Random Number Generators: Implementation and Statistical Analysis Using Dieharder

Kashev Dalmia
Email: dalmia3@illinois.edu

David Huang
Email: huang157@illinois.edu

***Abstract*—We implement in efficient C++ several kinds of different modern pseudo-random number generators, or PRNGs; a Linear Congruential Generator, a Lagged Fibonacci Generator, an Xorshift type generator, a Complementary Multiply with Carry Generator, and a Mersenne Twister Generator. We discuss the challenges in testing these kinds of PRNGs, and discuss one of the more common frameworks for testing these random number generators. Finally, we compare our results to what we expect from these implementations, as well as comment on the suitability of these PRNGs for use in simulation of systems.**

***Index Terms*—Random Number Generators, Psuedo Random Number Generators, PRNGs, Statistical Analysis, Dieharder**

I. Introduction

The applications of randomness are far reaching. From statistics, to simulation, to gaming, to cryptography, there is a large need for random number generators. Oftentimes, these generators must be fast to keep up with the demand produced by high speed applications, such as those used on modern servers to ensure proper security. It is also desirable that they be deterministic in some cases in order to produce reproducable and debuggable simulations. Thus, the creation of pseudo random number generators, or PRNGs, has become an important research area in modeling, computer simulation, and mathematics communities.

In the field of analysis of computing systems, simulation is an extremely important part of the modeling process. Today's stochastic models are so complex that finding analytic solutions is effectively impossible. As such, simulations driven by randomness allow a far more viable way to study the behavior of systems. This is the sort of need that motivates this project: a study of the theory behind random number generators, how to test them, and their suitability for this purpose.

A. *Kinds of Random Number Generators*

For the purposes of this paper, there are two kinds of Random Number Generators, or RNGs: Hardware RNGs, and Software RNGs. Hardware RNGs are used not only to output a random bit stream faster than a multipurpose CPU might be able to [1], [2], but also in order to gather entropy in a more unpredictable way, such as measuring thermal noise, detecting air moisture, or using unpredictable user interactions, such as keystrokes and mouse motion , to name a few. Often hardware RNGs are used to seed software RNGs. Readers interested in the seeding of RNGs with external entropy are encouraged to read [3].

Software RNGs, unless they draw upon these sources of entropy from nature for seeds, are in fact PRNGs, because the software that generates these sequences must be deterministic. In this paper, the term RNG will refer to software PRNGs, unless otherwise indicated. These software RNGs use mathematical models to take a finite amount of state and turn this into a sequence of seemingly random output, suitable for use in a variety of purposes. Though special purpose hardware RNG circuits may be fast, software RNGs are usually faster than repeatedly polling a sensor to gain outside entropy. They are also far more convenient to use, as all is required is a computer, rather than special purpose hardware.

Finally, there are several classes of software RNG based on quality of output. The highest class is those which are suited for cryptography. Generally, this includes passing the next bit test: that if k bits of a random sequence are known, then no polynomial time complexity algorithm should be able to guess the next bit [4]. On a similar vein, the previous bits of a random sequence must also be impossible to guess in polynomial time as this would expose the initial state of the generator, and thus the seed. These cryptographically secure random number generators tend to be slow and are not designed to be run more than a few times in quick succession, which makes them unsuitable for use in applications in which speed is necessary, like simulation.

This paper deals with RNGs which are not of cryptographic quality, but are nonetheless useful. These RNGs combine speed with memory complexity and quality of output.

B. This Project

In this project, we study and implement a few common random number generators in C++, from those which have been used historically, to those which are most commonly in use today. We test our generators using the Dieharder test suite [5]. Using Dieharder, we can push these RNGs to the point of failure, and therein see their flaws and sutability for various applications. We can also compare our results with the results of the RNGs built into the GNU Scientific Library, or GSL, and thus Dieharder. All of the software written for this project is available on Github [6].

In this paper, we first discuss the methods and challenges of testing random numbers in Section II. Then, we discussed the different classes of RNGs we study in Section III. Finally, we describe our software approach in Section IV and share our results in Section V.

II. Testing Random Number Generators

Testing random number generators can be tricky. An ideal random number generator can be considered to be independently, identically distributed samples of a uniform distribution $U(0, 1)$. However, the RNGs that we consider here are not truly random. So, the crux of the problem is how to determine what set of numbers appear random enough, such that they are suitable for use.

A. The Empirical Approach

Imagine flipping a coin 10 times and recording the number of heads and tails. For a fair coin, one would expect to get roughly the same number of heads and tails. If one got 10 heads, and only 1 tails, one might be inclined to suspect that this coin might not be a fair and truly random coin. However, this event has a distinct nonzero probability of this happening, described by the pdf of the binomial distribution:

$$\binom{10}{9}\frac{1}{2}^9\frac{1}{2} = 0.00977$$

In other words, even for a fair coin, in 1000 trials of 10 coin flips, one would expect to get 9 heads and 1 tails 9 or 10 times. Similarly, imagine if one had a different coin, and ran this same test, but always got exactly 5 heads and 5 tails, for all 1000 trials. Though on average we expect to get about the same number of heads and tails, this coin is also suspect, because for all these repeated trials, we should expect them to follow the binomial distribution discussed above.

If we increased the number of flips per trial to a very large n, and increased the number of trials to a very large t, the expected distribution should appear increasingly similar to the normal distribution, with mean heads percentage of $\mu = 0.5$. This is verified by the Central Limit Theorem [7].

So taking again our mystery coin, if we apply the test with reasonably sized n and t, and get a distribution of our sample means, we expect it to resemble a normal distribution but will likely not be perfect. There is a certain probability, if we assume that the underlying RNG is perfect, that the result we got would occur: a number on the range $(0, 1)$. We shall call this our p-value. If this p-value is very, very small (on the order of 10^{-6}), we might say with a certain degree of confidence that our assumption that the RNG is good is incorrect. However, we could be more confident if we repeated this process several times, and looked at the distribution of p-values.

We can find the probability of getting our p-value distribution using a Kolmogorov-Smirnov test [], which quantifies the difference between the expected distribution and the p-valuedistribution that we observe. This test yields a second p-value: the probability that the resultant distribution would occur given that the underlying RNG is indeed indistinguishable from a perfect one.

If this second p-value is low, then we can safely decide that the RNG being tested is not good. In other words, statistically speaking, the underlying RNG is differs from a perfect RNG by a significant amount. If this value is high, then, rather than accept that the RNG is 'good', we instead conclude that we cannot reject our hypothesis that it is 'good'.

In fact, this process is exactly what the Dieharder test suite does, and interested readers are encouraged to see [8] for further information. Our implementations of RNGs are used to pipe random numbers to Dieharder, which does this sort of analysis for several different kinds of tests, which are discussed next in Section II-B

B. Dieharder

Dieharder's man page states that "[...]dieharder can eventually contain all rng tests that prove useful in one place[...]" [5]. It aims to be a one-stop-shop for testing RNGs. It is the spiritual successor and improvement to George Marsagalia's Diehard testing tool, which in turn is a testing tool which took many tests from the famous Donald Knuth [9]. Though it is not the only set of tests or testing suite of this kind [10], [9], [11], [12], it holds the distinction of being readily available in the Ubuntu package repository, and having good documentation [8].

Dieharder has many tests, listed in Table I. It is worth discussing a few of the tests available in Dieharder to give the reader an idea of what sort of things are tested, and furthermore, what failing one of these tests might indicated about the quality of an RNG. Further information about tests not discussed is available in the Dieharder man pages and help flags.

1) The STS Monobit Test: Taken from the NIST Statistical Test Suite [11], this test is exactly the test described above in Section II-A. The number of ones are counted in a bit stream generated by a random number generator, and the same analysis is done on it. It is not an extremely rigorous test; some bad RNGs will still pass it but this test is good at weeding out extremely poor RNGs.

