A

PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation

MELISSA E. O'NEILL, Harvey Mudd College

This paper presents a new uniform pseudorandom number generation scheme that is both extremely practical and statistically good (easily passing L'Ecuyer and Simard's TestU01 suite). It has a number of important properties, including solid mathematical foundations, good time and space performance, small code size, multiple random streams, and better cryptographic properties than are typical for a general-purpose generator.

The key idea is to pass the output of a fast well-understood "medium quality" random number generator to an efficient permutation function (a.k.a. hash function), built from composable primitives, that enhances the quality of the output. The algorithm can be applied at variety of bit sizes, including 64 and 128 bits (which provide 32- and 64-bit outputs, with periods of 2^{64} and 2^{128}). Optionally, we can provide each b-bit generator with a $b-1$ bit stream-selection constant, thereby providing 2^{b-1} random streams, which are full period and entirely distinct. An extension adds up to 2^b-dimensional equidistribution for a total period of 2^{b2^b}. The construction of the permutation function and the period-extension technique are both founded on the idea of *permutation functions on tuples*.

In its standard variants, b-bit generators use a $2^{b/2}$-to-1 function to produce $b/2$ bits of output. These functions can be designed to make it difficult for an adversary to discover the generator's internal state by examining its output, and thus make it challenging to predict. This property, coupled with the ability to easily switch between random streams, provides many of the benefits provided by cryptographically secure generators without the overheads usually associated with those generators.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]; G.3 [**Probability and Statistics**]: Random number generation, statistical software

General Terms: Design, Algorithms, Performance, Theory, Security

Additional Key Words and Phrases: Random number generator, linear congruential generator, permutation functions, hash functions, TestU01

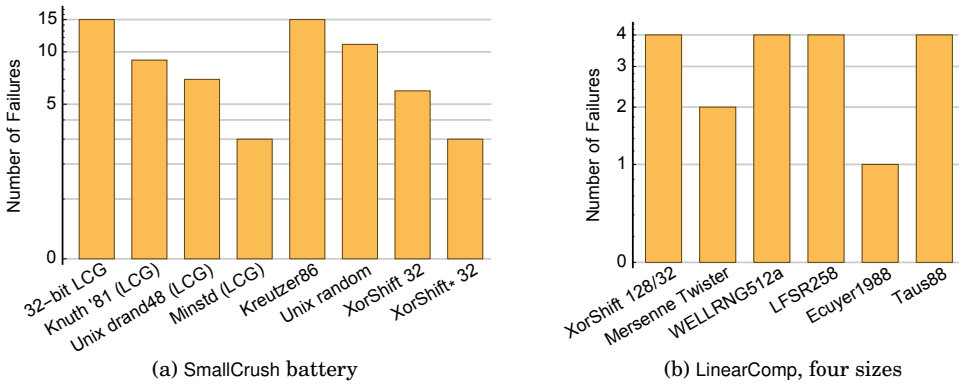
ACM Reference Format:
ACM Trans. Math. Softw. V, N, Article A (January YYYY), 46 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Random number generation is important throughout computer science. It is a key element to a huge number of areas, including global optimization, computational creativity, modeling and simulation, robotics, games, and many more besides. The correctness or performance of an algorithm can critically depend on the quality of the random number generation scheme it uses; its competitiveness can depend on the speed of its generator; and sometimes other criteria, such as security, matter even more. In short, generator quality matters and quality has many dimensions.

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-1219243. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

DOI: http://dx.doi.org/10.1145/0000000.0000000



(a) SmallCrush battery (b) LinearComp, four sizes

Fig. 1: Almost all generators in widespread use don't survive *very minimal* statistical scrutiny from TestU01 [L'Ecuyer and Simard 2007]. Even *one* failure indicates a problem.

The idea of solving problems using machine-generated random numbers has been with us since 1949, being first applied by a team at Los Alamos (lead by Nicholas Metropolis and including John von Neumann) that needed to model problems in physics [Metropolis and Ulam 1949]. In the years that have followed, there have been numerous advances in the area (and some horrendous missteps, such as the RANDU generator [Knuth 1997]), yet more than sixty years later, with random number generators in widespread use, there has been no single random number generator that provides all the properties we might consider valuable.

The current state of affairs is one of *islands*: there are the fast generators, but the fast ones mostly aren't statistically good. There are the statistically good ones, but they're mostly not fast. If you can find one that manages to be fast and statistically good, it won't be even remotely secure (i.e., its past and future output can be trivially predicted after observing only a few outputs). And if we add additional criteria, such as space usage or code size, to the mix, you are completely out of luck.

We will examine these criteria in depth in Section 2, but as a prelude to that, let us use the highly regarded TestU01 statistical test suite [L'Ecuyer and Simard 2007] to observe that today, many of the generators in widest use cannot withstand even *minimal* statistical scrutiny, sometimes failing statistical tests after generating only *a few thousand* numbers. Take a look at Figure 1—perhaps one of your favorite generators didn't do as well as you would have hoped![1] (The graph uses a log scale because failing even *one* statistical test is a significant problem. The combined tests require less than fifteen seconds to run.)

For years it has seemed that compromise was inevitable, and no one would be entirely happy. Academics would despair at the statistically weak generators that practitioners use, practitioners would shake their heads at the slow, or otherwise impractical, generators that academics propose, and cryptographers would wonder when someone would finally pay attention to security.

This paper changes that state of affairs. It introduces a new family of generators that set a new standard for combining desirable properties. Pleasantly, this new generation

[1] In Figure 1(b), TestU01's linear complexity test [Carter 1989; Erdmann 1992; L'Ecuyer and Simard 2007] is run at sizes 5000, 25000, 50000, and 75000 and usually completes in under a second. It is discussed in more detail in Section 2.1.2, which also discusses the entire process of applying statistical tests.

scheme applies a simple insight to a widely used technique. At its core, it is the idea of using a *randomized algorithm* to enhance randomness. Variations on the idea have been tried in the past (see Section 9 for related work), but the particular way randomization is achieved is new. I hope that by the end of the paper you will agree that the idea is obvious in retrospect.

1.1. Claims and Contributions

This paper introduces *the PCG family* of random number generators and the techniques that underlie them. In order to do so, it also articulates in detail (in Section 2) the desirable properties of a random number generator including performance, correctness, uniformity, and unpredictability, as well as sound mathematical grounding. It also describes some less well-known desirable features, such as k-dimensional equidistribution and seekability (a.k.a. jump-ahead).

With those criteria articulated, the paper then

— Describes a new permutation technique, founded on the idea of *permutation functions on tuples*, that can dramatically improve the output quality of a medium-quality random number generator while preserving important qualities such as uniformity. Although in this context the technique is applied to improving a random number generator, it has broader applications, including hashing.
— Applies that technique to the specific case of a base generator with weak statistical properties but other desirable ones (specifically, a linear congruential generator), and enumerates some members of the PCG family, including several that are simultaneously *extremely fast*, *extremely statistically good*, and *extremely space efficient*.
— Applies the technique in such a way that permutations may be made essentially uninvertable, offering far better *cryptographic security* than most mainstream generators (which offer none at all).
— Provides a low-cost period-extension technique, founded on the same ideas, that allows huge periods and k-dimensional equidistribution for arbitrary k.

The name for the family, PCG, stands for *permuted congruential generator*, combining both concepts that underly the generation scheme, namely permutation functions on tuples and a base linear congruential generator.

But in addition to describing the PCG family, it is also necessary to appraise its performance. To further that goal, the paper

— Develops a model for the performance of an *ideal* uniform random number generator with b bits of state, including the notion of the point at which such a generator becomes *overtaxed* and the constraints of uniformity make it unable to deliver truly random behavior. (The presentation includes a generalized solution to *the birthday problem* which may be useful in other contexts.)
— Using information about the TestU01 suite and the above model, determines an approximation of the point at which any uniform generator, even an ideal one, could be expected to fail TestU01's statistical tests.
— Draws on the work of L'Ecuyer & Simard [2007] to articulate a powerful way to quantitatively compare the statistical performance of different random number generation schemes capturing the concept of *headroom* to pass more stringent tests in the future.
— Uses the above statistical-performance comparison scheme, as well as time and space performance, to contrast PCG generators with existing ones.[2]

[2] Although I have included the most widely used and best performing existing generators, it would not be practical for me to contrast all prior work. Even the generators I do include are not *all* discussed in *every* section.

It also reviews some of the well-known (and less well-known) properties of linear congruential generators, both their flaws and some of their highly desirable properties.

2. DESIRABLE PROPERTIES

In this section, we will examine some of the properties that users of random number generators might reasonably hope for, namely, good statistical properties, good mathematical foundations, lack of predictability, cryptographic security, good time and space performance, small code size, multiple independent streams, a sufficiently long period, k-dimensional equidistribution, and a power-of-two modulus. In each case we'll examine what that property is, why it matters, and how well a representative selection of random number generators achieve each goal.

Although not all users require all these properties, all other things being equal, the more of these properties a random number generator can provide, the better. The PCG family, which we will discuss in subsequent sections, will provide them *all*.

2.1. Statistical Qualities

Perhaps the most obvious property a random number generator should satisfy is that it should "be random", but delineating exactly what this most fundamental property means is tricky [L'Ecuyer 2012], given that pseudorandom number generation typically uses deterministic algorithms. Generally speaking, we usually interpret this property to mean that a random number generator should conform to statistical expectations regarding random systems. These properties can be explored both mathematically and experimentally.

Mathematical properties can allow us to speak with authority about the behavior of a generator without needing to run it. Statistical properties that aren't easily determined through mathematical reasoning can be tested by experiment, using the mathematics of random systems to define expected behavior. In the next two subsections, we'll examine each of these properties.

2.1.1. Theory: Period & Uniformity. The mathematical characteristics of a random number generator matter a good deal, because they directly relate to the confidence we can have in it. Various authors [L'Ecuyer 2012; L'Ecuyer and Simard 2007; Knuth 1997] have (rightly!) admonished would-be inventors of new random number generators not to simply assemble an arbitrary collection of software gizmos in an unprincipled way and hope for the best, because doing so risks the later discovery that the generator has systemic bias or is broken in some other way.

The most fundamental mathematical concept underlying pseudorandom number generation is that of *period*. Any deterministic algorithm executed using finite memory must have finitely many states, and thus any random number generation algorithm must have a fixed period, after which it will repeat (or, as a rarely used alternative, just stop). It is usually necessary to approach period characterization mathematically because when periods become large, empirical verification becomes impractical.

The concept of *uniformity* builds on the notion of a fixed period. Uniformity requires that after a generator completes a full period all outputs will have occurred the same number of times. If a generator is uniform, we are assured that over the long term, it lacks bias.

Mathematical reasoning can allow us to make determinations about the uniformity of a random number generator without ever needing to see its implementation or run it. For example, let us consider a case where we can show that a generator must lack uniformity in its output. Consider a generator with b bits of state, but where one of the 2^b possible states is never used, (perhaps because the implementation must avoid an all-bits-are-zero state). The missing state would leave the generator with a

period of $2^b - 1$. By the pigeonhole principle, we can immediately know that it cannot uniformly output 2^b unique b-bit values. Moreover, if we desire k-bit values (where $k < b$), there will still be a size mismatch to overcome. (This issue arises in practice with *linear-feedback shift-register* generators [Tausworthe 1965]—the foundation for a large class of generators—limiting their use at small bit sizes unless additional remediation of some kind is performed.)

2.1.2. Practice: Empirical Testing, and Preexisting Generators. The saying "In theory, there is no difference between theory and practice. But, in practice, there is." certainly applies to random number generation.[3] The insights provided by mathematical analysis, while immensely valuable, are insufficient to verify the performance of a random number generator—current analytical techniques can reveal flaws, but they cannot fully characterize the required properties. Empirical testing is required, using tests that are themselves founded on a mathematical understanding of the behavior of random systems. In practice, empirical tests have felled random number generators with impressive mathematical credentials [L'Ecuyer and Simard 2007].

In 2007, L'Ecuyer & Simard [2007] made a very significant contribution to the world of random-number–generator testing when they created the TestU01 statistical test suite. Other suites, such as Diehard [Marsaglia 1995], had existed previously, but TestU01 (which included a large number of previously independently published tests, and applied them at scale) vastly increased the scope and thoroughness of the testing process. L'Ecuyer & Simard used their tests to review existing generators (something we will reprise in a much more abbreviated way momentarily), and the results were sobering for the field, because many well respected generators did not pass.

Test suites such as TestU01 work by performing some statistically well understood task using a candidate generator and then checking the plausibility of the results. Much like experiments in other fields of science, these results typically produce a p-value, but whereas scientists usually desire results where the null hypothesis—that the observations were merely due to chance—is ruled out, we desire the opposite, a result confirming that the observed behavior is consistent with chance.

Back in Section 1, we saw (in Figure 1) how TestU01's SmallCrush test battery revealed flaws in a number of well-used generators in mere seconds; now we will continue to illustrate the profound impact of TestU01. Figure 2 shows the number of test failures running the Crush and BigCrush batteries on a number additional generators. (L'Ecuyer & Simard [2007] and others have considered many more, my intent here is just to provide the reader with a glimpse of the landscape.) Following the advice of Vigna [2014a], the counts reflect failures for the generators both used normally and with their bits reversed; 64-bit generators have both their high 32 and low 32 bits tested.

Crush usually takes about an hour to run, whereas BigCrush takes about six hours. Interestingly, these intensive batteries include some tests that can be run quickly at small sizes, but were excluded from SmallCrush, presumably to keep the number of tests performed small. In particular, as Figure 1(b) showed, the "linear complexity" test [Carter 1989; Erdmann 1992] can actually find nonrandom behavior in the Mersenne Twister (a.k.a., mt19937) [Matsumoto and Nishimura 1998], Wellrng512a [Panneton et al. 2006], Taus88 [L'Ecuyer 1996], LFSR258 [L'Ecuyer 1999c], Ecuyer1988 [L'Ecuyer 1988], and all sizes of the unaugmented XorShift [Marsaglia 2003] generators in less than five seconds, using fewer than 75,000 generated numbers—sometimes many fewer. All of these generators were created by renowned figures in the world of random number generation, appeared in well-respected peer-reviewed venues, and all had impressive

[3] Although similar sentiments have been uttered by many, this variant of the saying is usually attributed to Jan L.A. van de Snepscheut.

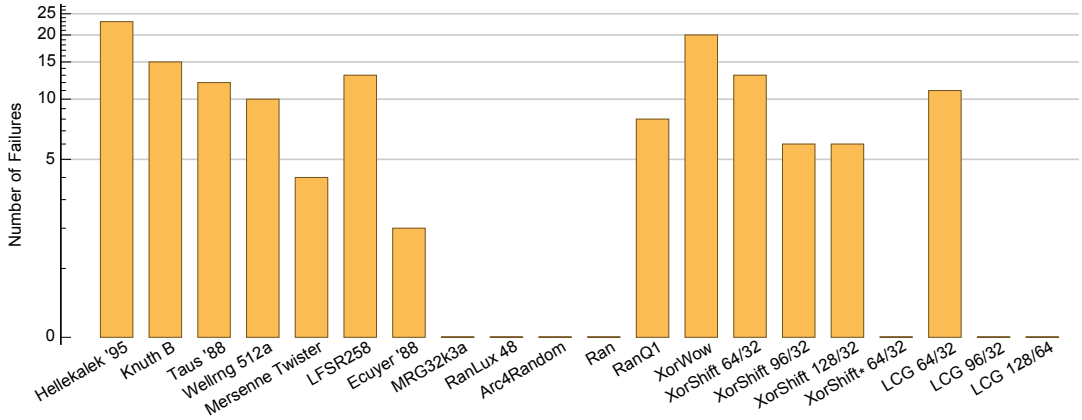


Fig. 2: Combined test failures for Crush and BigCrush from TestU01 for a number of common generators. *Zero* failures are desired. Many well-respected generators fail. Despite their flaws, LCGs actually pass at larger bit sizes.

mathematical credentials, yet they cannot withstand five seconds of empirical scrutiny with TestU01. Quite unsettling!

Also of note, RanQ1 [Press et al. 2007] and XorShift* 64/32 are essentially the *exact same* generator, yet the former fails the test suite and the latter passes. The difference is that the former markets itself as a 64-bit generator and fails because its low-order bits are weak, whereas the latter only returns the top 32 bits. The method that underlies this generator is notable because it follows a similar strategy to the one I will advocate in this paper: it uses a generator with known weaknesses (XorShift), and then applies an improvement step to the numbers it generates.

Finally, on the far right we have three different sizes of linear congruential generator [Lehmer 1951] with a power-of-two modulus, using constants suggested by L'Ecuyer [1999b]. Here we discover something that may surprise some readers: even though linear congruential generators were strongly represented in the very poor generators we saw in Figure 1, they actually *can* do well in empirical tests, provided that we give them enough state bits. Note that this property is not a foregone conclusion for *any* generator; in particular, generators that fail the "linear complexity" test appear to fail it at all bit sizes. Linear congruential generators are not without serious statistical issues (which we will examine in Section 4), but if you began reading this paper with the simplistic belief that "linear congruential generators are bad", perhaps that belief is starting to be eroded.

Let's review where we stand at this point. Only a small number of generators survive empirical testing, and the generators that are most widely used all fail. But thus far, we have only articulated one dimension of the desirable properties of a random number generator, and there are several more to cover.

2.2. Predicability, Repeatability, and Security

Another "natural" property a reasonable person might expect from a source of random numbers is a lack of obvious predictability. A die would hardly seem random if, when I've rolled a five, a six, and a three, you can tell me that my next roll will be a one.

Yet because the algorithms that we are concerned with are deterministic, their behavior is governed by their inputs, thus they will produce the same stream of "random" numbers from the same initial conditions—we might therefore say that they are only

random to an observer unaware of those initial conditions or unaware of how the algorithm has iterated its state since that point. This deterministic behavior is valuable in a number of fields, as it makes experiments reproducible. As a result, the parameters that set the initial state of the generator are usually known as the *seed*. If we want reproducible results we should pick an arbitrary seed and remember it to reproduce the same random sequence later, whereas if we want results that cannot be easily reproduced, we should select the seed in some inscrutable (and, ideally, nondeterministic) way, and keep it secret.

Knowing the seed, we can predict the output, but for many generators even without the seed it is possible to infer the current state of the generator from its output. This property is trivially true for any generator where its output *is* its entire internal state—a strategy used by a number of simple random number generators. For some other generators, such as the Mersenne Twister [Matsumoto and Nishimura 1998], we have to go to a little more trouble and invert its tempering function (which is a bijection; see Section 5), but nevertheless after only 624 outputs, we will have captured its entire internal state.

