# Enhancing the operational efficiency of quantum random number generators

Botond L. Márton, Dóra Istenes and László Bacsárdi, *Member, IEEE*

***Abstract*— Random numbers are of vital importance in today's world and used for example in many cryptographical protocols to secure the communication over the internet. The generators producing these numbers are Pseudo Random Number Generators (PRNGs) or True Random Number Generators (TRNGs). A subclass of TRNGs are the Quantum based Random Number Generators (QRNGs) whose generation processes are based on quantum phenomena. However, the achievable quality of the numbers generated from a practical implementation can differ from the theoretically possible. To ease this negative effect post-processing can be used, which contains the use of extractors. They extract as much entropy as possible from the original source and produce a new output with better properties. The quality and the different properties of a given output can be measured with the help of statistical tests. In our work we examined the effect of different extractors on two QRNG outputs and found that with the right extractor we can improve their quality.**

***Index Terms*—random numbers, statistical testing, quantum communication, QRNG**

## I. INTRODUCTION

QUANTUM TECHNOLOGIES are developing at a rapid speed in the modern world and they vastly differ from their classical counterparts. They offer new approaches for communication, cryptography or algorithm design. From an algorithmic standpoint they propose new and in many cases faster algorithms (for example Shor's algorithm for prime factoring or in the area of resource distribution [1]) which can utilize the unique phenomena present only in the world of quantum mechanics[2][3]. Two of the most developed technologies in the field are QRNGs and QKD (Quantum Key Distribution). QKD is mostly used as a building block in cryptographic solutions. One of these is the one-time pad encryption scheme, where parties use a different, unique random key for the encryption of each message. This is a mathematically proven secure method, with only one weakness, sharing the keys. QKD patches this weakness by providing a safe way to share the keys between the parties [4].

The authors are with the Department of Networked Systems and Services, Budapest University of Technology and Economics, Budapest, H-1117 Hungary.
E-mail: martonboti@gmail.com, idooori@gmail.com, bacsardi@hit.bme.hu.

The work was supported by the National Research Development and Innovation Office of Hungary (Project No. 2017-1.2.1-NKP-2017-00001). L. Bacsárdi thanks the support of the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

The application of random numbers ranges from dice simulators to cryptographic systems and mathematical simulations [5]. These various usages require different traits from the generators. High bitrate, quality, and safety are among the attributes the different applications expect.

Quantum generators make ideal outputs for most requirements, but their main quality is generating truly random numbers due to an underlying quantum phenomenon. Even so, they have their flaws, which mainly come from the limits of our physical tools.

The field of QRNGs is becoming more popular as some of the generators are already available on the commercial market (one of which is briefly introduced in Section II.A), see [6] for more. At the same time the field of randomness extraction also had interesting results. Ma et al investigated the effect of a Trevisian and Toeplitz extractor on a QRNG in [7]. In our work, we applied the Toeplitz extractor as well but the QRNG they used is based on a different generation mechanism. Qi and Bing tested a generator based on amplified spontaneous emission [8]. One of the QRNGs we worked with is also based on amplified spontaneous emission, but they used a different setup. In [9] Zhang, Xiao-Guang, et al presented a generator based on laser phase fluctuations, where they used a pipeline based solution with a Toeplitz extractor to achieve real-time processing. In our work we used the Toeplitz extractor, but the real-time operation was not one of our goals, therefore our implementation differs. Shakhovoy, Roman, et al. introduced a QRNG which works without the need for post-processing [10]. In current work we focused on investigation of QRNG, but another important question is the comparison of efficiency between QRNGs and PRNGs which was investigated by Martínez, Aldo C., et al in [11].

In our work we concentrated on two QRNGs, which were built at Budapest University of Technology and Economics (BME)[16]. Prior to our work, the generators were only tested without post-processing. In this paper, we present how extractors can improve the quality of two outputs from these generators. We implemented the extractors in Python, examined their applicability and their yielded results.

This article is structured as follows. In Section II we will introduce two popular generation methods used in QRNGs (on which the tested generators are based on) then we will show how can we measure the quality of random numbers and what is an extractor. After that in Section III we will present what we found during our testing, while Section IV contains our conclusion.

