A Random Number Generator Based on Electronic Noise and the Xorshift Algorithm

Mandana Ewert
University of Applied Sciences
Grantham-Allee 20, 53757 Sankt, Augustin, Germany
Tel: +49 2241 865 286
mandana.ewert@h-brs.de

ABSTRACT

This paper introduces a random number generator (RNG) based on the avalanche noise of two diodes. A true random number generator (TRNG) generates true random numbers with the use of the electronic noise produced by two avalanche diodes. The amplified outputs of the diodes are sampled and digitized. The difference between the two concurrently sampled and digitized outputs is calculated and used to select a seed and to drive a pseudo-random number generator (PRNG). The PRNG is an xorshift generator that generates 1024 bits in each cycle. Every sequence of 1024 bits is moderately modified and output. The TRNG delivers the next seed and the next cycle begins. The statistical behavior of the generator is analyzed and presented.

CCS Concepts

• Security and privacy→Cryptography

Keywords

Random number generator; true random number generator; pseudo-random number generator; xorshift-generator

1. INTRODUCTION

Random numbers are necessary for a variety of purposes such as generating data encryption keys, numerical simulation of physical phenomena in scientific and engineering fields, and analysis of physical experiments. The applications of random numbers vary over a wide range of scientific and technical fields to politics and games. They are fundamental to almost all secure computer systems.

Random numbers have been important for the humans since the first dice were carved out of bone over 5000 years ago. They have been used in diplomatic and military communication for encrypting the secret messages for thousands of years. Today with technological progress, the need for high quality random numbers is more than ever. In a network of computers and devices, every single encrypted connection consumes random numbers, and with a constantly growing demand for connectivity and an increased focus on privacy over the connections, the number of encrypted connections is only going to increase [1].

ICNCC 2018, December 14–16, 2018, Taipei City, Taiwan
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6553-6/18/12…$15.00
https://doi.org/10.1145/3301326.3301359

Randomness could be defined if something is algorithmically incompressible or irreducible. More precisely, a member of a set of objects is random if it has the highest possible complexity within the set [2]. In other words, random numbers are a sequence of numbers within an interval with an unpredictable progress. The sequence of random numbers must feature the following characteristics [3]:

•Uniform distribution of the numbers in the given interval
•Statistical independency of each number from the previous numbers
•Unpredictability of the future numbers

The uniformity of a sequence of bits could be examined with some well-defined tests. On the other hand, there is no such test to prove the statistical independency. Rather, a number of tests could be applied to show if there is some dependencies between the bits. If none of a number of tests could prove dependency between the bits of a sequence, the bits could be considered as independent for a defined level of confidence [3].

Unpredictability is a direct consequence of statistical independency. It cannot be proved directly. With true random sequences, each number is statistically independent of other numbers in the sequence and therefore unpredictable. Pseudo-random numbers, which are based on an algorithm, are deterministic and not statistically independent, even if the dependency cannot be proved. It means they are not unpredictable.

Additionally, high quality random numbers must have a high bit rate for todays' applications. They must also be independent of environmental parameters. These are the weaknesses of true random numbers generated based on physical sources. However, high quality and high speed are often inversely proportional to each other: excellent random number generators are often slow, where as poor random number generators are typically fast.

Despite their weaknesses, using true random numbers is indispensable as they provide the highest immunity of encrypted data to cryptanalytic attacks [4]. Computers are incapable of producing true random numbers as they are based on mathematical principles. To get true random numbers it is necessary to use a physical random phenomenon, such as thermal noise, atmospheric noise, avalanche noise or emission timing of radioactive decay. Using these phenomena as a source for an RNG is normally complex and inconvenient. Because of the complexity of design and inconvenience, the pseudo random number generators based on deterministic algorithms are often used instead.

Very fast PRNGs with excellent statistical behaviors could be implemented easily. Nevertheless, as their output is predictable, they are unusable for some applications such as cryptography. For cryptography, the science of making and breaking secret codes, robust random numbers are essential.

There have been many proposals and ideas to overcome the weakness of the PRNGs. One idea is to use an unpredictable physical random phenomenon as an entropy source and apply it to a PRNG. One popular solution is to implement a PRNG (like a linear-feedback shift register (LFSR)) and use a TRNG which generates a pool of true random numbers and apply the numbers in this pool as a seed (an initial state) [5].

This paper is organized as follows: Section II deals with the related work. Section III explains the center idea and the concept. (Figure 1) Section IV explains the system design and discusses the output signal. (Figure 2-3) Section IV presents and discusses the experimental results of the proposed random number generator. Section VI presents the conclusion and further works.

