# A Design for a Cryptographically Secure Pseudo Random Number Generator

**Conference Paper** · September 2019

DOI: 10.1109/IDAACS.2019.8924431



**3 authors:**

Benjamin Williams
University of Idaho
**1** PUBLICATION **1** CITATION
SEE PROFILE

Robert Hiromoto
University of Idaho
**79** PUBLICATIONS **1,353** CITATIONS
SEE PROFILE

Albert H. Carlson
Austin Community College
**27** PUBLICATIONS **43** CITATIONS
SEE PROFILE

The 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications
18-21 September, 2019, Metz, France

# A Design for a Cryptographically Secure Pseudo Random Number Generator

Benjamin Williams [1], Robert E. Hiromoto [1,2], Albert Carlson [3]
[1] University of Idaho, 1776 Science Center Drive Idaho Falls, ID 83402
will9847@vandals.uidaho.edu, hiromoto@uidaho.edu
[2] Center for Advanced Energy Studies, 995 MK Simpson Boulevard, Idaho Falls, ID 83401
(Affiliate)
[3] CipherLoc Corp., 825 S Main St, Suite 100, Buda, TX 78610
acarlson@CipherLoc.com

***Abstract*—We proposes the design of a cryptographically secure pseudo random number generator (CSPRNG) that involves the permutation of the internal state of the PRNG. The design of the CSPRNG is described and Dieharder test comparisons are made against AES. The results of these tests are described and a discussion of future work concludes the paper.**

***Keywords—cryptographically secure; pseudorandom number generation; permutations; statistical testing.***

## I. INTRODUCTION

Encryptions relies on two main techniques to try and hide patterns in language. Those techniques are "*confusion*," that is the changing of one language symbol for another, and "*diffusion*" which mixes up the order of symbols or bits representing the symbol [1]. However, with the right assumptions, *diffusion* becomes an instance of *confusion*, that is, ALL encryptions are instances of substitution, or S ciphers [2]. It is known that S ciphers do not adequately hide patterns [3], and for this reason the cryptographic modes were created to inject randomness [4] to hide the statistical properties of patterns.

The One-Time-Pad (OTP) is recognized to be the most mathematically secure encryption technique. Although, the OTP has the smallest amount of information accumulating for each cipher/key pair, it has been shown that the attack known as the Vernam attack has been used to break OTP encryptions. A defense against the Vernam attack is to maximize the entropy associated with the selection of the cipher/key pairs. But entropy implies randomness (uncertainty), and while desirable for security, 'true' randomness is impossible to create and reproduce between both ends of an encrypted communication stream, i.e., it is impossible to calculate a matching stream of random numbers at two different locations.

The best practical solution to 'true' randomness is addressed by the development of deterministic process that "looks" random. Typically, only a single RNG is used throughout the encryption process. Unfortunately, if the RNG is used for more than a short period of time, sometimes less than 625 accesses [5], an attacker can identify the RNG and through localized "pattern" analysis, accurately predict the next "random" number to be generated. A methodology must be implemented that does not allow "patterning" of an RNG as it is used for encryption and other security functions.

In the rest of this paper, we examine a new cryptographically secure pseudo random number generators (PRNG) that is developed using Melissa E. O'Neill's [6] PCG (permuted congruential generator) PRNG family as the basis. During this development we identified, as did others [7], several weaknesses of PCG that are not mentioned in the original paper, which made it largely unsuitable for multi-stream use in cryptography. However, we found advantages in the idea of permutations. The end result is a single extremely statistically good PRNG that appears to meet the criteria for a cryptographically secure PRNG.

## II. CRYPTOGRAPHIC SECURITY

To be cryptographically secure, a PRNG must meet two requirements. The first is the next-bit test. This asserts that given the previous sequence of bits generated by the algorithm, there is no polynomial-time algorithm that can predict the next bit with better than 50% accuracy. Tests that perform well in Dieharder can reasonably be construed to satisfy this requirement, as it is an element of statistical goodness.

The second requirement for a cryptographically secure pseudo-random number generator (CSPRNG) is that it should withstand "state compromise extensions". This means that if an attacker learned part or all of the current state, it would be impossible to reconstruct the previous stream of random numbers. In practice, this means that a cryptographically secure PRNG must use a many-to-one function somewhere in the process, such that any attempt at reversing the process will exponentially increase the number of possible streams with each step back.