DOI: 10.36244/ICJ.2021.2.2

## II. RANDOM NUMBER GENERATORS

### A. Generation methods

PRNGs generate a stream based on a mathematical algorithm and a starting point, the so-called seed. Although this makes it easy to generate numbers in high quantity, it also makes the output deterministic, and in turn prone to exploitation. Within possession of its algorithm and seed, which may be acquired through inspecting the output, the PRNGs upcoming outputs become easily predictable. This makes it highly unsafe to use them in applications with a high-security requirement, such as lottery or cryptographic solutions [12].

For TRNGs their entropy source comes from inherently random events, like radioactive decay, atmospheric noise or quantum mechanical events. Their set up is much more complex than the PRNGs, and their generation speed is also slower, but due to the high unpredictability of their source, their output is adequate for high-security uses.

QRNGs provide non-deterministic outputs in great quantities in a short time. and they are one of the most actively developed quantum computing technologies.

The two main producers of commercially available QRNG chips are ID Quantique (IDQ) and Quantum Numbers Corp (QNC). Both companies produce state of the art QRNG chips although the smallest commercial one belongs to IDQ, the Quantis QRNG chip. It contains a LED light source that emits random number of photons which are captured and counted by an image sensor, providing a set of easily accessible raw numbers. It also has a self-verification process, where if it detects any failure it starts an automatic recovery procedure instantly and notifies the user [13].

### B. Photon detection interval

Many types of optical QRNGs exist. A portion of them rely on a beam splitters and different amount of detectors. These tools can contribute greatly to the bias of a generator. In theory there are ideal equipments but in truth, perfect tools do not exist. Even a single detector's quantum efficiency is not 100% but using multiple detectors raises the problem of the two detectors differences [14].

The photon detection interval generator uses only one detector, as to mitigate the bias.

The distribution of the time between two detections is exponential with a probability density function $\lambda e^{-\lambda t}$ where $\lambda$ is the expected number of photons detected in a unit of time.

The time values are compared in pairs. For $t_1, t_2$ time values the generator returns 0 if $t_1 > t_2$ and 1 in case of $t_2 > t_1$. We restart the clock at each detection to eliminate correlation between the data. The time values of course have a certain amount of accuracy which makes equal values more probable. To overcome this issue we discard equal values [6].

### C. Amplified Spontaneous Emission

To achieve long ranges in fiber communication optical amplification is used. The basis for this technique is stimulated emission. During stimulated emission when a particle in excited state interacts with an incoming photon, the excited particle drops to a lower energy level emitting a new photon, whose properties are the same as the ones which started the process. For stimulated emission to be dominant over absorption, population inversion must be present. This means that there are more particles in excited state than in lower energy state. However, if stimulated emission is possible for a particle, than so is spontaneous emission, during which an excited particle randomly drops to a lower energy level while emitting a new photon with random properties. This photon then can cause stimulated emission thus creating amplified spontaneous emission, ASE. In an optical system this phenomenon is considered noise which fortunately can be measured, therefore it can be used as a basis for random number generation. During generation if there is no incoming signal in the amplifier, ASE will be the dominant interaction. Then the optical power can be sampled, giving statistically independent random variables. [6][15][16]

### D. Measuring the randomness

As we saw earlier, there are many ways to build a random number generator. But we need to determine the quality of the numbers (or the bits) which are coming out of the machine. The first problem is that we have to measure how random the output is. This means that we need to define what randomness is. This is a hard task, because we cannot tell certainly whether a given finite sequence of bits is random or not. In most cases we have to settle for a more practical solution. Instead of declaring that the output of a generator is truly random with absolute certainty, we will say that the output is closer to a true random source then a given limit. Therefore we can only say with a given probability, that the measured output is random or not, but if this probability is high enough, this approach is good for most usages.

The tests we can use on a generator (or the output of this generator) can range from the very simple to the more complex; but they have a common property: they require a finite number of bits. This means that firstly the length of the bit sequence is important. The longer the sequence is the better the precision of the tests. Secondly, this means that we can never look at the whole output of a generator, only a part of it and we have to make a decision based on this part. It is therefore possible that the generator will fail the same test that it passed earlier, because on the second run the new output will be different. To give an example of a simple test one can think about a truly random source, e.g. the uniform distribution. It puts out a 1 or a 0 bit with equal probability (50%), so if one looks at a longer and longer sequence from this source, one will find out the number of 1s and 0s is approaching the same number. This can be interpreted as a test: we count the 1 and 0 bits in the output of the generator and compare them to each other.