2. RELATED WORK

For generating fast and high quality random numbers there have been many proposals. Most of them use a physical source of entropy and apply it to an algorithm to improve the statistical behavior and to increase bit rate. The random number generator implemented in [6] uses the non-uniform quantization of the samples of avalanche diode noise. A PRNG is also implemented. The bit sequence generated by TRNG is xor-ed to the pseudo random sequence. The generator could get a bit rate up to 320 Mbit/s. This work [7] implements a true random number generator based on radioactive decay. A Geiger-Müller tube detects the decay events of a Thorium dioxide sample. They are transformed into a random bit stream on a Raspberry Pi single-board computer. The efficiency of the generator is 100%. On average, the system produces a random bit every 90 ms. It corresponds to a bit rate of about 11 Hz, which is very low. It can be used as a seed generator. In [8], an improved true random number generator (TRNG) is proposed, which comprises a low-bias hardware random number generator (HRNG) and a scrambler based on LFSR. The HRNG reduces both DC offset from the noise sources and offset voltage from the comparator to generate low-bias bit stream. The LFSR-based scrambler further reduces the bias to zero.

3. CENTER IDEA AND CONCEPT

Noise, a small fluctuation in current or in voltage is generated in all semiconductor devices. The intensity of these fluctuations depends on the device type, its manufacturing process, and operating conditions [9].

Avalanche noise is a form of noise produced by Zener or avalanche breakdown in a pn junction. In avalanche breakdown, holes and electrons in the depletion region of a reverse-biased pn junction acquire sufficient energy to create hole-electron pairs by colliding with silicon atoms. This noise is much higher than the other noises, because a single carrier can start an avalanching process that results in the production of a current burst containing many carriers moving together. These collisions are purely random and produce random current pulses but much more intense than other noises [10] [11].

The avalanche noise is associated with the current flow. The higher the external current, the more noise generated. The total current is the sum of the random chain reactions. If there is an avalanche breakthrough, avalanche noise dominates strongly other noises in a diode. Diodes when biased into the avalanche breakdown region produce significant amounts of excess noise with good stability, high bandwidth, and good statistical properties. The spectral density of avalanche noise depends on the actual voltage across the diode. It can be assumed that each electron that flows as having a multiplication factor, M. It can be written like that:

$$M = \frac{1}{1 - \left|\frac{V_D}{V_B}\right|^{\alpha}}$$

In this equation α is an exponent that varies between 3 and 6 depending on the semiconductor characteristics, V_B is the diode breakdown voltage and V_D is the actual voltage across the diode [12]. To achieve higher M, the diode voltage V_D must be as high as possible and nearly equal to V_B, and thus the diode must be in the avalanche breakdown region. In this area, the denominator of the formula will be almost zero and M will be a very large number.

This characteristic is used for generating the true random numbers. To achieve this, a circuit is needed to sample the produced noise. The sampled values must be amplified and converted from analog to digital. In order to avoid correlation between the samples, it is important to maintain a low bit rate. By using the xorshift algorithm in between each sampled pair, the final bit rate is increased while still avoiding the correlation problem.

4. SYSTEM DESIGN

The RNG proposed in this paper implements a TRNG based on avalanche noise of two diodes and a PRNG based on the xorshift algorithm [13]. A 2-channel, 8-bit A/D samples the amplified avalanche noise of two TVS diodes, and saves them in a file. The difference between two concurrently sampled outputs is used to select an initial state (seed) for an xorshift generator and to drive it.

The xorshift generator generates 32 32-bit numbers in each cycle. A matrix of 32x32 bits is constructed and moderately modified before output. The next cycle begins and the A/D delivers the next 8-bit number.

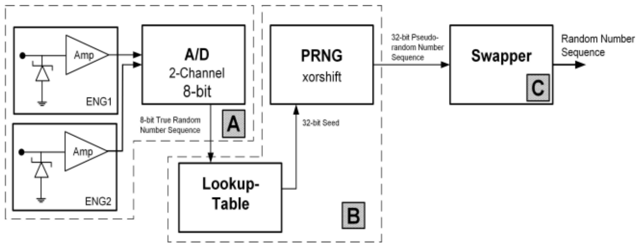


Figure 1. Block diagram of the RNG.

4.1 True Random Number Generator – Circuit Design

The avalanche effect of two TVS diodes has been used as an entropy source. They are on separate and isolated printed circuit boards. Both diodes concerned are P6SMB10. They generate high-amplitude noise, which is sampled after a two-stage amplification. Both transistors are BFR92A. In order to eliminate the influence of the environmental noise, the difference between the outputs of two circuits is used as the entropy source.

Figure 2 shows the electrical circuit diagram of a true random number generator, and Figure 3 shows the printed circuit board of a true random number generator.

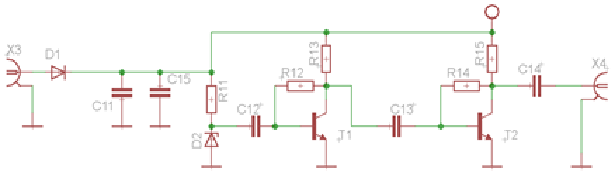


Figure 2. Circuit diagram of a true random generator.