## III. PCG

The PCG family of generators is built around a linear congruential generator (LCG) [8], primarily for speed. O'Niell points out that LCGs have a bad reputation, and she asserts that it is undeserved, because other generators of similar speed and complexity perform far worse. The output of the LCG is then sent through a series of permutations, composed primarily of bit-twiddling operations, and then used as the PRNG output.

PCG on its own merit is not cryptographically secure, nor does the author claim that it is. This is because linear congruential generators (LCGs) are trivially reversible, given the current state. This is perhaps the biggest hurdle to overcome.

In addition to this, we found that PCG fails especially badly when run in parallel streams with similar seeds. According to Sebastiano Vigna [9], one of the creators of the xoshiro and xoroshiro PRNG families, a pair of PCGs given similar seeds will continue to produce correlated output indefinitely, in contrast to most other PRNGs, which will rapidly diverge. Our results verify this. We found that a 64k block generator using 16,000 parallel PCG streams would consistently ‘catastrophically” fail Dieharder, unless seeded with very high quality randomness. Reducing the block size (and thus number of parallel PRNGs) dramatically improved tests results, and using different constants for each generator also significantly improved performance, which verified that the lack of divergence is the problem. Unfortunately, neither of these mitigation strategies is scalable.

We also found that the quality of the permutations varied dramatically. Many of the permutations suggested in O'Neill's paper offer no statistical improvement over the base LCG when used alone. Some, especially the multiplication permutation, actually _reduced_ the statistical quality of the output, even after high quality permutations had improved it substantially. By the end, we had reduced the permutations actually worth using to the xorshift and rotate permutations. Alone these permutations work quite well, and when combined correctly, they work extremely well.

## IV. THE DESIGN

### A. Implementation

The original implementation of the PCG generator went through the followed the steps during each iteration:

1. Update the state of the LCG generator.
2. Mutate a copy of the state using some number of permutations.
3. Return the mutated state, truncated to 32 bits.

This approach has two critical weaknesses. First, the LCG is easily reversible. The state is only changed by the LCG, where the LCG is a bijective function. This means, that given the current state and the constants used by the particular LCG implementation, it is trivial to find all previous and future states of the generator. The second weakness is that multiple LCG streams using the same constants will not diverge when similar seeds are used. This is the cause of poor statistical goodness in multi-stream implementations.

After several trials and setbacks, one successful approach emerged that displayed both “cryptographically secure” and scalable properties. This new approach alters O'Neill's implementation slightly, with the following steps:

1. Update the state of the LCG generator.
2. Mutate a copy of the state using some number of permutations.
3. Write the mutated copy of the state back to the original state.
4. Return the mutated state, truncated to 32 bits.

The novelty comes in Step 3. Step 3 rolls the permutations into the generator itself. The permutations, therefore, not only mutates the output of the generator but also mutates the generator state.

The corresponding C code for getting a full block (32,768 bytes) of random data is depicted in Fig. 1.

```
void get_random_block(block_state *states, result_block
*results) {
    uint64_t lcg_intermed_array[(*states).len];
    block_state intermed = {(*states).len,
lcg_intermed_array};
    block_lcg(states);
    block_r_bytexorshift(states, &intermed);
    block_r_rotate(&intermed, &intermed);
    block_r_shortxorshift(&intermed, &intermed);
    block_r_introtate(&intermed, &intermed);
    block_r_intxorshift(&intermed, &intermed);
    block_r_shortrotate(&intermed, &intermed);
    block_r_xorshift(&intermed, &intermed);
    block_r_byterotate(&intermed, states);
    block_trunc(states, results);
}
```

Figure 1. C code

where block_state is a struct type containing an array and length metadata. The array length is 1,024 in the final crypt1 implementation but can be changed with a constant. This represents 1,024 parallel RNGs. In the final implementation 9 multipliers and 11 incrementers are used for LCG, for 99 unique LCGs. (The index of the element in the state array is used to select the multiplier and incrementer for each stream.)