Predictability has security consequences, because it allows a variety of possible attacks, including denial of service attacks [Crosby and Wallach 2003]. If the generator is *invertible*, once we know its internal state, we can also run it backwards to discover the random numbers it generated in the past, potentially leading to severe security compromises [Goldberg and Wagner 1996; Gutterman and Malkhi 2005]. In contrast, *cryptographically secure* generators are not invertable. The only cryptographically secure generator in our sampling is Arc4Random from OpenBSD (which is equivalent to the trade-secret RC4 algorithm from RSA Security), although other, more-recent secure generators perform similarly (see Section 9). While security-related applications should use a secure generator, because we cannot always know the future contexts in which our code will be used, it seems wise for all applications to avoid generators that make discovering their entire internal state completely trivial.

One might wonder why more generators don't routinely keep more of their internal state to themselves. One answer is *speed*—outputting half as many bits could be seen as halving their speed.

2.3. Speed

Time performance is important for two reasons. If an algorithm makes heavy use of a random number generator, obviously its time performance will depend on the performance of its generator. Similarly, programmers must often choose between a randomized algorithm or a nonrandomized algorithm, and in those situations generator performance can influence their choice. For these reasons, we desire generators that are fast.

Figure 3 shows the time performance of the generators we considered in previous sections running an application that makes heavy demands on its random number generator—SmallCrush from the TestU01 suite [L'Ecuyer and Simard 2007].[4] The lower part of the bars represents the amount of time spent performing statistical checks and stays essentially constant regardless of the generator being tested; the upper part of the bar is the time spent in generator being tested. To facilitate comparison with Figure 2, the generators are listed in the same order, but to make the comparison easier the lower part of each bar is shaded differently depending on whether that generator passes or fails BigCrush—if you're viewing the graph in color, the ones that pass are shaded green.

[4] Full details of test setup and methodology are given in Section 10.

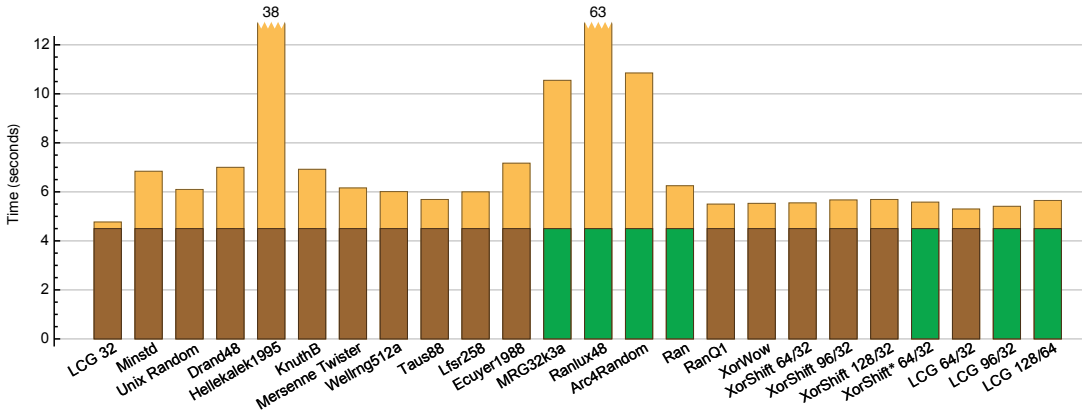


Fig. 3: Time performance of different generators running SmallCrush. Only a few of the generators that pass BigCrush are fast. The shaded lower parts of the graph represent the fixed overheads of the test suite—the shading also highlights the seven generators that pass BigCrush (as a reminder).

Many of the generators that pass BigCrush are slow, but there are *some* generators that have relatively good time performance while giving acceptable statistical performance. In particular, XorShift* 64/32 performs well,[5] but perhaps some readers are again surprised at the performance of linear congruential generators. LCG 96/32 is the fastest generator that passes all tests (if we needed 64 bits of randomness, LCG 128/64 would be the winner, but this application doesn't need 64 bits and so generating them is thus a waste of time). Also, although it is statistically very weak, the king of speed is LCG 32, running considerably faster than any of the other generators (240 times faster than the slowest generator, RanLux48).

Readers with lingering concerns about linear congruential generators might be pleased to know that there *is* a reason why we might prefer XorShift* 64/32 over LCG 96/32—it requires less space, which is our next topic for discussion.

2.4. Space, Period, and Output Range

Although most generator implementations require a constant amount of memory to store their state, the size of that constant matters. If an algorithm uses its own local or embedded random number generator, the space used to represent the state of the generator will contribute to the space overheads of the algorithm. Space usage also influences whether it seems wise to use several separate generators at once.

In addition, space usage can have an impact on speed. If the entire state of a random number generator can be represented in a single processor register, we may reasonably expect it to offer performance advantages compared to a generator that requires hundreds or thousands of bytes of memory to represent its state.

Figure 4 shows the space usage our sampling of preexisting random number generators; notice how widely different they are in how much space each uses (especially given the log scale!), and how little correlation there is between generator size and empirical performance (once again, the seven generators that pass TestU01's BigCrush battery are shaded differently (in green if you're viewing the graph in color).

[5] There are other XorShift-based generators that perform similarly [Vigna 2014a; 2014b], but XorShift* 64/32 works as an acceptable placeholder for them. These other generators are discussed briefly in Section 9.

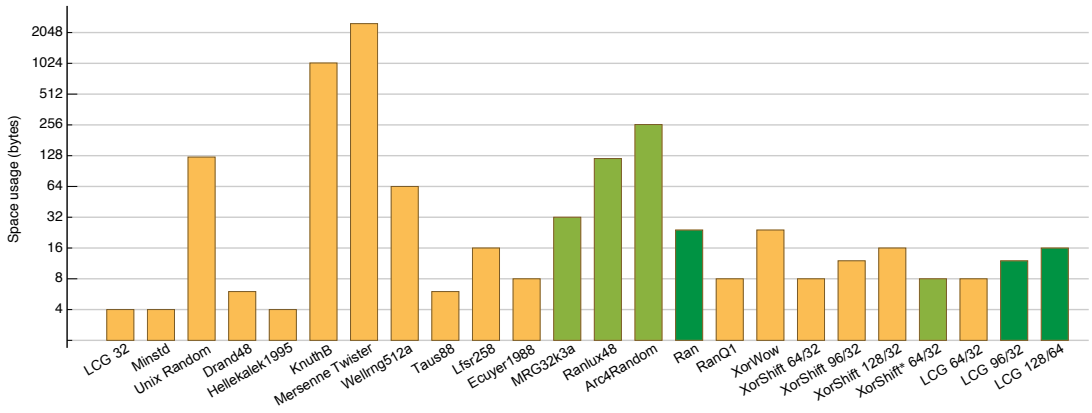


Fig. 4: Space usage of different generators. As with Figure 3, the seven generators that pass statistic tests are shaded differently, but here the group is divided in two—four generators that pass TestU01 but have some disuniformity when used as 32-bit generators are shaded differently. Only a few of the generators that pass BigCrush are modestly sized.

KnuthB [Knuth 1981; Bays and Durham 1976], one of the algorithms mandated by the C++11 standard library, taken from the previous edition of Knuth's *Art of Computer Programming* (but removed from the current edition [1997]) is particularly disappointing; we had already seen that it is slow, and it fails many statistical tests, but now we see that it is also huge.

In contrast, you may now feel a little more charitable to Hellekalek1995 [Hellekalek 1995] (an inversive generator); perhaps it is unreasonable to expect it to perform well in Crush or BigCrush with only 32 bits of state? Perhaps we ought to be delighted that it actually passed SmallCrush? We will address this question in Section 3.2.

2.4.1. The Effect of Period on Output Range and Uniformity. Although a generator with b bits of state can represent 2^b distinct states for a period of 2^b, some generators have fewer distinct states and a smaller period. For example, XorShift and XorShift* (and other generators, such as the Mersenne Twister, that can be seen as primarily employing a linear-feedback shift register) have a period of $2^b - 1$, and likewise generators based on arithmetic modulo a large prime will also have a non–power-of-two period. In Figure 4, Ran and the rightmost two LCG generators are the only ones that both pass BigCrush and have a power-of-two output range (they are thus shaded slightly differently from the other generators that pass this battery).

A power-of-two period (or a period divisible by a suitable power of two) is useful because it makes it trivial to have a uniform power-of-two output range. In practice, many applications need a random stream of x-bit values, for some x, and thus an output range of 2^r, where $r \geq x$, is highly desirable; $r = 32$ and $r = 64$ are particularly useful.

A non–power-of-two output range or period can add challenges. For example, MRG32k3a [L'Ecuyer 1999a], a generator with some positive properties (good statistical performance and long period) has an output modulus of $2^{32} - 209$, making it ill-suited for generating 32-bit integers—209 values will be never produced. Although unfortunate, such issues need not be a fatal flaw. There are a variety of fixes, including throwing away output or adding additional generators, but both techniques require additional time, space, or both. In cases where the output range is satisfactory and the

only issue is the generator's period, a common strategy is to have $b \gg r$ and the dismiss the disuniformity as "inconsequential".

A similar dismissal approach applies to Arc4Random, which has $256! \times 256^2 \approx 2^{1700}$ possible internal states, but has a period that varies depending on its initialization. Mister & Tavares [1999] discuss the period of Arc4Random's underlying RC4 algorithm in depth, and also claim that it exhibits bias, a nonuniformity claim confirmed by Basu et al. [2008], although no bias is detected by TestU01.

2.4.2. The Necessity of a "Large Enough" State Space and Period. If the period of a generator is too short, it could repeat itself while in use, which is undesirable. A larger state allows for a longer period because it allows more distinct states to be represented, but we quickly reach a point of diminishing returns. For example, given a choice between a generator with a period of 2^{128} and one with 2^{256}, we might do well to realize that if we had a trillion computers each examining one number per nanosecond, it would require more than ten billion years to mostly explore the period of the 2^{128} generator. (Even a period as "small" as 2^{56} would take a single CPU core more than two years to iterate through at one number per nanosecond.)

Long periods are sometimes advocated to support multiple random-number streams, a property we will address separately in Section 2.5.

2.4.3. The Importance of Code Size. Beyond the space used to represent the state of the generator there is also the space required for its code. Again, this space will be a constant, but the amount of space used can influence the speed of program execution by influencing cache usage and opportunities for function inlining.

Also, from a very practical perspective, the longer the code, the more likely it is to contain an implementation error. (From personal experience, I can say that implementation errors in a random number generator are challenging because they can be subtle, causing a drop in overall quality of the generator without entirely breaking it.)

2.4.4. A Summary of Size Issues. Thus, we desire a generator that uses enough space to provide a reasonable period for whatever application have in mind. Beyond that it is preferable to use as little additional space as possible, both for data and for code.

2.5. Seekability, Streams, and *k*-Dimensional Equidistribution

Let's turn to a few remaining properties that are useful for some applications: *seekability*, *multiple streams*, and *k-dimensional equidistribution*.

2.5.1. Seekability. Sometimes it is convenient if we can easily "jump ahead" an arbitrary k in the random-number sequence. For example, suppose we have a large simulation with two phases, where each phase requires 2^{36} random numbers. Wishing to be able to reproduce the results, we note the initial seed. After running the first phase, we discover that a mistake was made in the setup for the second part. It would be nice if we could rerun the second part without needing to rerun the first, but doing so requires that we advance the generator 2^{36} steps. We would thus prefer to be able to advance the generator k steps without it taking $O(k)$ time.

We say that a generator that provides a fast jump-ahead-k operation is *seekable*. Because generators are cyclic, if you jump ahead far enough you end up going around again, so we can also use jump-ahead to jump backwards.

Thanks to their mathematical foundations, most of the random number generators we've discussed actually are seekable, typically in $O(\log k)$ time. The papers that describe them do not always explicitly state how to do so, nor do their standard implementations always provide the feature, however—we will see an example of how seekability can be provided in Section 4.3.1.

In contrast, cryptographically secure random number generators are not seekable (over the long term) *by design*—we don't want people to be able to use a backward jump to discover numbers that they produced in the past. (Although a seekable generator can also prevent backward jumps by explicitly obliterating its state once its task is complete.)

2.5.2. Multiple Streams. If we wish to use multiple instances of a random number generator there is a potential hazard: unintended correlation between their outputs. For example, if we accidentally allow them to have the same internal state (perhaps we foolishly seeded both with the current time in seconds!), they will output the exact same numbers, which will hardly seem random.

One approach to providing streams to is demand a larger period (and thus larger state space) and then segment that period into areas designated for different streams. But, as we will see in Section 4.3, some generators can provide multiple streams at essentially no cost, without increasing the state space of the generator.

2.5.3. k*-Dimensional Equidistribution, Monkeys, and Shakespeare.* Another common use case for a random number generator is to provide points in a k-dimensional space, for some k. In the same way that we would like to avoid bias in the individual numbers a generator produces, we should also desire to avoid bias in the *pairs* (or triples, etc.) that it produces.

Many users are happy to know that a generator is merely *well distributed* across k dimensions but some users would prefer a much stronger property analogous to uniformity: that over the full period of the generator, *every possible* k-tuple will occur, and it will occur the same number of times.

An example from popular culture highlights this concept. There is a saying, popularized by Douglas Adams [1979] amongst others, that if you had an infinite number of monkeys typing on an infinite number of typewriters, they would produce all the great works of literature (and an inconceivable amount of dreck, too). We can cast that scenario into the world of random number generation. Suppose we have a generator that outputs 32-bit values (i.e., four bytes), and we grab its output in chunks of 16384 values at once. Each chunk will thus be 64 KB in size (which is 2^{19} bits). If we demand that the generator be 16384-dimensionally equidistributed, we can know that *all possible* 64 KB sequences of data must show up eventually over the full period of the generator, which must be at least $2^{2^{19}}$. Within that immensely huge collection of outputs lies *every* valid 64 KB zip file, some of which will contain great literature such as Hamlet. Thus, to make the saying more accurate, you don't need an infinite number of monkeys (k-tuples) to produce the works of Shakespeare, $2^{2^{19}}$ is ample.

An argument for k-dimensional equidistribution (and uniformity) goes like this: suppose you went and bought a lottery ticket every week, how would you feel if you discovered that the clerk at the store was handing you a fake ticket and pocketing the money because, at 259 million to one, you were never going to win anyway. You might, rightly, feel cheated. Thus as unlikely as any particular k-tuple might be, we ought to have *a real chance*, however remote, of producing it.

An argument against providing k-dimensional equidistribution (for large k) is that infinitesimal probabilities aren't worth worrying about. You probably aren't going to win the lottery, and your monkeys won't write Shakespeare. At least not without our rigging the game, something we'll discuss how to do in Sections 4.3.3 and 4.3.4.

2.6. Summary

We've now seen quite a litany of desirable properties, and assessed a number of popular, historically significant, and well regarded generators against those properties. Although

many of the generators have their niche—sometimes a sizable one—*none* offer all the properties we might reasonably desire.

One of the best performing was XorShift* 64/32, which adds a simple multiplicative improving step to the rather poor performing XorShift 64/32 generator.

But if you began reading the section with the belief that "linear congruential generators are bad" (a fairly widely-held belief amongst people who know a little about random number generation), you may have been surprised by how well they performed. We've seen that they are fast, fairly space efficient, and at larger sizes even make it though statistical tests that take down other purportedly better generators. And that's *without* an improving step. We will discuss such a step in Section 5, but before we develop better generators, we need to develop better techniques for comparing generators that pass statistical tests.

3. QUANTIFYING STATISTICAL PERFORMANCE

In the previous section we saw the vital importance of empirical testing, and how the TestU01 suite [L'Ecuyer and Simard 2007] had revealed statistical flaws in many previously well-respected generators. But although failing statistical tests can be a serious blow, we are left with the question of what we have learned when a generator passes a test battery. For example, if two generators A and B both pass TestU01's BigCrush battery, we might ask ourselves whether they are both equally good? If that were the case, it would mean that the linear congruential generators we saw passing BigCrush in Figure 2 were "just as statistically good" as any other generator that passes BigCrush, a state of affairs that would deeply trouble some readers.

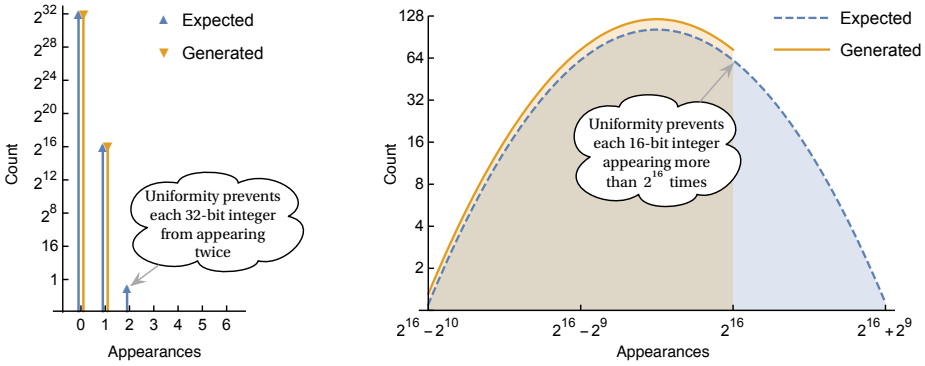
This section presents a way to go beyond mere pass/fail characterization to get a quantitative measure of how good the statistical performance of a random number generator is. The key observation is that as we limit the amount of state a random number generator has, we make it harder for it to perform well. This property doesn't just apply to some generators, as we shall see, it applies to even the most ideal uniform generator (as an absurd example, no generator, no matter how ideal, could perform well generating numbers with a single bit of state). Thus one way to determine the "statistical goodness" of a generation scheme is to reduce the size of its state to the point where it fails the test(s), and observe how close its pattern of failure is to the pattern we might expect from an ideal generator. In this section, we will develop the tools necessary to perform such an analysis.

3.1. Ideal Generators with Finite State Must Eventually Fail Statistical Tests

In Section 2.1.1, we introduced the notion of uniformity as a property we can and should demand of a deterministic random-number generator with finite state. This property requires that after a full period, all outputs will have occurred the same number of times. Uniformity is a generally desirable property because it can rule out certain kinds of bias, but it has some interesting ramifications—in some ways it requires a random number generator, even a theoretically ideal one, to be *less* random than it might otherwise be. We will see how uniformity (and the fixed-period property that is a prerequisite) eventually *requires* all uniform generators to fail reasonable statistical tests.