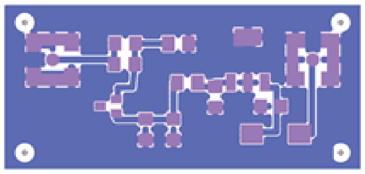
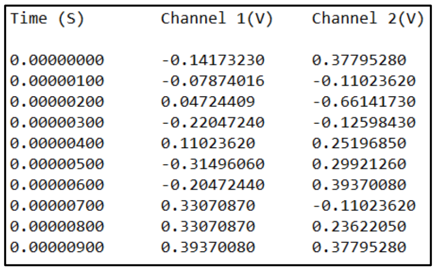


Figure 3. Printed circuit board of a true random number generator.

The sampled values are in the range of -2 to 2 volt. Figure 4 shows the sampled values for a time period of 10 µsec.



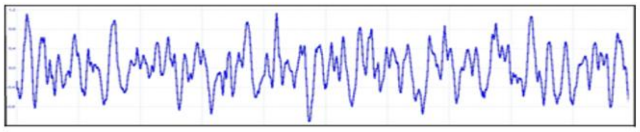
```
Time (S)          Channel 1(V)      Channel 2(V)

0.00000000        -0.14173230       0.37795280
0.00000100        -0.07874016       -0.11023620
0.00000200        0.04724409        -0.66141730
0.00000300        -0.22047240       -0.12598430
0.00000400        0.11023620        0.25196850
0.00000500        -0.31496060       0.29921260
0.00000600        -0.20472440       0.39370080
0.00000700        0.33070870        -0.11023620
0.00000800        0.33070870        0.23622050
0.00000900        0.39370080        0.37795280
```

Figure 4. Sampled values Channel 1 and 2 for a period of 10 µsec.

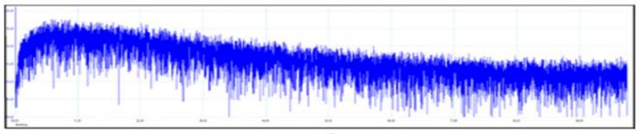
Figure 5a and Figure 5b show the amplified output of one diode in both time and frequency domains. Figure 5c shows recorded histograms of the captured data between -1.5 and 1.5 volt. As can be seen, it resembles a Gaussian distribution. After sampling the amplified outputs of the diodes, the two-channel 8-bit A/D converts the sampled values to digital and saves them as 8-bit numbers in a common file.

Finally, the difference of two simultaneously sampled outputs is calculated and saved. Figure 5d shows the distribution of 10^6 8-bit numbers between 0 and 255. Several tests confirmed the Gaussian distribution of these numbers.

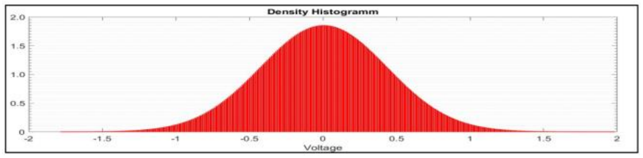
Every bit of the captured 8-bit numbers contains randomness or entropy, which could be used in different ways. One possibility is to use the most significant bit (MSB) which represents the zero crossing or sign changing of the number. This bit shows excellent statistical behavior and passes every statistical test. Nevertheless, it is not a good choice as the avalanche noise generated by diodes is not very fast and it should not be sampled by a sampling frequency higher than 1 MHz. Several diodes have been tested. They showed very same behavior. By using a sampling frequency higher than 1 MHz, there would be many zeros or ones in a row which means a dependency or a correlation between the neighboring numbers. This would make the sampled numbers unusable as an entropy source. As the frequency of 1MHz or the bit rate of 8 Mbit/s is not enough for todays' applications a transformation is needed.



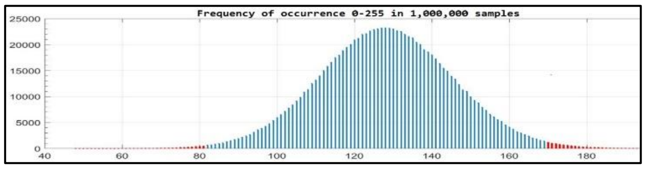
a



b



c



d

Figure 5. (a) Output of the TRNG in time domain, (b) in frequency domain (c) Probability density distribution and (d) histogram of the TRNG.

4.2 Pseudo Random Number Generator

Xorshift generators are fast, high quality pseudo-random number generators, which have been proposed by George Marsaglia [13]. Actually, they are a good choice for non-cryptographic applications as they are fast and easy to implement, have long periods and pass strict statistical tests. Nevertheless, they are not a good choice for cryptographic applications because they are based on mathematical or logical algorithms and their courses could be predicted if their parameters and the initial state (seed) of the generator are known.

