B. Williams, A. Carlson and R. Hiromoto. (Oct. 03-05, 2022). Novel Innovations for Improving the Quality of Weak PRNGs, *2022 13th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, Kharagpur, India, 2022, pp. 1-6, doi: https://doi.org/10.1109/ICCCNT54827.2022.9984611]

Novel Innovations for Improving the Quality of Weak PRNGs

Benjamin Williams
Computer Science
University of Idaho
Idaho Falls, U.S.
will9847@vandals.uidaho.edu

Albert Carlson
CSIT
Austin Community College
Austin, U.S.
albert.carlson@austincc.edu

Robert Hiromoto
Computer Science
University of Idaho
Idaho Falls, U.S.
hiromoto@uidaho.edu

Abstract—Pseudo-random Number Generators are used in an enormous number of computer applications, from video games and wireless network collision avoidance protocols to cryptographic security and Monte Carlo simulations. For most applications, certain requirements regarding speed and statistical quality must be met. This is typically a trade-off. Some modern generators add additional steps to fast, weak generators, to improve statistical quality. The improvement comes at the cost of speed but allows developers to compromise for an acceptable balance of speed and quality. This paper presents novel techniques that can be used to improve the statistical quality of weak PRNGs and examines how existing techniques can be generalized so they can be applied to any PRNG.

Index Terms—Randomness, Random Number Generators, Pseudorandom Number Generators, LCGs, MCGs, PCGs, Geffe Generators, Polymorphic RNGs, Composite Generators

I. Introduction

In Pseudo Random Number Generators (PRNGs), the critical factors are speed and statistical quality. Even in Monte Carlo simulations, where weak Linear Congruential Generators (LCGs) are often used for speed, quality is starting to be viewed as more important [1]. This is often a trade-off. For example, there is a mathematical proof showing the Blum Blum Shub PRNG[1] is cryptographically secure, but the generator itself is far too slow to be used in practical applications [2]. The fastest known PRNGs include LCGs, but these have very low statistical quality [3].

O'Neill's Permutation Congruential Generator (PCG) concept proves that fast PRNGs can be improved by adding mutations to the output [3], but the improvement is minor. This study takes the same starting point, fast LCGs, and explores a broader range of options for improving quality.

Another factor that becoming more important is memory usage. In the Internet of Things (IoT) devices, this is a critical factor. The Mersenne Twister algorithm uses over 2.4KB of memory [4], which is trivial even on mobile devices, but on microcontrollers used in IoT devices, like the Atmel SAM D family, which have 4KB to 32KB of Random Access Memory (RAM), this is very significant. The generator described here stores only 16 bytes of state.

[1] $x_{n+1} = x_n^2 \ mod \ M$, where M is the product of two large primes. The output is one or more of the least significant bits of x_{n+1}, and the initial seed must be co-prime to M but not 1 or 0.

II. Related Work

There is not a lot of recent work in terms of improvements to PRNGs. Researchers prefer to develop entire new PRNGs rather than finding discrete improvements that can be used to increase the quality of existing PRNGs or to construct new PRNGs in a modular manner.

Melissa O'Neill's PCG is one such PRNG. It combines LCGs with output permutations to produce a fast PRNG that is higher quality than LCGs on their own [3]. PCGs still suffer from some of the same weaknesses as LCGs [5]. While PCGs are better than LCGs on their own, there is little research on the benefits of individual output permutations. O'Neill does not do rigorous testing on each individual permutation. Even where a single innovation is added to an existing PRNG to create a new PRNG, rigorous testing is typically not done to quantify the impact of the innovation. Similarly, composite PRNGs like Mersenne Twister, which combine innovations from existing PRNGs in novel ways, do not quantify the impact of the individual innovations on the outcome.

This paper does not aim to present a new PRNG based on unquantified elements but rather presents a selection of innovations that produced significant improvements when applied to an existing low quality PRNG. The goal of this paper is to gain an understanding of the contribution of each innovative element. This would allow the construction of new PRNGs in a more systematic manner, building on excepted engineering practices.

III. Testing Methodology

The statistical quality of PRNGs is tested using the Dieharder test suite [6]. Popular randomness testing suites include Diehard [7], TestU01 [8], and NIST's test suite [9]. However, these test suites are all outdated and unmaintained. Dieharder is a suite of tests that is still actively maintained and contains most statistical tests that have been demonstrated to be of acceptable quality. Dieharder includes many tests from the previously mentioned test suites, as well as newer applicable test algorithms, making it the most comprehensive test suite presently available. The most recent release of Dieharder is dated from early 2020, and new tests are regularly added as they become available.