The main goal of these tests is to measure the randomness of the sequence which cannot be made with certainty as it was stated earlier, that's the reason why these tests are statistical tests. They take a statistical property (for example the number of 1s and 0s as mentioned above) and based on this result and a previously given criterion (for example: how far can the number of 1s and 0s differ from each other) can declare whether the sequence passed or not. Most of the tests fall under the statistical hypothesis test category. In the hypothesis test we want to accept or reject the null-hypothesis ($H_0$). During the testing of a random number generator the null-

hypothesis is that the generator is producing random numbers. The other hypothesis in the test is called the alternative hypothesis ($H_a$). $H_a$ is the opposite of $H_0$: it says that the generator isn't producing truly random numbers. The next step is to calculate a distribution function with the help of a probabilistic value (most of the time these are well known probabilistic values) while assuming that the null-hypothesis is true. After this we select a significance level (α) on this distribution. Generally, this is a very small value. In the RNG testing α tends to be around 1%. Lastly, we calculate the statistical value which the given test measures and compare it to the significance level. If it is below α, we reject the null-hypothesis and accept the alternative. If it is above it, we accept the null-hypothesis and reject the alternative. Based on our decision and the reality we have four possible outcomes. If we accepted the null-hypothesis and it is in fact true, we chose correctly (this has a probability of 1-α). It is the same if we rejected it and it was false in reality (the probability of this outcome is 1-β). The other two outcomes are called Type I and Type II error. The Type I error occurs when we rejected $H_0$, but it was true. This outcome has a probability of α and is called false positive. The Type II error is when we accept $H_0$, but it was false. It has a probability of β and is called false negative. Out of these two the Type I is more acceptable and with a good decision on the value of α we can fine tune it. In this case we falsely brand the RNG as "not random" in the test. But with the help of other tests we can still state at the end that it is in fact "random". The Type II error is harder to manage, because here a "not random" source passed the test it should not have. To lower the probability of the Type II error we have to choose an acceptable value for α and for the length of the sequence. The above mentioned information can also be interpreted as a so called p-value. The p-value is between 0 and 1 and it is the probability of getting results at least as extreme as the ones observed, given that the null-hypothesis is correct. In other words it is a metric showing how strong our evidences supporting the null-hypothesis are. To use the p-value we compare it to α and if it is below it we reject $H_0$. It is important to note here that α is used as a lower and 1- α is used as an upper bound and the p-values obtained throughout the test should follow a uniform distribution as well.

When we want to measure the randomness of a given bit sequence one test can only look at one property of the sequence. Therefore we need multiple tests which we can use and we need them to be different (in the sense that they are testing different properties). To solve the issue certain test were grouped together into a so-called test suite. Some of the suites are defined by standards, other are organized by various people.

An example for a standardized test suite is the NIST STS (National Institution of Standards and Technology Statistical Test Suite) [17] which consist of 15 different test and used widely in the world. Another test is the Diehard [18] and it is extended version the Dieharder [19] which are maintained by a community. The Dieharder suite consists of around 100 tests (it includes the NIST STS as well) which cover a large range of complexity. One of these test is the 32x32 binary rank test. This test takes 32 32-bit integer and builds a 32-by-32 matrix of 1s and 0s. Then it calculates the rank of this matrix and goes on for the next 32 number. Ranks less than or equal to 29 are rare, therefore they are treated as one rank. A Chi-squared test [20] is performed on the ranks 32, 31, 30, and ≤ 29,checking the uniformity of these rank groups.

One important question regarding these tests is when to use them. Using the tests must be part of the creation process of the generator. It is important during this time to run selected tests which might point to possible flaws in the design. After the generator is complete or when it is used in a real system monitoring the randomness of the output is vital for the underlying system which is using the numbers from the generator and for the maintenance of the generator as well. These tests can be used in real-time [21]. The NIST published several recommendations on which tests to use in which part of the generators lifecycle [22].