An xorshift random number generator (xorshift RNG) produces a sequence of $2^{32}-1$ integers x, or a sequence of $2^{64}-1$ pairs x,y, or a sequence of $2^{96}-1$ triples x,y,z, etc., by means of repeated use of a simple computer construction: exclusive-or (xor) a computer word with a shifted version of itself [13]. For 32-bit numbers Marsaglia proposes 81 triples of parameters (a, b, c) and 8 lines of C code in his paper. Any choice of the 81 a,b,c triples for 32-bit sequences, and any one of the eight lines of C code, will provide, for 32-bit words, an xorshift RNG with period $2^{32}-1$, for a total of 8×81 = 648 choices. He proposes a total of 2200 choices for the parameters a, b and c for 64-bit sequences.

4.2.1 32-bit xorshift generator

In the following sections, the two PRNG are described. The implementation of a 32-bit xorshift generator which is used to produce pseudo-random numbers is presented in this section. In the following section, the 64-bit generator is detailed. The initial state (seed) used by the xorshift generator is an 8-bit number supplied by A/D. A total of 88 static seeds are stored in a look-up table (LUT). 87 of them are for the 8-bit numbers with an occurrence probability higher than 0.001. The last static seed is stored for all of the other numbers with a very small probability of occurrence. After generating 32 numbers, the generator stores the last number to use it the following time if the A/D converter delivers the same 8-bit number, and a cycle ends. The new cycle starts and the A/D converter supplies the next seed.

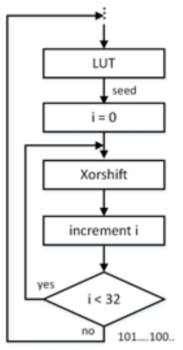


Figure 6. Flowchart of the generation of 32 32-bit numbers during a cycle.

The frequency of the A/D converter is 1 MHz. The xorshift generator is implemented in C. With a proper hardware the bit rate of final bit sequence will be:

1 MHz x 32 number x 32 bits/number= 1.024 Gbit/s

Several blocks of pseudo numbers have been generated and statistically tested for different sets of parameters, different lines of codes and different seeds. All of them pass almost every random test, except the Binary Matrix Rank Test of the NIST Test Suite. The program is written in C. For the final tests the following lines of codes are used:

```
x32 = seed; period = 32;
        for(int i=0; i < period; i++){
            x32 ^= x32 << 5;
            x32 ^= x32 >> 27;
            x32 ^= x32 << 25;
            *(x+k) = x32;
            ++k;
        }
    k=0; *(seeds+no) = *(x+31);
```

The purpose of the rank test is to check for linear dependence among fixed length substrings of the original sequence [14]. This test divides the binary sequence in 32x32-disjoint blocks. As we have chosen a 32-bit xorshift generator, in every row of every 32x32-matrix, the binary sequence of one pseudo-random number will appear. The rank of each matrix is then calculated. A common approach to finding the rank of a matrix is to reduce it to a simpler form by using Gaussian elimination (also known as row reduction). The rank of a matrix is the number of rows or columns with at least one nonzero element after row reduction [15]. If the rank of a matrix is smaller than a defined value, the test will fail. This is to be expected since every row of the matrices will be generated using the last row and the linear operation xor. Marsaglia too mentions this weakness in his paper. To address this weakness, non-linearity needs to be built into the matrices. The following section describes this process.

4.2.1.1 Swapping the first row with the main diagonal

A non-linear transformation is needed to change the ranks of the matrices. In order to avoid changing the results of the other tests, the elements of each matrix are only moderately moved. This process can be seen in Figure 7. The first row and the main diagonal of every matrix are swapped. The other elements of the matrices keep their positions. As the number of ones and zeros remains unchanged, the results of the frequency test of both matrices, on which many other tests are based remain the same. The other tests show very similar results.

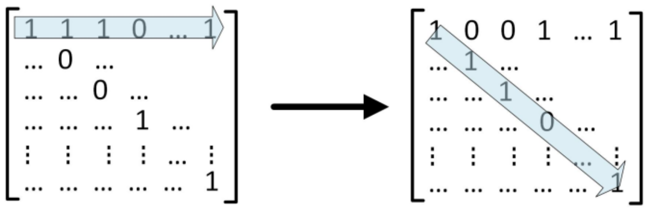


Figure 7. Swapping the first row with the main diagonal.

After a matrix is completed, the binary sequence is output and the program is given the following seed by the A/D to build a new matrix. Obviously, the new order of the bits will change the results of the statistical tests. Nevertheless, these changes are insignificant.

