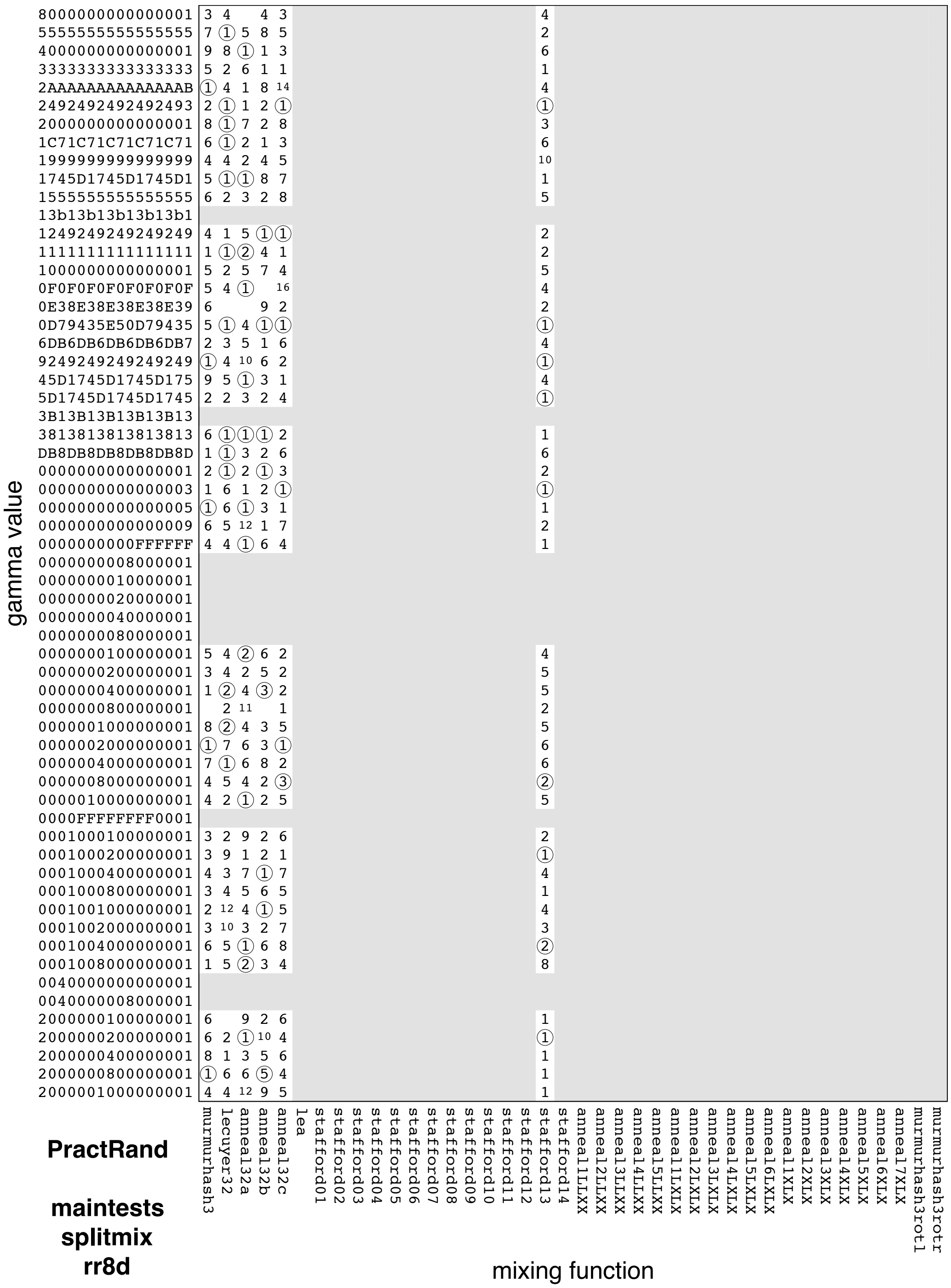


Fig. 21. PractRand_maintests_splitmix_rr8d_graph_0

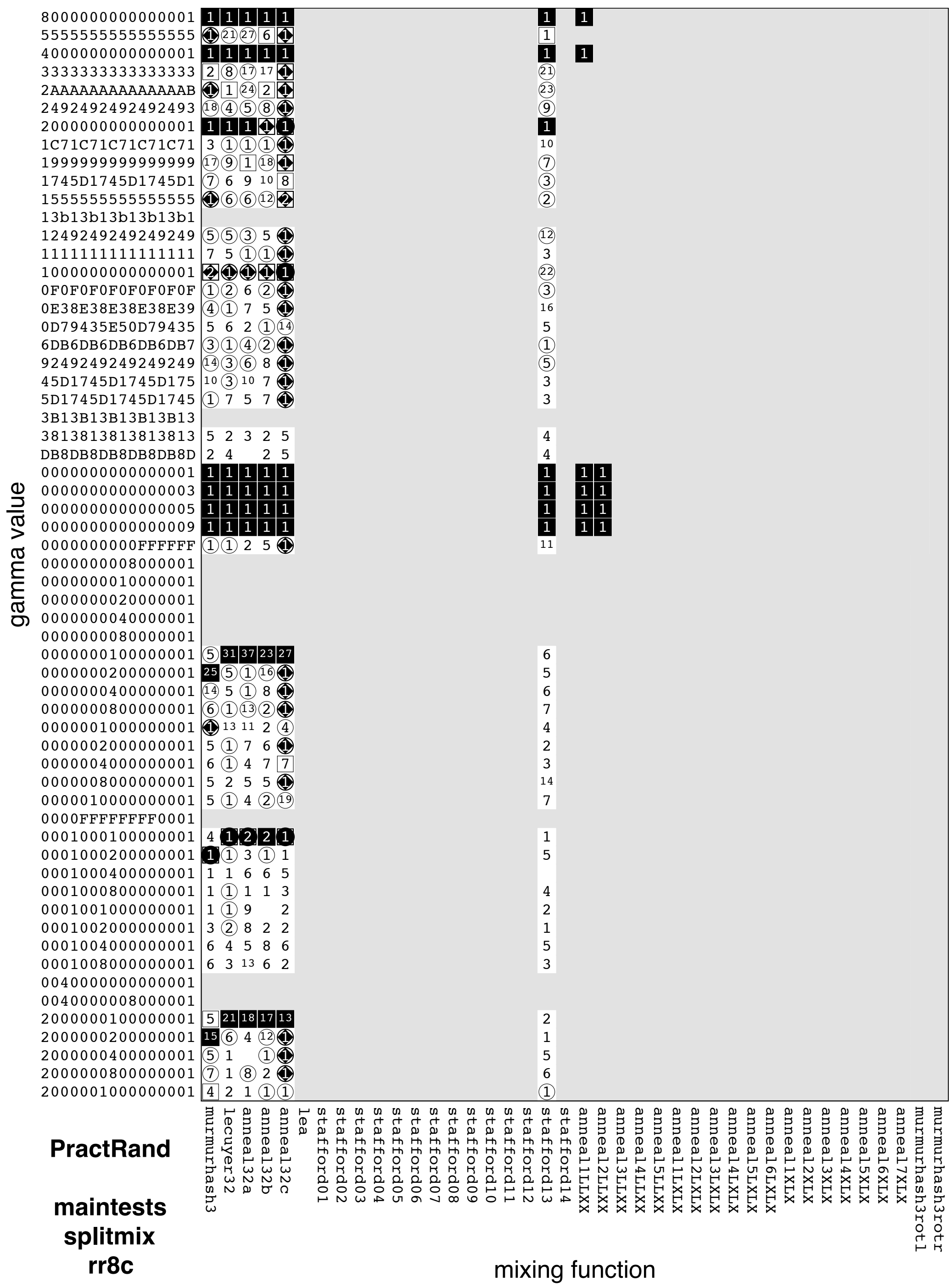


Fig. 22. PractRand_maintests_splitmix_rr8c_graph_0

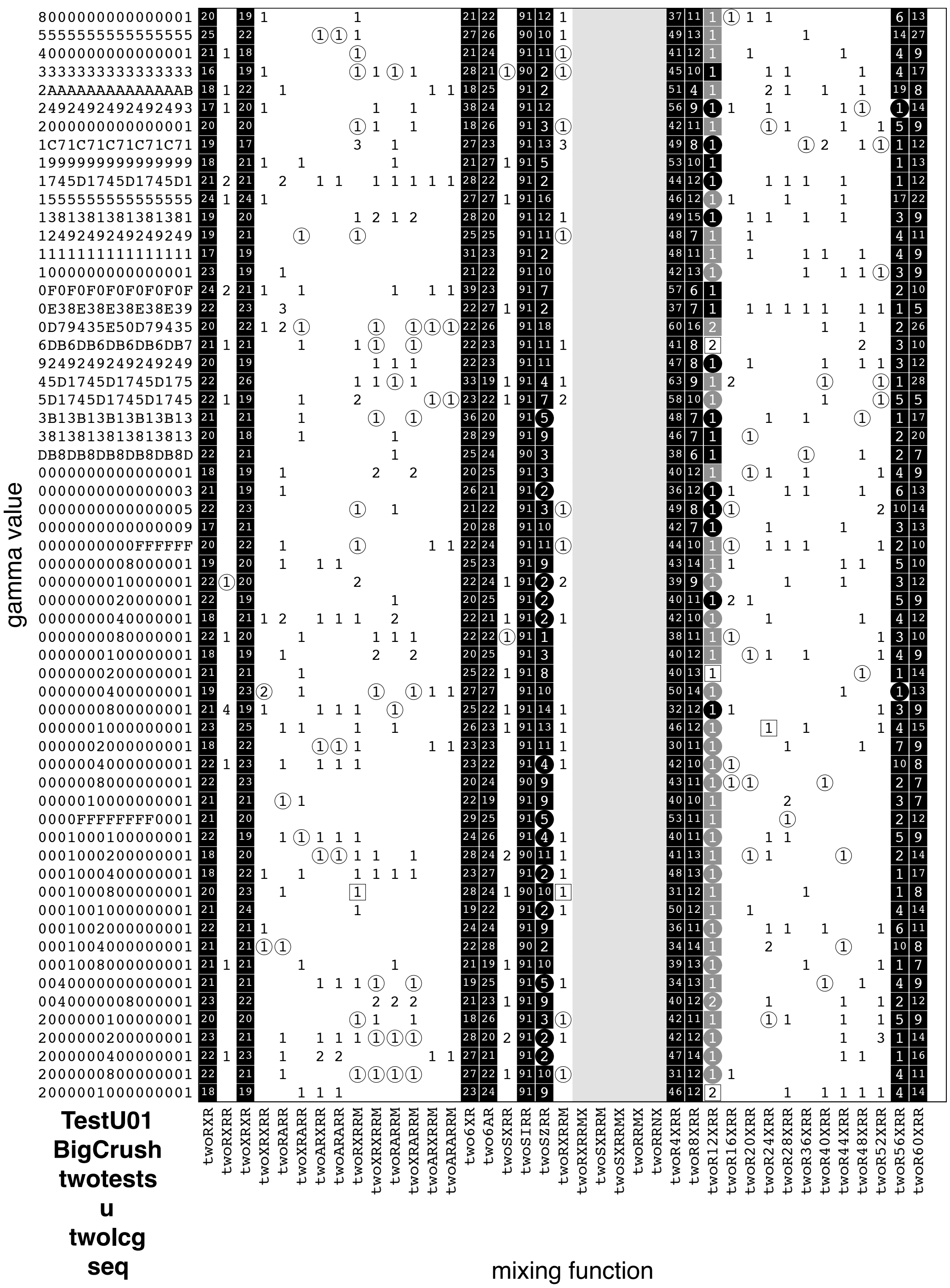


Fig. 23. TestU01_twotests_u_twolcg_seq_graph_0

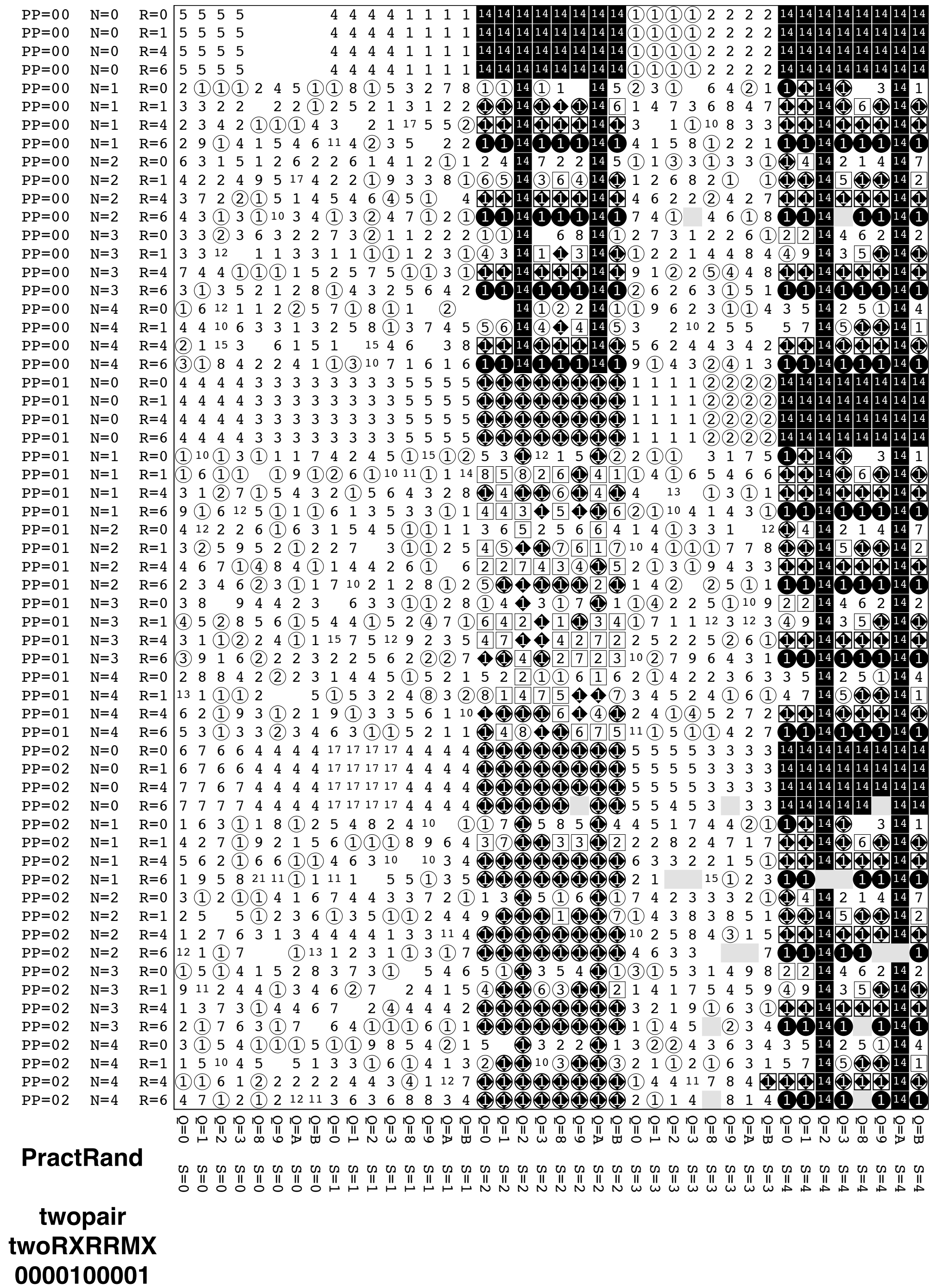


Fig. 24. PractRand_twopair_twolcg_twoRXRRMX_pair__0000100001_graph_0

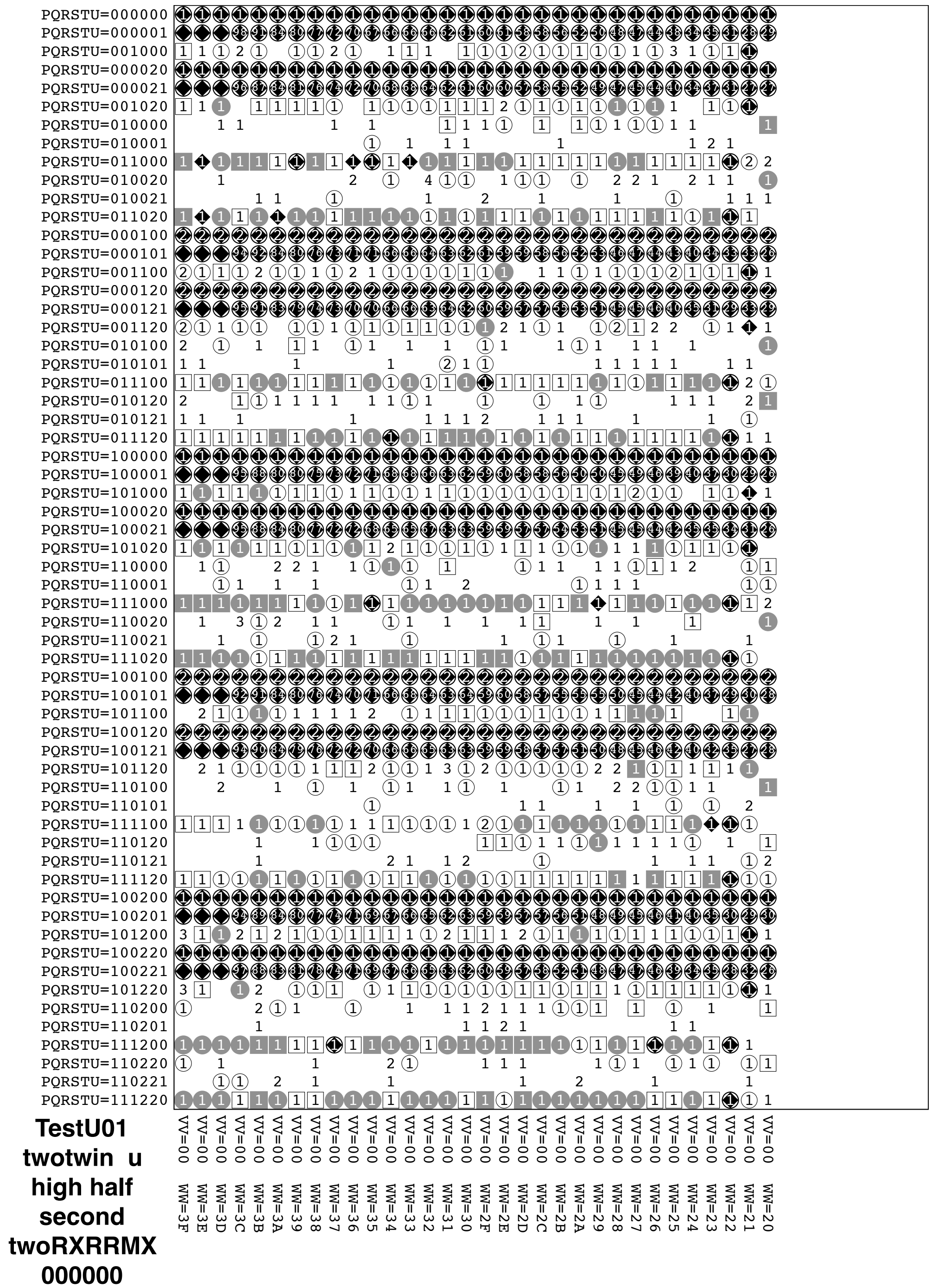


Fig. 25. TestU01_twotwin_u_twolcg_twoRXRRMX_1_twin__000000_graph_2

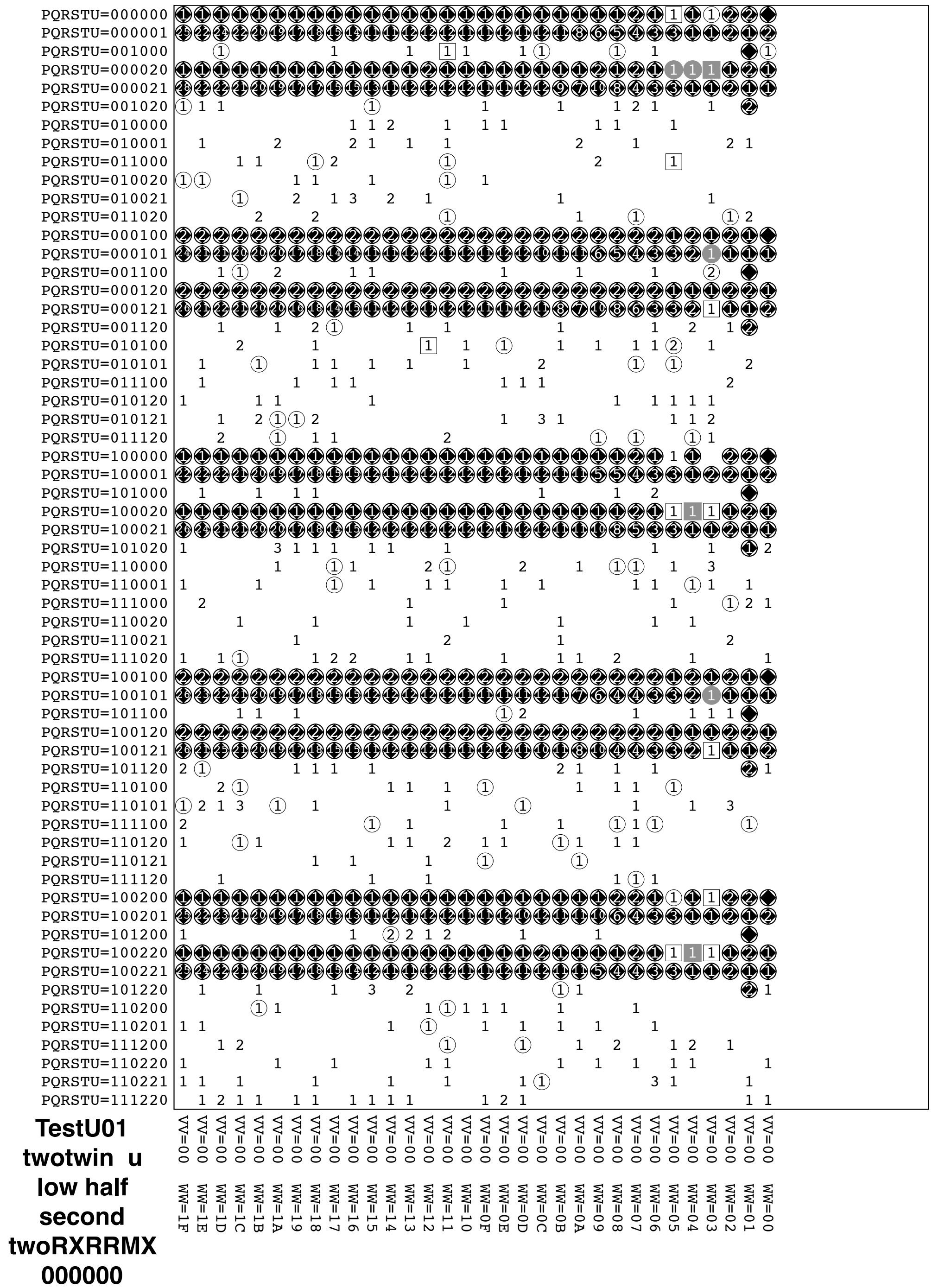


Fig. 26. TestU01_twotwin_u_twolcg_twoRXRRMX_1_twin__000000_graph_3

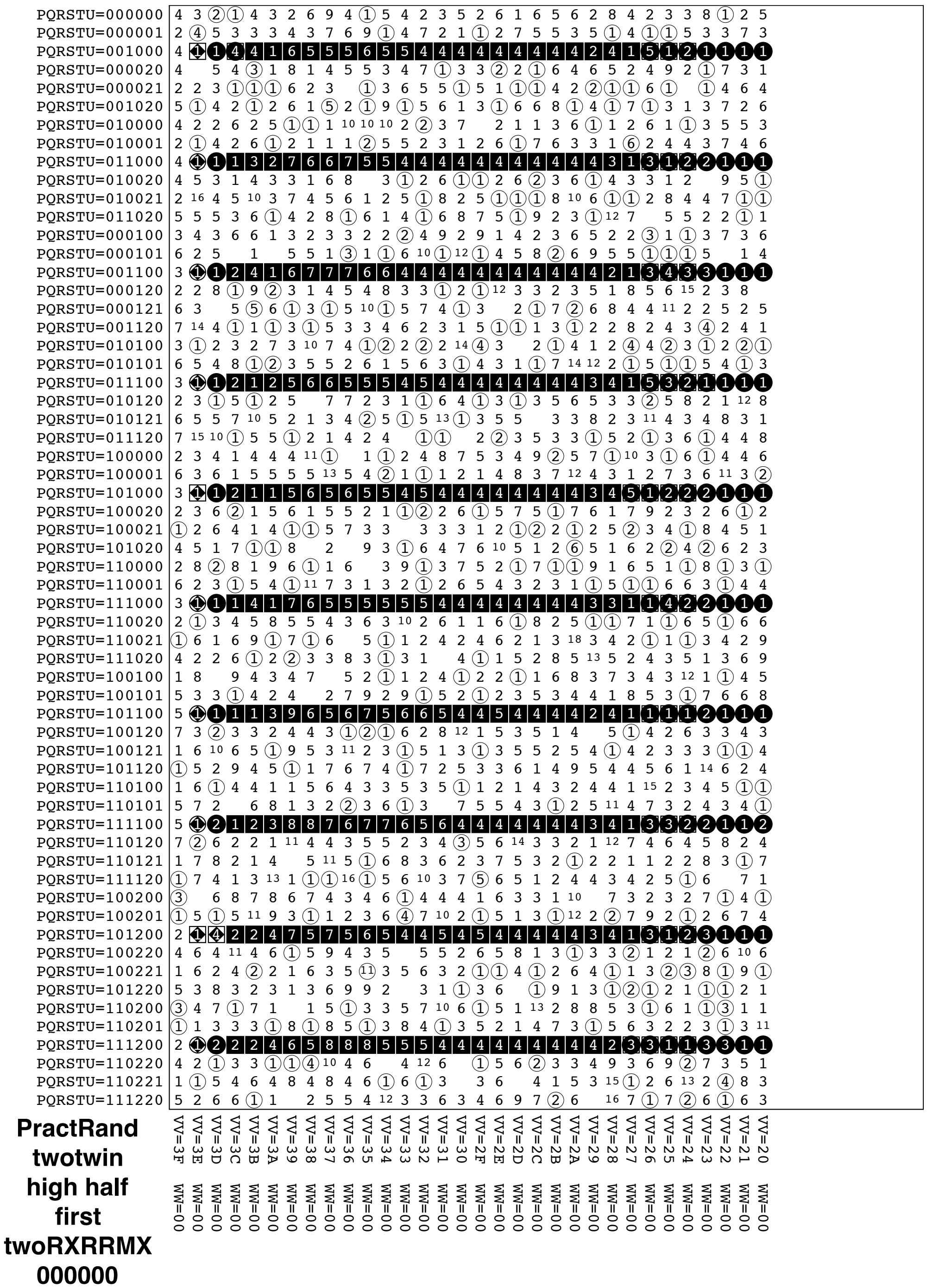


Fig. 27. PractRand_twotwin_twolcg_twoRXRRMX_0_twin__000000_graph_0

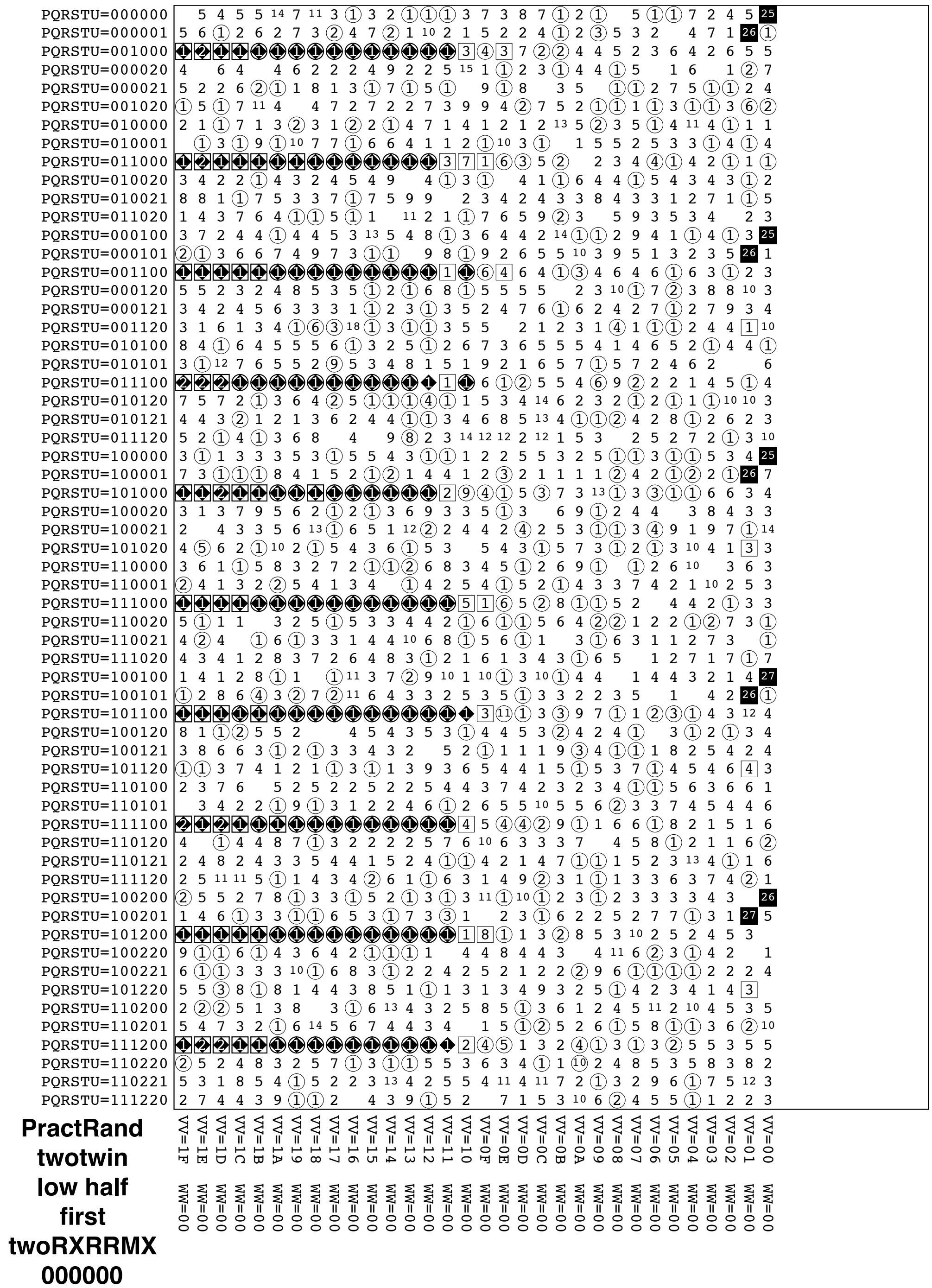


Fig. 28. PractRand_twotwin_twolcg_twoRXRRMX_0_twin__000000_graph_1

