Breaking Another Quasigroup-Based Cryptographic Scheme

Markus Dichtl, Pascale Böffgen

Siemens Corporate Technology

Abstract. In their paper "A Quasigroup Based Random Number Generator for Resource Constrained Environments", the authors Matthew Battey and Abhishek Parakh propose the pseudo random number generator LOQG PRNG 256. We show several highly efficient attacks on LOQG PRNG 256.

Keywords: quasigroup, pseudo random number generator, stream cipher

1 Introduction

In their paper "A Quasigroup Based Random Number Generator for Resource Constrained Environments" [BP12], the authors Matthew Battey and Abhishek Parakh propose the pseudo random number generator LOQG PRNG 256. They suggest to use LOQG PRNG 256 as a stream cipher by XORing the output of the generator with the data to be encrypted.

As one of the authors of this paper here once found a quasigroup based algorithm for post-processing biased random numbers ([MGK05]) very easy to attack ([Dic07a]), he felt that LOQG PRNG 256 could be easy prey for an attack as well. His hope was not in vain.

However, [MGK05] was by no means the only quasigroup based broken cryptographic algorithm. Another example is [DM03], for which [BP12] gives very bad statistical test results. The result from [BP12] about the compressability of the "random" bits generated by the generator from [DM03] is very impressive: The output can be compressed by a factor of about 1000; so, there seems to be very little (pseudo)entropy in these sequences.

A broken quasigroup based algorithm which received considerable attention in the cryptologic community was EDON-R ([GØM+08]). EDON-R was submitted to the SHA-3 contest. [Kli08] and [KNW08] demonstrate that EDON-R has severe shortcomings.

2 Quasigroups

It will turn out that LOQG PRNG 256 does not use the general concept of a quasigroup, so we could skip the definition of a quasigroup, but we give it anyway:

A *quasigroup* is a set Q with a binary operation $* : Q \times Q \to Q$ such that for all values a and b of Q, each of the two equations $a * x = b$ and $y * a = b$ has a unique solution. In other words, the function table of $*$ forms a Latin square, that is each element of Q is contained exactly once in each row and column.

3 The Simplified Quasigroup

In order to avoid the storage requirements of general quasigroups, the authors of LOQG PRNG 256 suggest the following construction: Instead of explicitly storing a function table, they apply addition modulo 256 as binary operation. Indeed, addition modulo 256 forms a group, and hence a quasigroup. However, in LOQG PRNG 256 addition modulo 256 is not used directly, but in a disguised form: Before adding two elements a and b, their values are used as indices into two tables r and c, and the elements found there are added modulo 256. Note that the indices of the arrays run from 0 to 255. The result of $a * b$ is $(r[a] + c[b]) \bmod 256$. The tables r and c each contain a permutation of the integers 0 through 255. To make things seem even more complicated, these permutations are not constant, but vary with time.

4 The Pseudo Random Number Generator to Be Attacked

The pseudo random number generator LOQG PRNG 256 specified in [BP12] works as follows:

In the initialisation phase, the tables r and c are filled with random permutations of the integers 0 through 255.

The values s_1 and s_2 are initialised to random values modulo 256.

The value i is initialised to zero.

The main loop of the random number generator is given in pseudocode:

```
while(true)
   {
   rand = (r[s_1]+c[s_2]) mod 256
   s_1 = s_2
   s_2 = rand
   aux = c[i]
   c[i] = c[s_1]
   c[s_1] = aux     //shuffle c[i] and c[s_1]
   aux = r[i]
   r[i] = r[s_2]
   r[s_2] = aux     //shuffle r[i] and r[s_2]
   i=(i+1) mod 256
   output(rand)
   }
```

5 Some Observations Leading to an Attack

We observe immediately that no general quasigroup operation appears in the pseudocode, but only an addition modulo 256 (in the first line of the loop of the pseudocode), whose summands are determined from the variables s_1 and s_2 by table lookup.

When we look closer at s_1 and s_2, we note that their values can be easily determined by an attacker. The new value s_2 is set to in the loop, is the easiest; it is identical to the value of rand, which is output at the end of the loop. The new value s_1 is set to, is not much more difficult; it is identical to the value s_2 is set to in the previous execution of the loop, or equivalently, to the rand output of the previous loop.

Since the value of the index i is initialised to a known value and increased by one in each round, its value is also known to the attacker.