Dieharder uses a chi-squared testing method [10] to determine the probability of the given pseudo-random stream being produced by a source of truly random values. Probabilities that are too high or too low are predictable and indicate poor quality. Due to the probabilistic nature of this testing, even an ideal PRNG is expected to fail approximately 1 test in 1,000 and report a weak result on approximately 1 test in 100. Statistical testing in general has some inherent weaknesses. One weakness is that the tests themselves cannot be proven to be perfect. A failure indicates that either the PRNG or the test itself is weak. With one exception (the Diehard Sums test [6]), retained for its historical significance, Dieharder avoids adding tests with known weaknesses. The other weakness is that while failing a test indicates a weakness in the PRNG with some degree of confidence, passing a test only indicates that the patterns that test is designed to look for are not present. Passing even 100% of Dieharder's tests does not prove that the PRNG under test does not have patterns that might easily be detected by some test that has not yet been discovered or added.

Despite these weaknesses, Dieharder is the most comprehensive PRNG test suite available and thus is the best option available. If other potential vulnerabilities exist in PRNGs, they are not currently well known, otherwise, tests would be developed to detect them. Those tests would then be added to Dieharder.

IV. Composite Generators

Combining multiple weak generators can often help to improve statistical quality. Multiple instances of the same generator are unlikely to yield any benefits. Using multiple different generators of the same class can be effective, and using generators from different classes can be even more effective. This practice dilutes statistical weaknesses, by combining the strengths and weaknesses of different generators. There are several known aggregation strategies that can be used to make composite PRNGs.

An advantage of this strategy that none of the other PRNGs presented here have is that it increases the potential entropy in the system. This is largely dependent on how the PRNGs are seeded. For example, if a seed of sufficient length to give each generator a unique, non-overlapping seed is used, greater entropy is available than if a shorter seed is reused[2]. Increased entropy provided by longer seeds increases both statistical quality and security [11].

[2]In testing, a short seed was used, rotated a different amount for each PRNG.

A. Geffe Generators

One example of a composite generator is the Geffe generator, which uses interleaving for PRNG aggregation [12]. A Geffe generator rotates through individual generators, either taking a series of values, one at a time or randomly selecting a generator each time a value is requested. In either case, the generators produce a limited number of values before being replaced. However, there is a slight speed penalty in exchange for increased quality.

The mechanics of Geffe Generators periodically replacing the underlying generators is another potential source of entropy. This mechanism is known as the time to live (TTL) process [13]. This regular injection of entropy can significantly improve resilience, by making the generator less vulnerable to state compromise [14].

B. Output Aggregation

Instead of rotating through generators, multiple generators can be run in parallel, and their output can be aggregated. This practice is significantly more expensive than Geffe generators, because each iteration requires all generators in the pool to iterate. In exchange for this overhead, the potential improvement in statistical quality is significantly higher. For aggregation, the outputs can be added, subtracted, or XORed. The outputs can also be aligned, or they can be offset using rotation. This allows the strengths and weaknesses of different generators to directly counter each other.

C. Partial Output Aggregation

Rather than aggregating the full output of each generator, it is sometimes preferable to aggregate only partial output. For example, it is well known that lower bits of an LCG generator are of lower statistical quality than higher bits [3]. It is therefore possible to use only the higher bits of several LCGs to produce an aggregate output of significantly higher statistical quality than could be obtained from just one LCG or from several with all of their output being aggregated. Additionally, when using partial output aggregation, the bits that are not used in the output serve to protect the security of the generator.

The aggregation strategy when using partial output aggregation depends heavily on how many generators are being used and what portion of their output is being used. This strategy was independently tested by concatenating the top 8 bits of four LCGs into a single 32 bit output. When concatenation will not work, rotations and XORs or other strategies can be used to fit the data to the output space. To maximize statistical quality, the partial outputs should be evenly distributed across the output space. Figure 1 demonstrates XORing successive rotations of 16 bit partial outputs from four generators, to produce a single 32 bit final output. The GNU C code for this is provided in Figure 2.