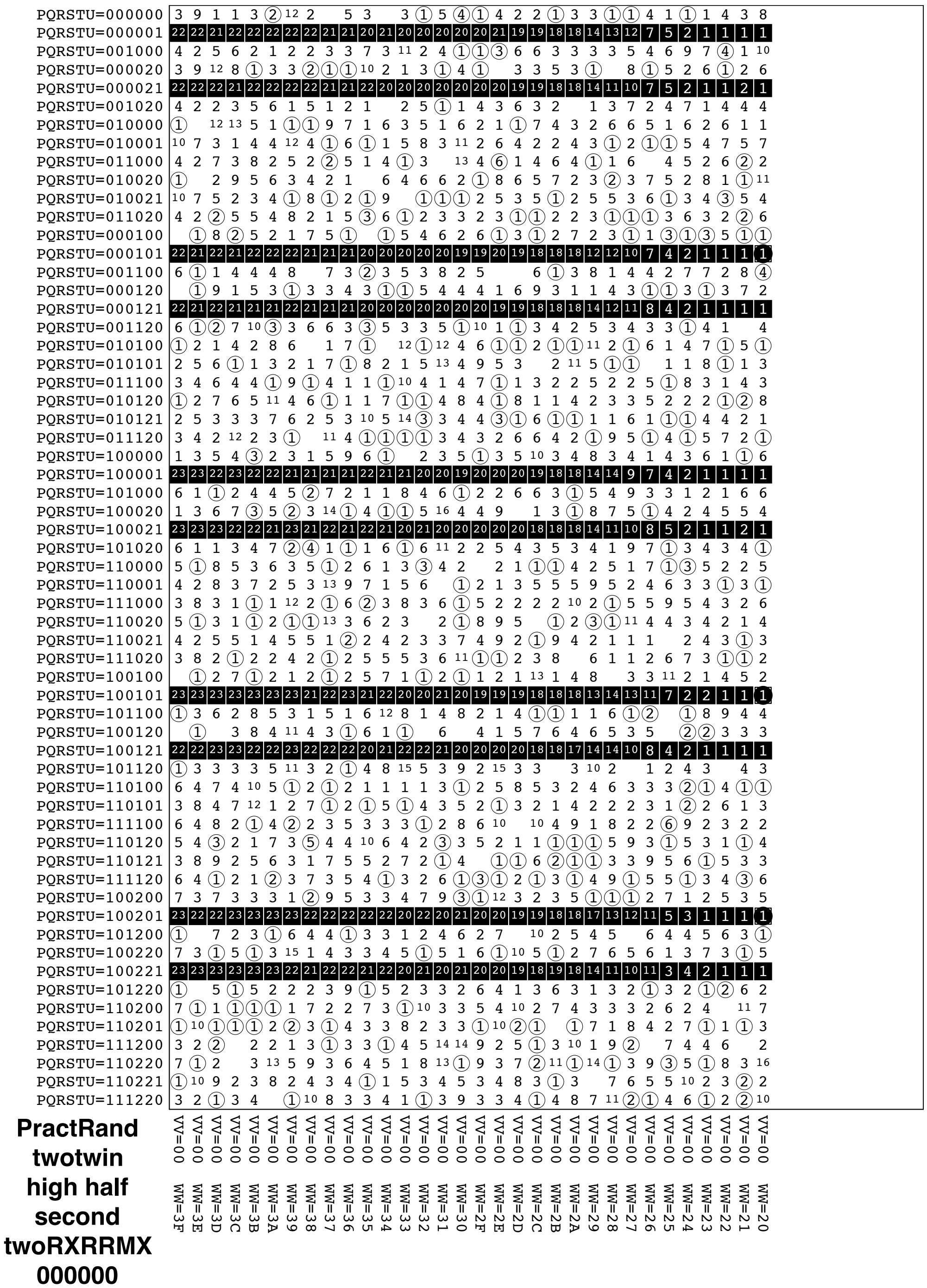


Fig. 29. PractRand_twotwin_twolcg_twoRXRRMX_1_twin__000000_graph_0

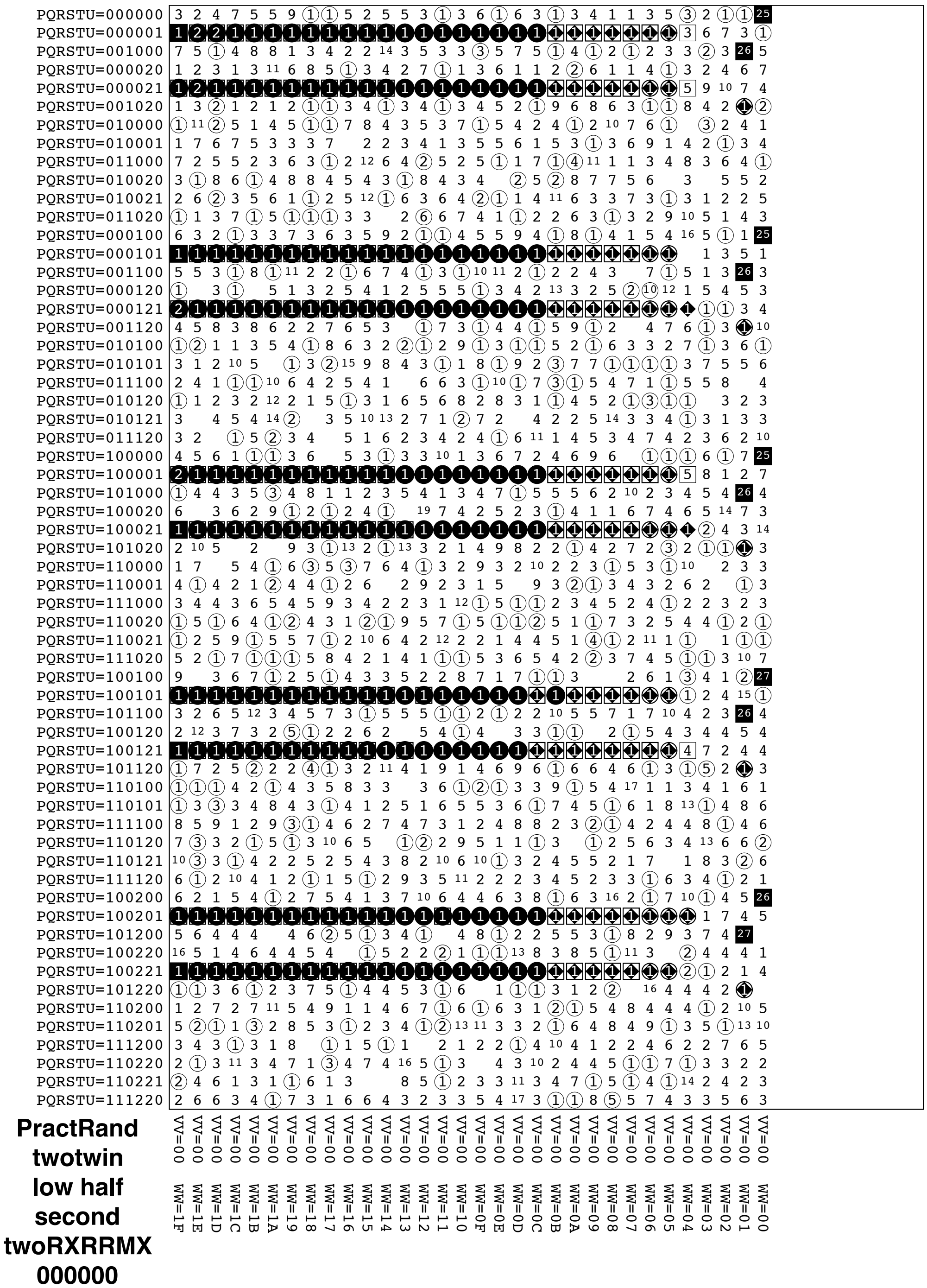


Fig. 30. PractRand_twotwin_twolcg_twoRXRRMX_1_twin__000000_graph_1

B DETAILS ABOUT DISTILLING BIGCRUSH REPORTS

The TestU01 BigCrush test suite runs 106 individual tests (L'Ecuyer and Simard 2013, pp. 148–152), computing 160 test statistics and p-values (L'Ecuyer and Simard 2007). A single test run typically prints about 110 kilobytes of information, which are summarized at the end in one of two forms. Here is an example of a summary produced when no significant anomalies are detected:

```
========= Summary results of BigCrush =========

Version: TestU01 1.2.3
Generator: u_xorgamma_stafford08_seq
Number of statistics: 160
Total CPU time: 05:12:44.09

All tests were passed
```