So, starting in the second execution of the loop, the attacker knows exactly in which way the algorithm manipulates its lookup tables r and c. In addition, in each execution she learns the modulo 256 sum of two elements of the tables. In the second execution of the loop, the attacker does not know yet from which position s_2 the index into the table c was chosen, as this value was determined in the first execution of the loop from the unknown initial value of s_1. However, starting with the third execution of the loop, the resulting values of s_1 and s_2 in the previous execution of the loop, and hence, the indices the summands come from are known to the attacker.

6 The Attack

Now, we want to combine our observations in order to attack.

We only start our attack in the third execution of the loop, but we take into account the rand values observed in the first two executions.

At this point, we know the values of s_1 and s_2, as they are the values of the rand output in the first two rounds. The value of i is 2, as it started with a value of 0 and was incremented twice in the first two executions of the loop.

We define the variables C_0, C_1, ... , C_{255} as the content of the table c at the beginning of the third execution of the loop. We define the table C as the table of the formal variables C_i corresponding to the content of the table c, and more specifically $C[i]$ as the formal variable corresponding to the content of $c[i]$ for all indices 0 through 255. In a completely analogous way, we also define the variables R_0, R_1, ... , R_{255} and the table R as the formal variables for the content of the table r.

At the start of the attack, i.e. at the beginning of the third execution of the loop of the algorithm, we initialise all table entries $C[i]$ with the formal variables C_i and all table entries $R[i]$ with the formal variables R_i.

Next, we evaluate the computation of *rand*. Here, we exploit for the first time that we learn the sum of two formal variables, namely that the equation $R[s_1]+C[s_2] \equiv rand \bmod 256$ holds. *rand* is the rand output at the end of the loop. We obtain a linear equation of the form $C_i+R_j \equiv k \bmod 256$ with concrete numbers i, j and k.

Then, we take into account the new assignments of s_1 and s_2. We set the new value of s_1 to the old value of s_2 and the new value of s_2 to the rand output of the current loop execution.

The next step in the attack is to take into account the shuffling of $c[i]$ and $c[s_1]$. We do this by exchanging the contents of $C[i]$ and $C[s_1]$.

Similarly, we take into account the shuffling of $r[i]$ and $r[s_2]$ by exchanging the contents of $R[i]$ and $R[s_2]$.

The value of i is updated by incrementing it.

Now, we iterate the steps described above, until we have reached a number of n equations. We will discuss the required number n subsequently in more detail. Obviously, $n \geq 512$ must hold, as we have to determine the values of 512 unknowns. For the moment, we suggest to choose n as 1976. Subsequent statistics will show that this is sufficient in 99 % of all attacks tried (see last paragraph of 7).

Now, it seems to be completely trivial to derive the values of the 512 variables C_0, C_1, ... , C_{255} and R_0, R_1, ... , R_{255} from the n linear equations, but we still have to overcome a tiny problem: We will never be able to find out the values of these variables! The good news is, however, that we do not have to. All equations we can derive from the output of the pseudo random number generator are of the form $C_i + R_j \equiv k \bmod 256$. Even if we know the sum k for all possible values of i and j, there is no unique solution for such a system of linear equations. Let us assume that we have found a solution of the system. Then, obviously, we obtain another solution if we increase all C_i values by a fixed integer a and decrease all R_i values by a. We solve this problem by arbitrarily fixing R_0 to zero. It turns out that then all remaining variables are uniquely determined. But how does our arbitrary choice of R_0 affect our ability to predict further output of the pseudo random number generator? Not at all, as all the sums of the form $C_i + R_j$ have the correct and unique value.

7 Attacking Even Sooner

Now, one may wonder whether we really have to wait until we can solve the system of 512 linear equations completely in order to successfully predict further output. In most cases, it is possible to predict output bytes much earlier. The individual steps of the attack we will describe now are as given above. For our improved attack, we must analyse in detail which other sums of the form $C_i + R_j$ we can derive from the ones we already know.