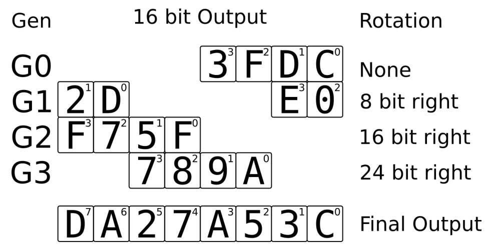

Fig. 1. 16-bit XOR with Successive Rotation

D. Bit-mixed Aggregation

Simple aggregation techniques often allow clear patterns of the component PRNGs to come through. Quality can be improved further by rearranging the bits during aggregation. By using bit-wise shifts, AND masks, and OR functions, this can be done efficiently and with low overhead. This technique also creates the potential to randomly select which bits are included in the output. However, the practice can come with a penalty to both speed and quality. Ideally, each output bit should only influence the aggregate output exactly one time, and an equal number of bits should be used from each generator in order to avoid statistical imbalances that could compromise the security of the PRNG.

The bit-mixing in Figure 3 uses the top 8 bits from each of four generators. Mixing is done using a binary value with four set bits and four clear bits with its inverse as masks, to guarantee each bit is used exactly once and that no generator contributes more bits than any other. The output is a 32 bit value, composed of the top 8 bits of the four generators mixed together in a way that makes it very difficult to reconstruct any part of the generator states without knowing both the mask values and how they are applied to each generator. The constants used here are 83 (a prime number with an equal number of set and clear bits) and 172 (the bit-wise inverse of 83), but any constant/inverse pair with four set bits and four clear bits should work.

V. Output Permutations

Output permutations, commonly used to improve the statistical quality of weak generators, are the basis of O'Neill's PCG [3]. PCGs are a class of generators that apply output permutations to a simple LCG. While most of the proposed output permutations consistently failed in testing with Dieharder, the rotate and xorshift permutations proved quite valuable in improving statistical quality [15]. Applying output permutations in series allows for fine-tuning the balance of speed and quality of a generator.

A. Partitioned Output Permutations

In Crypt1, Williams et al applied O'Neill's rotate and xorshift permutations to separate subsections of the output independently [15]. By splitting a 64 bit portion of the state into 32, 16, or 8 bit partitions and applying a permutation to each partition independently, movements of bits become possible that would have been far more difficult and expensive to achieve otherwise. Figure 4 illustrates the application of two heterogeneously partitioned rotate permutations executed in series, color coded for convenience[3]. In the resulting value, previously adjacent bits are completely separated, which no amount of rotates across the entire value could have done.

B. Heterogeneously Partitioned Output Permutations

A further innovation found in Crypt1 is heterogeneously partitioned output permutations [15]. This splits a single output value into partitions of different sizes, which allows for partitions to cross data type alignment boundaries that homogeneous partitions could not. For example, a 32 bit value might be partitioned into 8 bits for the top, 16 bits for the middle, and 8 bits for the bottom, allowing the middle partition to cross the center. Figure 5 shows how exchanging one of the homogeneous partitions for a heterogeneous permutation affects the results. Notice the middle two sections each have three colors. The 4-8-4 permutation allowed two bits (I and D) to cross the center boundary. For larger data types, more complex permutations can be used for more mixing. The best results are achieved with at least four permutations in series, mixing permutation types[4].

VI. Permutation Backfeeding

The final element of Crypt1, which produced a major increase in statistical quality, with almost no cost to speed, was permutation backfeeding [15]. Normally, output permutations are applied only to the output value [3]. In Crypt1, however, the result was written back to the state of the LCG after permutations were applied. Crypt1 shows that permutation backfeeding can be a very powerful method for improving the statistical quality of a PRNG. The process only costs a single memory write, which makes it very efficient. This procedure elevated Crypt1 from a fairly good PRNG to a cryptographically secure PRNG.

There is one improvement that could be made to Crypt1. The backfed state is also the output. This exposes part of the raw state of the underlying LCG. Crypt1 could be improved by backfeeding only the first four permutations. The final four would obfuscate the state used in the output.

VII. Quad LCG

The quad LCG PRNG was not created as a study of PRNGs. Instead, it was created for a different project where a fast, simple PRNG of moderate quality was temporarily required. LCGs are trivial to implement [3], so they were used as the starting point for the required

[3]Note that on real world CPUs, 4 bit rotate permutations would be quite expensive and are only used here due to space constraints. It might be helpful to imagine each letter as two bits, and the permutations as 16-16 rotated 6 bits and 8-8-8-8 rotated 4 bits.