Here is an example of a summary produced when, say, just one significant anomaly is detected:

```
========= Summary results of BigCrush =========

Version: TestU01 1.2.3
Generator: u_xorgamma_stafford07_seq
Number of statistics: 160
Total CPU time: 05:55:20.65
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

 Test p-value
----------------------------------------------
42 Permutation, t = 7 2.7e-4
----------------------------------------------
All other tests were passed
```

This is a typical result when a PRNG is fundamentally sound but one test "accidentally" falls outside the statistically desirable range, which is bound to happen by chance every so often.

Now, here is an example of a summary produced when many significant anomalies are detected:

```
========= Summary results of BigCrush =========

Version: TestU01 1.2.3
Generator: u_splitmix_stafford01_seq
Number of statistics: 160
Total CPU time: 06:01:33.54
The following tests gave p-values outside [0.001, 0.9990]:
(eps means a value < 1.0e-300):
(eps1 means a value < 1.0e-15):

 Test p-value
----------------------------------------------
10 CollisionOver, t = 14 1.6e-15
```

```
12 CollisionOver, t = 21 2.5e-4
21 BirthdaySpacings, t = 16 3.3e-24
27 SimpPoker, r = 27 eps
58 AppearanceSpacings, r = 27 1 - eps1
69 MatrixRank, L=1000, r=26 0.9995
75 RandomWalk1 H (L=50, r=25) 8.4e-8
96 HammingIndep, L=30, r=27 1.0e-138
102 Run of bits, r = 27 5.6e-4
----------------------------------------------
All other tests were passed
```

This is a typical result when a PRNG is fundamentally unsound. Note that a p-value may be reported in one of five forms: a floating-point value, eps, `1 - eps`, eps1, or `1 - eps1`. Tests 10, 21, and especially 96 have extraordinarily