Let us consider a uniform generator with b bits of state and 2^b period that produces r-bit random numbers.[6] Uniformity requires that by the time we have iterated through all 2^b states, we must have produced each of the 2^r possible outputs exactly 2^{b-r}

[6] In our discussion, we'll focus on b-bit generators where the size of the output range is 2^r, where $r \le b$, but the arguments also apply to arbitrary-sized output ranges.



(a) Birthday Problem, $b = 32,\ r = 32,\ n = 2^{16}$

(b) Generalized Birthday Problem, $b = 32,\ r = 16,\ n = 2^{32} - 2^{24}$

Fig. 5: Even an *ideal* uniform generator has limits: if overtaxed it deviates from true random behavior.

times, and that $b \geq r$. For example, if we ask a uniform generator with 32-bits of state to produce 32-bit random numbers, it can only produce each number exactly once (via the pigeonhole principle), but in a stream of truly random numbers, numbers can repeat—rolling a die and getting a six doesn't diminish your chance of getting a six on the next roll. A classical statistical problem, *the birthday problem*, models the probability of repeats.

In the birthday problem, we want to know how many people we need to have present for two people to share a birthday (assuming 365 possible birthdays, each with a 1 in 365 chance of occurring for a particular person). Although you may realize immediately that we won't need to see 365 people, many people are surprised at how few are required—only 23 for a 50% chance (and 70 for a 99.9% chance). Specifically, after seeing only 23 people, there will obviously be at least $365 - 23 = 342$ birthdays that we have not seen at all, but it is also likely that two people will share a birthday because the probability that all 23 people will *avoid* sharing a birthday is

$$1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \cdots \times \left(1 - \frac{22}{365}\right),$$

which works out to be approximately 0.49 (i.e., 49%).

Figure 5(a) applies the same idea to our random-number–generation example, so instead of 365 possible birthdays, we have 2^{32} possible 32-bit integers (in general, 2^r possible r-bit integers). The figure shows the situation after we have produced a mere 2^{16} random integers, 0.0015% of the period. The column for "0 appearances" is the number of 32-bit integers that haven't shown up at all. The column for "1 appearance" counts the number of 32-bit integers we've seen just once. And finally, "2 appearances" counts the number of 32-bit integers that we've seen occur twice—it is this column that is forced to be zero for our ideal generator (because every integer can occur only once), even though we would expect to have seen at least one 32-bit integer repeat itself by now.

The problem is that with only 32 bits of state, we could only output each 32-bit integer once. We could instead output some integers twice and some not at all but that would be biased. To allow numbers to repeat, we need more bits of state than bits of output. We will consider that case next.

Suppose we decide instead to have $r = b/2$—in our running example, that will have us generating 16-bit integers. Each number will appear 2^{b-r} times over the full period of the generator, so in our example of a generator with 32 bits of state, each 16-bit integer will show up 2^{16} times (i.e., 65536 times). You might think that we now have "birthday issues" taken care of, with each 16-bit integer being able to be repeated 65536 times, and it's true that we won't have problems early on, but we aren't out of the woods yet. Let's see why.

One way to envisage this situation is to imagine that inside the generator there is a bucket for each 16-bit integer it can output. Let's zoom in on just one, the bucket of sevens. Initially, the generator has 65536 sevens in the bucket, but every time it generates a seven as its output, it uses one up. At some point, it will run out of them entirely and not be able to output seven any more. In fact, by the time the generator reaches the very last output in its period, all the buckets except one will be empty and it will have no choice whatsoever about what to output. And *that* is not random at all! But that is the condition imposed by uniformity, which insists that when we complete the period every number has been output exactly 65536 times (to avoid bias).

Figure 5(b) characterizes our example generator when we are 99.6% of the way through its period (specifically, we have only 2^{24} outputs left before we have produced all 2^{32} outputs). At this point, each 16-bit integer will have been output 65280 times on average (which is $(2^{32} - 2^{24})/2^{16}$), but obviously there will be some random variation—some integers have been produced more, some less. In fact, the variation matches a Poisson distribution, which is where the curve of the graph comes from (on a log scale), and thus the standard deviation is approximately 2^8. If we had a true source of randomness (unconstrained by uniformity and amount of state) we ought to have seen quite a few of the 16-bit integers *more* than 65536 (i.e., 2^{16}) times—that is the blue/lighter part shown to the right of 2^{16} in the graph, which is the only x-axis label you need to pay attention to (but if you wonder why the x-axis doesn't start at zero, the graph is centered around 65280, the average, and goes four standard deviations to either side). The take-away from this graph is exactly the same as from the previous paragraph—as our uniform generator nears the end of its period, it stops being able to be properly random. And just like Figure 5(a), it is also an instance of a classical problem in statistics—this time it is the *generalized birthday problem*.

Although exact solutions to the generalized birthday problem exist [McKinney 1966], for simplicity we can use elementary statistics to model and approximate the solution. Let x be the expected number of random integers that occur more than 2^{b-r} times (i.e., the area under the blue/lighter part of the graph in Figure 5(b)). Using a Poisson approximation, we have

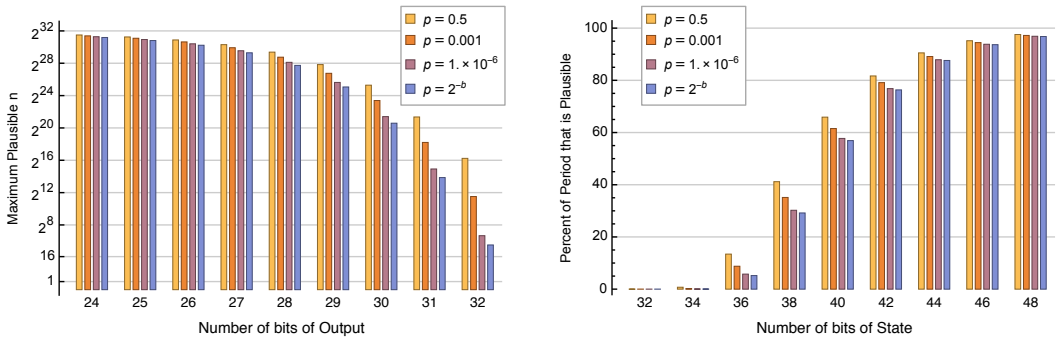
$$x \approx 2^r P(v > 2^{b-r} \mid v \sim \text{Poisson}(n/2^r)) = 2^r(1 - Q(2^{b-r}, n/2^r)),$$

where Q is the regularized gamma function. Note that x represents the expected number of events that *cannot occur* with a b-bit generator, so we desire a measure of how likely it is that we should have seen such an event by now—if it is quite unlikely, no one will notice that those events aren't there. Again we can use a Poisson approximation to calculate a p-value, given x as determined above:

$$p \approx P(w \neq 0 \mid w \sim \text{Poisson}(x)) = 1 - e^{-x}.$$

Figure 6 shows how we can apply the above formula to gain insight into the behavior of even the best uniform generator.[7] Figure 6(a) calculates how many numbers we can

[7] A small program, overtax.cpp, provided with the PCG library, solves this equation, allowing you to investigate these relationships for yourself.



(a) $b = 32$, varying r (b) $r = 32$, varying b

Fig. 6: Amount of usable output for an *ideal* uniform generator. To avoid being overtaxed very early in its period, a generator needs several more bits of state than it has bits of output.

generate before failing the birthday test with a 32-bit generator for different numbers of output bits (r), whereas Figure 6(b) assumes we desire 32 bits and shows what percentage of the period of the generator we can use for different state sizes (b bits). The figures show values for different p-values, including $p = 2^{-b}$, which has a certain appeal (with a period of 2^b, it's nice to be able to observe birthday problem events that occur with a probability of 1 in 2^b). This choice has another benefit: the $p = 2^{-b}$ columns of Figure 6(b) remain the same for other values of r—just start the x-axis labels at r instead of 32.[8]

Summary. From this discussion, we have shown that even an *ideal* uniform generator with finite state has a statistical test it will fail, and characterized exactly when it will do so. In essence, a generator "runs out of gas" as we push it close to its period (or much sooner if it has few bits of state and we insist on it outputting all of it). We call this situation a generator being *overtaxed*. We can avoid a generator being overtaxed two ways; ask for fewer numbers from it, or lengthen its period by adding more bits of state.

3.2. An Approximation for Limits on TestU01 Performance

We can use the concepts from the preceding section to approximate the number of failures we would expect running statistical tests (such as TestU01's test batteries) on an ideal uniform b-bit generator. If a test would overtax the generator by using too much of its period, we can presume it would fail the test because the generator has been taken into a zone where its ability to appear random is severely compromised. We will focus on TestU01.

The TestU01 suite is well documented, so we can know for each test in its batteries how many bits it uses from each random number, and we can measure exactly how many random numbers it needs. These two pieces of information are exactly what we need to determine whether or not it might overtax an ideal generator with a given amount of state.

Using information about TestU01 and the generalized–birthday-problem approximation from Section 3.1, I calculated, for each test in TestU01, the point at which an

[8] When $b = r$ and $p = 2^{-b}$, we can use about a hundred millionth of the period of the generator. That's a problem when $b = 32$ or $b = 64$, but when $b = 128$, a useable period of $2^{101.5}$ is not likely to be a problem!

ideal random number generator would have a birthday-problem p-value worse than 1 in 1000, and thereby determined the minimum number of state bits required to pass each test in the battery. This estimate is conservative because in practice, depending on the exact characteristics of the test, an overtaxed generator might pass the test anyway, either by chance or because the test is insensitive to birthday-problem issues.

This information indicates that an ideal generator can reasonably fail SmallCrush when $b < 32$, fail Crush when $b < 35$, and fail BigCrush when $b < 36$. Thus, in Section 2.1.2, I had indeed put Hellekalek1995 in a contest it couldn't hope to win—and its success in passing SmallCrush with only 32 bits of state *was* an achievement to be proud of.

In addition, we can now surmise that *in theory* an ideal generator could pass BigCrush using only 36 bits of state, but none of the generators we saw in Section 2.1.2 came close to this performance.

3.3. A Space-Based Metric for Generator Comparison

As we saw in Section 3.1, the point at which ideal generators start to fail statistical tests is closely connected to their period and how many bits of state they have. Empirical evidence suggests the same is true for real-world generators, as we will see momentarily. If a generator is failing statistical tests, giving it more state might allow it to pass, and, similarly, if it is passing the tests, forcing it to get by with fewer state bits can make it fail. This notion gives us a metric for determining whether one generator is better than another—give each one enough bits to pass statistical tests (and discard any that can't pass at any bit size), and then "put the screws on them", testing them at smaller and smaller state sizes to find the point at which they fail, as they eventually must.

If generation algorithm A passes a statistical test using b bits of state, and generation algorithm B requires at least $b + k$ bits of state, A is better—because if both are used in practice at the same size, A will have k bits more *headroom*. For example, suppose we will devote 64 bits to random number generation and we are trying to decide between two generators. If we have a test battery that A can pass with 40 bits of state, but B needs 63 bits to pass, then A has 24 bits more than it needs to pass our test, whereas algorithm B has only one additional bit. With only one bit of headroom, we can be reasonably concerned that if we gave B a slightly larger test (e.g., about double the size), it might not pass. Whereas we would hope that A would require a *much* larger test (about 2^{24} times larger) before it would be likely to fail.

This reasoning assumes that real-world generators behave analogously to ideal ones (e.g., needing 50% more state than an ideal generator, but otherwise being overtaxed in a similar way), but the empirical evidence given here and elsewhere [L'Ecuyer and Simard 2007] suggests that they do behave this way. L'Ecuyer and Simard [2001] explored birthday-test performance for a number of real-world generators, varying the size of a single test and keeping generator-state size the same. In this paper, we will examine the empirical performance of a small number of generators in the more comprehensive Crush and BigCrush batteries, keeping the test the same and varying bit size.

Figure 7 shows Crush and BigCrush performance for three random-number–generation algorithms that are straightforward to run at different sizes: LCG, XorShift, and XorShift*. This graph supports some intuitions that may have been building since Section 2.1.2. Specifically, LCG is quite poor at small bit sizes, much worse than XorShift, but it improves fairly quickly—at 88 bits of state LCG passes BigCrush. In contrast, XorShift begins from a better starting point but is never able to fully overcome its weaknesses as we add more bits of state. XorShift* outperforms both—a 40-bit XorShift* generator can pass BigCrush. Thus, if we had 64 bits to devote to state, the LCG and XorShift algorithms would be terrible choices, but XorShift* would be a good one. It will pass BigCrush with 24 bits of

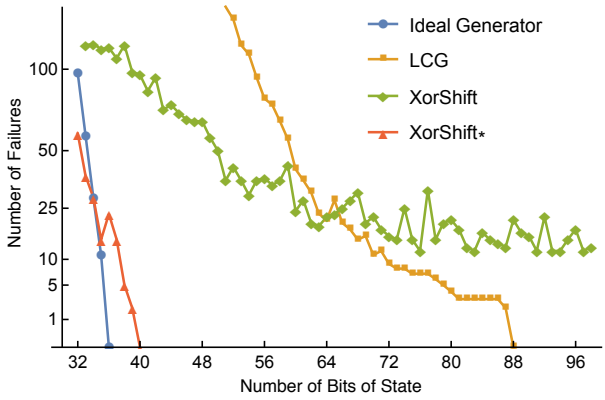


Fig. 7: BigCrush + Crush test failures at different bit sizes (32-bit output).

headroom—in other words, 24 bits more than it needs—and thus gives us reason to hope that it could withstand much more intense testing (although there is always the risk that an entirely new test could be developed that checks a previously untested statistical property and reveals a flaw).

XorShift* is a small change to XorShift—it permutes XorShift output using a multiplicative improvement step (discussed in more detail in Section 5.5.2). This simple change pushes it very close to the performance of an ideal generator. In fact, you may notice that at some points on the graph it actually *exceeds* our model of ideal performance. Recall that the theoretical model assumes that the tests TestU01 applies will detect all deviations from true randomness, but in practice some tests may be oblivious to birthday-problem–related issues.

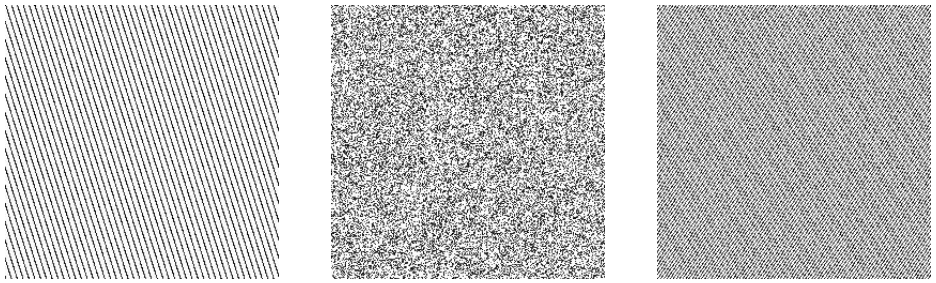
XorShift* would even be highly usable at 48 bits. Or at least it would be if its "minor" nonuniformity from a non–power-of-two period didn't start to become a problem at small sizes (an issue we mentioned previously in Section 2.4.1). In contrast, the statistical performance of the LCG scheme was less stellar—in a contest between LCG and XorShift, LCG wins, but compared to XorShift* it loses.

3.4. Summary

We now have a new concept, headroom, that quantifies statistical performance in pass/fail empirical statistical tests. Headroom is the difference between the number of bits of state a particular generation scheme requires to pass a given statistical test (or test battery) and the number bits that a given instance of that scheme actually has. Thus, a 128-bit LCG generator has 40 bits of BigCrush headroom because the LCG scheme can pass BigCrush with 88 bits of state. We have also discovered the concept of a theoretical limit on the minimum number of state bits that are required to pass a given statistical test (or test battery). In the case of TestU01's BigCrush battery, that minimum is 36 bits. Thus, although we might be even less impressed with generators that seem to require vastly larger state to pass BigCrush, we should not be especially impressed at the performance of the LCG scheme because its performance is very far from ideal. On the positive side, linear congruential generators do have other good properties, which we will review in the next section.

4. LINEAR CONGRUENTIAL GENERATORS

Linear congruential generators are one of the oldest techniques for generating a pseudorandom sequence of numbers; being first suggested by Lehmer [1951]. Despite its



(a) A bad LCG (b) True randomness (c) A "good" LCG

Fig. 8: 16-bit LCGs versus actual randomness.

age, the technique has had both enduring popularity, as well as a certain amount of ignominy (thanks a lot, RANDU [Knuth 1997]).

In this section, we will present the definition of LCGs and enumerate some of their well-known (and less well known properties). We will see that LCGs have statistical flaws, but also that they have almost all of the desirable properties that we discussed in Section 2. Because of their positive qualities, LCGs will form a key part of the PCG generation scheme (the "CG" in PCG).

4.1. Definition and Basic Properties

The generator uses a simple recurrence to produce random numbers:

$$X_{n+1} = (aX_n + c) \bmod m, \quad n \geq 0,$$

where m is the modulus ($m > 0$); a is the multiplier ($0 \leq a < m$); c is the increment ($0 \leq c < m$); and X_0 is the starting value, the seed ($0 \leq X_0 < m$). When $c > 0$ and $a > 0$, we call the generator a *linear congruential generator* (LCG), whereas in the special case that $c = 0$ and $a > 0$, we call it a *multiplicative congruential generator* (MCG), although others often call it a *Lehmer generator*.

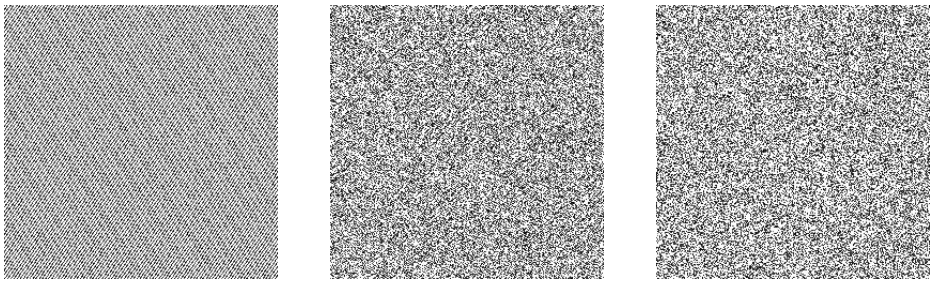
This technique has the potential to be fast given that it only requires three operations, a multiply, an add, and a modulus operation. If m is chosen to match the machine-word size, the modulus operation can be performed implicitly by the underlying hardware, reducing the number of operations to two. If we use an MCG, there is no need to perform the addition, thus allowing each step of the generator to be performed with a single multiply. In contrast to the machines of many years ago, CPUs in widespread use today can perform multiplies very quickly (e.g., at a throughput of one per cycle [Granlund 2012]), explaining the excellent performance we saw in Figure 3.