Notice that the second to last line of the function passes the state pointer as its second argument. Each permutation function takes a source and destination argument. The first permutation, the 8 bit partitioned xorshift, passes "states" as the source and "&intermed" as the destination. Normally, this is the last place "states" is used in an iteration, however in this case for convenience, we use "states" as the destination in the final permutation, the 8 bit partitioned rotate. This function applies all of the permutations to the state of the RNGs, and not just to the output.

### B. *Permutations*

The original PCG permutations introduced by O'Neill [6] are reasonably simple operations that act on the entire output variable at once. After testing these permutations individually, we determined that only the family of xorshift permutations passed as a statistically good generator, according to the Dieharder tests[1]. We also found that the family of rotate permutations also performed well; however, the application of these rotations lacked sufficient mixing of variable bits. As a consequence, a solution was adopted that split the variables into smaller types for which rotate permutations are performed on those smaller pieces individually. The rotate function is illustrated in Fig. 2.

```
uint64_t rotate16(uint64_t rand, uint8_t mod) {
    // Take only 4 bits
    char distance = mod >> 4;
    // Cast to 16-bit pointer, so we can access as array
    uint16_t* v = (uint16_t*)&rand;
    for (int i = 0; i < 4; i++)
      // Perform 16 bit rotate
      v[i] = __rorw(v[i], distance)
    return rand;
}
```

Figure 2. Permutations

This approach takes 64 bits of state and partitions it into four 16 bit values, which then rotates each 16-bit value independently. The upper 4 bits of the "mod" argument determine how far each partition is rotated. Two other homogeneously partitioned rotates were implemented, for 8 and 32 bit partitions. Applied to both rotates and xorshifts, this provided a total of eight permutations.

The homogeneously partitioned permutations performed well, and in crypt1. They were sufficient to compete with AES in statistical goodness, but they still missed some valuable potential. As a consequence, a heterogeneously partitioned permutation was introduced where the 64-bit input is split into some number of unequally-sized chunks, so long as the chunks added up to 64 bits in total. This would allow interactions between asymmetric portions of the variable. The heterogeneous partitioning code is illustrated in Fig. 3.

In Fig. 3, the heterogeneous partitioning code rotates the first 8 bits, the next 16 bits, the next 32 bits, and then the final 8 bits, all independently. This allows mixing across boundaries that could not be crossed with homogeneous partitions.

Figure 4 illustrates how heterogeneous partitioning allows bit mixing that is otherwise impossible. The first four rows show the four homogeneous partitioning models, labeled with the bit size of each partition. The bottom row illustrates the partition model used in the above function. The blue bar shows how the second partition, which is 16 bits wide, does not line up with any of the homogeneous models. The 32 and 64 bit models do not split that region, but they also do not rotate that region exclusively.

```
uint64_t rotate8_16_32_8(uint64_t rand, uint8_t mod) {
    char distance = mod >> 3;
    void* ptr = &rand;
    uint8_t *first = ptr;
    ptr += sizeof(uint8_t);
    uint16_t *second = ptr;
    ptr += sizeof(uint16_t);
    uint32_t *third = ptr;
    ptr += sizeof(uint32_t);
    uint8_t *fourth = ptr;
    *first = __rorb(*first, distance);
    *second = __rorw(*second, distance);
    *third = __rord(*third, distance);
    *fourth = __rorb(*fourth, distance);
    return rand;
}
```

Figure 3. Heterogeneous Partitioning Code.

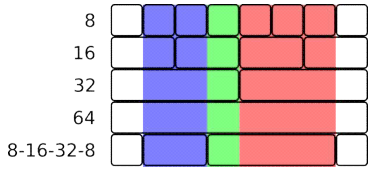


Figure 4. Heterogeneous Partitioning Scheme

The homogeneous models do not allow bits coming from one edge of that region to go directly onto the other edge, because there are other bits intervening. The red and green bars illustrate how bits on the right and left halves of the variable cannot be mixed effectively in any of the homogeneous models but can be mixed by the 32-bit partition in the heterogeneous model. The 64-bit model does allow some motion in that region, but again, there are many intervening bits interfering with effective mixing. Partitioning models like 16-32-16 and 8-8-32-8-8 allow the center 50% of bits to be mixed more effectively, and there are also other off-center models that allow action over the middle of the variable. By combining different partitioning models with different rotation distances, it is theoretically possible to shuffle the bits of a variable into any configuration, dramatically increasing the period and the possible output sequences.