4.2.2 64-bit xorshift generator

The implementation of the 64-bit xorshift generator is presented here. A total of 2200 choices for the parameters have been proposed by Marsaglia. Several values for the parameters a, b and c have been tested. All runs resulted in consistent behavior. All the NIST Test Suite tests were successfully completed. While at first glance, everything seems fine, a closer look reveals that using long streams of bits in every cycle decreases the real-randomness and increases the pseudo-randomness. By using just 16 numbers, this situation is avoided and in addition, the same bit rate of 1.024 Gbit/s is maintained.

1 MHz x 16 number x 64 bits/number= 1.024 Gbit/s

The following set of parameters have been used to generate random numbers and to evaluate the generator:

```
x64 ^= x64 << 21; x64 ^= x64 >> 13; x64 ^= x64 << 52;
```

5. EXPERIMENTAL RESULTS

Randomness is a probabilistic property; that is, the properties of a random sequence can be characterized and described in terms of probability. For probability analysis of the generated numbers the NIST Test Suite has been used. The NIST Test Suite is a statistical package consisting of 15 tests that were developed to test the randomness of (arbitrarily long) binary sequences produced by either hardware- or software-based cryptographic random or pseudo-random number generators [14].

In this Test Suite every statistical test is formulated to test a specific null hypothesis (H0). The null hypothesis under test is that the sequence being tested is random. Associated with the null hypothesis there is also an alternative hypothesis (Ha) which allege that the sequence is not random. It is necessary to choose a relevant randomness statistic to accept or to reject the null hypothesis. Under an assumption of randomness, such a statistic has a distribution of possible values. A theoretical reference distribution of this statistic under the null hypothesis and a critical value is determined by mathematical methods. Every test calculates a test statistic value and compares it to the critical value. If the test statistic value exceeds the critical value the null hypothesis will be rejected. Otherwise, it will be accepted. There could be two types of errors. Type I error describes, that the bit sequence is random but the null hypothesis has rejected it. The probability of this error is called the level of significance (α). It is an important parameter for these tests. Type II error describes that although the bit sequence is non-random the null hypothesis is accepted. The probability of the type II error is normally presented by β. Every test calculates a *p-value* which is a strength of the evidence against the null hypothesis. If the calculated *p-value* is higher than or equal to α (the level of significance) the null hypothesis is accepted and the test passes. Otherwise, the null hypothesis is rejected and test fails. An α of 0.01 indicates that one would expect one sequence in 100 sequences to be rejected.

The first test is the Frequency or Monobit Test. It counts the numbers of ones and zeros in a bit sequence. If they are approximately the same, the test passes. With the selected parameters, this test will fail if there are more than 501300 or less than 498700 ones or zeros in a sequence of 10^6 bits. All subsequent tests depend on the passing of this test. The Block Frequency Test determines the proportion of ones and zeros in a block with the length of M. The Run Test counts the number of ones with at least one zero before and at least one zero after it. The Longest Run Test counts the number of ones with at least one zero before and at least one zero after it in a block of length M. The Rank Test checks for linear dependency among fixed length substrings of the original sequence. The FFT Test checks if there is a periodicity in the sequence. The focus of the Overlapping Template Matching test and the Non-Overlapping Template Test is the number of occurrences of pre-specified target strings.

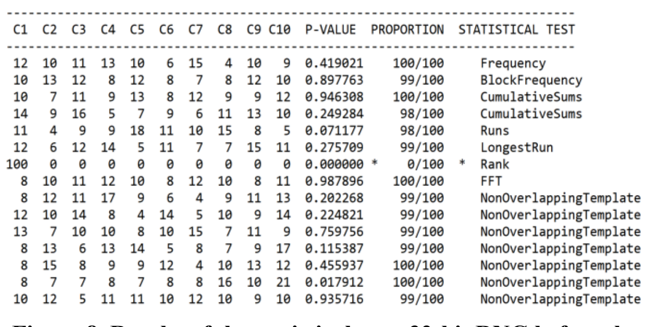
All tests have been carried out for 100 sequences of 10^6 bits, a significance level of $\alpha = 0.01$, block length of Frequency Test M = 128, of Non-Overlapping Template Test M = 9, of Overlapping Template Test M = 9, of Approximate Entropy Test M = 10, of Serial Test M = 16 and of Linear Complexity Test M = 500.

5.1 32-bit generator

100 sequences of 10^6 bits are captured before and after the swapping procedure and tested with the NIST Test Suite.