LCGs and MCGs are also space efficient; they only need enough space to store the previous value of the recurrence. In the case where $m = 2^k$ (which is convenient for k-bit machine words), the maximum period of the LCG variant is 2^k whereas the maximum period of the MCG variant is 2^{k-2} [Knuth 1997].

LCGs and MCGs are conceptually simple, well understood, easy to implement (provided $m = 2^k$, making overflow a blessing, not a curse), and typically fast. Yet, as I have alluded to a several times already, in some quarters, they have a poor reputation. Let's consider why...

4.2. Visualizations, Intuition Pumps, and the Flaw in LCGs

Take a look at Figure 8. It provides a visualization of the output of a three would-be random number generators. The one in the middle probably matches what your conception of random noise looks like, whereas the two on either side probably do not.



(a) 16-bit LCG (b) 17-bit LCG (c) 18-bit LCG

Fig. 9: The same LCG with different state sizes; using less of the period makes the crystalline structure much less obvious (although it is still there).

Before we proceed further, let us discuss how they were made. The output of a random number generator is grouped into *pairs*, which are then used to provide the (x, y) coordinates to plot a point on a $2^8 \times 2^8$ grid. Each picture (even later ones that seem very sparse), plots 2^{15} points, consuming 2^{16} random numbers. The diagram also captures temporal properties of the generator; earlier points are rendered a lighter gray than later ones (the first point is in 50% gray, so areas where there are no points at all remain clearly distinguished). Although images such as these (and 3D variants) are not new, they do not appear to have been given a short and simple name. Let's call them *randograms*.

4.2.1. Randograms as Intuition Pumps. Randograms are informal, but they can nevertheless be useful. You may not feel absolutely certain that Figure 8(b) is truly random, but it is likely that you are sure that Figures 8(a) and 8(c) are emphatically *not*. As such, they drive our *intuitions* about what is more or less random.

As with other intuitive thinking, we have to exercise caution because our intuitions can lead us astray (e.g., seeing *one* LCG look bad in *one* context and assuming that *all* LCGs are bad in *all* circumstances), so we should never draw conclusions based on intuitions alone, but they nevertheless have a useful place as a tool. Much of the theory that underlies LCGs owes its origins to these kinds of intuitions.

Before we move on to that theory, however, let's take a moment to realize that the intuitions we gathered from Figure 8 may have overstated the case against LCGs. Figure 8(c), which is repeated in Figure 9(a), used the *entire* period of a 16-bit LCG, something we know (from Section 3.1) to be questionable for even an *ideal* generator (and LCGs are *not* ideal), so we might wonder what the diagrams would have looked like with an extra bit or two of state. Figure 9(b) shows 17 bits, and you may still notice some regularities, but they are somewhat subtle. Regularities are still there in the output of the 18-bit generator depicted in Figure 9(c), but they are too subtle to clearly see. We should not be surprised by this result, given what we discussed in Section 2.4 (e.g., Figure 7)—we that know LCGs perform better with more bits.

Thus we have seen that we must take care in the intuitions we form, and we should pay attention to context; if we only see LCGs depicted with poor multipliers or using their entire period, we may mistakenly believe they are worse than they really are. Likewise, we should be careful about giving too much credit too quickly—even though Figure 9(c) looks random, flaws are still there.

4.2.2. The Flaw in LCGs. Marsaglia [1968] showed that even with the most well-chosen multiplicative constants, using an LCG to choose points in an n-dimensional space will generate points that will lie on, at most, $(n!m)^{1/n}$ hyperplanes. He wrote:

> all multiplicative congruential random number generators have a defect—a defect that makes them unsuitable for many Monte Carlo problems and that cannot be removed by adjusting the starting value, multiplier, or modulus. The problem lies in the "crystalline" nature of multiplicative generators—if n-tuples $(u_1, u_2, \ldots, u_n)$, $(u_2, u_3, \ldots, u_{n+1})$ of uniform variates produced by the generator are viewed as points in the unit cube of n dimensions, then all the points will be found to lie in a relatively small number of parallel hyperplanes. Furthermore, there are many systems of parallel hyperplanes which contain all of the points; the points are about as randomly spaced in the unit n-cube as the atoms in a perfect crystal at absolute zero.

When $m = 2^k$ there is a further problem. The period of the b^{th} bit of an LCG (where bits are numbered from the right, starting at one) is 2^b, thus although the period is 2^k, only the high bits are good and the lower order bits exhibit a clear repeating pattern. For an MCG, the bottom two bits remain fixed, resulting in a period of 2^{k-2} and bit b ($b \geq 2$) having a period of 2^{b-2} [L'Ecuyer 1999b].

On the positive side, at least the flaws in LCGs are very well understood, and (as we saw in Section 2.1.2) statistical flaws are hardly unique. For example, we saw that linear-feedback–shift-register–based generators were brought down in statistical tests (in mere seconds!). I would claim they fail because they have the opposite problem to that of LCGs—instead of being too regular, I would describe them as "too lumpy"; output for some sections of the period is decidedly improbable. Moreover, we saw in Section 3.3 that at least sometimes statistical flaws in a generator have an easy fix (e.g., the multiplication step in XorShift*), so there is every reason to be hopeful.

So, for now let us set the statistical flaws to one side in the hope that we can fix them (we'll see how in Section 5) and turn to other properties where LCGs fare a little better.

4.3. The Good Stuff: Seekability, Multiple Streams, *k*-Dimensional Equidistribution

LCGs may have statistical problems, but on the positive side they can provide all of the auxiliary properties we discussed in Section 2.5, namely seekability, multiple streams, and k-dimensional equidistribution.

4.3.1. Seekability. Thanks to the simple linear properties of the recurrence, we can actually jump ahead an arbitrary i steps in the random sequence using the formula

$$X_{n+i} = \left(a^i X_n + \frac{c(a^i - 1)}{a - 1} \right) \bmod m.$$

Brown [1994] gives an algorithm that performs this calculation quickly (specifically in $O(\log i)$ steps), without using division, using an algorithm analogous to fast exponentiation.

4.3.2. Multiple Streams. Although there are rules for the choice of constants [Hull and Dobell 1962], if we pick a power-of-two modulus and a good multiplicative constant, the only constraint on c for a full period generator is that c is odd and $0 < c < m$ (or $c = 0$ for an MCG). Every choice of c results in a *different* sequence of numbers that has none of its pairs of successive outputs in common with another sequence. We can prove this property as follows: Suppose we had two generators with additive constants c and d that transitioned from x to the same value y, thus

$$y = ax + c = ax + d \mod m,$$

and the only solution to this equation is that $c = d$.

It's worth noticing that in many programming languages, we can allow distinct random number generators to have their own distinct streams at *zero cost*. In languages

like C and C++, we can base the constant on the memory address of the state, whereas in other languages (e.g., Python), every object has a unique integer *id* that we can use.

4.3.3. Party Tricks. The above two properties allow us to perform a silly party trick with an LCG. Let's assume we're working with a 64-bit LCG. First, let us observe that

$$3935559000370003845 \times 3203021881815356449 + 11742185885288659963 \equiv_{2^{64}} 2406455411552636960$$

This equation captures a particular 64-bit LCG as it advances. The first number is one of Pierre L'Ecuyer's multiplicative constants for a good LCG [1999b], the second number is actually the little-endian encoding of the bytes of "`!Thanks,`" and likewise the last number encodes "` Pierre!`". The additive constant is contrived to make the sum work. Thus by using a contrived constant, we can make the generator produce any pair of two outputs we desire (provided that one is even and one is odd); I have contrived to make it produce the bytes of "`!Thanks, Pierre!`". Using seekability, we can backstep the generator so that it will produce this output at some arbitrary point of our choosing in the future, possibly during execution of L'Ecuyer's test suite, TestU01 [2007].

4.3.4. Very Long Periods & k*-Dimensional Equidistribution.* Some generators, most notably the Mersenne Twister [Matsumoto and Nishimura 1998], tout their long period and k-dimensional equidistribution, so we should consider whether LCGs can provide these properties.

First, let's step back for a second and observe (paraphrasing L'Ecuyer [2012]) that these properties *alone* are trivial to provide. They are true of a simple counter! A counter of b bits will toil its way through every bit pattern, showing it exactly once, for a period of 2^b. I could thus set aside 1 MB of RAM, use it as a counter, and claim it as a 262144-dimensionally equidistributed generator with a period of $2^{8388608}$, but no one would be very impressed with it.

But this terrible idea does provide a clue for how to build a k-dimensionally equidistributed generator out of k LCGs—treat each generator like a digit in a k-digit counter. For this approach to work, the generator will need the concept of *carry* to advance the the count of its leftward neighbor, but we can use the same approach we use with ordinary digits—when we hit zero, we advance our neighbor.

Also, you might be concerned that while the rightmost generator/digit counts up, the others will stay the same—we will have k-dimensional equidistribution but terrible randomness. But adding one isn't the only way to count. Imagine a three-digit counter, counting from 000 to 999; we could add 001 each time, but we could instead add 111. Observe that the sequence goes

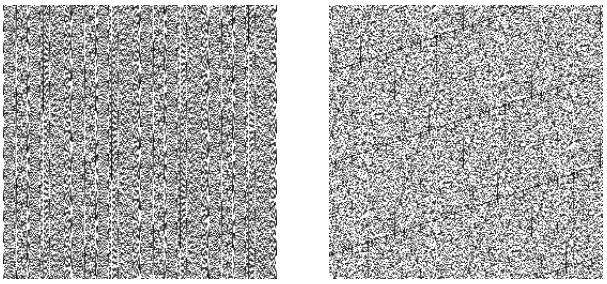
$$\ldots, 888, 999, 110, 221, 332, 443, \ldots.$$

In other words, we advance all the numbers each step, but we perform an additional advance when carry occurs. In fact, it doesn't matter *how* we advance the digits to the left, so long as we break them out of lockstep.

Thus, if we wished to have a three-dimensionally equidistributed LCG generator, we could advance all three of them by writing:

```
state1 = mult * state1  +  inc1;
state2 = mult * state2  +  inc2*(1 + (state1 == 0));
state3 = mult * state3  +  inc3*(1 + (state2 == 0));
```

and then read out the three values in turn. We could perform the carry-advance with an extra LCG iteration step, but because all that matters for "advancing the digit to the left" is performing *some* kind of extra advance and it doesn't matter what kind, the code merely uses an extra increment.



(a) 2 × 8-bit LCGs (b) 2 × 9-bit LCGs

Fig. 10: Using two tiny LCGs with two-dimensional equidistribution.

Notice that in the code above each LCG has its own additive constant to create distinct random streams, otherwise there would be embarrassing synchronicities between their values. It would have been more random to use distinct multipliers, too, but it isn't as critical.

There was nothing special about choosing three-dimensional equidistribution. We can build a k-dimensionally equidistributed generator for any k, and if the period of the base generators is 2^b, the period of the combined generator will be 2^{kb}. Finally, although we think of it as producing a k-tuple, we can also think of it as producing k values each in turn. Thus, if all we care about is having a generator with a long period, we can use this technique to achieve that goal.

Amazingly, in some ways this scheme actually does *better* at being k-dimensionally equidistributed than the Mersenne Twister does. For example, the Mersenne Twister cannot generate the all-zeros state, but the Mersenne Twister also struggles to behave randomly when its state is almost-all-zeros [Panneton et al. 2006]. In contrast, thanks to the distinct additive constants, our gang of k generators will at some point arrive at an all-zeros state, but then advance to entirely different positions and continue on their way.

In addition, we can reprise the party trick we saw in the previous section, on a larger scale (and this time without needing to contrive an additive constant). For example, you could set up 2^{14} LCGs for a total state space size of 64 KB, set all the states so that it'll read out a Zip file for Hamlet, and then backstep the generator s steps. Now, for s steps it will appear entirely random, but then it will suddenly output a Zip file containing Hamlet, and then return to more conventional random output.[9]

Figure 10 shows randograms drawn using two tiny LCGs following the above scheme for equidistribution. Clearly there is blatant structure there, but we shouldn't expect much else for such low-quality generators. Also, in the previous diagrams, we've thrown away half the bits, whereas here we've used them all and we are thus only halfway through the period. Thanks to exact two-dimensional equidistribution, if we use the complete period we'll shade the square entirely, with each point occurring exactly once.

Even though we've seen that long periods and k-dimensional equidistribution are not much of a challenge for LCGs, the structure we can see in Figure 10 reminds us that they *are* statistically weak.[10]

[9] If you want to try this trick yourself, don't forget to allow for the possibility of carry. Odds are that it won't happen, but you should check.

[10] In addition to the flaws of LCGs themselves, this scheme for k-dimensional equidistribution can add additional statistical flaws due to the uniformity of the underlying generators from which it is made. We have glossed over those issues and their remediation because they won't be a concern for the scheme the

4.4. Summary

Despite their flaws, LCGs have endured as one of the most widely used random-number generation schemes, with good reason. They are fast, easy to implement, and fairly space efficient. As we saw in Section 3.3, despite poor performance at small bit sizes, they continue to improve as we add bits to their state, and at larger bit sizes, they pass stringent statistical tests (provided that we discard the low-order bits), actually outperforming many more-complex generators. And in a surprise upset, they can even rival the Mersenne Twister at its principle claims to fame, long period and equidistribution.

Nevertheless, there is much room for improvement. From the empirical evidence we saw in Section 3.3 (and the much more thorough treatment of L'Ecuyer & Simard [2001], who observe that LCGs are only free of birthday-test issues if $n < 16p^{1/3}$, where n is the number of numbers used and p is the period), we can surmise that we may observe statistical flaws in a 128-bit LCG after reading fewer than 2^{47} numbers (which is more than BigCrush consumes but nevertheless isn't that many—an algorithm could plausibly use one number per nanosecond and 2^{47} nanoseconds is less than two days).

There are many ways to seek to improve a random number generator. One well-explored option is to combine multiple generators, but doing so adds a time and space penalty. In addition, for a fair comparison, we should always compare at the same overall bit size—it doesn't make much sense to compare a combination of four 64-bit generators against a single LCG struggling along with 64 bits—we should compare the combined 4×64-bit generator to a 256-bit LCG. Because of these issues, we'll take a different tack—rather than add more generators, we'll improve the one we have.

5. PERMUTATION FUNCTIONS AND *K*-TO-1 UNIFORM FUNCTIONS

Several random number generators (e.g., XorShift*, Mersenne Twister) use a final step to improve the output of a base generator. Instead of just outputting the generator's internal state as-is and trying to find a "more random" way to transition from one internal state to another, they instead adopt a more sophisticated method to turn the the generator's internal state into its output. Specifically, they perform a uniform scrambling operation, remapping the output in some way so as to enhance its randomness. The key idea is a *permutation function*.

A permutation function is a bijective function (i.e., 1 to 1 and onto) where the domain, range, and codomain are all the same. For example, $f(x) = x + 1 \bmod m$ is a permutation function on $\mathbb{Z}_m$ (i.e, integers modulo m), but we might reasonably expect that applying such a function would do little to make a random sequence seem more random. For our task, we desire permutation functions that *scramble* the output in some way.

XorShift* 64/32, for example, applies the multiplicative step of an MCG to improve its output. It uses a 64-bit multiplicative constant recommended by L'Ecuyer [1999b]. We know that this step will act as a scrambling function because it lies at the heart of another random number generator (in essence, we're borrowing its trick), and we can also reasonably hope that this step won't somehow inject bias because MCGs are uniform, not biased.

Interestingly, we need have no concerns about bias when applying a permutation function to an ideal random number generator, because *whatever* the permutation, it makes *no difference* to its statistical performance; either way, every possible output has a 1 in m possibility of showing up (where m is the modulus).

PCG family uses (described in Section 7.1), but, generally speaking, the more bits of state the underlying generators have, the better.

Thus, if we have a pseudorandom number and pass its output through a permutation function, there are three possibilities:

(1) It makes no difference, the statistical properties of the generator are unchanged (exactly as would happen for a true source of randomness).
(2) It improves the statistical properties of the generator, masking its flaws.
(3) It worsens the statistical properties of the generator, exposing its flaws.

Unfortunately, a problem with applying permutation functions (or at least doing so naïvely) is that we increase the chance that a user might trigger the third option, as we shall see.

5.1. The Perils of Invertability

Suppose that, excited by the idea of permutation functions, you decide to always improve the random number generators you use with a multiplicative step. You turn to L'Ecuyer's excellent paper [1999b], and without reading it closely (who has time to read papers these days!), you grab the last 32-bit constant he lists, 204209821. You are then surprised to discover that your "improvement" makes things worse! The problem is that you were using XorShift* 32/32, a generator that already includes multiplication by 747796405 as an improving step.[11] Unfortunately, 204209821 is the multiplicative inverse of 747796405 ($\bmod\ 2^{32}$), so you have just turned it back into the far-worse–performing XorShift generator! Oops.

Because of their 1-to-1 nature, *all* permutation functions are invertible, so this issue applies broadly. Of course, it's much more unlikely that you'll accidentally invert the tempering function of the Mersenne Twister, but, as mentioned in Section 2.2, a hacker might do so entirely deliberately.

Thankfully, there is a variation on permutation functions that makes inverting them more difficult—k-to-1 uniform functions.

5.2. Avoiding Invertability: k-to-1 Uniform Functions

If our function is k-to-1 rather than 1-to-1 (i.e., where *exactly* k unique inputs map to every available output), there are k possible inputs that could have lead to each output, leaving anyone hoping to invert the function with k possibilities. As k gets larger, the concept of a meaningful inverse vanishes.

Functions of this kind are widely used in computer science, usually going by the name *uniform hash functions*. In fact, we could stop at this point, just use a linear congruential generator and produce output by hashing its entire state using a high-quality off-the-shelf hash function. But even though this strategy might work, we would actually find performance to be mediocre (see Section 9 on related work, which discusses this approach), and, without careful examination of the properties of the hash function, we might not have full confidence that it was not somehow introducing bias. Thus, instead of stopping there, we will press on, and design something specifically for the task.

5.3. Permutation Functions on Tuples

This section contains the key idea that underlies all PCG generators (the "P" in PCG). The idea is simple, yet powerful when applied to our task.