## V. Statistical Testing

For statistical testing, we used two suites. TestU01 [10] was developed in 2007, and it appears to be the most commonly uses suite for statistical testing of PRNGs. Dieharder [11] is a PRNG statistical test suite, initially starting development in 2003, and based originally on George Marsaglia's "Diehard battery of tests" [12] with many more added as NIST and others developed new tests for their own suites. The vast majority of the testing is done using Dieharder, which has been maintained through at least 2016 and contains a much larger suite of

[1]Dieharder's documentation expects high quality generators to test weak on only around 1% of tests.

tests that are capable of far more intensive testing. TestU01 is used primarily as a second opinion on iterations that already performed well in Dieharder.

Dieharder tests by calculating the probability of the output of the PRNG, as measured by a series of tests. Outputs with very low and very high probabilities are flagged as weak or failed. According to Dieharder's documentation, a statistically good PRNG will test as weak on about one in every hundred tests. On the machine used for testing, a single run of tests with default settings took around 30 minutes.

Dieharder also has a "test to destruction" mode. This mode will run each test until it fails. No statistical testing method is perfect, so even with a perfect RNG, each of Dieharder's tests will eventually fail if run for long enough. As such, it is recommended to compare the results of "test to destruction" with the results of testing a known good PRNG.

## VI. Results

Tables 1 and 2 depict the Dieharder test comparisons between AES and the crypt1 PRNG described in this paper. The significance of the p-values is used primarily as a reference to how far from 0.5 they deviate. Excessively high or low p-values suggest statistical flaws in the RNG or flaws in the tests themselves. We test our RNG against an AES generated RNG, which is accepted as a good RNG, to control for flaws in the Dieharder tests. Thus if AES fails at a particular point, it is expected that any good RNG will fail near that same point. Note, however, that even extreme p-values have a non-zero probability of occurring even in true randomness, thus for a good generator, each Dieharder test with the default settings should get a "WEAK" result 1 in 100 tests and very rarely FAIL results are expected to occur for good RNGs.

Consequently, even for tests with the same outcomes, a comparison of their final p-values can be used to gain limited insight into their comparative strength. For example, in diehard_operm5, AES had a final p-value around 0.30 and crypt1 had a final p-value around 0.70, which are both 0.2 from 0.5, making them statistically equivalent. On diehard_craps, on the other hand, AES finished with 0.97, and crypt1 finished with 0.49, which tells us that AES may be slightly weaker on this test. This is a very weak assertion though, because both of those are well within the range of high probability outcomes. What is more interesting is when one test comes out WEAK and the other comes out PASS, because that is where the p-values deviate significantly from each other.

It is also worth noting that tests that came out WEAK on the final pass (those that are WEAK with 100,000 psamples) may be approaching the limits of the test. Even good generators may have higher odds of testing WEAK here than just 1 in 100.