### *E. Extractors*

With the help of the statistical tests we mentioned in the previous section we can measure the quality of the numbers produced by a generator while we are building it. This helps us to see how far are we in the development. If we are not satisfied with the results, we can try to make the construction better with for example a new layout or with the help of more precise components. But there is point where we cannot improve the system further just by fine tuning because the physical implementation of an RNG cannot be 100% efficient or the physical phenomenon which the generator is based on hasn't got a high enough entropy. This means that we have to find another way to improve the quality of the generated numbers which comes after the generation phase. This is the post-processing, where we aim to improve the original output of the generator by making a new with better properties.

During post-processing we use extractor functions or algorithms. Their main goal is to extract as much entropy from the original source as possible and to create a new output whose entropy is as close to the original source as possible and has a better quality [23]. Previously we mentioned that a good random number generator is close to a truly random source or indistinguishable from it. Now we will define what this means. The distance of two random variable can be written as:

$$d(X,Y) = \max_{a \in A} |P_X(a) - P_Y(a)|$$

where $X$ and $Y$ are random variables of the same sample space $A$. If we think about our generator and a truly random source as a random variable can modify the definition to this:

$$d(X, U) = \max_{a \in A} |P_X(a) - P_U(a)| \leq \varepsilon.$$

In this inequality $X$ is random variable (our generator), $U$ is a random variable representing the uniform distribution (a truly random source) and ε is an upper bound for the distance. If $X$ satisfies this inequality we say that $X$ is ε uniform.

The next step is to measure the entropy of the source, because the main objective of the extractors is to extract as much entropy as possible and we need a way to compare the new output to the old one. There are different ways to measure the entropy for example the Shannon entropy but in the case of extractors the min-entropy is the mostly used version.

The definition of the min-entropy is the following:

$$H_\infty(X) = \min_x \left\{ \log\left(\frac{1}{P(X=x)}\right) \right\}$$

where log is the base 2 logarithm and $X$ is a random variable. With the help of this definition we can calculate the minimum entropy for the source if we want an ε-uniform bit sequence with length $m$. For the uniform distribution the probability of all possible outcome is $2^{-m}$. This means that the min-entropy is $m$ and this is the value we want to reach (or get close to it).

We can distinguish different extractors. The first group is the deterministic extractors. These extractors use a source denoted with $C$, a min-entropy, an input with a length of $n$ and have an output with length $k$ and of course they are ε-uniform. Because they are deterministic the output only depends on the input, this means that for the same input they will produce the same output.

The second group is the so-called seeded extractors. They have the same properties as the deterministic extractors, but they also have a seed with length $d$. The seed is used as an initialization vector just like with a hash function for example. During the creation of the new output these extractors are using seed as well as the input. This means that the same input won't result in the same output (of course if the seed is the same it will). The seed has to be a random sequence because only then will it provide the desired effect of altering the output in a hard to reverse way. But producing a long random sequence could be a hard task, therefore we want to minimize the length of seed while at same time maximize the possible length of the output.

In our testing we chose and implemented 8 extractor algorithms. The extractors we picked cover a wide range of different properties. We have simple ones, which manipulate the bits with logical operators to produce the output. But we also have more complex algorithms, which use techniques that are widely used in cryptography for example. Now we would like to introduce some of these extractors.

*1) The XOR Operator as an Extractor*

The XOR logical operator is one of the most used operator in computer science ranging from RAID technology to cryptography but it can also be used as a very simple extractor [24].

The XOR operator can be used effectively to lower the bias of the source but only if the bits are independent. The easiest way to use this extractor is to go over the original output of the generator and use the XOR on the bits in pairs. This means that the new output will have half the length of the original one. We can go further an use the XOR $n$ times always using the new output as the input for the next XOR. Doing so will lower the length of the generated output at the end $1/(n+1)$ times the original.

Although this extractor is very simple, can lower the bias of the source and can be quickly computed, it is not used, because the independency of the output bits cannot be guaranteed every time and it has a heavy effect on the length of the output (therefore the possible bitrate of the generator).

*2) The Von Neumann Extractor*

The Von Neumann extractor was created by John Von Neumann and it is the first extractor to be created [25]. Because it is the first extractor its main aim is to eliminate the bias of the source (like the XOR).