---

[11] This generator, although plausible, is not a widely used generator nor mentioned elsewhere in the paper (except implicitly as a plotted point in Figures 7 and 15). It fails statistical tests, including SmallCrush. Normally we use XorShift* 64/32, which performs better, but that would not work for our accidental inversion scenario—thanks to its 32-bit output, its 64-bit multiplication is not so easy to invert.

Suppose you have two permutation functions $f : A \to A$ and $g : B \to B$, and you would like a permutation function on the Cartesian product of their domains, $A \times B \to A \times B$. One possible function would be

$$fg(a, b) = (f(a),\ g(b)).$$

Let us call this a *simple* permutation function on $A \times B$ tuples because we are about to examine another, more powerful, option. Suppose instead that rather than applying two functions to both sides of the tuple at once, we focus all our efforts on one side, say, the right-hand side. We set up a *family* of permutation functions on B, where we have a function f_a for each $a \in A$. In this case, the following function is *also* a permutation function:

$$f_*(a, b) = (a,\ f_a(b)).$$

Let us examine two arguments as to why f_* is a permutation function on $A \times B$. First, we can observe that the Cartesian product marries *all possible combinations* of elements of A with elements of B, thus, every element of B is passed to *each* of the permutation functions in the family exactly once. Second, more practically, there is an obvious way to invert f_*, given that a was passed through unchanged—thus we can apply the inverses for all the functions in our family in the exact same manner.

Obviously, there was nothing special about applying f on the right-hand side; if we had a family, l, that operates on the left-hand side, we could perform $l_*(a, b) = (l_b(a),\ b)$. We can also compose multiple permutations together, whether they are simple or family-based, and the idea extends to k-tuples because, for example, $A \times B \times C$ is isomorphic to $(A \times B) \times C$. Finally, because our topic of interest is a random number generator with b bits of state, it's worth realizing that we can break numbers in $\mathbb{Z}_{2^b}$ (numbers represented with b bits) into pairs from the Cartesian product $\mathbb{Z}_{2^k} \times \mathbb{Z}_{2^{b-k}}$; in other words, using the k leftmost bits as the first member of the pair and the remaining bits as the second.

In the context of random number generation this technique is powerful because it allows a random number generator to *apply its own randomness to itself*, in a uniform way. In particular, if a comes from a random distribution, the permutation functions in the family f will be applied in an entirely unbiased, yet random, way. This property is valuable in general, but it is extremely useful for linear congruential generators where the high bits are fairly random and the low bits are not (a property we previously discussed in Section 4.2.2).

Finally, if we drop a from the result, the function becomes a $|A|$-to-1 uniform function. We will have woven together several permutation functions, selected at random, and then removed any clue as what came from where. (Note also that, using the same arguments we have already made, we can apply function families exactly analogously for k-to-1 uniform functions as we do for permutation functions—if $f_a : B \to C$ is a family of k-to-1 functions for every $a \in A$, then $f_* : A \times B \to A \times C$ is also a k-to-1 function.)

The application of powerful composable permutation functions is the cornerstone of the PCG generation scheme, coupled with the technique of discarding enough information to make the permutation impractical to invert. In principle, a PCG generator could use a bizarre function family of entirely unrelated permutation functions, weaving them together randomly, but in deference to both execution speed and our sanity, we will focus on families of very closely related permutation functions.

5.4. A Gallery of Permutation Primititves

In this section, we will look at a few permutation functions (and k-to-1 uniform functions) that are particularly well suited to turning the internal state of an LCG into an output value.

To assist our intuitions, I will provide the same "randogram" visualizations we saw in Section 4.2; in particular, we will apply them to the *pathologically bad* LCG we saw in Figure 8(a). In practice, we would not be using such a bad LCG, but the example is nevertheless useful for developing an informal notion of what the projection performed by the permutation is doing—if I had used a more realistic LCG such as the one from Figure 9(c), improvement would still be occurring, but you wouldn't be able to see it.

5.4.1. Notation. Before we get started, let's define some notation.[12]

— If f and g are functions, $f \circ g$ is their composition.
— If $\langle p, q\rangle$ is a pair, $\pi_1\langle p, q\rangle = p$ and $\pi_2\langle p, q\rangle = q$ (i.e., the usual projection functions).
— $\text{split}_t(n) : \mathbb{Z}_{2^b} \to \mathbb{Z}_{2^t} \times \mathbb{Z}_{2^{b-t}}$ is the bitwise partition function, making a pair from the top t bits of n and the remaining bits of n, defined for all b.
— $\text{join}\langle p, q\rangle : \mathbb{Z}_{2^t} \times \mathbb{Z}_{2^b} \to \mathbb{Z}_{2^{t+b}}$ is the bitwise join function, undoing any split, defined for all t, b.
— If $n, m \in \mathbb{Z}_{2^b}$, $n \oplus m$ is their bitwise exclusive or, and $n \otimes m$ is multiplication $\bmod 2^b$.
— If $n \in \mathbb{Z}_{2^b}$ and $0 < c < b$, $p \circlearrowright c$ is the bitwise clockwise rotation of c bits.
— If $n \in \mathbb{Z}_{2^b}$ and $0 < c < b$, $p \triangleright c \in \mathbb{Z}_{2^{b-c}}$ drops the lower c bits (the unsigned right shift operator in C, >>). We could define it as $\pi_1 \circ \text{split}_{b-c}$.
— If $n \in \mathbb{Z}_{2^b}$ and $0 < c < b$, $p \triangleleft c \in \mathbb{Z}_{2^{b+c}}$ adds the c zero bits on the right-hand side of n (the left shift operator in C, <<).
— If $n \in \mathbb{Z}_{2^b}$ and $0 < c < b$, $p \boxright c \in \mathbb{Z}_{2^c}$ keeps the upper c bits and drops the lower ones. It's just a convenient way to say $n \triangleright (b - c)$.
— If $n \in \mathbb{Z}_{2^b}$ and $0 < c < b$, $p \boxleft c \in \mathbb{Z}_{2^{b-c}}$ keeps the lower c bits and drops the upper ones (usually done by masking in C). We could define it as $\pi_2 \circ \text{split}_{b-c}$.

In all situations, we will want to permute b bits of state from the generator into r bits of state to provide as output from the generator. Thus we will desire a 2^{b-r}-to-1 uniform function. Let us call this function the *output function*.

5.4.2. Dropping Bits Using a Fixed Shift. We have already discussed one simple 2^{b-r}-to-1 output function: dropping low-order bits. For LCGs and MCGs, the low-order bits have a very short period so this simple transformation significantly improves their statistical properties. In the randograms we looked at in Figures 8 and 9, we were looking at the high 8 bits of a 16-bit generator; in the first eight images of Figure 11 we move our 8-bit window down to see the increasing weakness of the generated output. Thus, taking the high 8 bits seems wise.

If we were coding our toy generator (which has 16 bits of state and 8 bits of output) in C, we would most likely write `state >> 8` to drop the low 8 bits and keep the high ones, but in more formal notation and general terms, we could say that for r result bits from b bits of state, the 2^{b-r}-to-1 uniform function that we're applying is just the $\triangleright$ function, $\pi_1 \circ \text{split}_b$.

5.4.3. Dropping Bits Using a Random Shift. Dropping the low 8 bits is a 2^8-to-1 uniform function, as are all the other bit-dropping options, which are depicted in Figure 11(a . . . h). Perhaps that gives you an idea; they could be a family, couldn't they? We don't have many bits to play with, so we only have two choices, a family of two members or a family of four. Figure 11(i) shows the result of allocating the high bit to family-member selection, and Figure 11(j) shows the result of allocating the top two bits. Note that the more high bits we use, the weaker the generators we will be combining. It

[12] As with most papers, you can adopt the "read the paper but skip the math" approach. Everything given in formal notation is also explained in English.

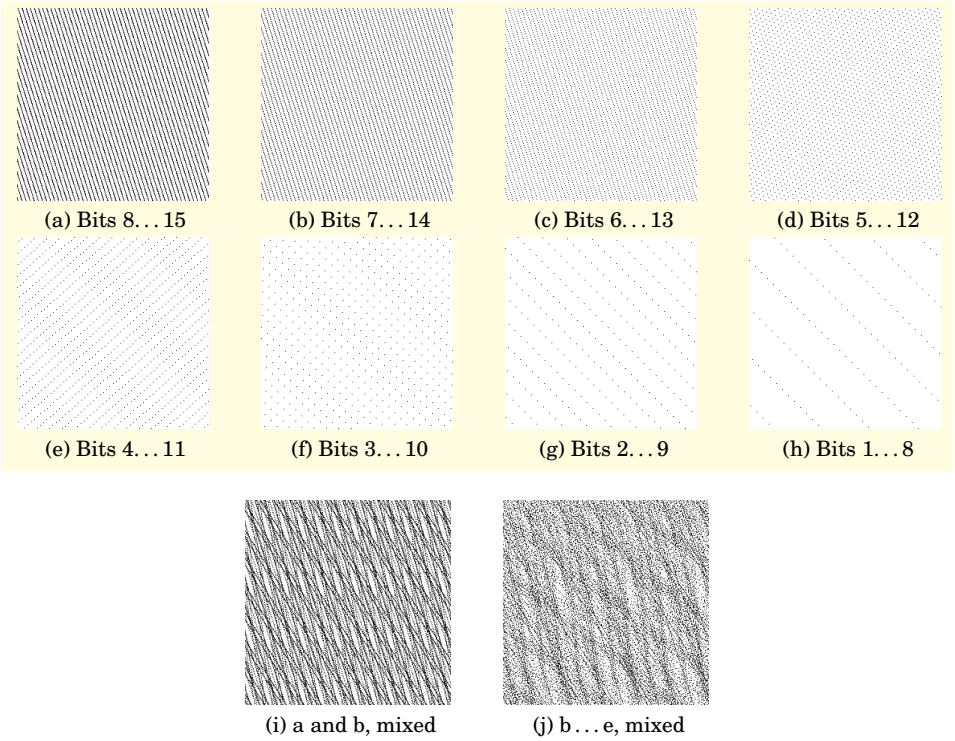


Fig. 11: Using shifts to drop bits, both fixed shifts and random ones.

might be tempting to instead build a family from a, b, c, and d instead, but that would be *cheating* and we would no longer have a permutation function.

From an intuitive perspective, this transformation is clearly an improvement. We began with a terrible LCG, and although the result clearly has some structure it certainly *seems* at a gut level like it is an improvement.

Let's take a second to describe these concepts formally. On b bits, our family of shifts is

$$f_c(n) = (n \triangleright c) \boxtimes r,$$

where r is the number of result bits we are to produce. If we use t top bits to decide on the shift, our output function becomes $\pi_2 \circ f_* \circ \text{split}_t$.

Now let's see it as C code, this time for a 64-bit generator with 32 bits of output, using a family size of eight. I'll include the multiplication step to give us the *entire code* for random number generation. I'll write the code tersely, writing the constants directly and allowing the machine-word size to perform truncation implicitly.

```
state = state * multiplier + increment;
uint32_t output = state >> (29 - (state >> 61));
```

That's *it*. Two lines. It's so short that it might be hard to believe (especially if you started reading the paper in the middle looking for the good stuff—yes, I know there are readers who do that). The first line is a standard state advance for an LCG, and the second applies our output function, which is a 2^{32}-to-1 uniform function based on function families. From Figure 11(j), where we applied the analogous output function to

a pathologically bad 16-bit LCG, we have strong intuition that the transformation is an improvement.

As you might expect for such short source code, it compiles to a handful of instructions and executes blazingly fast (we will look at the performance of the PCG family in detail in Section 6.1). But what about quality?

First, let's look at things from a theory perspective. In essence we are running eight (related) linear congruential generators, with moduli and periods between 2^{61} and 2^{54}, and randomly choosing between them using another LCG with period 2^{64}, but thanks to the properties of permutation function families, we guarantee that every one of those eight LCGs is given exactly the same number of turns, doing so in such a way as to preserve uniformity. Even though the bits of each underlying LCG have a short period, the high bit of the LCG, which has a 2^{64} period, affects the shuffling, meaning that all bits of the output have a 2^{64} period.

Despite every bit of the output having a 2^{64} period, we should realize that we're moving a window over the same generator, and thus even as we move the window, it is possible for *some* flaws to still leak through. If, for example, we applied this output function to a generator that had a tendency to sometimes output big clumps of zero bits, wiggling a window around would do little to mask it. For LCGs, however, these issues are likely to be less of a problem because their chief issue is *too much* regularity.

Empirically, we can turn to TestU01 [L'Ecuyer and Simard 2007] for some quantitative data. As we saw in Section 3.3, an unadorned LCG passes BigCrush with 88 bits of state. This algorithm can be applied at any size b where $b \geq r+2$, so we can test it at a variety of sizes. It passes with 58 bits of state, making an empirically good 64-bit LCG-based generator practical.

That's great, but perhaps you're concerned that six bits isn't enough headroom—we could reasonably surmise that if someone developed MassiveCrush that ran for a month using 32 GB of RAM, this generator would fail. That's an entirely valid concern. But, as we shall see, in the world of PCG generators, this one, which I'll name PCG-RS (where RS is for random shift), is the *weakest* of those we'll name. Its only redeeming features are its simplicity and speed.[13]

5.4.4. Rotations, Fixed and Random. In the last section, we took a well-known action on LCGs, dropping bits, and randomized that. This time, we'll focus on a well-known *property*, specifically that the high bits are the most random. They are—unless we move them somewhere else, which we can do with a bitwise rotate. Figure 12 captures the effect on the lattice of a rotation—in essence it changes our perspective on it.

People don't often rotate LCGs, but all these different lattice structures are prime candidates for mixing together randomly by considering all eight possible rotations as a family of permutation functions. Thus, the family is

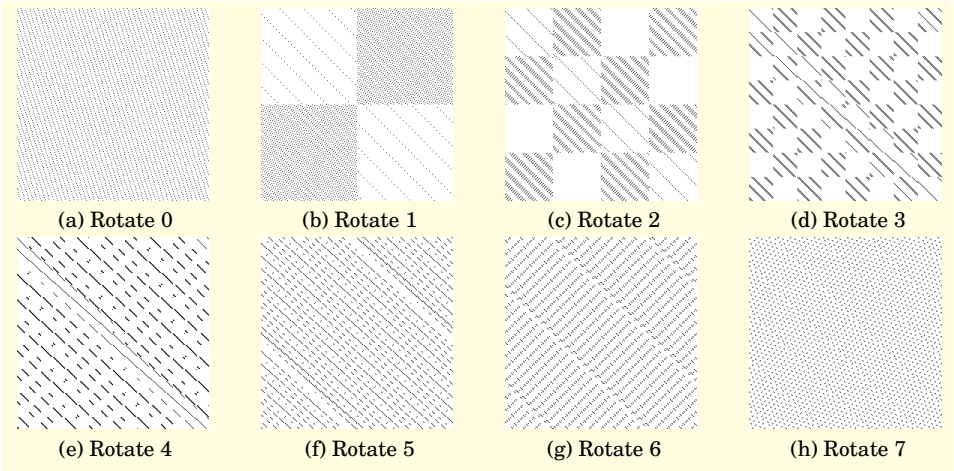
$$f_c(n) = (n \boxdot r) \circlearrowleft c,$$

where r is the number of bits of result. If we use $t = \log_2 r$ top bits to select the rotation, our output function is, as before, $\pi_2 \circ f_* \circ \text{split}_t$.

We can see the result in our running example in Figure 12(i). Again, we have considerably improved the randomness of our pathologically bad LCG, supporting the intuition that this family of permutations is useful. In fact, if you compare the result with Figure 11(j), you may think it looks a little "more random". If so, you'd be right.

First, rotation is a significant perturbation of the lattice structure creating novel lattices that would never have been generated by a vanilla LCG. Second, whereas

[13] To me, it also has sentimental value because it was the first one I came up with.



(a) Rotate 0 (b) Rotate 1 (c) Rotate 2 (d) Rotate 3

(e) Rotate 4 (f) Rotate 5 (g) Rotate 6 (h) Rotate 7

(i) a . . . h, mixed

Fig. 12: Applying rotates to bits 6 . . . 13.

random shifts needed to use bits 4 . . . 11 some of the time, we are drawing the lattice only from bits 6 . . . 13 (while bits 14 . . . 16 provide the amount of random rotation to do).

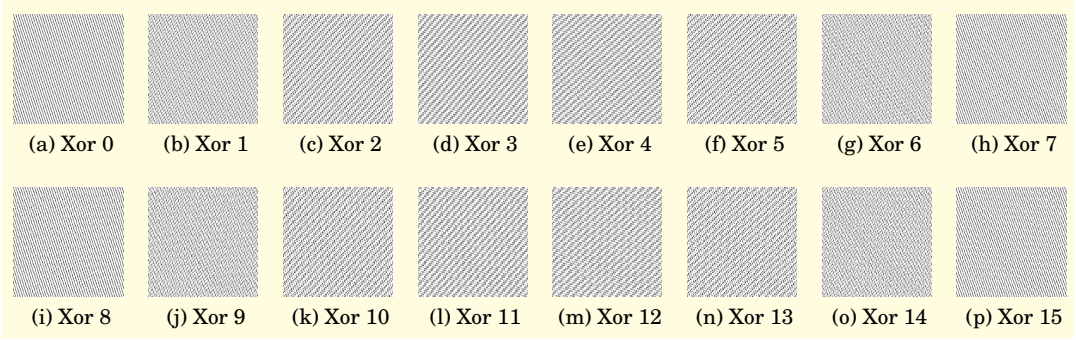
As with random shifts, the permutation has changed the period of all output bits. For our toy 16-bit generator, they all have period 2^{16}. Because we are using bits 6 . . . 13 to provide the input to the rotation, this subgenerator has an overall period of 2^{13} and will repeat eight times, but for each repeat, it is placed at a unique rotation (by bits 14 . . . 16), and all the rotations for all numbers produced by the sub-generator are different.

As for empirical testing, random rotations (PCG-RR) passes BigCrush at 53 bits, which is a significant improvement over PCG-RS (which passed at 58 bits). We might surmise that a ReallyMassiveCrush test designed to overtax a 64-bit incarnation of PCG-RR will need to work 2048 times as hard as BigCrush, which means running for well over a year and using 1TB of RAM to do so. But random rotation was also low hanging fruit—we can still do better.