TABLE 1. Dieharder AES test results

| | Dieharder | | AES | | |
|---|---|---|---|---|---|
| | test_name | ntup | psamples | p-value | Assessment |
| 1 | diehard_birthdays | 0 | 66000 | 0.00000092 | FAILED |
| 2 | diehard_operm5 | 0 | 100000 | 0.30576615 | PASSED |
| 3 | diehard_rank_32x32 | 0 | 100000 | 0.61673773 | PASSED |
| 4 | diehard_rank_6x8 | 0 | 100000 | 0.00207153 | WEAK |
| 5 | diehard_bitstream | 0 | 100000 | 0.78987437 | PASSED |
| 6 | diehard_opso | 0 | 100000 | 0.03566565 | PASSED |
| 7 | diehard_oqso | 0 | 100000 | 0.00006196 | WEAK |
| 8 | diehard_dna | 0 | 4300 | 0.00000081 | FAILED |
| 9 | diehard_count_1s_str | 0 | 100000 | 0.00823229 | PASSED |
| 10 | diehard_count_1s_byt | 0 | 100000 | 0.44539943 | PASSED |
| 11 | diehard_parking_lot | 0 | 20100 | 0.00000088 | FAILED |
| 12 | diehard_2dsphere | 2 | 100000 | 0.61041125 | PASSED |
| 13 | diehard_3dsphere | 3 | 100000 | 0.29515048 | PASSED |
| 14 | diehard_squeeze | 0 | 100000 | 0.64208327 | PASSED |
| 15 | diehard_sums | 0 | 1100 | 0.00000007 | FAILED |
| 16 | diehard_runs | 0 | 11900 | 0.00000077 | FAILED |
| 17 | diehard_craps | 0 | 100000 | 0.96712511 | PASSED |
| 18 | marsaglia_tsang_gcd | 0 | 27300 | 0.00000087 | FAILED |
| 19 | sts_monobit | 1 | 100000 | 0.05558083 | PASSED |
| 20 | sts_runs | 2 | 100000 | 0.22416225 | PASSED |
| 21 | sts_serial | 1 | 100000 | 0.14148169 | PASSED |
| 22 | sts_serial | 2 | 100000 | 0.78964866 | PASSED |
| 23 | sts_serial | 3 | 100000 | 0.12489156 | PASSED |
| 24 | sts_serial | 4 | 100000 | 0.58266529 | PASSED |
| 25 | sts_serial | 5 | 100000 | 0.08031215 | PASSED |
| 26 | sts_serial | 6 | 100000 | 0.87705157 | PASSED |
| 27 | sts_serial | 7 | 100000 | 0.35888505 | PASSED |
| 28 | sts_serial | 8 | 100000 | 0.46167686 | PASSED |
| 29 | sts_serial | 9 | 100000 | 0.34564016 | PASSED |
| 30 | sts_serial | 10 | 100000 | 0.51870631 | PASSED |
| 31 | sts_serial | 11 | 100000 | 0.43771321 | PASSED |
| 32 | sts_serial | 12 | 100000 | 0.65361182 | PASSED |
| 33 | sts_serial | 13 | 100000 | 0.40249368 | PASSED |
| 34 | sts_serial | 14 | 100000 | 0.24499522 | PASSED |
| 35 | sts_serial | 15 | 100000 | 0.23944811 | PASSED |
| 36 | sts_serial | 16 | 100000 | 0.5709769 | PASSED |

TABLE 2. Dieharder crypt1 test results

| | Dieharder | | crypt1 | | |
|---|---|---|---|---|---|
| | test_name | ntup | psamples | p-value | Assessment |
| 1 | diehard_birthdays | 0 | 77000 | 0.00000099 | FAILED |
| 2 | diehard_operm5 | 0 | 100000 | 0.72237221 | PASSED |
| 3 | diehard_rank_32x32 | 0 | 100000 | 0.12428824 | PASSED |
| 4 | diehard_rank_6x8 | 0 | 100000 | 0.01657774 | PASSED |
| 5 | diehard_bitstream | 0 | 100000 | 0.0019744 | WEAK |
| 6 | diehard_opso | 0 | 100000 | 0.45272381 | PASSED |
| 7 | diehard_oqso | 0 | 100000 | 0.00169381 | WEAK |
| 8 | diehard_dna | 0 | 5300 | 0.00000051 | FAILED |
| 9 | diehard_count_1s_str | 0 | 100000 | 0.03324601 | PASSED |
| 10 | diehard_count_1s_byt | 0 | 100000 | 0.00080845 | WEAK |
| 11 | diehard_parking_lot | 0 | 29400 | 0.000001 | FAILED |
| 12 | diehard_2dsphere | 2 | 100000 | 0.49646436 | PASSED |
| 13 | diehard_3dsphere | 3 | 100000 | 0.73603753 | PASSED |
| 14 | diehard_squeeze | 0 | 100000 | 0.87766305 | PASSED |
| 15 | diehard_sums | 0 | 1400 | 0.0000003 | FAILED |
| 16 | diehard_runs | 0 | 56600 | 0.00000091 | FAILED |
| 17 | diehard_craps | 0 | 100000 | 0.49276848 | PASSED |
| 18 | marsaglia_tsang_gcd | 0 | 23500 | 0.00000099 | FAILED |
| 19 | sts_monobit | 1 | 100000 | 0.02603976 | PASSED |
| 20 | sts_runs | 2 | 100000 | 0.45014286 | PASSED |
| 21 | sts_serial | 1 | 97500 | 0.02784438 | PASSED |
| 22 | sts_serial | 2 | 97500 | 0.07664873 | PASSED |
| 23 | sts_serial | 3 | 97500 | 0.82785512 | PASSED |
| 24 | sts_serial | 4 | 97500 | 0.93979936 | PASSED |
| 25 | sts_serial | 5 | 97500 | 0.50837805 | PASSED |
| 26 | sts_serial | 6 | 97500 | 0.74134683 | PASSED |
| 27 | sts_serial | 7 | 97500 | 0.56281584 | PASSED |
| 28 | sts_serial | 8 | 97500 | 0.08090197 | PASSED |
| 29 | sts_serial | 9 | 97500 | 0.26458637 | PASSED |
| 30 | sts_serial | 10 | 97500 | 0.22651285 | PASSED |
| 31 | sts_serial | 11 | 97500 | 0.1677171 | PASSED |
| 32 | sts_serial | 12 | 97500 | 0.23751934 | PASSED |
| 33 | sts_serial | 13 | 97500 | 0.66984901 | PASSED |
| 34 | sts_serial | 14 | 97500 | 0.00351755 | WEAK |
| 35 | sts_serial | 15 | 97500 | 0.33460998 | PASSED |
| 36 | sts_serial | 16 | 97500 | 0.62845924 | PASSED |