The operation of the algorithm is very simple, but just like at the XOR it is important that the bits are independent. It takes two bits as input and based on the values of these two it produces one or no bit. If the two bits are equal it discards the two bits. If they different it will give out the first one as the output. For a uniform source the new output will have the quarter of the length of the original.

The Von Neumann extractor has the same problems as the XOR. Although it is easy to use and it can eliminate the bias, it has a heavy toll on the length of the output.

*3) Other variants of the Von Neumann extractor*

Since the Von Neumann extractor was the first extractor, many have modified its operation. The two main problems the original design had are that it discards to many bits of the original bitstream and only has 2 bit long input. To overcome these issues the iterating [26] and the $N$ bit Von Neuman extractors have been created [27][28].

In the case of the Iterating Von Neumann the original extractor is used as a building block. The discarded bits are reused as new input, but before this they are modified with different operators. For the N bit Von Neumann extractor the original design was extended in such a way that the length of the input can be longer than two bits.

*4) H Function*

The $H$ function was created by Markus Dichtl [29] and just like the previous algorithms this extractor can also be simply implemented with logical gates, but compared to them it can achieve better result (as we will see in the tests).

It takes 16 bits as an input and gives out 8 bits as output and presumes that the bits are independent. In this area it is similar to the XOR. The algorithm works in the following way: We take the input bits and make two groups. The first is $a_1$ which is the first 8 bit, the second is $a_2$ which is the next 8 bit. The output of the algorithm is

$$H(a_1, a_2) = \left(a_1 \text{ XOR } \text{rotate\_left}(a_1, 1)\right) XOR\ a_2.$$

Where rotate_left($a_1$,1) means rotating the $a_1$ to the left with 1 step by taking the leftmost bit and putting it in the rightmost position.

Although the $H$ function produces a new sequence with half the length of the original one, it can better reduce the bias compared to the XOR operator. It can be implemented simply with logic gates and it is very efficient to use.

*5) Hash Function As Extractors*

Hash functions were not designed with the intent to be used as extractors but today they can be used as extractor algorithms for example during key derivation in cryptography [30].

A deterministic function which takes an $m$ bit length input and gives out an $n$ bit length output have to have specific properties to be called a hash function. These include collision

resistance, the avalanche effect, one-way property etc. The properties of a hash function make it a good choice for extraction. The output values are uniformly distributed and one bit difference between two inputs result in a bigger difference between the outputs. Other than that the length of the output can be the same as the input therefore the bitrate of the generator does not change. The physical implementation of a hash function can be achieved with good efficiency because there are specific hardware components which are designed for the fast computation of specific hash functions.

One of these hash functions is the Toeplitz hashing [31], where a Toeplitz matrix is used during extraction where the input bits (divided into smaller groups) as a vector is multiplied with this matrix.

*6) Using S-boxes as extractors*

Substitution boxes (S-boxes) are mostly used in symmetric key encryption algorithms. For example they are used in the Data Encryption Standard (DES) [32]. They take an $m$ long bits of input and give out bits of $n$ length, substituting the input for the output. As their main goal in encryption systems is to increase confusion, they can be used as extractors [33].

## III. Testing The Generators And The Extractors

Our main objective during the testing was to find out how can the different extractors improve the quality of the original outputs from the generator. During the testing of the two generators we firstly implemented the extractors we previously introduced. For running the test we used the Dieharder test suite which we introduced in the previous chapter. This test suite has a command line program which can be used on Linux based operating systems and provides a variety of possible arguments which can be given to the test [34].

In the implementation phase we decided that for the N bit Von Neumann extractor we will implement the N=4 case and for the iterative Von Neuman extractor we use 2 iteration. For the extractor which uses the Toeplitz matrix we generated Toeplitz matrix with the help of a PRNG. For the S-boxes we used the one which can be found in DES. For the hash function we chose the SHA-256.

After we implemented the extractors we had to choose the tests we wanted to run. We chose 19 test from the Dieharder test suite from which 16 was part of the Dierharder and 3 was part of the NIST STS. We only chose this subset of the Dierharder tests, because the generators were already tested with the NIST STS in previously published paper [21] and our main goal was to demonstrate the effect of the extractors on the original output. Therefore the results we will be presenting in the following subsections cannot be taken as a thorough statistical test of the generators.