5.5. Xor and Xorshifts

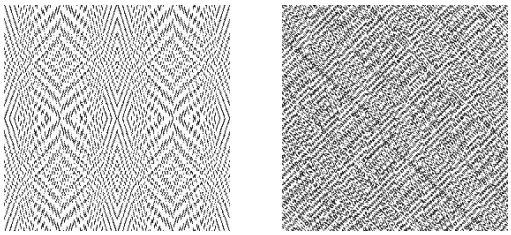
A bitwise xor operation is a permutation, it flips some bits in its target, and can be inverted simply by repeating the operation. Figure 13(a . . . p) shows the results of applying different xors to some of the bits in our running example. The pictures are small because it's just more of what we've already seen—they're simply a permutation on the lattice. And, once again, they form a family,

$$f_c(n) = n \oplus g(c),$$



(a) Xor 0 (b) Xor 1 (c) Xor 2 (d) Xor 3 (e) Xor 4 (f) Xor 5 (g) Xor 6 (h) Xor 7

(i) Xor 8 (j) Xor 9 (k) Xor 10 (l) Xor 11 (m) Xor 12 (n) Xor 13 (o) Xor 14 (p) Xor 15



(q) Maximize Periods (r) Avoid Invertability

Fig. 13: Applying Xors to bits 9. . . 12.

where g is defined as follows: if $n \in \mathbb{Z}_{2^p}$ and $c \in \mathbb{Z}_{2^q}$, then $g : \mathbb{Z}_{2^q} \to \mathbb{Z}_{2^p}$ is defined as $g(c) = c \lhd (p - q)$; in other words, it is the function that pads c out with zeros on the righthand side so that it has the same number of bits as n.

But, something is different this time. Unlike previous operations we've considered, applying this family of permutation functions gives us a well-known and well-analysed operation, (rightward) *xorshift*, which is already known to be a permutation function.[14] Thus we are able to see them both through the perspectives of previous work and though the lens of permutation families.

Conventional wisdom about xorshift would have us focus on maximizing the period of the bits. If that is our goal, we should write our output function, o, as

$$o(n) = \text{join}(f_*(\text{split}_{r/2}(n \boxdot r))).$$

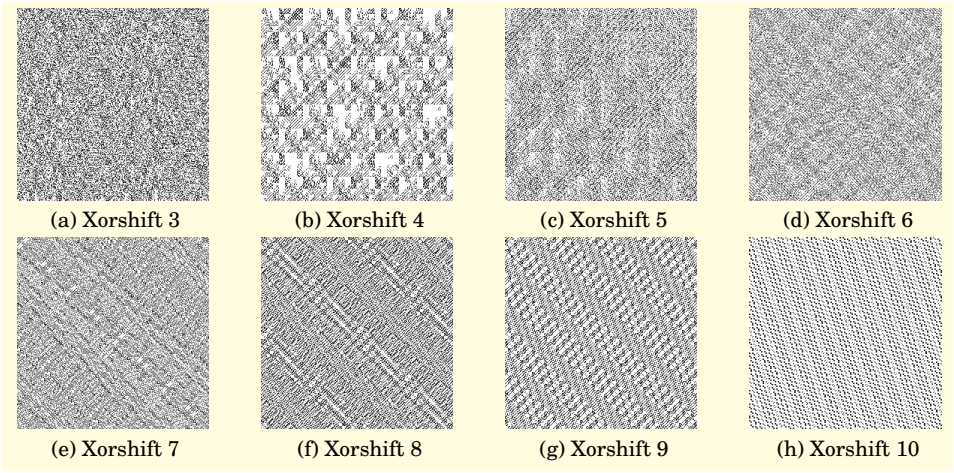
In other words, focus on the top r bits, and then perform an xorshift of $r/2$ bits.

Thus, for a 16-bit generator wanting 8 bits of result, the periods of the bits of output would be, from high bit to low bit, $2^{16}, 2^{15}, 2^{14}, 2^{13}, 2^{16}, 2^{15}, 2^{14}, 2^{13}$. That option is shown in Figure 13(q) and you'll immediately notice that something looks different. What's going on. . . ?

Pursuing the goal of maximizing bit periods, we haven't constructed the output function following our usual rules. In the two previous examples, we defined it as $\pi_2 \circ f_* \circ \text{split}_t$, and we could have done so here (with $t = r/2$), as shown in Figure 13(r).

What is the difference between these two options? The first, maximize–bit-periods, version is trivially invertible because we kept the top bits in the output rather than throwing them away. The second, hard-to-invert, version adds a valuable property but

[14] In this construction, f_c can only represent xorshifts where the shift is at least as large as the number of bits being xored; smaller shifts must be performed in stages via function composition. We won't use f_c to perform any xorshifts where this issue arises.



(a) Xorshift 3 (b) Xorshift 4 (c) Xorshift 5 (d) Xorshift 6

(e) Xorshift 7 (f) Xorshift 8 (g) Xorshift 9 (h) Xorshift 10

(i) a . . . h, mixed

Fig. 14: Applying Xorshifts.

at some cost; the periods of the bits are now $2^{16}, 2^{15}, 2^{14}, 2^{13}, 2^{12}, 2^{11}, 2^{10}, 2^{9}$, which is no better than they were to begin with.

At this point, we're left in something of a quandary, tugged in different directions by these two principles. We could go with gut intuition based on the diagrams, which probably leads us towards the second, or we could insist that bit periods are what matters and stick with the first.

For a moment, let's adopt an experimentalist approach and see if TestU01 can help settle the argument (mindful of any stern disapproving looks, as the test suite ought to be used as final confirmation, not a design tool!). It votes with its feet for the second approach by a staggering margin—it passes at 60 bits, whereas the conventional-wisdom approach doesn't pass until 70 bits; you could even claim that TestU01 considers the second option a thousand times better! Perhaps applying an invertible permutation to a low-quality random number doesn't fool TestU01?

But we can hardly make decisions based on diagrams, intuition and speculation about how "perceptive" TestU01 is. And it's worse, because these aren't the only xorshifts we could have done—there are several ways we could have constructed and applied the permutation family. Wait a second... There are several permutations we could apply, as in, a family of them. And we're having a hard time choosing... So perhaps we should choose the xorshift randomly? That is exactly what we will do in the next section!

5.5.1. Random Xorshift. Once again we'll apply a familiar strategy to our example. We will use the top three bits to choose members of a family of eight permutations, but this time the permutation is an xorshift. You can see the results in Figure 14.

Constructing the function family is a little challenging because we will use the top t bits two ways at once—we will use them to both define *how much* of an xorshift to apply and also to be part of the xorshift itself, all without breaking any of the rules. (We could avoid this dual-use approach by grabbing t spare bits that we otherwise wouldn't need from elsewhere in the state, and xoring them with the top t bits to create t more bits with the same period but different values. But xors take time, and we don't always have spare bits at really small sizes. So instead we'll use the same bits two ways.)

To do so, let us define our family function $f_c : \mathbb{Z}_{2^{b-t}} \to \mathbb{Z}_{2^r}$, where $c \in \mathbb{Z}_{2^t}$, as

$$f_c(n) = ((n \mathbin{⋗} c) \boxminus r) \oplus (c \lhd (r - c)),$$

where the $⋗$ operator is rightward xorshift (i.e., $x ⋗ y = x \oplus (x \rhd y)$), and apply it in the usual way, $\pi_2 \circ f_* \circ \text{split}_t$, where t is the number of top bits we wish to devote to selecting a random xorshift (in essence I have performed the top part of the xorshift manually, but the bottom part using the $⋗$ operator).

This equation may look a bit daunting, but we can check it easily. If we set $r = b$, the family functions should be permutations, and in this case we can quickly verify that for all c, f_c is straightforward to invert.

As you might hope, this output function is an improvement over our previous ones, and is able to pass BigCrush at only 45 bits. Given that this function is really the composition of two functions, we shouldn't be surprised that it performs better, but this improvement also reminds us that the composition of permutations may be better than single ones, an idea we'll look at in more detail in Section 6.

5.5.2. Modular Multiplication. The final permutation function we'll consider doesn't need function families. Is is the multiplication step that forms the core of any good MCG. As we discussed earlier, it is sometimes used in combination by other generators as an improving step. For example, there are known weaknesses in XorShift generators, so XorShift* generators such as RanQ1 [Press et al. 2007] add an MCG multiplication as final step to improve their output.

In contrast, adding an MCG multiplication (alone) to an LCG or MCG makes very little difference to the quality of the output (because two multiplications are identical to one combined multiplication), inspiring a rule of thumb that it is better to combine *different* approaches rather than add more of the same thing [Press et al. 2007].

But our previous two output functions, xorshift and randomized xorshift, pushed good random bits to the right, and thus a multiplication can shift the randomness back to the left again and won't be undoing our earlier multiplications.

5.5.3. And Many More... Just Follow the Rules. These are just a few of a myriad of possible permutations open to us. There are other examples of simple permutations that can be cast into the framework of function families and applied as a tuple permutation, and I would encourage you to think of some yourself. To that end we will finish with a *practical* perspective on how to write your own. I hope this discussion will be particularly useful to any readers who were daunted by the mathematical formalisms used in the preceding subsections (although, if you are a programmer, I would encourage you to recognize that math is much like another kind of code).

Here are the rules for building new permutation functions, in English:

— Bits are precious. You can throw them away but you can't make new ones, and you can't even duplicate the ones you have.
— Divide the bits into two *disjoint* groups, the target bits and the control bits.
— The control bits determine *how* to mess with the target bits (sometimes I call them the *opcode*). The "messing" is only limited by your imagination, but it must be a permutation. If you can't undo it and be back with what you had, it's not a permutation.

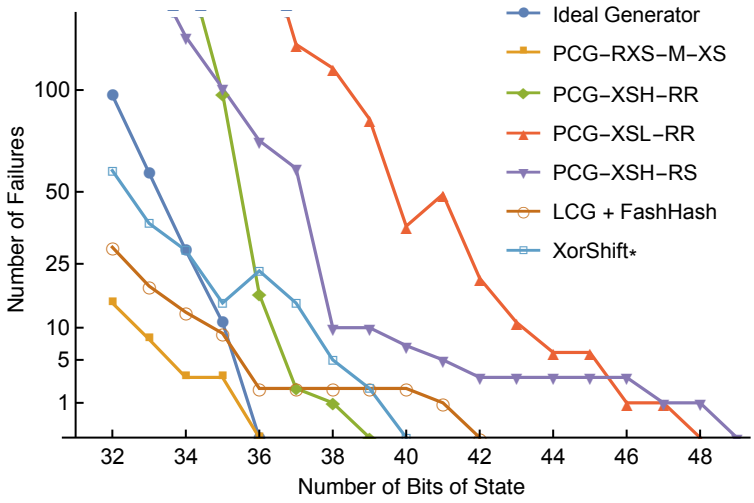


Fig. 15: BigCrush + Crush test failures at different bit sizes.

— While bits are acting as control bits, they must *not* be modified.
— You can do as many rounds as you like, so bits that were used as control bits in one round can become target bits in the next.

The above rules are just a restatement of the mathematics, nothing new. (I find this perspective useful, however. It's how I think when writing code. It's like a game, and it's actually quite easy to play.)

5.5.4. Summary. In this section we've built some useful output functions that can efficiently scramble the output of a linear congruential generator. Scrambling itself was never the difficulty, it was doing so in as few operations as possible. All the functions except MCG multiply are composed from very small number of very fast bitwise operations.

In the next section, we will embrace the composability of permutations to do even better.

6. SOME MEMBERS OF THE PCG FAMILY

Having constructed some basic building blocks, we can compose them together to produce even stronger output functions that form highly usable generators at a variety of bit sizes. The task now becomes one not of theory but engineering; we must balance trade-offs. For example, a particular output function may offer excellent statistical results, but lose out to others because it isn't *quite* as fast as they are. Different applications need different things, so we can tailor the output function to favor a particular criterion.

Because there are a myriad of ways we could combine permutation functions, we will only discuss a small subset of them, ones that are interesting in some way.

6.1. Performance

Figure 15 shows the statistical performance of the members of the PCG family that I have chosen to highlight (we will cover them individually in Section 6.3).[15] All the

[15] Figure 15 also includes the performance of applying an off-the-shelf hash function, Fast Hash, which we will discuss in Section 9, Related Work.

chosen members pass BigCrush at 49 bits or less, providing ample headroom for use at 64 bits. Some members pass at much fewer, most notably PCG-RXS-M-XS, which passes at 36 bits, the minimum permitted by the theoretical model introduced in Section 3.2. In fact, at sizes less than 36 bits, it exceeds those predictions, which is possible because the model assumes that *all* the tests in BigCrush will notice when a random number generator is forced, by its uniformity, to deviate from truly random behavior, but whether a particular test actually detects the deviation depends on what exactly it is testing and how stringently it is doing so.[16]

The other generators turn in weaker (but still excellent) statistical performance in furtherance of another goal, speed. The statistically weakest ones are only intended for fine-tuning 128-bit generators that, by virtue of their large size, can already pass BigCrush with significant headroom unassisted.

In contrast, Figure 16 and Table I summarize other properties of our family members beyond statistical performance, including their speed. For comparison, the bar chart and table include the most notable and best performing generators from Section 2 (although for space reasons some popular generators from Figure 1 that do not even survive even minimal statistical testing are not included). All the competing generators except XorShift 64/32 and Arc4Random do not survive statistical testing. As a reminder, XorShift 64/32 and RanQ1 are the exact same generator, but the latter claims that you can use all 64 bits of its output, whereas the former does not.

Both the bar chart and the table group the generators into two categories, those that expose the state of the generator in their output such that it can be *trivially* reconstructed from the generator's output and those that do not. This issue is discussed in more detail in Section 6.2.2.

The full measurement details are given at the very end of the paper (in Section 10) because they're uninteresting for most readers, but there are some points worth noting. All the PCG generators are *very* fast; so much so that major speed differences occur depending on whether a single-cycle addition operation is present or not (as witnessed by the MCG vs. LCG variants). Similarly, issues such as register selection and instruction scheduling make a difference at this low level, so I have given every generator a chance to shine (even non-PCG ones), by compiling them all with two compilers and two optimization settings (purportedly choosing between optimizing for speed and code size) and picking the fastest code.

Table I also shows the size and periods of these generators. All of the PCG generators except those marked EXT are very compact. The EXT variants use the extension techniques that will be described in Section 7.1. For example, PCG-XSH-RS 64/32 (EXT 1024) provides 1024-dimensional equidistribution and a period of 2^{32830}.

From this data, you might surmise that the fastest generators are the most preferable, but, for many applications, it is likely that a 0.1 nanosecond difference in the time to generate a random number may not make a significant difference to performance—even for demanding applications, it is probable that memory bandwidth is likely to be more of a concern than tiny differences in generator speed.

6.2. Implementation Notes

Before we discuss specific generators, we will examine some commonalities between their implementations, as provided by the PCG library.[17]

---

[16] For the curious, it is only the MaxOft test that brings down PCG-RXS-M-XS at 35 bits. Depending on luck, sometimes it passes, but too often it gives a 1 in 10,000 p-value, and that's just not quite good enough. See Section 10 for more about test methodology.

[17] The PCG library is available at http://www.pcg-random.org/.

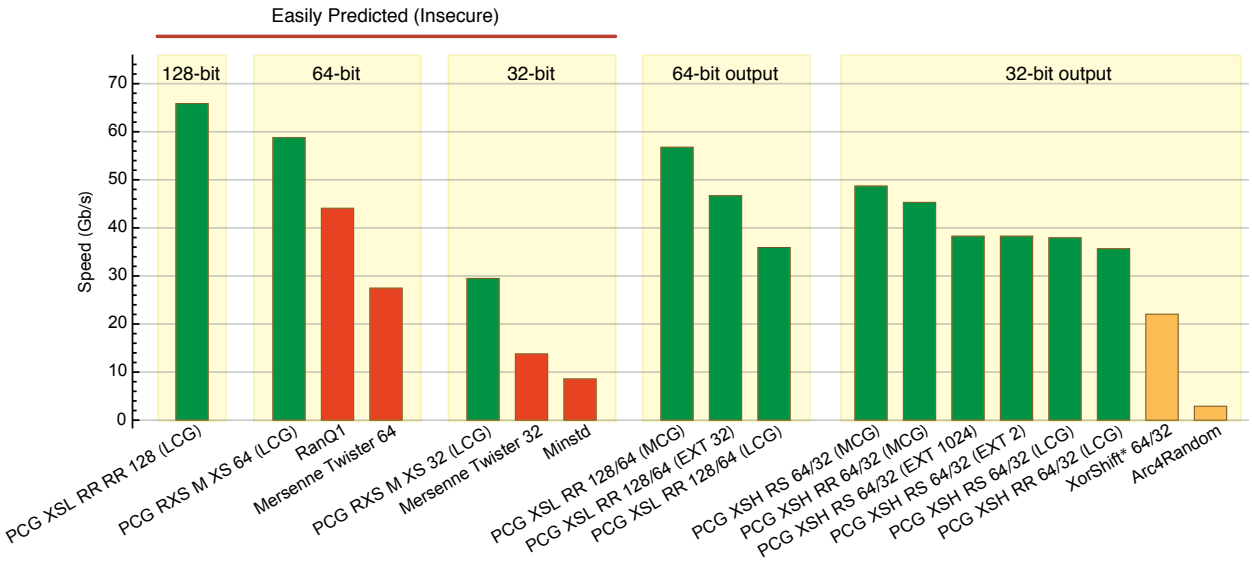


Fig. 16: Benchmark performance contrasting PCG generators against widely used generators. Higher Gb/s is better. All members of the PCG family are faster than other generators in their class. RanQ1, Mersenne Twister, and Minstd also fail statistical tests.