We define a matrix m with 256 rows and columns. Its entry m_{ij} contains our information about the sum $C_i + R_j$. Since we initially have no information at all, the matrix entries are initialised to the symbol *nil*. As we learn equations of the form $C_i + R_j \equiv k \bmod 256$, we update the matrix by setting m_{ij} to k. But occasionally, we can derive new matrix entries from the ones we already know: If we have non-*nil* entries at m_{ij}, m_{il}, and m_{kl} with i, j, k, and l all distinct, we obtain $m_{kj} \equiv C_k + R_j = (C_i + R_j) - (C_i + R_l) + (C_k + R_l) \equiv (m_{ij} - m_{il} + m_{kl}) \bmod 256$. So, if we have *nil* at m_{kj}, we replace it with $(m_{ij} - m_{il} + m_{kl}) \bmod 256$.

Symmetrically, if we have non-*nil* entries at m_{ij}, m_{kj}, and m_{kl} and *nil* at m_{il}, we replace *nil* at m_{il} with $(m_{ij} - m_{kj} + m_{kl}) \bmod 256$.

Geometrically, this means that if we know matrix entries at three corners of a rectangle in the matrix, we can determine the matrix entry at the fourth corner. Here, we only consider rectangles whose sides are parallel to the rows and columns of the matrix.

For any matrix entry we derive by using the "rectangle rule" explained above, we also try whether we can derive further matrix entries from it.

The blue curve (upper curve) in Fig. 1 shows for 1000 samples how may output bytes of the generator we need in order to fill the whole matrix by using the "rectangle rule". The results are in ascending order. The smallest values are 1118, 1128, 1137; the biggest values are 2793, 2836, 2970. The mean is 1714.45; the median is 1666.5.

We can also exploit the Latin square property of the matrix m, that is each entry appears exactly once in each row and column. So, even if we are not able to predict an output of the pseudo random number generator, we can predict that all non-*nil* entries in the corresponding row and column are impossible as the next output of the pseudo random number generator. Occasionally, the entries in a row and column leave only one possibility for the entry at the intersection of the row and the column. Hence, one can use Sudoku-like arguments to derive additional entries of the matrix.

The red curve (lower curve) in Fig. 1 shows for 1000 samples (in ascending order) how many output bytes of LOQG PRNG 256 we need in order to fill the whole matrix by using the "rectangle rule" and the "Sudoku rule" just explained. The smallest values are 971, 994, 1014; the biggest values are 2223, 2245, 2287. The mean is 1417.39; the median is 1391. The value for sample 990 is 1976, i.e. in 99 % of our attacks, 1976 output bytes of LOQG PRNG 256 were sufficient in order to predict all further output (see second to last paragraph of 6).

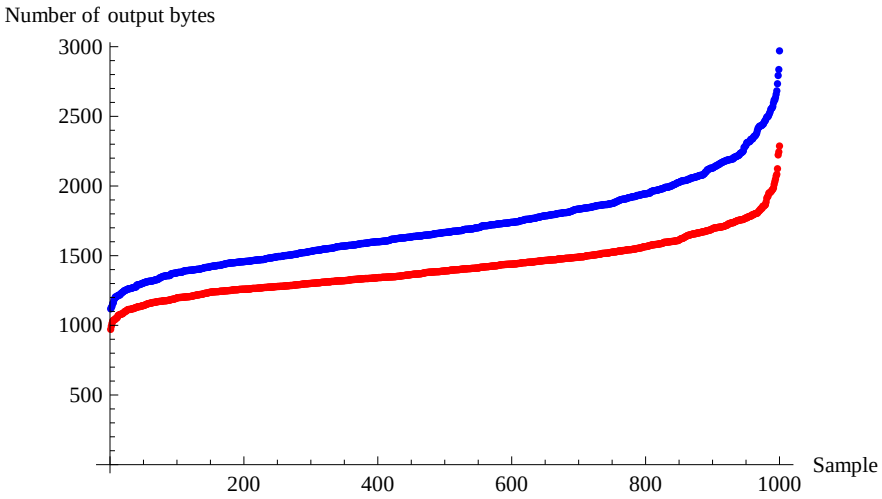

Fig. 1. Number of LOQG PRNG 256 output bytes necessary to predict all future output bytes. Blue curve (upper curve): only "rectangle rule". Red curve (lower curve): both "rectangle rule" and "Sudoku rule". The results have been sorted in ascending order.