5.1.1 Results before the swapping procedure

Figure 8 shows the results of one of the statistical tests applied to the random numbers generated before the swapping procedure. Only the Rank Test fails. This demonstrates the vulnerability of the random numbers since with enough time and effort, the 32-bit numbers in a 32-block could be cracked.



```
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 12  10  11  13  10   6  15   4  10   9  0.419021    100/100     Frequency
 10  13  12   8  12   8   7   8  12  10  0.897763     99/100     BlockFrequency
 10   7  11   9  13   8  12   9   9  12  0.946308    100/100     CumulativeSums
 14   9  16   5   7   9   6  11  13  10  0.249284     98/100     CumulativeSums
 11   4   9   9  18  11  10  15   8   5  0.071177     98/100     Runs
 12   6  12  14   5  11   7   7  15  11  0.275709     99/100     LongestRun
100   0   0   0   0   0   0   0   0   0  0.000000 *    0/100  *  Rank
  8  10  11  12  10   8  12  10   8  11  0.987896    100/100     FFT
  8  12  11  17   9   6   4   9  11  13  0.202268     99/100     NonOverlappingTemplate
 12  10  14   8   4  14   5  10   9  14  0.224821     99/100     NonOverlappingTemplate
 13   7  10  10   8  10  15   7  11   9  0.759756     99/100     NonOverlappingTemplate
  8  13   6  13  14   5   8   7   9  17  0.115387     99/100     NonOverlappingTemplate
  8  15   8   9   9  12   4  10  13  12  0.455937    100/100     NonOverlappingTemplate
  8   7   7   8   7   8   8  16  10  21  0.017912    100/100     NonOverlappingTemplate
 10  12   5  11  11  10  12  10   9  10  0.935716     99/100     NonOverlappingTemplate
```

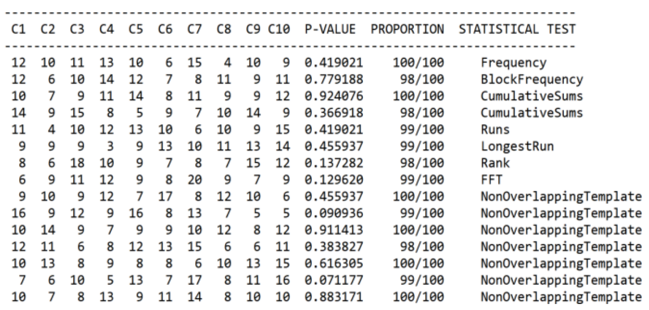
Figure 8. Results of the statistical tests 32-bit RNG before the swapping procedure.

5.1.2 Results after the swapping procedure

In this section, the random numbers which were reordered with the swapping procedure are tested with the NIST Test Suite. An excerpt of the results is listed in Figure 9. As we see, all statistical tests have been passed successfully. The final *p-values* are high and the calculated *p-values* for every test C1 to C10 are almost uniformly distributed between 0 and 1. This confirms the robustness and the stability of the generator.

The swapping procedure changes just the order of the bits in every 1032 sequence of bits. Reordering of the bits should not change the results of the frequency test. This can be seen in the results presented in Figures 8 and 9. Nevertheless, very small changes are possible, as every 10^6 of bits includes 976 blocks of 1024 bit sequences. It remains 576 bits. The order of these 576 bits could change the results of the frequency test. This situation is not represented in the results of these tests, but rather can be seen in Section 5.2.

The remaining tests analyze the order rather than the number of the bits in a given block. As such, the changes in the results of these tests are more obvious. Some of them are improved and some not. All of them remain within the acceptable range which itself is very limited.



```
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 12  10  11  13  10   6  15   4  10   9  0.419021    100/100     Frequency
 12   6  10  14  12   7   8  11   9  11  0.779188     98/100     BlockFrequency
 10   7   9  11  14   8  11   9   9  12  0.924076    100/100     CumulativeSums
 14   9  15   8   5   9   7  10  14   9  0.366918     98/100     CumulativeSums
 11   4  10  12  13  10   6  10   9  15  0.419021     99/100     Runs
  9   9   9   3   9  13  10  11  13  14  0.455937     99/100     LongestRun
  8   6  18  10   9   7   8   7  15  12  0.137282     98/100     Rank
  6   9  11  12   9   8  20   9   7   9  0.129620     99/100     FFT
  9  10   9  12   7  17   8  12  10   6  0.455937    100/100     NonOverlappingTemplate
 16   9  12   9  16   8  13   7   5   5  0.090936     99/100     NonOverlappingTemplate
 10  14   9   7   9   9  10  12   8  12  0.911413    100/100     NonOverlappingTemplate
 12  11   6   8  12  13  15   6   6  11  0.383827     98/100     NonOverlappingTemplate
 10  13   8   9   8   8   6  10  13  15  0.616305    100/100     NonOverlappingTemplate
  7   6  10   5  13   7  17   8  11  16  0.071177     99/100     NonOverlappingTemplate
 10   7   8  13   9  11  14   8  10  10  0.883171    100/100     NonOverlappingTemplate
```

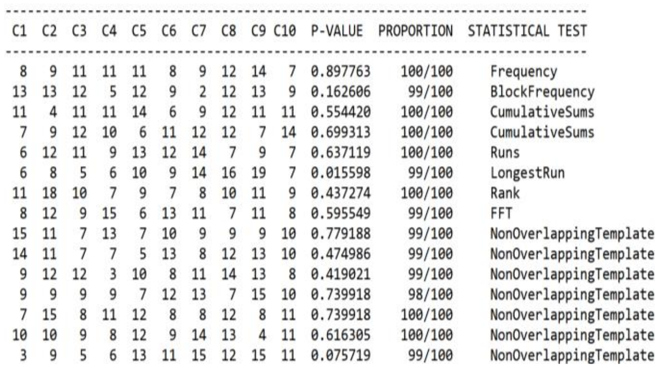
Figure 9. Results of the statistical Tests 32-bit RNG after the swapping procedure.