Method	Period	State (Bits)	Output (Bits)	Speed (ns/rand)	Speed (Gb/s)	Best Compiler
PCG XSL RR RR 128 (LCG)	2^{128}	128	128	1.81	65.86	g++ -O2
PCG RXS M XS 64 (LCG)	2^{64}	64	64	1.01	58.77	g++ -O2
RanQ1	$2^{64}-1$	64	64	1.35	44.09	g++ -Os
Mersenne Twister 64	$2^{19937}-1$	20032	64	2.17	27.48	g++ -O2
PCG RXS M XS 32 (LCG)	2^{32}	32	32	1.01	29.49	g++ -Os
Mersenne Twister 32	$2^{19937}-1$	20032	32	2.16	13.79	g++ -O2
Minstd	$2^{31}-2$	64	31	3.36	8.59	g++ -O2
PCG XSL RR 128/64 (MCG)	2^{126}	128	64	1.05	56.79	clang++ -O2
PCG XSL RR 128/64 (EXT 32)	2^{2174}	2176	64	1.29	46.36	clang++ -Os
PCG XSL RR 128/64 (LCG)	2^{128}	128	64	1.70	35.92	g++ -Os
PCG XSH RS 64/32 (MCG)	2^{62}	64	32	0.61	48.72	g++ -O2
PCG XSH RR 64/32 (MCG)	2^{62}	64	32	0.66	45.31	g++ -O2
PCG XSH RS 64/32 (EXT 1024)	2^{32830}	32832	32	0.78	38.29	g++ -O2
PCG XSH RS 64/32 (EXT 2)	2^{126}	128	32	0.78	38.29	g++ -O2
PCG XSH RS 64/32 (LCG)	2^{64}	64	32	0.79	37.95	g++ -O2
PCG XSH RR 64/32 (LCG)	2^{64}	64	32	0.84	35.67	g++ -O2
XorShift* 64/32	$2^{64}-1$	64	32	1.35	22.05	g++ -Os
Arc4Random	$2^{1699}\sim$	2064	32	10.29	2.90	g++ -O2

Table I: Benchmark performance.

6.2.1. Four Variants. Each implementation is available in four variants: an *extra-fast* variant that uses a MCG rather than an LCG to avoid performing addition (but reduces the period by a factor of four); a *normal* variant where the LCG adds an arbitrary fixed constant; an *automatic-unique-stream* variant that, at zero space overhead, provides a distinct stream compared to its compatriots; and an *explicit-switchable-stream* variant that lets the user select a different stream at any point.

These variations are provided by the underlying LCG generation scheme, as we discussed in Section 4.3.2. As we mentioned there, the trick used by the *automatic–unique-stream* variant involves using the address (or "object id") of the state to create the additive constant and incurs zero additional space cost.

6.2.2. Security Considerations. Thanks to the address-space–layout randomization performed by most modern operating systems, the automatic–unique-stream variant can produce a different sequence of random numbers every time the program is run, *even if* it is seeded with the same value. In many situations, this feature is advantageous, especially in systems where unpredictability is an asset, but in settings where repeatability is desired, we can instead use the normal or explicit-switchable-stream versions instead.

In addition, most of the PCG variations presented in the next section have an output function that returns only half as many bits as there are in the generator state. But the mere use of a $2^{b/2}$-to -1 function does *not* guarantee that an adversary cannot reconstruct generator state from the output. For example, Frieze et al. [1988] showed that if we simply drop the low-order bits, it is possible for an adversary to discover what they are. Our output functions are much more complex than mere bit dropping, however, with each adding at least *some* element of additional challenge. In addition, one of the generators, PCG-XSL-RR (described in Section 6.3.3), is explicitly designed to make any attempt at state reconstruction especially difficult, using xor folding to minimize the amount of information about internal state that leaks out.[18] It should be used when a fast general-purpose generator is needed but enhanced security would also be desirable. It is also the default generator for 64-bit output.

For sensitive applications, the explicit–switchable-stream variant is probably the best choice, because it almost triples the number of unknown bits (e.g., in a 128-bit generator, the output function is a 2^{64}-to-1 uniform function requiring a 64-bit guess to invert, and an additional 127 bits of additive constant must be guessed, too, for a total of 191 unknown bits).[19] If more bits of unknown state are desired, users can turn to the extension that provides k-dimensional equidistribution, described in Section 7.1.

Although these properties make it extremely challenging for an agent observing the external behavior of a program to guess the state of its random number generator, we should also consider the case where the entire program state becomes compromised. For this reason, in programs performing a sensitive task, once a set of random numbers has been produced, it should advance the generator state by an arbitrary amount and change the additive constant, thereby destroying the previous state and making it impossible to recover numbers that were previously generated. Section 7.2 describes a technique that can allow this action to happen automatically.

Finally, in some situations the above security considerations may be counterproductive. Sometimes users actually desire a b-bit generator that iterates through b-bit integers with each one occurring exactly once—sometimes for the no-repeats property

[18] Related work on the difficulty of inferring a generator's internal state from its output is discussed in Section 9.

[19] Note that a secret additive constant does not *by itself* add much security, it can only hope to do so in combination with an appropriately constructed output function [Stern 1987]

and sometimes because space is at a severe premium. In this situation, we must use a 1-to-1 output function, which is, by definition, invertible, and allows full recovery of generator state. To discourage their use by casual users and warm the hearts of security professionals, these variants are explicitly marked *insecure*. In contrast, the normal variants are *not* explicitly labeled as secure—that would be premature; additional scrutiny is required for such a claim. We will return to these issues in Sections 8 and 9 which discuss future and related work, respectively.

6.2.3. Instruction-Level Parallelism. The task of calculating the output function and advancing the state are disjoint; one does not require anything from the other. Thus we can adopt a simple strategy of calculating the output function using the *previous* value of the state while, *at the same time*, we advance the state. In modern CPUs, which typically have many working registers and can execute multiple instructions per clock, it is advantageous to apply this strategy to allow the CPU to get more done at once.

Note, however, that when the state is very large, giving the CPU more to do at once can actually be counterproductive, so for optimal performance on a given CPU both options can be tried. The implementations in the PCG library are structured to allow this change to be performed trivially.

6.3. Specific Implementations

Now let us examine the five family members we are focusing on in more detail. As with PCG-RS in Section 5.4.3, sample code will be given tersely for a specific bit size. The version of the code in the PCG library is longer, uses named constants and can be applied at multiple bit sizes. (I strongly recommend that people use the library rather than copy code from this paper.)

Note also that although the generators are presented with mnemonic names based on the permutations they perform, users of the PCG library should rarely select family members by these mnemonics. The library provides named generators based on their properties, not their underlying implementations (e.g., `pcg32_unique` for a general-purpose 32-bit generator with a unique stream). That way, when future family members that perform even better are discovered and added (hopefully due to the discoveries of others), users can switch seamlessly over to them.

6.3.1. 32-bit Output, 64-bit State: PCG-XSH-RR. Here the design goal is to be a good all-purpose random number generator. The intent is to balance speed with statistical performance and reasonable security, charting a middle-of-the-road path. (It's the generator that I recommend for most users.)

The strategy is to perform an xorshift to improve the high bits, then randomly rotate them so that all bits are full period. Hence the mnemonic PCG-XSH-RR, "xorshift high (bits), random rotation".

With 64 bits of state, the 32-bit output function can be coded tersely as

```
output = rotate32((state ^ (state >> 18)) >> 27, state >> 59);
```

assuming that `rotate32(`v`,`r`)` denotes an unsigned 32-bit bitwise clockwise rotation of r bits on v (i.e., $v \circlearrowright r$). Note that the top five bits specify the rotation, leading to the constants above ($64-5=59$, $32-5=27$, and $\lfloor(5+32)/2\rfloor=18$).

6.3.2. 32-bit Output, 64-bit State: PCG-XSH-RS. Because 32-bit output, 64-bit state is likely to be a common use case, there is an alternative generator that makes a slightly different trade-off—slightly worse statistical performance for slightly greater speed. This version performs a random shift rather than a random rotation. (There is an implied fixed shift in PCG-XSH-RR so this version performs slightly less work.) The

difference between the two is minor, and for most users PCG-XSH-RR is probably the better choice.

With 64 bits of state, the 32-bit output function can be coded tersely as

```
output = (state ^ (state >> 22)) >> (22 + (state >> 61));
```

Note that the top three bits specify the amount of the random shift; in this case a shift of up to seven bits ($64-3=61$, and $32-3-7=22$).

6.3.3. 64-bit Output, 128-bit State: PCG-XSL-RR. This generator is intended for more demanding applications, either those that need 64-bit values or those that need enhanced security. Because it has 128 bits of state, headroom is less of a concern, so the generator is instead optimized for performance and security. Performance is a particular concern at 128 bits because for many architectures the 128-bit value will be represented by two 64-bit processor registers, and some operations (such as shifts) require several steps to perform.

Thus this generator begins with a 64-bit rightward xorshift. According to BigCrush's divinations, this step does essentially nothing to improve quality, but that is not the goal. First, we needed to perform some kind of xorshift to have both a high-period target value and a high-period rotation value for the next step, and 64-bit shift is trivial for the compiler to implement (it doesn't actually do any shifting at all if the high and low halves are in distinct registers, just a direct xor). The second and more important goal is obfuscation. The resulting value is our whole state folded in on itself. There are 2^{64} possible states that could generate this value, making it hard to determine generator state from the output. Finally we make the real improvement step, a random rotation, which ensures that all the bits are full period.

This generator uses 128-bit arithmetic, which is efficiently supported by many of today's languages (including C and C++ with many popular compilers, including GCC and Clang). Platforms that do not make 128-bit arithmetic easy can use a two-dimensional version of PCG-XSH-RR or PCG-XSH-RS instead, using the techniques outline in Section 7.1.

The mnemonic PCG-XSL-RR stands for "xorshift low (bits), random rotation". With 128 bits of state, the 64-bit output function can be coded tersely as

```
output = rotate64(uint64_t(state ^ (state >> 64)), state >> 122);
```

assuming that `rotate64(`v`,`r`)` denotes an unsigned 64-bit bitwise clockwise rotation of r bits on v (i.e., $v \circlearrowright r$). Note that the top six bits specify the rotation, leading to the constants above ($128-6=122$, and $128/2=64$).

6.3.4. 32-bit (or 64-bit) Output & State: PCG-RXS-M-XS (Insecure). This generator is the most statistically powerful of the ones we are considering—it can pass BigCrush with only 36 bits of state, the theoretical minimum. In a cruel twist of fate, we will give it the rottenest job, generating 32 bits of randomness from 32 bits of state. It's cruel because the generator has no choice but to perform its permutation in an invertible way (because the number of output bits and the number of state bits are the same), and so all of its statistical goodness can be undone by anyone who knows its permutation function, unmasking the lowly LCG it has inside (but given the intricateness of the permutation, that is unlikely to happen by accident).

All the output functions we have seen up to now were designed to throw away some of the bits, but this output function must improve them all. It does so by composing three output transformations. First, it performs a random xorshift (described in Section 5.5.1), which improves the middle bits of the output but necessarily leaves the upper (control) bits untouched. Then it performs a multiplication step, which improves the upper bits.

Finally, it improves the lowest bits using an xorshift to xor the bits from the upper third of the number with the bits in the lower third (another random xorshift would work here too, but these bits only need modest improvement and the fixed shift is faster).

This somewhat complex output function allows PCG-RXS-M-XS to pass SmallCrush with 32 bits of both output and state, which is a rarely seen achievement (although Hellekalek1995 [Hellekalek 1995] does also manage it, although at dramatically worse speed, as we previously discussed in Section 2.4). We cannot expect the 32-bit variant to pass Crush or BigCrush—as we have already seen, even a theoretically ideal generator needs 36 bits of internal state to do so.

PCG-RXS-M-XS can also generate 64 bits of output with 64 bits of state. It can likewise provide 128 bits of output with 128 bits of state, but the next generator is custom designed for that task.

Notwithstanding its speed and compactness, all but the most space-constrained applications should prefer a more secure generator, unless the "every number appears only once" property is especially useful.

6.3.5. 128-bit Output & State: PCG-XSL-RR-RR (Insecure). This generator is a simple tweak to PCG-XSL-RR to make it suitable for returning the whole random state (as inadvisable as that is). It just adds a random rotation to the high-order bits. It exists for the same reason that PCG-XSL-RR does; we could use PCG-RXS-M-XS at 128 bits, but if the state is split across two 64-bit registers, multiplication would be slow, and so we turn to other means instead. It's not shown on the graph of statistical performance because it has the same performance as PCG-XSL-RR.

7. EXTENSIONS & TRICKS

We're almost done at this point, but there are a couple of final tricks left to cover. Because PCG permutations are composable, we also gain *extensibility*. We can allow an external agent of some kind to *interpose* its permutation before we apply one of the permutations we've discussed in the previous sections, or apply its own permutation afterwards. This strategy allows us to implement a variety of useful facilities, including k-dimensional equidistribution for large k at very low cost, and enhanced cryptographic features.

7.1. Very Long Periods and *k*-Dimensional Equidistribution

Although we can achieve very long periods and k-dimensional equidistribution for arbitrary k using the counting techniques we discussed in Section 4.3.4, there is a better method available to us via the extension mechanism.

As usual, let's suppose we have a b-bit generator producing r-bit values, which we shall call the *base* generator. The *extended* generator consists of those b-bits of state and k additional r-bit values (where $k \geq 1$), which we shall call the *extension array*. We use the state of the base generator to choose a value from the extension array (by passing the base-generator state through a *selector function*, $s : \mathbb{Z}_{2^b} \to \mathbb{Z}_k$). The final output of the combined generator is produced by xoring the selected value from the extension array with the random number produced by the base generator (i.e., the result from applying its output function, $o : \mathbb{Z}_{2^b} \to \mathbb{Z}_{2^r}$, to its state). Restated using the formalisms introduced in Section 5.4, the function family for the combined generator is

$$f_c(\langle n_1, \ldots, n_k \rangle) = n_{s(c)} \oplus o(c),$$

where $c \in \mathbb{Z}_{2^b}$ is the state of the base generator and the argument to f is the extension array (as a k-tuple). Thus, $f_* : \mathbb{Z}_{2^b} \times (\mathbb{Z}_{2^r} \times \cdots \times \mathbb{Z}_{2^r}) \to \mathbb{Z}_{2^b} \times \mathbb{Z}_{2^r}$ and the output function for the extended generator is $\pi_2 \circ f_*$.

The selector function can be an arbitrary function, but for simplicity let us assume that $k = 2^x$ and that selection merely drops bits to choose x bits from the state. Two obvious choices for choosing x bits are taking the high-order bits and taking the low-order bits. If we wish to perform party tricks like the ones we discussed in Section 4.3.4, it makes sense to use the low-order bits because they will access all the elements of the extension array before any elements repeat (due to the property of LCGs that the low-order l bits have period 2^l). Conversely, if we use the high-order bits, the extension array will be accessed in a more unpredictable pattern.

Thus far we have an output function that uniformly maps the state of the combined generator to an r-bit value, but we must also define how the extension array advances. If the combined generator advances both the base generator and the extension array though all possible states, we will have a uniform generator (whereas if we do not advance it at all, we will not!). We can advance the extension array through all possible states using the same strategy we saw in Section 4.3.4. In particular, we can either choose to advance the array elements each time they are accessed or less frequently. The only *requirement* is that we advance the state of the extension array every time the primary generator crosses zero so that over the full period of the combined generator every possible extension array state is married with every possible base-generator state.

This technique extends generator's period from 2^b to 2^{kr+b} (the factor of k comes from the k r-bit values in the extension array, and b from the base generator). Because the only limit on k is that $k \le 2^b$ and $r \le b$, the maximum possible period is $2^{(2^b+1)b} \approx 2^{b2^b}$; thus the true constraint on period is actually available memory. If the selector function uses the low-order bits of the state to provide the index into the extension array, each element of the extension array will be accessed in turn (in some arbitrary but well-defined sequence), thereby providing k-dimensional equidistribution.

In practice, for base generators with many bits of state (i.e., 128 bits or more), we can neglect to advance the state of the extension array at all, because we are only required to advance it when the main generator completes a full period, and that is going to take a *very* long time.

In Figure 16 and Table I, PCG-XSH-RS 64/32 is shown with two-dimensional and 1024-dimensional equidistribution using this mechanism (the EXT 2 and EXT 1024 variants), whereas PCG-XSL-RR 128/64 (EXT 32) provides a longer period but opts for greater unpredictability rather than 32-dimensional equidistribution. In all cases, the base generator is an MCG, and the extended variant actually runs faster than the simple LCG variant, showing that the overheads of the extension mechanism are very low.

7.2. Enhanced Cryptographic Security

We can also use the above mechanism for added cryptographic security. The extension array of k xor values adds more state to the generator, making it even more inscrutable. Moreover, even though it is technically a violation of uniformity, a cryptographic extension can use the low-order bits to perform an action after every 2^i numbers have been generated, for some i (by performing it when those i bits are all zero). One option, for example, would be to reset the generator using an external source of entropy.

7.3. Somewhat-Secure Seeding

The seed is fundamental to a random number generator—if you know it, you can repeat its output. Thus security-conscious applications must pay particular attention to choosing their seed in a way that cannot be easily guessed. Current best practice is obtain random bits from the seed from a trustworthy external source provided by the system, or to implement a comprehensive seed-management scheme (e.g., as suggested

by Schneier & Ferguson [2003; 2010]). But practitioners writing applications where security isn't a particular concern may not be willing to devote the extra execution time necessary to apply these methods and have a tradition of falling back to fast-but-weak options.

The permutation functions of PCG can tread a middle ground, allowing a couple of quickly accessible local sources of entropy to be hashed and combined to produce a seed that is at least somewhat obscure in minimal time.

8. CONCLUSION & FUTURE WORK

I hope that you will agree that the PCG family of generators significantly raises the bar for what we should expect from a random number generator. It shows that it *is* possible to have one generator that provides all the properties we consider desirable.

I very much hope that others will build on this work. I have outlined some members of the PCG family, and they perform very well, but there is always room for improvement. Possibly there is a PCG generator that performs as quickly as PCG-RS with the statistical performance of PCG-RXS-M-XS, for example; I would like to hope so.

Likewise, I would encourage people working with other generation schemes to use some variation of the assessment methodology I have set out here—in particular the concept of *headroom*. In fact, I believe that this concept merits further exploration. Does "16 bits of headroom" mean the same thing for two different generators? Does the steepness of the different lines in Figures 7 and 15 mean something about the character of the generator? Surely it must. . . ?

Thoughout the paper, I have not hesitated to point out the predictability of most general purpose generators and the security concerns such predictability poses. I have argued that the PCG family is at least a step in the right direction, but I have been reluctant to make any strong claim about its cryptographic security. It isn't *trivial* to crack the state of a PCG generator, particularly PCG-XSL-RR, but a full cryptographic analysis is beyond the scope of this paper, and, in any case, is much better performed by people with a vested interest in finding flaws in the permutation functions. I hope there are no such flaws, but at least if there are, the issue won't be architectural; stronger, more secure output functions could then be developed.

Permutation functions on tuples are a general technique that can probably also be fruitfully applied to other areas, in particular hashing, but in that domain there are also powerful engineering tradeoffs to consider. The issue is not performing the best hash, it is performing the best you can in a given time. Determining the best trade-off remains an open question, but at least another tool is available for the task.