2) The Diehard Birthdays Test: Taken from Diehard, the Birthdays Test is so named for the Birthday Paradox: That in some set of n randomly chosen people in a room, the probability that two of them share the same birthday becomes high astonishingly quickly, reaching 99.9% at $n = 70$ [13], assuming the distribution of birthdays is uniform. If these birthdays are placed in a year then the distance between pairs birthdays should follow a Poisson distribution. The test in Dieharder places 512 birthdays in a 24 bit 'year' and determines if the the resultant distribution is in fact Poisson using the p-value analysis described above.

TABLE I

AVAILABLE TESTS IN DIEHARDER VERSION 3.31.1.

#	Test Name	Reliability
0	Diehard Birthdays	Good
1	Diehard OPERM5	Good
2	Diehard 32x32 Binary Rank	Good
3	Diehard 6x8 Binary Rank	Good
4	Diehard Bitstream	Good
5	Diehard OPSO	Suspect
6	Diehard OQSO	Suspect
7	Diehard DNA	Suspect
8	Diehard Count the 1s (stream)	Good
9	Diehard Count the 1s (byte)	Good
10	Diehard Parking Lot	Good
11	Diehard Minimum Distance (2d Circle)	Good
12	Diehard 3d Sphere (Minimum Distance)	Good
13	Diehard Squeeze	Good
14	Diehard Sums	Don't Use
15	Diehard Runs	Good
16	Diehard Craps	Good
17	Marsaglia and Tsang GCD	Good
100	STS Monobit	Good
101	STS Runs	Good
102	STS Serial (Generalize)	Good
200	RGB Bit Distribution	Good
201	RGB Generalized Minimum Distance	Good
202	RGB Permutations	Good
203	RGB Lagged Sum	Good
204	RGB Kolmogorov-Smirnov	Good
205	Byte Distribution	Good
206	DAB DCT	Good
207	DAB Fill Tree	Good
208	DAB Fill Tree 2	Good
209	DAB Monobit 2	Good

3) RGB Lagged Sums: Written by the main Dieharder author Robert G. Brown, this test is the only test in the Dieharder suite which tests for lagged correlations. Imagine that, overall, an RNG's output seemed very random, but every seven bits, the same pattern emerged throughout the generation. This test checks for that situation, with several different lag numbers, when running all tests. The lagged bits are taken, used to create single precision numbers between 0 and 1, then t of these numbers are summed. A single trial should have a mean $\mu = 0.5t$. Then, the p-values are determined from several trials of t sample sums. Interested readers are encouraged to see [8] for a more discussion.

C. Interpreting Dieharder Results

The authors of Dieharder are careful to avoid saying that Dieharder can verify that a random number generator appears truly random. Rather, Dieharder serves as a set of tests which can weed out poor RNGs, and say with certainty that they do not appear truly random under scrutiny. This is not to say that they are not useful; for instance, if one is simply creating a website which provides a virtual die for a user to roll, then the RNG powering the die need not necessarily pass every Dieharder test (perhaps only the Diehard Craps Test, which specifically tests using an RNG to power dice for a game), but rather pass a majority of them, and still fulfill the requirements. However, if one is running Monte Carlo simulations, an RNG of a higher quality may be desired, which passes more tests.

A system designer must take into account the trade offs between quality of RNG output, memory efficiency, speed, and complexity of implementation when choosing an RNG for a system. These days, with computers as fast as they are, and many programming languages having already chosen a de facto RNG, this choice is hidden from most users, but is still relevant to mathematicians, those who work with large simulations, or others concerned with extremely high quality of randomness and speed.

What does failing a Dieharder test look like? We look back to the p-value analysis; This final p-value represents the probability that, given that the underlying PRNG resembles a true RNG, the series of our tests would yeild the results that they did. The final p-value is a probability on the range $(0, 1)$. If this probability is extremely low or extremely high, then the result is considered to be a test failure.

In Dieharder specifically, if the p-value p is $p > \%99.95$ or $p < \%0.05$, then the test is regarded as failed, which is an extremely high threshold. If $p > \%99.5$ or $p < \%0.5$, then the test is marked as weak performance on the part of the RNG. In either the case of the failure or weakness result, it may be worth re-running the test using a different seed to see if this result is consistent. After all, there is some degree of randomness involved, and finding an area of zero ambiguity is nigh impossible.

D. The Theoretical Approach

Apart from the empirical approach described above and used in Dieharder, there is also theoretical analysis that can be done with a priori knowledge of the RNG's implementation. For instance, the period of an RNG can be time consuming and memory inefficient to measure empirically, but with knowledge of the generating algorithm, these can be deduced. Additionally some RNG algorithms (specifically the Linear Congruential Generator discussed in Section III-A) have a property where their output falls in planes which can be determined easily, if one knows where to look.

This sort of analysis exposes flaws of specific generator algorithms, but doesn't always provide an even footing on which to analyze the RNGs. The periods of the generators are discussed in each RNG class's section, along with other strengths and weaknesses, but the focus of our project is on the performance of these RNGs in Dieharder.

III. Classes of Random Number Generators

Here we describe a few different kinds of random number generators, the math that makes them work, give examples of each kind, and give their strengths and weaknesses.

A. Linear Congruential Generators

The Linear Congruential class of RNGs, or LCG for short, is a very old and relatively simple random number generator. They are based on the algebraic formula of a line, modulo another number. The following recurrence relation defines the LCG.

TABLE II
Table of common LCG parameters.

Name	m	a	c
MINSTD	$2^{32}-1$	48271	0
glibc	2^{31}	1103515245	12345
RANDU	2^{31}	65539	0

$$X_{n+1} = (aX_n + c) \mod m \qquad (1)$$

Where the parameters are chosen with the following restraints:

$$\begin{aligned} m, &\qquad 0 < m \\ a, &\qquad 0 < a < m \\ c, &\qquad 0 \le c < m \\ X_0, &\qquad 0 \le X_0 < m \end{aligned}$$

If $c = 0$, this particular class of of RNG is called a Lehmer, or Park-Miller RNG [14], [15].

The period of such generators depends heavily on the chosen parameters. It can be at most m, and as a result, m is often chosen to be the largest number which can fit in the number of bits of the output; usually 2^{32} or 2^{64}. Some common parameter sets in use include MINSTD, and glibc. A set of parameters notoriously deemed unfit for use is called RANDU, and is discussed below. Values for these parameters can be found in Table II.

1) Strengths: LCG generators are tiny. They store a single word of state, are easy to seed, and with proper parameter selection, can be quite effective RNGs. However, some of the flaws greatly outweigh the lightweight nature and speed benefits of LCGs for all but the most basic of platforms, discussed next.

2) Weaknesses: LCGs have a fatal flaw that is amplified by bad parameter selection: their outputs fall in easily identifiable planes, as proved by Marsaglia [16]. This is most easily demonstrated by viewing the output

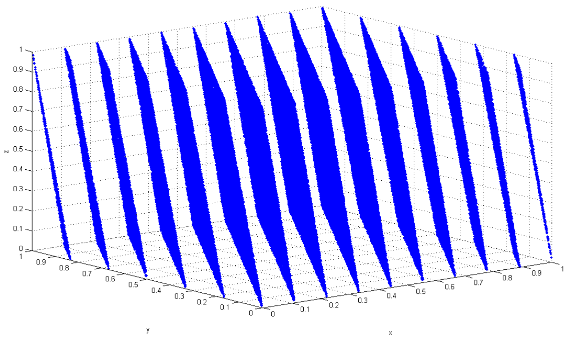

Fig. 1. The problem with LCGs demonstrated by RANDU; a MATLAB plot of consecutive outputs from RANDU used as (x, y, z) points and plotted [18].

of a notoriously bad set of parameters: RANDU. In Figure 1, the output of RANDU is treated as (x, y, z) pairs as it is generated, then these points are plotted. It is obvious that all of the output falls in the same set of planes, and is not even an approximation of a truly random number generator. Better parameter choices alleviate this phenomenon by adding the number of planes that would come up in a test like this, and making them closer together, but the basic problem is unavoidable.