[4]Crypt1 uses eight heterogeneous permutations [15].

```
// gen0-gen3 are of type uint32_t and preseeded.
// __rord(uint32_t, int) is the GNU C 32 bit, right rotate intrinsic for x86.

uint32_t result =        (gen0 & 0xFF)      ^ __rord(gen1 & 0xFF00,  8) |
                  __rord(gen3 & 0xFF, 24) ^        (gen0 & 0xFF00)      |
                  __rord(gen2 & 0xFF, 16) ^ __rord(gen3 & 0xFF00, 24) |
                  __rord(gen1 & 0xFF,  8) ^ __rord(gen2 & 0xFF00, 16);
```

Fig. 2. C example of 16-bit XOR with Successive Rotation

```
// states[] is a uint32_t array of generator states

uint32_t rand =
    (states[0] >>  24 & 0b01010011) << 16 | (states[0] >> 24 & 0b10101100) << 0 |
    (states[1] >>  24 & 0b10101100) << 16 | (states[1] >> 24 & 0b01010011) << 8 |
    (states[2] >>  24 & 0b01010011) << 24 | (states[2] >> 24 & 0b10101100) << 8 |
    (states[3] >>  24 & 0b10101100) << 24 | (states[3] >> 24 & 0b01010011) << 0;
```

Fig. 3. C example of Bit-mixed Aggregation

0bABCD:EFGH:IJKL:MNOP	Initial Value
0bFGHA:BCDE:NOPI:JKLM	Rotate right by 3 partitioned 8-8
0bHAFG:DEBC:PINO:LMJK	Rotate right by 2 partitioned 4-4-4-4
0bHAFG:DEBC:PINO:LMJK	Final Value

Fig. 4. 16-bit Homogeneously Partitioned Rotates

0bABCD:EFGH:IJKL:MNOP	Initial Value
0bFGHA:BCDE:NOPI:JKLM	Rotate right by 3 partitioned 8-8
0bHAFG:PIBC:DENO:LMJK	Rotate right by 2 partitioned 4-8-4
0bHAFG:PIBC:DENO:LMJK	Final Value

Fig. 5. 16-bit Heterogeneously Partitioned Rotates

PRNG. Quality improvement was necessary, because LCGs are known to be extremely weak. Since that the higher order bits of LCGs are stronger, the top 8 bits of four parallel LCGs were used to provide the necessary 32 bits. Concatenating the four 8 bit values into a single 32 bit value was the obvious aggregation strategy. However, this could allow an attacker to attack each LCG independently. The solution to this problem was bit-mixing to obfuscate the source generator of each bit. An xorshift permutation was applied to the final output to prevent raw state from being exposed in the output. Upon testing to discover whether some statistical weaknesses in the output of the main project were due to the PRNG or the project itself, it was determined that this PRNG has far higher statistical quality than expected. This prompted a study on this type of PRNG.

A. Implementation

The PRNG produced uses four LCGs, each with unique constants obtained from a list of known good constants [15]. Each iteration separately advances each LCG one step. Since the statistical quality of the bits produced by an LCG is higher for more significant bits, only the 8 most significant bits of each LCG were used to produce the output. This also has the benefit of keeping 75% of the LCGs' states private, making the PRNG more resistant to statistical attacks on its output. Thus, this is known as a 128/32 PRNG.

Aggregation is accomplished through bit-mixing. The strategy for this practice employs shifts and masks to use half of the bits of each PRNG in one part of the output and half in another. This must be done carefully, using each bit exactly and only once. During testing it was found that a failure to follow this rule caused much lower statistical quality, likely due to a strong correlation between reused bits. Before this bug was corrected, the PRNG was failing 34 of the 114 tests run by the current version of Dieharder. After correction, the PRNG did not fail any of the tests, and it only produced weak results on one test. The code used for the aggregation is shown in figure 3[5]. After bit-

[5]The patterns used are 83 and 172. 83 was selected from a list of 8 bit prime numbers with an equal number of 0s and 1s. Any 8 bit value with four 1s and four 0s would probably work.

mixing, an xorshift permutation with a partition scheme 8-16-8 was applied, as a final step. No backfeeding was used in this PRNG, again, for simplicity.