8 Two More Attack Statistics

In the previous section, we found out how many output bytes of LOQG PRNG 256 we need in order to predict *all* future output bytes. But it is also interesting to know when we are able to

predict *for the first time* which output byte comes next. Fig. 2 displays for 1000 samples the minimum number of output bytes we need in order to predict the next output byte. The results are again in ascending order. The smallest values are 10, 13, 13. The mean is 221.8; the median is 234.

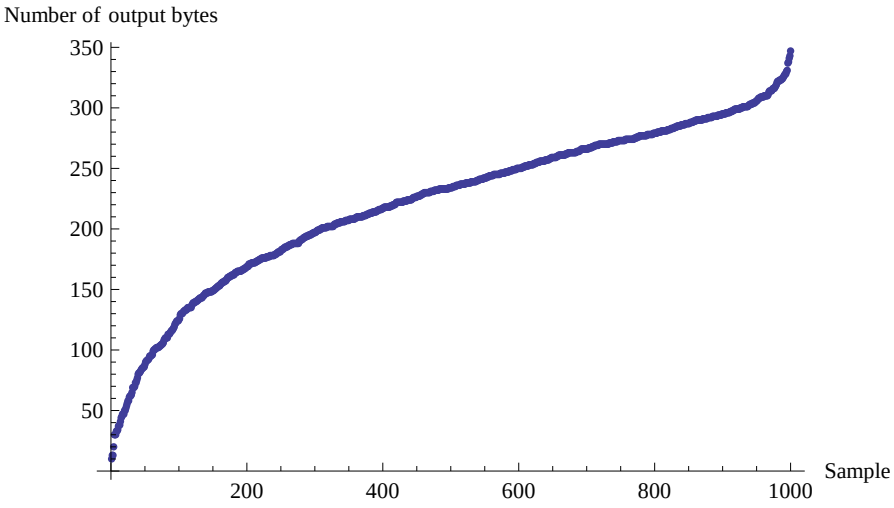

Fig. 2. Minimum number of output bytes necessary to predict the next output byte, based on 1000 trial attacks with random seeds. The results have been sorted in ascending order.

Furthermore, we wanted to know how likely it is at any time to be able to predict the next output byte. Fig. 3 contains these figures for 1000 samples.

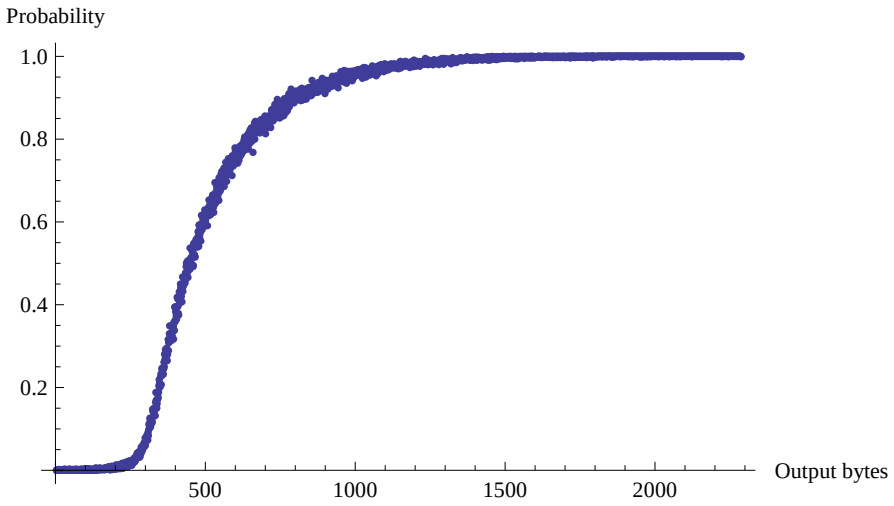

Fig. 3. Probability for being able to predict the next LOQG PRNG 256 output byte as a function of the number of previous LOQG PRNG 256 output bytes.

Some examples:

- After 437 output bytes of the generator, we can predict the next output byte with a probability of 50 %.
- After 777 output bytes of the generator, we can predict the next output byte with a probability of 90 %.
- After 1196 output bytes of the generator, we can predict the next output byte with a probability of 99 %.

To make these probabilities completely clear, we should define their semantics. When we try to predict output bytes, we do not guess (although in some cases we could very well), but we look at the entries of our attack matrix. If we find the entry *nil* at the relevant place, we say we cannot predict the output. In the other case that we find a non-*nil* entry in the matrix, which occurs with the probabilities given above, we predict the LOQG PRNG 256 output byte as the matrix entry, and we are certain that our prediction is correct.