Another issue is the period length; at most the period length is m, which cannot be easily increased without increasing the number of bits in the word being operated on, which is not practical on modern architectures. In fact, experts recommend not even considering generators with periods $< 2^{50}$ for anything but the most trivial uses [17].

Due to these problems, most LCGs pass some tests in Dieharder, but do not stand up to more intense scrutiny, like the lagged sums tests. RANDU's performance is patently awful, and stands up to its infamy. MINSTD fares better, but still does not pass as many tests as more sophisticated generators.

Despite these flaws, these extremely small and fast RNGs may still be suited to small embedded architectures and applications where speed is significantly more important than quality.

B. Xorshift Generators

A far more attractive choice than LCGs for those trying to implement RNGs with a lot of memory constraints, speed requirements, or a combination of the two on an embedded platform, is the Xorshift generator. Discovered and described by George Marsaglia [19], they are significantly more adept at creating sufficiently long periods for more applications.

Xorshift RNGs work by keeping a small amount of state (in many example implementations, four numbers), then repeatedly XORing these numbers with shifted versions of themselves. By doing this, the period of the generation can be $2^{32*4} = 2^{128}$ with very little code. The following is a short example for how easily Xorshift generators can be implemented. Let `t`, `x`, `y`, and `w` be seed values. Then, let `a`, `b`, and `c` be the parameters of the RNG. Then, the C style code is as simple as the following:

```c
// Random seed numbers
uint32_t t, x, y, w;
// Fixed constants for shifting
const uint32_t a, b, c;
uint32_t get_rand(void){
    t = x ^ (x << a);
    x = y;
    y = z;
    z = w;
    w = w ^ (x >> b) ^ t ^ t( >> c);
    return w;
}
```

Picking parameters for the directions and amounts to shift each state variable by is also relatively easy, and is discussed in [19], along with every set of appropriate parameters a, b, and c for 32 bit and 64 bit words.

1) Strengths: Xorshift RNGs are very fast on modern architectures, and use very little state. They have longer periods than LCGs, and are easier to choose parameters for. These RNGs also perform far better on the Dieharder tests than LCGs, and pass the majority of them rather convincingly.

2) Weaknesses: However, the periods of Xorshift generators are not as large as some other generators. They are suitable for most uses, but in large simulations and Monte Carlo analysis, where massive amounts of random numbers are needed, the relatively small period could affect results.

C. Lagged Fibonacci Generators

Lagged Fibonacci Generators, or LFGs, owe their name to the general form of the recurrence relation that defines them, which is reminiscent of the Fibonacci recurrence relation:

$$X_n \equiv (X_{n-r} \star X_{n-s}) \mod m, \quad 0 < s < r \tag{2}$$

Here, $\star$ can be any operation, usually addition, subtraction, or logical XOR, and m is a modulus which ensures the output is in a proper range. Usually, $m = 2^{32} - 1$, to give maximum period and avoid correlation issues that can arise when $m = 2^{32}$. The generator stores the past r items of state, then uses the recurrence relation above in Equation 2 to generate the next output. The two lag constants r and s are specially chosen to avoid correlations.

For example, in a popular implementation called R250, $\star$ is defined as XOR, $r = 147$, and $s = 250$ [20]. Another popular implementation is RANLUX, which adds a few wrinkles to a simpler recurrence style generator, called Subtract with Carry [21], [22]:

$$X_n \equiv (X_{n-r} - X_{n-s} - c_{n-1}) \mod m \tag{3}$$

Here, r and s are the usual lag constants where $0 < s < r$, and c is defined as:

$$\begin{aligned} c_n &= 0 \qquad \text{if } X_{n-r} - X_{n-s} - c_{n-1} \geq 0, \\ c_n &= 1 \qquad \text{otherwise.} \end{aligned}$$

In essence, this is a simple LFG using subtraction where each term is possible decremented based on whether or not the previous operation required a borrowed term.

RANLUX takes this generator and adds skipping element parameter p, designed to remove unwanted correlations at the expense of speed. After R random numbers are generated, $p - R$ random numbers are thrown away [23], [24]. The default C++11 parameters for the RANLUX 24 bit engine are $r = 24$, $s = 10$, $p = 223$, and $R = 23$.

1) Strengths: Lagged Fibonacci generators can store a wide range of amount of state and their flexibility results in many generators of this type being created and seeing common use. The operations are relatively fast, and have larger periods when a few different twists like the subtract with carry has been applied [22]. As a result, good implementations generally do well on most of the Diehardertests, with sporadic failures or weak results.

2) Weaknesses: LFGs have a wrinkle in understanding their simplicity; they are difficult to seed properly, and require special initialization to make sure that there are no linear correlations in the initial seed data, lest the entire generator become extremely predictable [25]. Furthermore, their flexibility also allows for a wide range of quality so care must be taken when choosing parameters. Some parameter sets may produce poor generators, although this problem is not as pronounced as in LCGs. They are in general, slower and require more state than Xorshift generators, particularly when using more complex variants, such as RANLUX. They also do not have periods as large as Xorshift or Multiply with Carry Generators, discussed next.

D. Multiply With Carry Generators

Multiply With Carry generators, or MWCs are very similar to the previously discussed LCG generators. The primary difference is that rather than adding a constant c during each operation, a value is carried over and added from the previous operation. The concept of variable lag as a parameter is also borrowed from Lagged Fibonacci generators. This givens MWCs the general form:

$$X_{n+1} = (aX_{n-r} + c_n) \mod m \tag{4}$$

Where c_n is given by:

$$c_n = \left\lfloor \frac{aX_{n-1-r} + c_{n-1}}{m} \right\rfloor \tag{5}$$

The value of c_n is simply the 'carry' or overflow from the previous operation.

The combination of variable lag and nonconstant addends allow MWC generators to have much longer periods than their LCG counterparts as well as greater randomness. Intuitively, this is due to the storage of additional data about the prior sequence. In LCG generators, the amount of overflow from each operation is simply discarded via the modulo operation. Here, that overflow is stored and used as part of the internal state.

The period of such a generator is heavily dependent on the selection of the lag r and the modulus m [?]. Computing the period is difficult but is made easier by selecting parameters such that $p = am^r - 1$ is prime. There may be many lags r that satisfy this for a given a and m but this set may be narrowed if p is a safe prime, that is, if both p and $(p-1)/2$ are prime. If this is not the case, factoring $p-1$ must be factored to determine the optimal lag. Unfortunately, factoring is a difficult problem so, in practice, the Complementary Multiply With Carry variant is most often used. This generator is described by the form:

$$X_{n+1} = (m-1) - (aX_{n-r} + c_n) \mod m \tag{6}$$

This alternative form has period related instead to $p = am^r + 1$, which makes $p - 1 = am^r$ far easier to factor. This eases the process of finding parameter sets that yield the greatest period.

1) Strengths: Built on the very lightweight LCG generator, MWC generators share the desirable property of being very fast, as it is computed using a small number of basic arithmetic operations. Unlike its predecessor, MWC generators can have very long periods, making them quite useful in most applications.

2) Weaknesses: Just like other generators utilizing lag, MWC generators require additional memory to store its state compared to LCG generators, making it slightly less desirable when used in memory-constrained applications. Also like other lagged generators, seeding a MWC generator may require some finesse not needed in seeding the LCG generator.