B. Testing Results

The testing data cited for bit-mixing aggregation was generated using a static seed, with a single run of Dieharder's battery of tests. This single test is not sufficiently rigorous, thus additional testing was done using random seeds obtained from /dev/urandom (a non-blocking source of randomness generated by the Linux kernel, derived from any source of randomness provided by the CPU or system). Each set of tests was executed 10 times (each time with a new random seed), to produce a statistically significant sample size. Recall that even given a truly random input, Dieharder tests are expected to return weak results up to 1% of the time and failures up to 0.1% of the time. Weak and failed tests can indicate weaknesses in the tests themselves, rather than the PRNG under test. This test data is presented in Table I.

where

- "LCG" is just one LCG, without any special techniques to improve the output. The results are provided here as a baseline, and they are consistent with what is known about the quality of LCGs.
- "Parallel" uses four LCGs and aggregates the 8 most significant bits of each using concatenation. The improvement is striking, with no failed tests and a decreased number of weak tests.
- "Bit-Mixed" uses four parallel LCGs and aggregates using the bit-mixing strategy. This improvement here is much less pronounced, though this may be because there is not much room for improvement. There are 48% fewer weak results than "Parallel" though, which is still quite significant.
- "Full" is the full generator, with four parallel LCGs, partial output aggregation of the top 8 bits of each generator through bit-mixing, followed by an 8-16-8 xorshift output permutation. The improvement does not appear to be very big, but it does comes out to 5% fewer weak tests than "Bit-Mixed". The rate of weak tests is only 70% higher than should be expected from true randomness, which is quite good for a PRNG.

In addition to providing statistical testing, Dieharder also provides speed data on the PRNG under test. Included in Table I is the average speed of each PRNG version across the 10 tests. It might seem incongruous that the parallel LCGs are running 15% faster than a single LCG, given that it is doing four times the work plus some additional aggregation with every iteration. There is a drop in speed after adding bit-mixed aggregation, but adding the xorshift permutation appears to increase speed. This speed data merely indicates that the overhead is so low that unpredictable variations in OS scheduling and other system processes make a bigger difference to performance than any of the additional elements. Even the slowest speed achieved in testing is extremely fast for a PRNG of this quality. To accurately measure performance, it would be necessary to either use profiling software or compile the code into assembly and count instructions. This is currently unnecessary, as even the slowest average speed achieved is more than sufficient for nearly all existing PRNG applications.

A weakness of this particular approach is the lack of independent testing for each element. Bit-mixing was not tested on a single LCG, nor were bit-mixing and the output permutation tested independently of the parallelism. The data indicates that each additional element provided significant improvement upon the previous elements. The magnitude of the benefits is not clear. It appears that the parallel, partial output aggregation provided the greatest benefit, but without testing the other elements independently, this cannot be guaranteed.

The limits of Dieharder's default testing parameters were reached with parallel, partial output aggregation. It is clear that the techniques used in this generator have a very powerful ability to improve the statistical quality of weak PRNGs, but the full magnitude is unclear. The final testing for Crypt1 used Dieharder's "test to destruction" mode, and then its results were compared with results from AES_OFB, a known high quality PRNG [15]. These short test runs using default settings really only provide a very rough quality metric. In short, while the above Dieharder results indicate that this PRNG is significantly higher quality than most other PRNGs, the test data is insufficient to compare this PRNG with other higher quality PRNGs. Additional testing is necessary, using Dieharder's "test to destruction" mode, to provide sufficiently high resolution data to make such comparisons[6].

VIII. Conclusion

The relative ease with which a high quality PRNG can be created, using a low quality PRNG and a small number of additional elements is surprising. This suggests, and the evidence confirms, that it is not difficult to produce very fast PRNGs that can compete in quality with popular PRNGs that are high quality but relatively slow. There is significant room for more research, but the set of elements presented here provide some powerful options for building custom PRNGs that can provide both high speed and quality.

The number of innovations tested here is small. This does not consider all of the past innovations currently in use, which need testing and cataloging to build a more comprehensive catalog of elements. Perhaps the biggest limitation is one of testing randomness in general. While Dieharder is an excellent statistical testing suite for PRNGs, it suffers from the same problem as any kind of statistical randomness testing. This means that as new

[6]This testing was not completed for this publication, because it takes several months on high performance hardware to complete.