tiny p-values. Test 27 also has a tiny p-value, but we are not told exactly what it is, only that it is less than 10^{-300}. Test 58 has a p-value that is very close to 1; we are not told exactly how close, only that the difference is less than 10^{-15}.

We will refer to each test explicitly listed before the phrase “`All other tests were passed`” as an *anomaly* of the test run. The distillation software for BigCrush test runs distills a set of anomalies into a pair of integers (f, c) (a *failure level* and a *count*) in this manner:

- If a test run file is missing for some reason, then $(f, c) = (-1, 0)$.
- If a test run file is present but is incomplete or otherwise malformed for some reason, then $(f, c) = (-2, 0)$.
- If a test run file is present and all tests were passed, then $(f, c) = (0, 0)$.
- Otherwise, the test run file was present and well-formed but reported anomalies.
  - The reported p-value for each anomaly is transformed into a floating-point number as follows:
    - A p-value listed as `eps` or `1 - eps` becomes `1.0e-300`.
    - A p-value listed as `eps1` or `1 - eps1` becomes `1.0e-15`.
    - A p-value already listed as a floating-point value greater than `0.5` is subtracted from `1.0`.
    - A p-value already listed as a floating-point value not greater than `0.5` remains as is.
  - Next, the floating-point p-value is categorized into one of five *failure levels*:
    - If it is less than `1.0e-300`, it is failure level 5.
    - Otherwise, if it is less than `1.0e-15`, it is failure level 4.
    - Otherwise, if it is less than `1.0e-6`, it is failure level 3.
    - Otherwise, if it is less than `1.0e-4`, it is failure level 2.
    - Otherwise, it must be less than `1.0e-3`, and it is failure level 1.
  - Next, let f be the highest failure level among all anomalies for the test run is identified, and let c be the number of anomalies having that highest failure level.

For charting purposes, the pair of integers (f, c) is then reduced to a symbol and/or number in this way:

- $(0, c)$ becomes whitespace (no anomalies).
- $(1, c)$ becomes the number c (minor anomalies, of which many are expected when doing thousands of tests).
- $(2, c)$ becomes the number c within a circle.
- $(3, c)$ becomes the number c within a square.
- $(4, c)$ becomes the number c in white on a solid black circle.
- $(5, c)$ becomes the number c in white on a solid black square.
- $(-1, c)$ becomes a light gray square (indicating missing data).
- $(-2, c)$ becomes $\times$ on a light gray square (indicating malformed data).