E. Mersenne Twister Generators

One of the most commonly used RNGs, the Mersenne Twister (MT), was originally discovered by [?]. It is essentially an extension on top of the classical linear feedback shift register (LFSR). A simple LFSR is comprised of a bitwise shift-register where its input is determined by some combination of its current contents, traditionally the XOR of a select few bits from the register. While the LFSR is itself a RNG, its linear nature makes it very easily predictable. The Mersenne Twister improves upon the base concept by obfuscating the internal state of the register before feeding it into the XOR operation as well as applying additional transformations on the output to make the resulting sequence more uniform.

Mathematically, the MT operation for determining the next number in the sequence is:

$$X_{k+n} = X_{k+m} \oplus (X_k^u | X_{k+1}^l l)A \tag{7}$$

Where w is width of each word, X_k^u is the upper $w - r$ bits of the kth word, $X_{k+1}^l l$ is the lower r bits of the $k+1$th word, and A is matrix of the form:

$$\begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & a_w ... a_0 \end{pmatrix}$$

Colloquially, this is a matrix where the bottom row is a vector a and the upper rows are comprised of a single identity matrix shifted to the right by one column. In effect, left multiplying by this matrix yields a vector whose bits are masked by a and right shifted once. Again, this is similar to the operation of a simple LFSR except that the inputs to the XOR are masked before being processed.

Finally, to generate the desired random numbers, the sequence built from the above equations are transformed once more as such:

$$y := x \oplus (x >> u) \tag{8}$$

$$y := y \oplus ((y << s) \cdot b) \tag{9}$$

$$y := y \oplus ((y << t) \cdot c) \tag{10}$$

$$y := y \oplus (y >> l) \tag{11}$$

This is done so that each possible word occurs the same number of times within one period, with the exception of the all-zero word, which is less represented due to the underlying LFSR backbone.

The vast amount of parameters needed to define an instance of the MT algorithm along with the option of additional flexibility means that there exist many variants of this generator. For example, the 32-bit generator used in C++11 (MT19937) uses the following parameters: w=32, n=624, m=397, a=0x9908B0DF, b=0x9D2C5680, c=0xEFC60000, u=11, s=7, t=15, l=18.

1) Strengths: Mercenne Twister generators were originally designed to address concerns regarding the quality of randomness provided by prior generators. It is unsurprising then that these generators generally perform very well in the Dieharder tests. They are also well-known for their incredibly long periods ($2^{19937} - 1$). Due to the popularity of this general type of generator, a number of specialized variants have also been created to mitigate its shortcomings, such as TinyMT, which boasts a miniscule internal state of only 127 bits. Of course, such variants do not come without drawbacks. In the case of TinyMT, shrinking the internal state size also necessarily decreases the length of the period. Still, this type of flexibility means that it is possible to gain the benefits of the MT generator tailored to the resource limits and constraints of specific applications, providing a great deal of versatility to the algorithm.

2) Weaknesses: The complex design of this generator unfortunately leads to slower performance, although it is certainly not the slowest of the bunch. Also, much like Lagged Fibonacci generators, MT generators store a great deal of state. This is especially true for MT generators since its implementations typically require much more state storage than its Lagged Fibonacci counterparts. Similarly, care must be taken when initializing the generator upon seeding. For example, large numbers of 0s in the initialization vector can cause the algorithm to produce long series of 0s as output until the internal recurrence breaks out of this pattern. It's also worth mentioning that despite its high quality outputs, the base MT algorithm does not pass the next bit test and is not suited for strict cryptographic applications.

IV. Software Implementation

For this project, we use the Dieharder binary on Ubuntu in its raw input mode. This allows us to use a shell pipe to send raw random numbers from an arbitrary generator into Dieharder for analysis.

There are two major components included in our code: C++ source to build an RNG binary, and a small Python script to call the RNG binary and pipe it into Dieharder appropriately. We chose C++ because these tests can take a long time if the generator is slow, and C++ is a good compromise between its ability to allow for modular code and its speed.

The specific RNG algorithms implemented are as follows:

- Mersenne Twister
- RANDU

TABLE III
TABLE DISPLAYING DIEHARDER RESULTS FOR THE IMPLEMENTED RNGS.

RNG Name	Passes	Weaks	Fails
STL	108	5	1
MT19937	108	5	1
RANDU	6	3	105
MINSTD	107	6	1
R250	107	6	1
RANLUX	79	8	27
Xorshift	107	7	0
CMWC	113	1	0

- MINSTD
- R250
- RANLUX
- Xorshift
- CMWC

Each RNG algorithm inherits from a common base class, which allows for the RNG binary to choose the RNG in use via a command line argument. Random 32 bit numbers are written to `stdout` in raw binary, which is in turned piped to Dieharder for analysis. Finally, the output of Dieharder can be optionally piped to file.

All code for this project is available on Github [6], with some selected source files in Appendix B.The project was tested using Ubuntu 14.04, gcc version 4.9.1, and Python version 3.4.2. Note that Dieharder is required to run examples, and can be installed via your package manager if available, or built from source [5].

V. RESULTS AND ANALYSIS

Overall, the RNGs that we implemented performed as expected. All RNGs (with the notable and expected exception of RANDU) passed the majority of the tests, with at most one failure, and a few weak results which could very possibly pass if tested with different seeds. Statistics for passes, weaks, and failures are presented in visual form in Figure 2 and tabularly in Table IV.

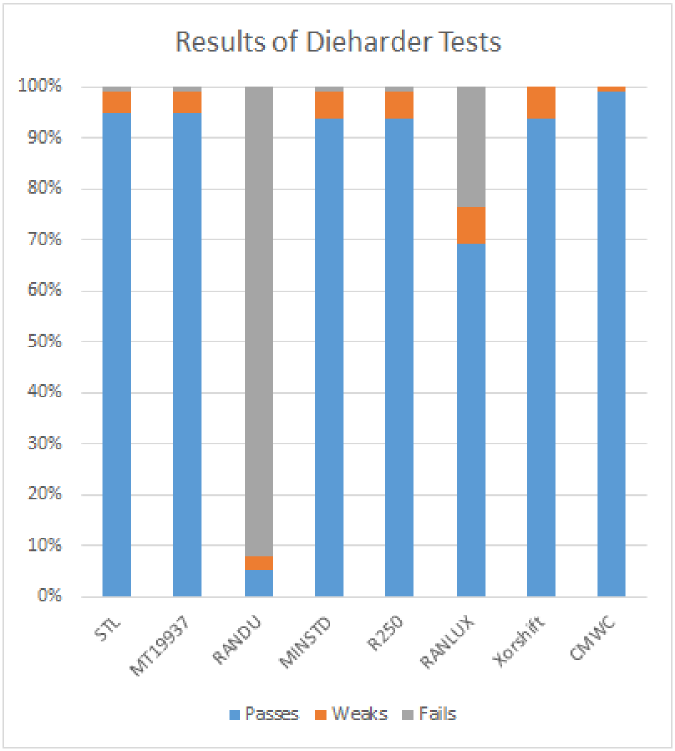

Fig. 2. Percentage bar graph of results of Dieharder test results for implemented RNGS.

Another metric worth measuring is RNG speed. This metric has been alluded, but is formally measured in random numbers per second. Luckily, Dieharder generates this metric already. As expected, the STL implementation of the Mersenne Twister is quite fast (as is much of the STL), and the RANLUX implementation is slow (because it must discard so many numbers to keep the quality of output high). Of the other RNGs, our Mersenne Twister is the most complicated, and as a result it is the slowest. RANDU is simple and fast, and the other algorithms hover around the same speed. The data is presented graphically in Figure 3 and tabularly in Table III

It is important to remember that these speeds must be considered to be relative. These RNGs must be piped through the shell, as opposed to the GNU Scientific Library RNGs built into the Dieharder binary, and as a result, they can be faster.