The statistics in 7 and 8 show how efficient our attack is.

9 What Made the Attack Possible?

Two properties of the suggested pseudo random number generator made the very efficient attack possible:

1. The same data are used as the output of the algorithm and for shuffling the internal data. This enables an attacker to keep track of the shuffling. A similar problem occurs when using shuffling to enhance the cryptographic strength of pseudo random number generators. The shuffling algorithm suggested by D. Knuth in [Knu81] is not suitable for cryptographic purposes, as an attacker learns from the output the details of the shuffling operations executed. In [Dic07b], a solution for the required cryptographic shuffling is suggested.
2. The attack is made much more efficient by the fact that no general quasigroup, but a disguised version of addition modulo 256 is used. In principle, a similar attack is also possible for a general quasigroup, but it would take many thousands of bytes of observed output to be able to predict further output. The designers of LOQG PRNG 256 aimed at an algorithm for resource constrained environments. With their choice of a somewhat disguised additive group modulo 256 with low storage requirements, they simultaneously made the attack much more efficient.

10 Another Property of LOQG PRNG 256

While implementing the attack on LOQG PRNG 256, we discovered another, probably not desired property of this algorithm. The values of the variables $C[s_1]$ of our attack in the i-th execution of the loop of LOQG PRNG 256 are with a high probability equal to C_{i-2}. Fig. 4, based on 10000 trials with random keys, gives the probability of this equality to hold as a function of i for $5 \leq i \leq 255$.

This observation is explained as follows: First, the current random value, which is called s_2, is used for row shuffling. The rows $r[i]$ and $r[s_2]$ are swapped. In the next execution of the loop of LOQG PRNG 256, the previous value of s_2 becomes s_1, which still another execution of the main loop later is used as an index into the row permutation. If the value of $r[i]$ still contains the value it was initialised to, the equation given above holds, but it might have been modified in another row swapping operation with a certain probability. This probability increases as the number of previous executions of the main loop of LOQG PRNG 256 increases. This phenomenon is clearly visible in Fig. 4.

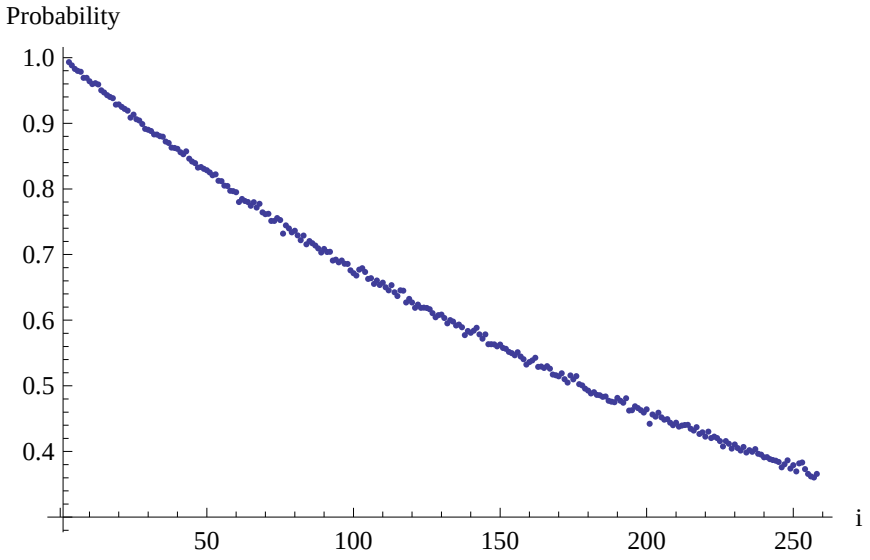

Fig. 4. Probability, as a function of i, of the equation $C[s_1] = C_{i-2}$ to hold at the beginning of the i-th execution of the main loop of LOQG PRNG 256.

Although such a structure seems quite undesirable in a cryptographic algorithm, this observation does not improve our attack. As the attack finds out with certainty which row permutations occurred, it gains nothing from knowing that some specific permutations occur with high probability.