### A. *Comments regarding Tables 1 and 2 outcomes*

Test 1 – diehard_birthdays: crypt1 went for 11,000 more psamples than AES before failure.

Test 2 – diehard_operm5: Both passed; ending p-values are comparable.

Test 4 – diehard_rank_6x8: crypt1 tested stronger here, though WEAK results are expected in 1 of 100 tests even for good RNGs.

Test 5 – diehard_bitstream: AES tested stronger here, though see the above comment on WEAK results.

Test 7 – diehard_oqso: These three tests are suspected to be unreliable, according to Dieharder's documentation.

Test 10 – diehard_count_1s_byt: AES tested strong here, though see the above comment on WEAK results.

Test 11 – diehard_parking_lot: crypt1 went for 9,300 more psamples than AES before failure.

Test 15 – diehard_sums: This test is considered entirely unreliable, according to Dieharder's documentation.

Test 16 – diehard_runs: crypt1 went for 44,700 more psamples than AES before failure.

Test 18 – marsaglia_tsang_gcd: AES went for 3800 more psamples than AES before failure.

Test 21 thru 36 – sts_serial: crypt1 tests were terminated early from here down. This section seems to indicate fairly comparable results, though crypt1 was interrupted 2,500 psamples short of completion.

## VII. CONCLUSION

The biggest conclusion we came to is that PCG is a poor choice for the basis of a CSPRNG. It just has too many flaws to produce good quality output without so many permutations that it will function very slowly. It is especially poor for block generation, as the output quality is directly correlated to the quality of randomness in the seed. This means that a significant subset of possible seeds will produce poor quality randomness, requiring each seed to be tested before being approved for cryptographic use.

Permutations, on the other hand, turned out to be a very good idea. The biggest issue with permutations is finding high quality ones. Once found, however, applying them is simple, and they can dramatically improve statistical quality.

PCG includes another weakness not mentioned above or in the paper. This is the fact that the distance of rotations and shifts in the rotate and xorshift permutations is determined by bits from the data being mutated. The paper appears to claim that the permutations increase randomness, however using some of the data being mutated to determine how it is mutated does not actually increase overall randomness. It merely increases the appearance of randomness. A better solution might be to use a 128 bit PRNG as the core, use 64 bits of the state for the output, and then use some of the other 64 bits for permutation parameters. This would avoid the double dipping used in PCG, cramming a bit more randomness into the output. Doing this also substantially increases the period of the generator, avoiding another problem mentioned in the PCG paper[2].