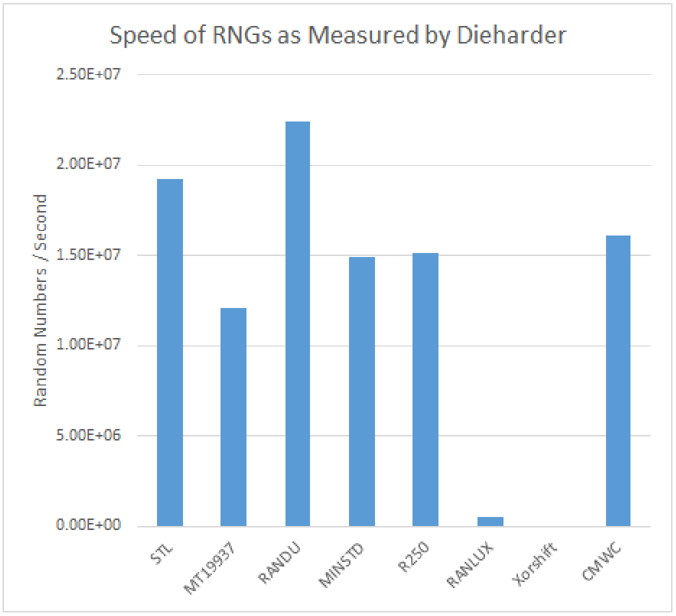

Fig. 3. Speed of implemented RNGs in random numbers per second as measured by Dieharder.

TABLE IV
TABLE DISPLAYING SPEEDS OF IMPLEMENTED RNGS AS MEASURED BY DIEHARDER.

RNG Name	Speed
STL	1.92E+07
MT19937	1.21E+07
RANDU	2.24E+07
MINSTD	1.49E+07
R250	1.51E+07
RANLUX	5.13E+05
Xorshift	2.54E+00
CMWC	1.61E+07

VI. CONCLUSION

Hopefully at this point, one is convinced not only that testing random number generators is a tricky art which is difficult to get right, but that despite this, there are many reasonable choices for RNGs which pass the tests which we have. So, what is a user of Random Numbers to do?!

It turns out that the choice is more or less made for most users. By far, the most common random number in service today is the Mersenne Twister. It is the default RNG in Python, MATLAB, the GNU Scientific Library, as well as being available and encouraged in C++11. The trade offs for memory and speed efficiency simply don't matter as much to most users, especially with modern computers. Thus, a RNG with a large period like the Mersenne Twister, which is also proved to be well distributed in higher dimensions, seems like an obvious choice.

Not everyone agrees with these programming language designers! George Marsaglia, a noted RNG researcher, developed equally performant on test suites like Dieharder, generators of the CMWC variety which are faster, store less state, and have equally long periods [26]. The developers of the Mersenne Twister also have a different generator called WELL, developed by the same researchers who wrote the Mersenne Twister, which has similar arguments for its usefulness [27]. These same researchers also have created different versions of the Mersenne Twister, which take greater advantage of modern CPU architecture to gain higher performance [28], [29]. Still other RNGs designed specifically for GPUs are gainin popularity in gaming systems [30], [31]. These generators are on the cutting edge of random number generation research, and as such, it will take longer for programming langauge desingers to adopt them.

Ultimately, it is the job of the system designer to carefully weigh the tradeoffs of different random number generation techniques and chose one for their system. Luckily, they will have tools like Dieharder to aid them in their decision.

ACKNOWLEDGMENT

The authors would like to thank Professor William H. Sanders for his guidance and the course staff for the University of Illinois course ECE/CS 541: Computer Systems Analysis.

APPENDIX A
RESULTS FROM DIEHARDER

This section includes the raw results outputs from a run of Dieharder for some of our implemented RNGs. Plaintext files with these results, and the full results for the rest of the RNGs, can be found in the Github repository [6].