5.2 64-bit generator

100 sequences of 10^6 bits are captured and thoroughly tested with the NIST Test Suite.

5.2.1 Results before the swapping procedure

Figure 10 shows the results of one of the statistical tests applied to the generated numbers with the parameters `a=21, b=13, c=52.` The results show that all the tests are passed successfully. The same results have been achieved with various values for the parameters `a`, `b` and `c`. This generator looks to be stable and secure. However, every block of 1024 bits is based on the deterministic xorshift algorithm. This is a vulnerability which can not be proven mathematically.

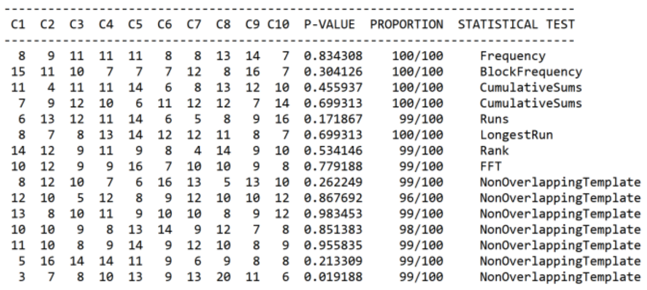


```
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  8   9  11  11  11   8   9  12  14   7  0.897763    100/100     Frequency
 13  13  12   5  12   9   2  12  13   9  0.162606     99/100     BlockFrequency
 11   4  11  11  14   6   9  12  11  11  0.554420    100/100     CumulativeSums
  7   9  12  10   6  11  12  12   7  14  0.699313    100/100     CumulativeSums
  6  12  11   9  13  12  14   7   9   7  0.637119    100/100     Runs
  6   8   5   6  10   9  14  16  19   7  0.015598     99/100     LongestRun
 11  18  10   7   9   7   8  10  11   9  0.437274    100/100     Rank
  8  12   9  15   6  13  11   7  11   8  0.595549     99/100     FFT
 15  11   7  13   7  10   9   9   9  10  0.779188     99/100     NonOverlappingTemplate
 14  11   7   7   5  13   8  12  13  10  0.474986     99/100     NonOverlappingTemplate
  9  12  12   3  10   8  11  14  13   8  0.419021     99/100     NonOverlappingTemplate
  9   9   9   9   7  12  13   7  15  10  0.739918     98/100     NonOverlappingTemplate
  7  15   8  11  12   8   8  12   8  11  0.739918    100/100     NonOverlappingTemplate
 10  10   9   8  12   9  14  13   4  11  0.616305    100/100     NonOverlappingTemplate
  3   9   5   6  13  11  15  12  15  11  0.075719     99/100     NonOverlappingTemplate
```

Figure 10. Results of the statistical tests 64-bit RNG before the swapping procedure.

5.2.2 *Results after the swapping procedure*

To counter the weakness of the 64-bit generator, the first row and the main diagonal of every 32x32 matrix are swapped. The results are shown in Figure 11.



```
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  8   9  11  11  11   8   8  13  14   7  0.834308    100/100     Frequency
 15  11  10   7   7   7  12   8  16   7  0.304126    100/100     BlockFrequency
 11   4  11  11  14   6   8  13  12  10  0.455937    100/100     CumulativeSums
  7   9  12  10   6  11  12  12   7  14  0.699313    100/100     CumulativeSums
  6  13  12  11  14   6   5   8   9  16  0.171867     99/100     Runs
  8   7   8  13  14  12  12  11   8   7  0.699313    100/100     LongestRun
 14  12   9  11   9   8   4  14   9  10  0.534146     99/100     Rank
 10  12   9   9  16   7  10  10   9   8  0.779188     99/100     FFT
  8  12  10   7   6  16  13   5  13  10  0.262249     99/100     NonOverlappingTemplate
 12  10   5  12   8   9  12  10  10  12  0.867692     96/100     NonOverlappingTemplate
 13   8  10  11   9  10  10   8   9  12  0.983453     99/100     NonOverlappingTemplate
 10  10   9   8  13  14   9  12   7   8  0.851383     98/100     NonOverlappingTemplate
 11  10   8   9  14   9  12  10   8   9  0.955835     99/100     NonOverlappingTemplate
  5  16  14  14  11   9   6   9   8   8  0.213309     99/100     NonOverlappingTemplate
  3   7   8  10  13   9  13  20  11   6  0.019188     99/100     NonOverlappingTemplate
```

Figure 11. Results of the statistical tests 64-bit RNG after the swapping procedure.