There is also significant room for additional permutations, even within the xorshift and rotate families. There may even be room for sufficient improvement of the multiply permutation to make it viable again. We only tried a homogeneous partitioning strategy, with partitions that were all the same size. This allows for a significant amount of bit shuffling that the un-partitioned versions could not do, but it is still lacking in inter-partition movement. For example, it is possible with a 32-32 bit partitioned rotate to move the upper and lower bits around, but it is not possible to move bits between the upper and lower half except with the 64-bit un-partitioned version. The un-partitioned version will only move around the bits on the ends. A 16-32-16 heterogeneously partitioned rotate, however, would allow the upper 16 bits of the lower 32 bits to be shuffled with the 16 lower bits of the upper 32 bits. In essence, this allows the shuffling of the center, not just the outside edges. With 64 bits of state and a minimum sized rotate operation of 8 bits, we can actually get almost 50 different partitioning strategies, by adding heterogeneous partitioning. The same can be applied to the xorshift permutation, for almost 100 permutations. Applying this same strategy to the multiply and shift permutations could even produce some versions of those that are statistically good.

Of course, with 100 to 200 total permutations, hand picking and testing combinations loses feasibility, but this does provide room for an additional strategy. Using a 128-bit generator, some of the bits not used in the output could be used to select permutations, making the output even less predictable and adding a definite many-to-one function to satisfy the second requirement for cryptographic security.

Overall, we learned a lot from this design. The generator produced seems to be quite strong and satisfy the requirements of a CSPRNG, but there is plenty of room for improvement.

## VIII. Future Work

The next step involves taking everything we learned from this study and applying it to another novel innovation. The next generator will be the Interleaved Permutation Generator (IPG).

The IPG will start with known good quality base generators, instead of LCG. Five such generators have

[2] As a PRNG reaches around 75% of its period, its statistical goodness starts breaking down rapidly, because it begins to "run out" of some values.

been selected and tested so far. The first is a variation on LCG suggested by Sebastiano Vigna in his discussion on the weaknesses of PCG. The others are variations on the xoshiro and xoroshiro generators. All generators are 128 bit. This will provide much longer periods and more bits that can be used for permutation selection, permutation parameters, and possibly even PRNG selection.

The IPG will use a much larger variety of permutations, mostly drawn from the heterogeneously partitioned versions of the xorshift and rotate permutations. It may also include some heterogeneously partitioned multiply and shift permutations, if good ones can be found.

Finally, the IPG will interleave the outputs of the five generators. This will massively increase the period of the final generator. We hope that this strategy will allow the strengths of each generator to make up for the weaknesses of the others. A generator selection strategy will also be tested, which will dynamically select a generator for each state element each iteration, provably guaranteeing the state mutation function to be many-to-one and thus irreversible.

## ACKNOWLEDGEMENT

We gratefully acknowledge the support of CipherLoc Corp. under grant number UA2881.

## REFERENCES

[1] C.E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, v. 28, pp. 656 – 715, 1949.

[2] H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, v. 228, no. 5, pp. 15 – 20, 1973.

[3] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd Edition, John Wiley and Sons:Hoboken, NJ., 1996.

[4] P. Garret, *The Mathematics of Coding Theory*, Pearson/Prentice Hall:Upper Saddle River, NJ., 2004.

[5] M. Matsumoto and M. Saito, "SIMD-oriented Fast Mersene Twister: a 128 bit Pseudorandom Number Generator," Monte Carlo and Quasi-Monte Carlo Methods, 2008.

[6] M.E. O'Neill, "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation," HMC-CS-2014-0905, Harvey Mudd College, Sept. 2014

[7] http://PCG.di.unimi.it/PCG.php

[8] F.B. Brown, Y. Nagaya, "THE MCNP5 RANDOM NUMBER GENERATOR," American Nuclear Society 2002 Winter Meeting November 17-21, 2002 Washington, DC.

[9] S. Vigna, "Mirai botnet attack hits thousands of home routers, throwing users offline," ZDNet, Nov. 29, 2016.

[10] http://simul.iro.umontreal.ca/testu01/tu01.html

[11] https://webhome.phy.duke.edu/~rgb/General/dieharder.php

[12] G.S. Marsaglia, "The DIEHARD Battery of Tests of Randomness," https://web.archive.org/web/20160125103112/

[13] http://stat.fsu.edu/pub/diehard/