A. Mersenne Twister

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  3.69e+06  | 334501002|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.94658958|  PASSED
      diehard_operm5|   0|   1000000|     100|0.96198385|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.24240504|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.56033672|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.36069880|  PASSED
        diehard_opso|   0|   2097152|     100|0.40862193|  PASSED
        diehard_oqso|   0|   2097152|     100|0.45443937|  PASSED
         diehard_dna|   0|   2097152|     100|0.06000922|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.17495967|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.79645250|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.43824831|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.17246784|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.16472438|  PASSED
     diehard_squeeze|   0|    100000|     100|0.60317887|  PASSED
        diehard_sums|   0|       100|     100|0.02682330|  PASSED
        diehard_runs|   0|    100000|     100|0.60254832|  PASSED
        diehard_runs|   0|    100000|     100|0.46677063|  PASSED
       diehard_craps|   0|    200000|     100|0.32427347|  PASSED
       diehard_craps|   0|    200000|     100|0.27961217|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.23485073|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.82596264|  PASSED
         sts_monobit|   1|    100000|     100|0.85246403|  PASSED
            sts_runs|   2|    100000|     100|0.35817371|  PASSED
          sts_serial|   1|    100000|     100|0.74162598|  PASSED
          sts_serial|   2|    100000|     100|0.16597190|  PASSED
          sts_serial|   3|    100000|     100|0.91870927|  PASSED
          sts_serial|   3|    100000|     100|0.60604836|  PASSED
          sts_serial|   4|    100000|     100|0.49807683|  PASSED
          sts_serial|   4|    100000|     100|0.73416252|  PASSED
          sts_serial|   5|    100000|     100|0.99500583|   WEAK
          sts_serial|   5|    100000|     100|0.82089520|  PASSED
          sts_serial|   6|    100000|     100|0.69150250|  PASSED
          sts_serial|   6|    100000|     100|0.90675682|  PASSED
          sts_serial|   7|    100000|     100|0.50223133|  PASSED
          sts_serial|   7|    100000|     100|0.25121204|  PASSED
          sts_serial|   8|    100000|     100|0.79597609|  PASSED
          sts_serial|   8|    100000|     100|0.74387173|  PASSED
          sts_serial|   9|    100000|     100|0.48188135|  PASSED
          sts_serial|   9|    100000|     100|0.32245638|  PASSED
          sts_serial|  10|    100000|     100|0.85675457|  PASSED
          sts_serial|  10|    100000|     100|0.58565755|  PASSED
          sts_serial|  11|    100000|     100|0.27398289|  PASSED
          sts_serial|  11|    100000|     100|0.96699373|  PASSED
          sts_serial|  12|    100000|     100|0.99571085|   WEAK
          sts_serial|  12|    100000|     100|0.99998250|   WEAK
          sts_serial|  13|    100000|     100|0.83769749|  PASSED
          sts_serial|  13|    100000|     100|0.70406949|  PASSED
          sts_serial|  14|    100000|     100|0.33108799|  PASSED
          sts_serial|  14|    100000|     100|0.33874542|  PASSED
          sts_serial|  15|    100000|     100|0.62491377|  PASSED
          sts_serial|  15|    100000|     100|0.97355969|  PASSED
          sts_serial|  16|    100000|     100|0.52289877|  PASSED
          sts_serial|  16|    100000|     100|0.91101554|  PASSED
         rgb_bitdist|   1|    100000|     100|0.71168623|  PASSED
         rgb_bitdist|   2|    100000|     100|0.50192972|  PASSED
         rgb_bitdist|   3|    100000|     100|0.98142915|  PASSED
         rgb_bitdist|   4|    100000|     100|0.77984554|  PASSED
         rgb_bitdist|   5|    100000|     100|0.66101093|  PASSED
         rgb_bitdist|   6|    100000|     100|0.91816382|  PASSED
         rgb_bitdist|   7|    100000|     100|0.80046681|  PASSED
         rgb_bitdist|   8|    100000|     100|0.79769030|  PASSED
         rgb_bitdist|   9|    100000|     100|0.39761538|  PASSED
         rgb_bitdist|  10|    100000|     100|0.72563831|  PASSED
         rgb_bitdist|  11|    100000|     100|0.94635747|  PASSED
         rgb_bitdist|  12|    100000|     100|0.00790713|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.66619425|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.29126310|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.61539740|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.69380013|  PASSED
    rgb_permutations|   2|    100000|     100|0.54292104|  PASSED
    rgb_permutations|   3|    100000|     100|0.84339338|  PASSED
    rgb_permutations|   4|    100000|     100|0.65721535|  PASSED
    rgb_permutations|   5|    100000|     100|0.63741550|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.91499091|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.58803936|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.45161465|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.19256312|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.78964801|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.99999945|  FAILED
      rgb_lagged_sum|   6|   1000000|     100|0.92564798|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.42382792|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.86728057|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.94533364|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.86319260|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.67230198|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.60087872|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.53686765|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.59498101|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.25928645|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.99649444|   WEAK
      rgb_lagged_sum|  17|   1000000|     100|0.57762461|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.61051907|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.99642829|   WEAK
      rgb_lagged_sum|  20|   1000000|     100|0.70037387|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.48495808|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.95914643|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.30312748|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.26160988|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.13145792|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.84738587|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.16458415|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.12698620|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.96405239|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.67856463|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.24194016|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.37704298|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.65808472|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.63940335|  PASSED
             dab_dct| 256|     50000|       1|0.31841243|  PASSED
Preparing to run test 207.  ntuple = 0
        dab_filltree|  32|  15000000|       1|0.88641557|  PASSED
        dab_filltree|  32|  15000000|       1|0.89984663|  PASSED
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.61511593|  PASSED
       dab_filltree2|   1|   5000000|       1|0.50735769|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.27042876|  PASSED
```

B. RANDU

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  7.88e+06  |3038070828|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.00000000|  FAILED
      diehard_operm5|   0|   1000000|     100|0.00000000|  FAILED
  diehard_rank_32x32|   0|     40000|     100|0.00000000|  FAILED
    diehard_rank_6x8|   0|    100000|     100|0.00000000|  FAILED
   diehard_bitstream|   0|   2097152|     100|0.00000000|  FAILED
        diehard_opso|   0|   2097152|     100|0.00000000|  FAILED
        diehard_oqso|   0|   2097152|     100|0.00000000|  FAILED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.00000000|  FAILED
diehard_count_1s_byt|   0|    256000|     100|0.00000000|  FAILED
 diehard_parking_lot|   0|     12000|     100|0.00000000|  FAILED
    diehard_2dsphere|   2|      8000|     100|0.00000000|  FAILED
    diehard_3dsphere|   3|      4000|     100|0.00000000|  FAILED
     diehard_squeeze|   0|    100000|     100|0.00000000|  FAILED
        diehard_sums|   0|       100|     100|0.00000000|  FAILED
        diehard_runs|   0|    100000|     100|0.00581517|  PASSED
        diehard_runs|   0|    100000|     100|0.00088099|   WEAK
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
         sts_monobit|   1|    100000|     100|0.00132032|   WEAK
            sts_runs|   2|    100000|     100|0.05116931|  PASSED
          sts_serial|   1|    100000|     100|0.04426849|  PASSED
          sts_serial|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  15|    100000|     100|0.00000000|  FAILED
          sts_serial|  15|    100000|     100|0.00000000|  FAILED
          sts_serial|  16|    100000|     100|0.00000000|  FAILED
          sts_serial|  16|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   1|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   2|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   3|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   4|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   5|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   6|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   7|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   8|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   9|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  10|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  11|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  12|    100000|     100|0.00000000|  FAILED
rgb_minimum_distance|   2|     10000|    1000|0.00000000|  FAILED
rgb_minimum_distance|   3|     10000|    1000|0.00000000|  FAILED
rgb_minimum_distance|   4|     10000|    1000|0.00000000|  FAILED
rgb_minimum_distance|   5|     10000|    1000|0.00000000|  FAILED
    rgb_permutations|   2|    100000|     100|0.97069761|  PASSED
    rgb_permutations|   3|    100000|     100|0.28310835|  PASSED
    rgb_permutations|   4|    100000|     100|0.66914232|  PASSED
    rgb_permutations|   5|    100000|     100|0.00000705|   WEAK
      rgb_lagged_sum|   0|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   1|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   2|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   3|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   4|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   5|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   6|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   7|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   8|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|   9|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  10|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  11|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  12|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  13|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  14|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  15|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  16|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  17|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  18|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  19|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  20|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  21|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  22|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  23|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  24|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  25|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  26|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  27|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  28|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  29|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  30|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  31|   1000000|     100|0.00000000|  FAILED
      rgb_lagged_sum|  32|   1000000|     100|0.00000000|  FAILED
     rgb_kstest_test|   0|     10000|    1000|0.00000000|  FAILED
     dab_bytedistrib|   0|  51200000|       1|0.00000000|  FAILED
             dab_dct| 256|     50000|       1|0.00000000|  FAILED
Preparing to run test 207.  ntuple = 0
        dab_filltree|  32|  15000000|       1|0.00000000|  FAILED
        dab_filltree|  32|  15000000|       1|0.00000000|  FAILED
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.00000000|  FAILED
       dab_filltree2|   1|   5000000|       1|0.00000000|  FAILED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|1.00000000|  FAILED
```

C. Xorshift

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  7.18e+06  |1616344042|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.49902432|  PASSED
      diehard_operm5|   0|   1000000|     100|0.25474793|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.03165397|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.25767730|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.20672267|  PASSED
        diehard_opso|   0|   2097152|     100|0.25133362|  PASSED
        diehard_oqso|   0|   2097152|     100|0.66694775|  PASSED
         diehard_dna|   0|   2097152|     100|0.75118125|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.64648928|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.11279025|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.41532341|  PASSED
```

```
    diehard_2dsphere|   2|      8000|     100|0.65294377|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.41726522|  PASSED
     diehard_squeeze|   0|    100000|     100|0.88235090|  PASSED
        diehard_sums|   0|       100|     100|0.30287804|  PASSED
        diehard_runs|   0|    100000|     100|0.61241815|  PASSED
        diehard_runs|   0|    100000|     100|0.93971074|  PASSED
       diehard_craps|   0|    200000|     100|0.94690900|  PASSED
       diehard_craps|   0|    200000|     100|0.88014864|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.66822887|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.23714996|  PASSED
         sts_monobit|   1|    100000|     100|0.07263718|  PASSED
            sts_runs|   2|    100000|     100|0.59150620|  PASSED
          sts_serial|   1|    100000|     100|0.98865452|  PASSED
          sts_serial|   2|    100000|     100|0.67923671|  PASSED
          sts_serial|   3|    100000|     100|0.40038667|  PASSED
          sts_serial|   3|    100000|     100|0.74590304|  PASSED
          sts_serial|   4|    100000|     100|0.76395907|  PASSED
          sts_serial|   4|    100000|     100|0.45655805|  PASSED
          sts_serial|   5|    100000|     100|0.72571182|  PASSED
          sts_serial|   5|    100000|     100|0.48061111|  PASSED
          sts_serial|   6|    100000|     100|0.81233938|  PASSED
          sts_serial|   6|    100000|     100|0.65075997|  PASSED
          sts_serial|   7|    100000|     100|0.78242940|  PASSED
          sts_serial|   7|    100000|     100|0.54454691|  PASSED
          sts_serial|   8|    100000|     100|0.69263905|  PASSED
          sts_serial|   8|    100000|     100|0.71216880|  PASSED
          sts_serial|   9|    100000|     100|0.25173055|  PASSED
          sts_serial|   9|    100000|     100|0.25828734|  PASSED
          sts_serial|  10|    100000|     100|0.35354245|  PASSED
          sts_serial|  10|    100000|     100|0.12568077|  PASSED
          sts_serial|  11|    100000|     100|0.99890521|   WEAK
          sts_serial|  11|    100000|     100|0.59350010|  PASSED
          sts_serial|  12|    100000|     100|0.81969952|  PASSED
          sts_serial|  12|    100000|     100|0.98941712|  PASSED
          sts_serial|  13|    100000|     100|0.99975291|   WEAK
          sts_serial|  13|    100000|     100|0.71872451|  PASSED
          sts_serial|  14|    100000|     100|0.99917709|   WEAK
          sts_serial|  14|    100000|     100|0.99773009|   WEAK
          sts_serial|  15|    100000|     100|0.96041847|  PASSED
          sts_serial|  15|    100000|     100|0.90680839|  PASSED
          sts_serial|  16|    100000|     100|0.55058080|  PASSED
          sts_serial|  16|    100000|     100|0.22192587|  PASSED
         rgb_bitdist|   1|    100000|     100|0.95111713|  PASSED
         rgb_bitdist|   2|    100000|     100|0.46876584|  PASSED
         rgb_bitdist|   3|    100000|     100|0.07937752|  PASSED
         rgb_bitdist|   4|    100000|     100|0.31077302|  PASSED
         rgb_bitdist|   5|    100000|     100|0.23549435|  PASSED
         rgb_bitdist|   6|    100000|     100|0.37517346|  PASSED
         rgb_bitdist|   7|    100000|     100|0.90181028|  PASSED
         rgb_bitdist|   8|    100000|     100|0.71075597|  PASSED
         rgb_bitdist|   9|    100000|     100|0.42126443|  PASSED
         rgb_bitdist|  10|    100000|     100|0.17021069|  PASSED
         rgb_bitdist|  11|    100000|     100|0.10316838|  PASSED
         rgb_bitdist|  12|    100000|     100|0.42217482|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.24582710|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.51767092|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.96320260|  PASSED
```