A closer look shows that the results have changed slightly as a result of the reordering of the bits. In addition, a small change in the Frequency test can also be seen. After the swapping procedure the *p-value* of one 10^6 bit sequence is moved from C8 to C9. Although it means a better *p*-value for the sequence, this decreases the uniformity of the values in C1 to C10, and consequently decreases the final *p-value* from 0.897763 to 0.834308. This is the result of the 576 remaining bits in every 10^6 bit sequence.

The results confirm the stability and the robustness of the generator. The non-linearity of the procedure of re-ordering the bits which is presented in this work improves robustness by making the random numbers difficult to crack. This makes the need for an increased amount of memory which the process requires worthwhile.

5.3 Bit rate of the RNG

The combination of the presented TRNG and the PRNG generate high quality random numbers with a bit rate of 1.024 Gbit/s. It is possible to increase the bit rate by generating more numbers in a sequence or by using a 96- or higher-bit xorshift generator.

5.4 Temperature Dependency

As the temperature rises, the leakage current rises too and the noise decreases accordingly. Light can also release free electrons, called photo-current, in the depletion region of the diode so that the noise level decreases. Thus, the noise in an avalanche breakdown is maximal at low temperatures and a dark ambient [16]. In order to verify the temperature dependency of the generated random numbers, several tests have been carried out under varying temperatures. Although the amplitude of the amplifier is shifted or changed as temperature increases, the experimental results stay the same. This means that increasing the temperature from 20 ºC to about 50 ºC has no significant impact on the generator.

6. CONCLUSION

The random number generator based on avalanche noise and the xorshift algorithm, which is proposed in this paper, satisfies the NIST Test Suite. Due to the partial physical nature of the generator, the numbers generated cannot be predicted. This makes the generator suitable for numerous applications including cryptography. The TRNG generates 8-bit numbers with a frequency of 1 MHz. The xorshift generator generates a sequence of 1024 bits between two samplings. The bit rate of the generator is 1Gbit/sec. It could be increased by using a 64, 96 or 128-bit xorshift generator and by generating more numbers in every cycle.

7. REFERENCES

[1] [Online]. Available: https://conferences.oreilly.com/oscon/oscon2013/public/schedule/detail/28777.

[2] G. J. Chaitin, Exploring RANDOMNESS, London: Springer, 2002.

[3] W. Stalling, Cryptography and Network Security, England: Pearson, 2014.

[4] H. G. Katzgraber, “Cornell University Library,” 9-20 August 2010, Oldenburg, Germany. [Online]. Available: https://arxiv.org/pdf/1005.4117.pdf..

[5] . M. (Intel), „Intel® Digital Random Number Generator (DRNG) Software Implementation Guide,“ 15 May 2014. [Online]. Available: https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide. [Zugriff am 7 March 2017].

[6] P. M. Z. B. V. O. S. M. Miroslav Peric, „High speed random number generator for section key generation in encryption devices,“ 20 January 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6716186/. [Zugriff am 28 February 2017].

[7] M. S. J. F. I. H. Daniel Rüschen. [Online]. Available: https://www.medit.hia.rwth-aachen.

[8] Y. L. C.-H. H. Wei Mao, „Zero-bias true random number generator using LFSR-based scrambler,“ 05 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8050474/.

[9] Konczakowska und B. M. Wilamowski, „Noise in Semiconductor Devices,“ [Online]. Available: http://www.eng.auburn.edu/%7Ewilambm/pap/2011/K10147_C011.pdf. [Zugriff am 1 2 2017].

[10] P. R. Gray, Analysis and design of Analog Integrated Circuits

[11] Texas Instrument, „ti,“ 2007. [Online]. Available: http://www.ti.com/lit/an/slva043b/slva043b.pdf

[12] Tektronix, „Noise Figure - Overview of Noise Measurement Methods,“ 2014. [Online]. Available: https://de.tek.com/document/.../noise-figure-overview-noise-measurement-methods

[13] G. Marsaglia, „ Xorshift RNGs,“ 2003. [Online]. Available: https://www.jstatsoft.org/article/view/v008i14

[14] J. S. J. N. M. Andrew Rukhin, „http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html,“ April 2010. [Online]. [Zugriff am 23 February 2017].

[15] G. Bärwolff, Höhere Mathematik, Berlin: Spektrum, 2009.

[16] Vishay Semiconductor - Jochen Krieger, „The Noise of Avalanche Breakdown Diodes - Vishay,“ 20 July 2017. [Online]. Available: https://www.vishay.com/docs/85966/thenoiseofavalanchebreakdown.pdf.