TABLE I
Diehard Test Results

PRNG Version	PASSED	WEAK	FAILED	WEAK%	FAILED%	Speed
LCG	562	50	528	4.4%	46.3%	6.690e+07/s
Parallel	1101	39	0	3.4%	0%	7.725e+07/s
Bit-Mixed	1120	20	0	1.8%	0%	5.898e+07/s
Full	1121	19	0	1.7%	0%	6.366e+07/s

Aggregate results, 10 runs per PRNG version

tests are developed, any compendium of effective techniques for improving pseudo-randomness will need to be tested against those new tests to eliminate techniques that have undiscovered flaws. This will be an ongoing effort, unless some revolutionary new technique is discovered that can comprehensively prove the quality of PRNGs.

The incremental contributions listed in this paper (generator composition, partial output aggregation, bit-mixing aggregation, output permutations, permutation backfeeding, partitioning modes) have proven to increase the statistical quality of simply engineered PRNGs. These additions demonstrate that PRNGs can be made much stronger using simple techniques with relatively low overhead. This suggests that complex strategies with high overhead are unnecessary in improving PRNGs.

IX. Future Work

There is significant room for more research in this domain. More techniques for improving the statistical quality of weak PRNGs undoubtedly exist and are merely waiting to be discovered. There is also still room for more testing, both of the individual elements presented and in the generator presented. There is also some potential here for the creation of modular PRNG systems.

There are three places where there is significant room for improvement in testing. One is in the individual elements presented. Some elements, especially those which were discovered in previous research, already have some testing, however often without the rigor provided by Dieharder. The new elements presented here have limited testing with Dieharder and would benefit from more rigorous testing, independent of other elements.

The second area where there is more room for testing is the complete PRNG presented here. Using Dieharder, it has undergone more testing than most PRNGs, however, it has not been tested to the standards established with Crypt1 to ensure cryptographic statistical quality. For this, it will be necessary to test this PRNG using Dieharder's "test to destruction" mode, the result of which must then be compared to those of a known high quality PRNG, like AES_OFB.

The final place where testing can be improved is the development of a standard testing methodology using Dieharder. Work on this is already underway. The methodology used in this research is one step in that process.

There is also significant potential here for breaking down other elements commonly used in PRNGs, to create a larger set of elements that can be used to dynamically generate new PRNGs. Modular PRNGs like this could have huge implications for security, especially in the context of Polymorphic RNGs used in Polymorphic encryption as well as Geffe generators.

References

[1] P. L'Ecuyer, "Random numbers for simulation," Communications of the ACM, pp. 85 – 98, 1990.

[2] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," SIAM Journal on Computing, pp. 364–383, May 1986.

[3] M. E. O'Neill, "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation," Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, 9 2014.

[4] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimendsionally equidistributed uniform pseudorandom number generator," ACM Transactions on Modelling and Comutpute Simulation, vol. 8, no. 1, pp. 3 – 30, 1998.

[5] S. Vigna, "The wrap-up on pcg generators." https://pcg.di.unimi.it/pcg.php, Accessed 20 June 2022.

[6] R. G. Brown, D. Eddelbuettel, and D. Bauer, "Robert g. brown's general tools page." https://webhome.phy.duke.edu/~rgb/General/dieharder.php, 2022.

[7] M. M. Alani, "Testing randomness in ciphertext of block-ciphers using diehard tests," IJCSNS International Journal of Computer Science and Network Security, vol. 10, no. 4, 2010.

[8] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," ACM Transactions on Mathematical Software, vol. 33, no. 22, pp. 1 – 40, 2007.

[9] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," Tech. Rep. 800-22, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, 4 2010.

[10] R. L. Ott and M. T. Longnecker, An Introduction to Statistical Methods and Data Analysis 7_{th} edition. Cengage Learning, 2016.

[11] C. Shannon, "Communication theory of secrecy systems," Bell System Technical Journal, vol. 28, pp. 656 – 715, 1949.

[12] P. Geffe, "How to protect data with ciphers that are really hard to break," Electronics, pp. 46, 99–100, 1973.

[13] A. Carlson, Set Theoretic Estimation Applied to the Information Content of Ciphers and Decryption. PhD thesis, University of Idaho, 2012.

[14] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs, "Security analysis of pseudo-random number generators with input: /dev/random is not robust," in Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, (New York, NY, USA), p. 647â 658, Association for Computing Machinery, 2013.

[15] B. Williams, R. Hiromoto, and A. Carlson, "A design for a cryptographically secure pseudo random number generator," pp. 864–869, 09 2019.