```
rgb_minimum_distance|   5|     10000|    1000|0.33004027|  PASSED
    rgb_permutations|   2|    100000|     100|0.85682126|  PASSED
    rgb_permutations|   3|    100000|     100|0.97107041|  PASSED
    rgb_permutations|   4|    100000|     100|0.46006773|  PASSED
    rgb_permutations|   5|    100000|     100|0.00182706|   WEAK
      rgb_lagged_sum|   0|   1000000|     100|0.03326173|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.05387456|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.93556085|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.56177954|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.53075714|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.97957264|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.99626119|   WEAK
      rgb_lagged_sum|   7|   1000000|     100|0.84475689|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.20696248|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.86572183|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.77684529|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.80172300|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.16379969|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.07773692|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.52496663|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.79522715|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.48450975|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.01868482|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.98448891|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.34024310|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.97691977|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.33271752|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.99307319|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.36269171|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.07769534|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.06889774|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.95699166|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.69000388|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.48259760|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.25775601|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.95312206|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.80795102|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.75798961|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.77393295|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.42869597|  PASSED
             dab_dct| 256|     50000|       1|0.29261724|  PASSED
Preparing to run test 207.  ntuple = 0
        dab_filltree|  32|  15000000|       1|0.90526013|  PASSED
        dab_filltree|  32|  15000000|       1|0.88055907|  PASSED
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.74399501|  PASSED
       dab_filltree2|   1|   5000000|       1|0.70764739|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.99626005|   WEAK
```

APPENDIX B
SELECTED SOURCE FILES

This section includes some source files from the code base. Interested readers are encouraged to view the Github Repository for all code [6].

The first file included is `rng.h` in Section B-A. This header file defines the interfaces for all the implemented RNGs. The next included sources are `xorshift.h` and `xorshift.cpp`, in Sections B-B and B-C, respectively. This should give the reader a feel for what the internals of the generators look like. Finally, the Python calling script, `rngs.py` in Section B-D, which demonstrates hooking up the produced RNG binary to Dieharder, is included as well.

A. rng.h

The source for `rng.h`, the basic skeleton of all implemented generators.

```
/*
 * rngs
 * ECE 541 Project 2
 * Kashev Dalmia - dalmia3
 * David Huang   - huang157
 */

#ifndef RNG_H
#define RNG_H

#include <cstdint> // for fixed width integer types.
/*
 * Convenience typedef for the integer type being used.
 */
typedef uint32_t fuint;

namespace rng
{
    class RandomNumberGenerator {
    public:
        RandomNumberGenerator(){};
        virtual ~RandomNumberGenerator(){};
        virtual void seed(fuint seed_num) = 0;
        virtual fuint operator()() = 0;
    };
}

#endif /* RNG_H */
```

B. xorshift.h

The source for `xorshift.h`, an example generator implementation.

```
/*
 * rngs
 * ECE 541 Project 2
 * Kashev Dalmia - dalmia3
 * David Huang   - huang157
 */

#ifndef XORSHIFT_H
#define XORSHIFT_H

#include "rng.h"

namespace rng
{
    class Xorshift : public RandomNumberGenerator {
    public:
        Xorshift();
        void seed(fuint seed_num);
        fuint operator()();
    private:
        fuint x, y, z, w;
    };
}

#endif /* XORSHIFT_H */
```

C. *xorshift.cpp*

The source for `xorshift.cpp`, an example generator implementation.

```
/*
 * rngs
 * ECE 541 Project 2
 * Kashev Dalmia - dalmia3
 * David Huang   - huang157
 */

#include "xorshift.h"

namespace rng
{
    Xorshift::Xorshift() :
            RandomNumberGenerator(),
            x(0),
            y(0),
            z(0),
            w(0)
    {}

    void Xorshift::seed(fuint seed_num) {
        static constexpr uint64_t g = 48271;
        static constexpr uint64_t n = (1UL << 32UL) - 1UL; // 2^32 - 1
        x = seed_num;
        /*
         * Use MINSTD to finish populating state.
         */
        y = static_cast<fuint>((static_cast<uint64_t>(x) * g) % n);
        z = static_cast<fuint>((static_cast<uint64_t>(y) * g) % n);
        w = static_cast<fuint>((static_cast<uint64_t>(z) * g) % n);
    }

    fuint Xorshift::operator()() {
        fuint t = x ^ (x << 11);
        x = y;
        y = z;
        z = w;
        w = w ^ (w >> 19) ^ t ^ (t >> 8);
        return w;
    }
}
```

D. rngs.py

The source for `rngs.py`, the main code for the RNG binary.

```
#!/usr/bin/env python3
# rngs
# ECE 541 Project 2
# Kashev Dalmia - dalmia3
# David Huang   - huang157

"""␣The␣called␣script␣for␣this␣project.␣Wraps␣the␣C++␣based␣RNG,␣passes
␣␣␣␣necessary␣options␣to␣it,␣and␣then␣pipes␣the␣output␣to␣dieharder.␣Note␣that
␣␣␣␣this␣script␣requires␣Python3.
"""

import argparse
import enum
import os
import subprocess


@enum.unique
class Generator(enum.Enum):
    all = -1
    stl = 0
    mt19937 = 1
    randu = 2
    minstd = 3
    r250 = 4
    ranlux = 5
    xorshift = 6
    cmwc = 7
    # Add more generator types here! Remember to update main.cpp.


def check_generator_type(value):
    try:
        ival = int(value)
        for gen in Generator:
            if gen.value == ival:
                return gen
    except ValueError:
        for gen in Generator:
            if gen.name == value:
                return gen

    raise argparse.ArgumentTypeError(
        "{}␣is␣an␣invalid␣generator␣type.".format(value))
```

```
def main():
    """␣Parse␣command␣line␣arguments,␣pass␣appropriate␣ones␣to␣the␣RNG,
␣␣␣␣␣␣␣␣and␣pipe␣to␣dieharder.
␣␣␣␣"""