11 Statistical Test Results

The main argument the authors of [BP12] give for the strength of their algorithm are statistical test results. They apply some test benches to data generated by various pseudo random number generators. Since in most tests the number of rejections for their pseudo random number generator was lower than for the competitors, they conclude that their generator is better. This argument neglects the probabilistic nature of statistical testing. For the NIST tests ([RSN+01]), the worst test result reported in Figure 1 of [BP12] was 984 acceptances in 1000 tests. As the NIST tests are based on an acceptance level of 99 percent, even perfect random numbers are accepted as random at an average of 990 out of 1000 test cases. A result of 984 or less acceptances in 1000 tests is by no means uncommon; when testing perfect random numbers 1000 times, it occurs with a probability of 4.79 percent. As in Figure 1 of [BP12] the results of 14 tests are reported for five generators, numbers of 984 should be observed several times, even when testing perfect random number generators.

The only conclusion one can draw from the results of Figure 1 in [BP12] is that none of the five algorithms showed deviations from random behaviour. The interpretation in [BP12] that LOQG PRNG 256 outperformed the other algorithms is not valid. The authors of [BP12] were just a little lucky when choosing the seeds for their tests.

We are convinced that rerunning the tests of Figure 1 in [BP12] with different random seeds would not lead to "better" results for LOQG PRNG 256 compared to the other algorithms, in the majority of cases. However, we have not considered it worth the effort to do all these tests for an algorithm already broken.

What the statistical results reported in [BP12] show indeed convincingly are the very bad statistical properties of the quasigroup based algorithm from [DM03].

12 Conclusion

Clearly, the pseudo random number generator suggested in [BP12] is not suitable for security purposes like stream cipher encryption as suggested in the paper, since we have shown how to attack it very efficiently.

Our conclusions about the general suitability of quasigroups for cryptographic purposes are not quite clear. Although cryptographic algorithms based on quasigroups seem to have a tendency to be broken, we wonder whether the quasigroups are to blame. They are interesting combinatorial objects, but maybe the correct way of using them for cryptographic algorithms has not yet been found.

References

[BP12] Matthew Battey and Abhishek Parakh, *A Quasigroup Based Random Number Generator for Resource Constrained Environments*, Cryptology ePrint Archive, Report 2012/471, 2012, `http://eprint.iacr.org/`.

[Dic07a] Markus Dichtl, *Bad and Good Ways of Post-processing Biased Physical Random Numbers*, FSE (Alex Biryukov, ed.), Lecture Notes in Computer Science, vol. 4593, Springer, 2007, pp. 137–152.

[Dic07b] Markus Dichtl, *Cryptographic Shuffling of Random and Pseudorandom Sequences*, Symmetric Cryptography, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2007, `http://drops.dagstuhl.de/opus/volltexte/2007/1014/`.

[DM03] V. Dimitrova and J. Markovski, *On Quasigroup Pseudo Random Sequence Generators*, 1st Balkan Conference in Informatics, Thessaloniki, 2003, pp. 393–401.

[GØM+08] Danilo Gligoroski, Rune Steinsmo Ødegård, Marija Mihova, Svein Johan Knapskog, Ljupco Kocarev, Ale Drápal, and Vlastimil Klima, *Cryptographic Hash Function EDON-R*, Submission to NIST, 2008, `http://people.item.ntnu.no/~danilog/Hash/Edon-R/Supporting_Documentation/EdonRDocumentation.pdf`.

[Kli08] Vlastimil Klima, *Multicollisions of EDON-R Hash Function and Other Observations*, 2008, `http://cryptography.hyperlink.cz/BMW/EDONR_analysis_vk.pdf`.

[Knu81] Donald E. Knuth, *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*, Addison-Wesley, 1981.

[KNW08] Dmitry Khovratovich, Ivica Nikoli, and Ralf-Philipp Weinmann, *Cryptanalysis of Edon-R*, 2008, `http://ehash.iaik.tugraz.at/uploads/7/74/Edon.pdf`.

[MGK05] Smile Markovski, Danilo Gligoroski, and Ljupco Kocarev, *Unbiased Random Sequences from Quasigroup String Transformations*, Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers, Lecture Notes in Computer Science, vol. 3557, Springer, 2005, `http://www.iacr.org/cryptodb/archive/2005/FSE/3135/3135.pdf`, pp. 163–180.

[RSN+01] 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*, NIST special publication 800-22, National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, 2001, See `http://csrc.nist.gov/rng/`.