Some of the most interesting things in computer science rely on random number generation. I am delighted that the PCG family can be there to help. It is also fitting that after helping support randomized algorithms for years, we have an excellent strategy for randomness and it is itself *a randomized algorithm*, because that's the kind of bootstrapped self-referentiality that makes computer science so wonderful.

9. RELATED WORK

For a broader perspective on random number generation, there are several good summaries. Pierre L'Ecuyer gives an excellent recent summary of the field in a chapter in the *Handbook of Computational Statistics* [2012]. Knuth's *Art of Computer Programming* [1997] also covers the area in depth. Press et al.'s *Numerical Recipes* [2007] also provides sound practical advice. Schneier & Ferguson [2003; 2010] provide an excellent grounding on cryptographic security.

Press et al. [2007] advocate strongly for combined generators, arguing that there is strength in combining multiple generation techniques with different properties. The PCG family uses a single base generator, but is in some sense a combined approach

because the underlying mechanisms of its output functions are quite different from the underlying mechanisms its LCG base generator.

This paper advocates for generators whose output range is 2^r, for some positive integer r, because these generators are useful for generating streams of bits (see Section 2.4.1). In contrast, some generators, such as MRG32k3a [L'Ecuyer 1999a], are not designed with a power-of-two output range in mind, instead using a generator with a prime modulus to produce floating-point values in the range 0–1. Unfortunately, producing random floating-point values uniformly is tricky (assuming every floating-point value should have some chance of appearing), because the range of values from 0–1 that can be represented in floating-point is distinctly nonlinear. In 2014, Taylor Campbell explained the issue in detail and presented an algorithm for uniformly generating floating-point values.[20] The essence of the algorithm is to repeatedly subdivide the 0–1 interval using a random choice. Interestingly, this algorithm requires a random bitstream, consuming a variable number of random bits, up to 1024 bits in the worst case.

The techniques in this paper obviously have strong connections to hashing. In fact, if we apply a 32-bit rendition of one of the best general-purpose integer hash functions, Fast Hash[21], twice to the output of a linear congruential generator, we can also pass BigCrush at 36 bits (and fail the exact same test at 35 bits that PCG-RXS-M-XS did, namely MaxOft). But if the Fast Hash is applied just once, the generator doesn't pass until 42 bits as shown in Figure 15. More importantly, the speed of either generator is mediocre compared to the ones presented in Section 6.3.

The idea of using a random number generator's own randomness to improve its output is a very old one, being suggested by Bays & Durham [1976]. Typically, however, the idea has been to shuffle the order of the output or discard some of it. Discarding output is problematic because, unless undertaken with extreme care, it breaks uniformity. Shuffling tends to be a local operation that may not adequately mask problems with the generator and can also prove memory intensive. KnuthB [Knuth 1981] adopts the shuffling strategy, and, as we saw in Section 2.1.2, it does not fare well in empirical tests. PCG generators have an advantage because they permute the output at a much lower level, and because they keep some of their state to themselves.

As mentioned in Section 6.2.2, Stern [1987] observed that Knuth's [1981] advice to drop the low-order bits and keep LCG parameters secret is not sufficient for cryptographic security. Frieze et al. [1988] used the *LLL algorithm* [Lenstra et al. 1982] to provide a practical reconstruction algorithm. More recently, however, Contini [2005] showed that secret truncated linear congruential generators may not necessarily be insecure for properly chosen parameters, giving hope that the obfuscation provided by PCG's permutation functions may also be sufficient to make the problem hard. The use of xor folding in PCG-XSL-RR is somewhat reminiscent of the *shrinking generator* proposed by Coppersmith et al. [1994] (which combined two generators with xor), but it is probably more akin to Meier & Staffelbach's *self-shrinking generator* variant [1994], which has suffered cryptographic attack (e.g., Zenner et al. [2001; 2006]), although these attacks appear to be computationally expensive.

A good general description of possible cryptographic attacks on a random number generator is given by Kelsey et al. [1998], and PCG has at least been designed with those attacks in mind, but currently lacks a full cryptographic analysis. In contrast, there are several generators that have had a such an analysis—Schneier & Ferguson [2003; 2010] present Fortuna (a successor to Yarrow [Kelsey et al. 2000]), a modern cryptographically secure generator that takes special care to ensure it is seeded in a safe way. Fortuna's goal

[20] Available at http://mumble.net/~campbell/tmp/random_real.c.

[21] Available at https://code.google.com/p/fast-hash/.

is excellent cryptographic security for use as a stream cipher, rather than a fast rate of random-number generation. In general, generators designed for stream ciphers may not have the same concerns about statistical properties (such as uniformity) compared to general-purpose random number generators. For example, Arc4Random is not uniform [Mister and Tavares 1999; Basu et al. 2008].

Turning to speed, Vigna [2014a; 2014b] has also suggested that simple low-overhead generators can produce excellent results. His generators appear to be an improvement in the world of XorShift* generators, but the improvement comes at a space cost, and they do not perform as well as the PCG generators described here.

Other recent authors have also focused on attempting to marry good statistical performance with fast execution and not-unreasonable space usage. Neves & Araujo [2012] suggest a nonlinear generator that, although it does not perform as well as PCG on desktop hardware, appears to perform well on GPUs, whereas thus far PCG generators have not been specialized for that domain.

Several recent authors [Vigna 2014a; 2014b; Neves and Araujo 2012] have produced 128-bit (or more) generators that pass the BigCrush battery. Presumably their generators really are superior to the basic 96-bit LCG we saw passing BigCrush and running quickly in Sections 2.1.2, 2.3, and 3.3, but it would strengthen their results to see the comparison made. Hopefully the framework presented in Section 3 could be useful in that task.

10. TEST METHODOLOGY

Timing tests were run on a Intel "Haswell" Core i7-4770 CPU, running at 3.40GHz with Turbo-boost disabled, which happens to be the same architecture used by Vigna [2014a; 2014b]. (I achieved very similar results, including sub-nanosecond generation, with an older computer, a Mid 2012 Retina MacBook Pro, with an Intel Core i7-3720QM CPU, running at 2.60GHz.) SmallCrush timings use the values reported by the TestU01 suite. Microbenchmark timings were performed using a custom framework that measured both cycle times using the timestamp counter, and real time using nanosecond-resolution timing. The test suite measures and subtracts loop overhead, and care is taken with compiler optimization to make sure that all tests are fair. In particular, the code is optimized for both size and speed, but heavy optimization settings are avoided to prevent loop unrolling from obscuring the picture. If anything, this condition is unfair to the PCG generation scheme because its tiny code size makes unrolling very viable and can enhance its performance beyond that reported in the paper.

Because the code for PCG generators is extremely small and fast, executing in very small numbers of cycles, issues such as register selection and instruction scheduling become significant. Thus for overall fairness, code is compiled (as 64-bit) with multiple compilers (GCC 4.8.2 and Clang 3.4) and the fastest code used. It turns out that this approach was very favorable to the Mersenne Twister—the GCC implementation is considerably faster than the one used provided by the Boost library or Clang. The OpenBSD code for Arc4Random was converted to a C++ class to allow it to be fairly compared with the other generators, which were all written in C++. The full timing code is provided with the PCG library.

In the interests of space, the data presented in Section 6.1 focuses exclusively on 64-bit performance on desktop hardware, but the PCG family also performs well on 32-bit hardware, including lower-end systems.[22] For example, in tests run on a dual-core Cortex-A7 ARM CPU at 1 GHz (in an Allwinner A20 (sun7i) SoC, as might be found in an embedded system or low-end smartphone), PCG-XSL-RR 64/32 (LCG) outperforms the 32-bit Mersenne Twister by a factor of 2.07. Interestingly, on the ARM architecture there

[22] More performance data, including 32-bit timings can be found at http://www.pcg-random.org/.

is no difference in performance between MCG and LCG variants because ARM has a combined integer multiply-add instruction.

Statistical tests employed two machines, one with 48 cores and one with twelve. Neither is particularly powerful or new, but their ability to apply parallelism to the task of running TestU01 allowed them to devote over a year of CPU time running batteries of tests on various generators at various sizes in a little over a week.

For TestU01, generators are tested both normally and with their bits reversed. Also, because TestU01 only expects 32-bit resolution (and actually only tests at 31 bits at a time), 64-bit generators have both their high and low 32 bits tested. Tests that give a p-value worse than 1 in 10^9 are considered clear fails, whereas tests where the p-value is worse than 1 in 1000 are rerun five times. If a test shows a worse than 1 in 1000 result in two or more of its retests, it is also considered a fail. In theory, given the number of generators tested, there was some chance that an unlikely event would have muddied the waters, but the only such event was the reversed variant of an every-permutation-but-the-kitchen-sink generator, PCG-XSH-RR-M-XS, passing BigCrush at 35 bits. So it *can* be done. If you're lucky.

Downloads

Source for the PCG family of generators, in C and C++ is available at http://pcg-random.org.

Acknowledgments

This work owes a huge debt to Pierre L'Ecuyer: when I needed LCG constants, it was his table of constants I used; when I needed a summary of a background topic, I found his summaries were enlightening; and, most importantly, when I needed empirical tests, it was his TestU01 suite that provided them. TestU01 shook up the world of random number generation, telling us that the ones we were all using were not good enough.[23]

But it was Stephan T. Lavavej's talk, " rand() Considered Harmful" (given at Microsoft's 2013 *Going Native* conference) that actually inspired the work in the first place.[24] He gave advice that mirrored advice that I myself had given in the past, to avoid linear congruential generators and use "a good generator like the Mersenne Twister". Somehow hearing those words spoken by someone else triggered my critical sense and sent me down this path (although a stack of midterms I didn't want to grade no doubt helped, too).

Many students and colleagues at Harvey Mudd College have been extremely helpful in refining the idea and its presentation. Claire Connelly caught numerous issues in two detailed copy-edit readings. Christopher Stone listened to me explain the concept of function families and provided their name, and his review comments helped enhance the paper's clarity and rigor. Jim Boerkoel's feedback helped Section 3 become a section in its own right. Nick Pippenger read a very early draft of the paper and provided both warm encouragement and valuable feedback. Maria Klawe listened to an oral description of the technique and encouraged me to formalize my intuitions (some of those formalizations do not appear here; I hope that they will appear a future paper). Art Benjamin assured me that my statistical model of the generalized birthday problem was "obvious" given that it was built using elementary methods. And finally, early on when I was reluctant to believe I could have really have stumbled onto anything that hadn't been discovered before, it was Zach Dodds who assured me that sometimes it is the simple ideas that get overlooked and encouraged me to write this paper.

[23] "Or maybe it was the zeros that weren't up to spec!" — Christopher Stone.

[24] Available at http://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful.

REFERENCES

Douglas Adams. 1979. *The Hitchhiker's Guide to the Galaxy*. Arthur Barker, Edinburgh.

Riddhipratim Basu, Shirshendu Ganguly, Subhamoy Maitra, and Goutam Paul. 2008. A Complete Characterization of the Evolution of RC4 Pseudo Random Generation Algorithm. *Journal of Mathematical Cryptology* 2, 3 (2008), 257–289.

Carter Bays and S. D. Durham. 1976. Improving a Poor Random Number Generator. *ACM Trans. Math. Software* 2, 1 (March 1976), 59–64.

Forrest B Brown. 1994. Random number generation with arbitrary strides. *Transactions of the American Nuclear Society* 71 (Nov. 1994), 202–203.

Glyn D. Carter. 1989. *Aspects of Local Linear Complexity*. Ph.D. Dissertation. Royal Holloway College, University of London.

Scott Contini and Igor E. Shparlinski. 2005. On Stern's Attack Against Secret Truncated Linear Congruential Generators. In *Proceedings of the 10th Australasian Conference on Information Security and Privacy (ACISP'05)*. Springer-Verlag, Berlin, Heidelberg, 52–60.

Don Coppersmith, Hugo Krawczyk, and Yishay Mansour. 1994. The Shrinking Generator. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '93)*. Springer-Verlag, London, UK, UK, 22–39.

Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 3–3.

Eva D. Erdmann. 1992. *Empirical tests of binary keystreams*. Ph.D. Dissertation. Royal Holloway College, University of London.

Niels Ferguson and Bruce Schneier. 2003. *Practical Cryptography* (1 ed.). John Wiley & Sons, Inc., New York, NY, USA, Chapter Generating Randomness, 155–184.

Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. 2010. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Chapter Generating Randomness, 137–161.

Alan M. Frieze, Johan Hastad, Ravi Kannan, Jeffrey C. Lagarias, and Shamir Adi. 1988. Reconstructing Truncated Integer Variables Satisfying Linear Congruences. *SIAM J. Comput.* 17, 2 (1988), 262–280.

Ian Goldberg and David Wagner. 1996. Randomness and the Netscape browser. *Dr Dobb's Journal* 21, 1 (1996), 66–71.

Torbjörn Granlund. 2012. *Instruction latencies and throughput for AMD and Intel x86 Processors*. Technical Report. gmplib.org.

Zvi Gutterman and Dahlia Malkhi. 2005. Hold Your Sessions: An Attack on Java Session-id Generation. In *Proceedings of the 2005 International Conference on Topics in Cryptology (CT-RSA'05)*. Springer-Verlag, Berlin, Heidelberg, 44–57.

Peter Hellekalek. 1995. Inversive Pseudorandom Number Generators: Concepts, Results and Links. In *Proceedings of the 27th Conference on Winter Simulation (WSC '95)*. IEEE Computer Society, Washington, DC, USA, 255–262.

Thomas E. Hull and Alan R. Dobell. 1962. Random Number Generators. *SIAM Rev.* 4, 3 (1962), 230–254.

John Kelsey, Bruce Schneier, and Niels Ferguson. 2000. Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In *Proceedings of the 6th Annual International Workshop on Selected Areas in Cryptography (SAC '99)*. Springer-Verlag, London, UK, UK, 13–33.

John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Cryptanalytic Attacks on Pseudorandom Number Generators. In *Proceedings of the 5th International Workshop on Fast Software Encryption (FSE '98)*. Springer-Verlag, London, UK, UK, 168–188.

Donald E. Knuth. 1981. *The Art of Computer Programming, Volume 2 (2nd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Pierre L'Ecuyer. 1988. Efficient and Portable Combined Random Number Generators. *Commun. ACM* 31, 6 (June 1988), 742–751.

Pierre L'Ecuyer. 1996. Maximally Equidistributed Combined Tausworthe Generators. *Math. Comp.* 65, 213 (Jan. 1996), 203–213.

Pierre L'Ecuyer. 1999a. Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Operations Research* 47, 1 (1999), 159–164.

Pierre L'Ecuyer. 1999b. Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure. *Math. Comp.* 68, 225 (Jan. 1999), 249–260.

Pierre L'Ecuyer. 1999c. Tables of Maximally Equidistributed Combined LFSR Generators. *Math. Comp.* 68, 225 (Jan. 1999), 261–269.

Pierre L'Ecuyer. 2012. Random Number Generation. In *Handbook of Computational Statistics*, James E. Gentle, Wolfgang Karl Härdle, and Yuichi Mori (Eds.). Springer Berlin Heidelberg, 35–71.

Pierre L'Ecuyer and Richard Simard. 2001. On the Performance of Birthday Spacings Tests with Certain Families of Random Number Generators. *Mathematics and Computers in Simulation* 55, 1-3 (Feb. 2001), 131–137.

Pierre L'Ecuyer and Richard Simard. 2007. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Transactions on Mathemathical Software* 33, 4, Article 22 (Aug. 2007), 40 pages.

Derrick H Lehmer. 1951. Mathematical Methods in Large-Scale Computing Units. In *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery*. Harvard University Press, 141–146.

Arjen K. Lenstra, Hendrik W. Lenstra Jr., and László Lovász. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261, 4 (1982), 515–534.

George Marsaglia. 1968. Random Numbers Fall Mainly in the Planes. *Proceedings of the National Academy of Sciences* 61, 1 (1968), 25–28.

George Marsaglia. 1995. *The Marsaglia Random Number CD-ROM, including The DIEHARD Battery Of Tests Of Randomness,*. Technical Report. Florida State University.

George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14 (4 7 2003), 1–6.

Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, 1 (Jan. 1998), 3–30.

Earl H. McKinney. 1966. Generalized birthday problem. *Amer. Math. Monthly* 73, 4 (April 1966), 385–387.

Willi Meier and Othmar Staffelbach. 1994. The self-shrinking generator. In *Communications and Cryptography*. Springer, 287–295.

Nicholas Metropolis and Stanislaw Ulam. 1949. The Monte Carlo Method. *J. Amer. Statist. Assoc.* 44, 247 (1949), 335–341. PMID: 18139350.

Serge Mister and Stafford E. Tavares. 1999. Cryptanalysis of RC4-like Ciphers. In *Proceedings of the Selected Areas in Cryptography (SAC '98)*. Springer-Verlag, London, 131–143.

Samuel Neves and Filipe Araujo. 2012. Fast and Small Nonlinear Pseudorandom Number Generators for Computer Simulation. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM'11)*. Springer-Verlag, Berlin, Heidelberg, 92–101.

François Panneton, Pierre L'Ecuyer, and Makoto Matsumoto. 2006. Improved Long-period Generators Based on Linear Recurrences Modulo 2. *ACM Trans. Math. Software* 32, 1 (March 2006), 1–16.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing* (3 ed.). Cambridge University Press, New York, NY, USA.

Jacques Stern. 1987. Secret Linear Congruential Generators Are Not Cryptographically Secure. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science (SFCS '87)*. IEEE Computer Society, Washington, DC, USA, 421–426.

Robert C. Tausworthe. 1965. Random numbers generated by linear recurrence modulo two. *Math. Comp.* 19, 213 (April 1965), 201–209.

Sebastiano Vigna. 2014a. *An experimental exploration of Marsaglia's xorshift generators, scrambled*. Computer science research (corr / cs.ds). arXiv.org.

Sebastiano Vigna. 2014b. *Further scramblings of Marsaglia's xorshift generators*. Computer science research (corr / cs.ds). arXiv.org.

Erik Zenner, Matthias Krause, and Stefan Lucks. 2001. Improved Cryptanalysis of the Self-Shrinking Generator. In *Information Security and Privacy*, Vijay Varadharajan and Yi Mu (Eds.). Lecture Notes in Computer Science, Vol. 2119. Springer Berlin Heidelberg, 21–35.

Bin Zhang and Dengguo Feng. 2006. New Guess-and-determine Attack on the Self-shrinking Generator. In *Proceedings of the 12th International Conference on Theory and Application of Cryptology and Information Security (ASIACRYPT'06)*. Springer-Verlag, Berlin, Heidelberg, 54–68.