C DETAILS ABOUT DISTILLING PRACTRAND REPORTS

The PractRand test suite runs for an indefinite amount of time, normally producing intermediate reports after processing 2^m bytes of generated pseudorandom values for all integer values of m starting with $m = 27$. (We have chosen to provide command-line arguments that cause additional reports to be produced after processing 0.375×2^{40}, 0.75×2^{40}, 1.25×2^{40}, 1.5×2^{40}, 1.75×2^{40}, 2.25×2^{40}, 2.5×2^{40}, 2.75×2^{40}, 3×2^{40}, 3.25×2^{40}, 3.5×2^{40}, and 3.75×2^{40} bytes. We also provide a command-line arguments that terminate the test run either after the first report that prints "FAIL" or after testing 4 terabytes of data, whichever comes first.) For an intermediate report produced after processing 2^m bytes of generated pseudorandom values, PractRand computes $4m - 56$ separate statistics; thus the first report (for $m = 27$) reports 52 test results, and the report for $m = 42$ (4 terabytes) reports 112 test results.

A single test run that gets all the way to 4 terabytes typically prints about 5 kilobytes of information. Here is an example of a run that was terminated after examining 64 gigabytes of generated values:

```
rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 128 megabytes (2^27 bytes), time= 2.6 seconds
 Test Name Raw Processed Evaluation
 [Low4/64]BCFN(2+1,13-5) R= +9.9 p = 4.8e-4 unusual
 ...and 51 test result(s) without anomalies

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 256 megabytes (2^28 bytes), time= 5.9 seconds
 no anomalies in 56 test result(s)

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 512 megabytes (2^29 bytes), time= 11.6 seconds
 no anomalies in 60 test result(s)

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 1 gigabyte (2^30 bytes), time= 22.2 seconds
 no anomalies in 64 test result(s)

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 2 gigabytes (2^31 bytes), time= 42.6 seconds
 no anomalies in 68 test result(s)

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 4 gigabytes (2^32 bytes), time= 82.4 seconds
 no anomalies in 72 test result(s)

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 8 gigabytes (2^33 bytes), time= 161 seconds
 no anomalies in 76 test result(s)

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 16 gigabytes (2^34 bytes), time= 319 seconds
 Test Name Raw Processed Evaluation
```

```
 Gap-16:B R= +5.8 p = 1.9e-4 suspicious
 ...and 79 test result(s) without anomalies

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 32 gigabytes (2^35 bytes), time= 631 seconds
 Test Name Raw Processed Evaluation
 Gap-16:B R= +8.3 p = 1.2e-6 very suspicious
 [Low4/64]DC6-9x1Bytes-1 R= +5.7 p = 3.7e-3 unusual
 ...and 82 test result(s) without anomalies

rng=splitmix_anneal32a_seq, seed=0x5555555555555555
length= 64 gigabytes (2^36 bytes), time= 1253 seconds
 Test Name Raw Processed Evaluation
 Gap-16:A R= +10.0 p = 1.0e-6 very suspicious
 Gap-16:B R= +17.5 p = 1.1e-10 FAIL !
 [Low1/64]DC6-9x1Bytes-1 R= -4.2 p =1-5.0e-3 unusual
 ...and 85 test result(s) without anomalies
```

This is a typical result when a PRNG is fundamentally unsound. Note that PractRand reports not only a p-value for each anomaly but also a word or phrase assessing that p-value; we choose to rely on these nonnumerical assessments in distilling the reports. The complete set of assessments used, in increasing order of severity, is `unusual`, `suspicious`, `SUSPICIOUS`, `very suspicious`, `VERY SUSPICIOUS`, and `FAIL`. (PractRand may further print a varying number of exclamation points after the word "`FAIL`" but we choose to ignore those.)

The distillation software for PractRand test runs distills a set of anomalies into a pair of integers (f, c) (a *failure level* and a *count*) in a manner not too different from the strategy used for BigCrush:

- If a test run file is missing for some reason, then $(f, c) = (-1, 0)$.
- If a test run file is present but is incomplete or otherwise malformed for some reason, then $(f, c) = (-2, 0)$.
- If a test run file is present and no anomalies were observed after processing 4 terabytes of generated pseudorandom data, then $(f, c) = (0, 0)$.
- Otherwise, the test run file was present and well-formed but reported one or more anomalies.
  - Each anomaly is categorized into one of 11 *failure levels*:
    * If the assessment was `FAIL`:
      · If the assessment occurred for $m < 31$, it is failure level 11.
      · Otherwise, if the assessment occurred for $m < 34$, it is failure level 10.
      · Otherwise, if the assessment occurred for $m < 37$, it is failure level 9.
      · Otherwise, if the assessment occurred for $m < 40$, it is failure level 8.
      · Otherwise, if the assessment occurred for $m < 42$, it is failure level 7.
      · Otherwise, it is failure level 6.
    * If the assessment was `VERY SUSPICIOUS`, it is failure level 5.
    * If the assessment was `very suspicious`, it is failure level 4.
    * If the assessment was `SUSPICIOUS`, it is failure level 3.
    * If the assessment was `suspicious`, it is failure level 2.
    * If the assessment was `unusual`, it is failure level 1.
  - Next, let f be the highest failure level among all anomalies for the test run, and let c be the number of anomalies having that highest failure level.

For charting purposes, the pair of integers (f, c) is then reduced to a symbol and/or number in this way:

- $(0, c)$ becomes whitespace (no anomalies).
- $(1, c)$ becomes the number c (minor anomalies; many are expected when doing thousands of tests).
- $(2, c)$ becomes the number c within a circle.
- $(3, c)$ becomes the number c within a square.
- $(4, c)$ becomes the number c in white within a solid dark gray circle.
- $(5, c)$ becomes the number c in white within a solid dark gray square.
- $(6, c)$ becomes the number c in white within a solid black diamond.
- $(7, c)$ becomes the number c in white within a solid black diamond within a circle.
- $(8, c)$ becomes the number c in white within a solid black diamond within a square.
- $(9, c)$ becomes the number c in white on a solid black circle.
- $(10, c)$ becomes the number c in white on a solid black circle within a square.
- $(11, c)$ becomes the number c in white on a solid black square.
- $(-1, c)$ becomes a light gray square (indicating missing data).
- $(-2, c)$ becomes $\times$ on a light gray square (indicating malformed data).