    # Change to script directory
    abspath = os.path.abspath(__file__)
    dname = os.path.dirname(abspath)
    os.chdir(dname)

    parser = argparse.ArgumentParser(
        description="Wrapper␣script␣for␣RNGs.␣Sets␣up␣C++␣RNG␣and␣pipes␣to"
                    "dieharder␣for␣analysis.",
        formatter_class=argparse.RawTextHelpFormatter)

    helpstring = "The␣generator␣can␣be␣any␣specified␣number␣or␣name:\n"
    for gen in Generator:
        helpstring += "{}␣|␣{}\n".format(gen.value, gen.name)

    parser.add_argument("generator", type=check_generator_type,
                        help=helpstring)
    parser.add_argument("--directory", type=str,
                        help="Directory␣to␣output␣files␣to",
                        action=’store’, default="results")
    parser.add_argument("--file", action=’store_true’,
                        help="Output␣to␣file␣or␣not.")

    parser.add_argument(’args’, nargs=argparse.REMAINDER)

    args = parser.parse_args()

    # Create Results Directory if it doesn’t exist, if the file option is
    # turned on
    if args.file and not os.path.exists(args.directory):
        os.makedirs(args.directory)

    # Check for running all generators.
    if args.generator == Generator.all:
        print("Using␣all␣generators...")
        for gen in Generator:
            if gen != Generator.all:
                if args.file:
                    pipestring = "␣>␣{}/{}.txt".format(args.directory, gen.name)
                else:
                    pipestring = ""
                print("Running␣generator␣{}...".format(gen.name))
                subprocess.call("./bin/rngs␣{}␣|␣dieharder␣-g␣200␣{}{}"
                                .format(gen.value,
                                        "␣".join(args.args),
                                        pipestring),
                                shell=True)

    # Run a single generator.
    else:
        if args.file:
            pipestring = "␣>␣{}/{}.txt".format(args.directory,
                                               args.generator.name)
        else:
            pipestring = ""
        print("Using␣generator␣{}...".format(args.generator.name))
        subprocess.call("./bin/rngs␣{}␣|␣dieharder␣-g␣200␣{}{}"
                        .format(args.generator.value,
                                "␣".join(args.args),
                                pipestring),
                        shell=True)

if __name__ == '__main__':
    main()
```

References

[1] C. Saiprasert, C.-S. Bouganis, and G. A. Constantinides, "An optimized hardware architecture of a multivariate gaussian random number generator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 1, pp. 2:1–2:21, Dec. 2010. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/1857927.1857929

[2] M. Barel, "Fast hardware random number generator for the tausworthe sequence," in *Proceedings of the 16th Annual Symposium on Simulation*, ser. ANSS '83. Los Alamitos, CA, USA: IEEE Computer Society Press, 1983, pp. 121–135. [Online]. Available: http://dl.acm.org.proxy2.library.illinois.edu/citation.cfm?id=800042.801454

[3] C. Hennebert, H. Hossayni, and C. Lauradoux, "Entropy harvesting from physical sensors," in *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '13. New York, NY, USA: ACM, 2013, pp. 149–154. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/2462096.2462122

[4] A. C. Yao, *Theory and applications of trapdoor functions*, 1982, pp. 80–91.

[5] R. G. Brown, D. Eddelbuettel, and D. Bauer. (2013) Dieharder: A random number test suite. [Online]. Available: http://www.phy.duke.edu/~rgb/General/dieharder.php

[6] K. Dalmia. (2014) kashev/rngs: A project for ECE 541 project #2 @ UIUC. Investigating Random Number Generators. [Online]. Available: https://github.com/kashev/rngs

[7] Y. V. Prokhorov, "Central limit theorem," in *Encyclopedia of Mathematics*. [Online]. Available: http://www.encyclopediaofmath.org/index.php?title=Central_limit_theorem&oldid=26376

[8] R. G. Brown, D. Eddelbuettel, and D. Bauer, "DieHarder: A Gnu Public License Random Number Tester," 2013.

[9] G. Marsaglia. (1995) The marsaglia random number cdrom including the diehard battery of tests of randomness. [Online]. Available: http://www.stat.fsu.edu/pub/diehard/

[10] E. K. Donald, *The art of computer programming*, 1999, vol. 3.

[11] L. E. Bassham, III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," Gaithersburg, MD, United States, Tech. Rep., 2010.

[12] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1268776.1268777

[13] M. Abramson and W. Moser, "More birthday surprises," *American Mathematical Monthly*, pp. 856–858, 1970.

[14] W. H. Payne, J. R. Rabung, and T. P. Bogyo, "Coding the lehmer pseudo-random number generator," *Commun. ACM*, vol. 12, no. 2, pp. 85–86, Feb. 1969. [Online]. Available: http://doi.acm.org/10.1145/362848.362860

[15] S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find," *Commun. ACM*, vol. 31, no. 10, pp. 1192–1201, Oct. 1988. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/63039.63042

[16] G. Marsaglia, "Random numbers fall mainly in the planes," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 61, pp. 25–28, 1968.

[17] P. L'Ecuyer, "Testing random number generators," in *Proceedings of the 24th Conference on Winter Simulation*, ser. WSC '92. New York, NY, USA: ACM, 1992, pp. 305–313. [Online]. Available: http://doi.acm.org/10.1145/167293.167354

[18] L. Sanchez. (2006) File:Randu.png. [Online]. Available: http://commons.wikimedia.org/wiki/File:Randu.png

[19] G. Marsaglia, "Xorshift rngs." [Online]. Available: http://www.jstatsoft.org/v08/i14/paper

[20] S. Kirkpatrick and E. P. Stoll, "A very fast shift-register sequence random number generator," *Journal of Computational Physics*, vol. 40, no. 2, pp. 517–526, 1981.

[21] G. Marsaglia and A. Zaman, "A new class of random number generators," *The Annals of Applied Probability*, pp. 462–480, 1991.

[22] G. Marsaglia, "Random number generators," *Journal of Modern Applied Statistical Methods*, vol. 2, no. 1, pp. 2–13, 2003.

[23] M. Lüscher, "A portable high-quality random number generator for lattice field theory simulations," *Computer physics communications*, vol. 79, no. 1, pp. 100–110, 1994.

[24] F. James, "Ranlux: A fortran implementation of the high-quality pseudorandom number generator of lüscher," *Computer Physics Communications*, vol. 79, no. 1, pp. 111–114, 1994.

[25] M. Matsumoto, I. Wada, A. Kuramoto, and H. Ashihara, "Common defects in initialization of pseudorandom number generators," *ACM Trans. Model. Comput. Simul.*, vol. 17, no. 4, Sep. 2007. [Online]. Available: http://doi.acm.org/10.1145/1276927.1276928

[26] G. Marsaglia, "Seeds for random number generators," *Commun. ACM*, vol. 46, no. 5, pp. 90–93, May 2003. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/769800.769827

[27] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Trans. Math. Softw.*, vol. 32, no. 1, pp. 1–16, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132973.1132974

[28] M. Saito and M. Matsumoto. SIMD-oriented fast mersenne twister (SFMT): twice faster than mersenne twister. [Online]. Available: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html

[29] ——. Tiny mersenne twister (TinyMT): A small-sized variant of mersenne twister. [Online]. Available: http://www.math.sci.hiroshima-u.ac.jp/~%20m-mat/MT/TINYMT/index.html

[30] J. Passerat-Palmbach, C. Mazel, and D. R. C. Hill, "Pseudo-random number generation on gp-gpu," in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org.proxy2.library.illinois.edu/10.1109/PADS.2011.5936751

[31] F. Zafar, M. Olano, and A. Curtis, "Gpu random numbers via the tiny encryption algorithm," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 133–141. [Online]. Available: http://dl.acm.org.proxy2.library.illinois.edu/citation.cfm?id=1921479.1921500