After we chose the tests we set up the testing environment. We gathered data from the two generators. In case of the generator which is based on the arrival times of photons the size of the data was bigger. After this we ran the tests on the original output as well as the new ones which were produced by the 8 implemented extractor. We summarized the result in tables. In the rows we can see the tests, in the columns we can see the name of the tested outputs. If the generator PASSED the given test we can see the p-value it has achieved, if it failed it we can see an "F".

### A. Testing the ASE generator

The first generator we tested was the ASE generator. First of all we have to note that during the creation of the original output we deliberately introduced oversampling into the creation process. This resulted in a higher bitrate, but as we will see it heavily effected the quality of the numbers.

Table 1 shows the results of the original output as well as 4 simple extractors. We can see the effect of the oversampling. The original output could only pass 1 test out of 19. The simple extractors could slightly improve the quality, only 1 or 2 more tests were successful with their help. This correlates with the previously mentioned information about these extractors.

Table 2 shows the results of the 4 more complex extractors. As we can see they performed much better compared to the previous ones. The $H$ function performed really good considering it is simple construction and the hash function could almost eliminate all the failed tests. During the testing of this generator, we found that if there is a problem in the creation process (here, for example oversampling) with the help of extractors we cannot eliminate it perfectly, but we can mitigate the effect it has on the quality of the numbers. This is important, because there could be an underlying system which uses the number created by the generator and it requires a high bitrate. If we can only provide the desired bitrate with oversampling then the extractors could help us meet some of the quality requirements.

### B. Testing The Generator Based On The Arrival Times Of Photons

The second generator we tested was the one which is based on the arrival times of photons.

Table 3 shows the results of the original and the 4 simple extractors. We can see that the original output achieved a good result, only 2 out 19 tests failed. The explanation for the 2 failed test is the minimal inaccuracy of the hardware components in the generator (for example the photon sensor). The extractors couldn't improve the quality of the output to a perfect case but the XOR for example only failed 1 test. Table 4 shows the results for the second group of extractors. As we can see they performed better. The $H$ function and the hash function were able to achieve a perfect result, eliminating the 2 failed test in the original one. The other 2 extractor achieved good results as well. We can conclude from the testing of this generator that with the help of extractors we can eliminate the negative effects the physical implementation introduces to the system.

## IV. Conclusion

In our paper we presented the concept of QRNGs and also briefly presented two of techniques which are used in these generators during the creation of the numbers. We introduced selected tests which can be used to determine the quality of the generated numbers on a probabilistic basis. After this we presented the idea of extractors and showed where they fit into in the lifecycle of the generator.

We have presented 8 extractors, together with their operation and listed some of their strengths and weaknesses. In the last part of our paper we concentrated on these 8 extractors and their effect on the quality of the outputs produced by two QRNGs. We ran several statistical tests to determine how the extractors effect the properties of the numbers and presented the outcome of these tests on both generators. After we performed the tests we concluded that post-processing can be utilized to enhance the output of the generators, but we have to select the right extractors as not all of them can perform equally. While there are ones which can greatly increase the number of passed tests, they can also decrease the possible output speed of the generator. Another important property we found was that in case of a miscalibration during the generation process inside the generator the extractors can to a certain degree mitigate the negative effects. The chosen tests do not cover all the aspects which are needed for a deep statistical testing of the complete post-processing with these extractors, therefore as a future improvement it can be studied.

TABLE I
THE RESULTS FOR THE GENERATOR BASED ON AMPLIFIED SPONTANEOUS EMISSION PART 1.

| **Name of the test** | **H function** | **S-box** | **Toeplitz-matrix** | **SHA256** |
|---|---|---|---|---|
| *diehard_birthdays* | 0.946 | 0.136 | F | F |
| *diehard_operm* | 50.187 | 0.359 | F | 0.414 |
| *diehard_rank_32x32* | 0.467 | 0.101 | F | 0.861 |
| *diehard_rank_6x8* | 0.419 | F | F | 0.497 |
| *diehard_bitstream* | F | F | F | 0.822 |
| *diehard_opso* | F | F | F | 0.231 |
| *diehard_oqso* | 0.010 | F | F | F |
| *diehard_dna* | 0.035 | F | F | 0.382 |
| *diehard_count_1s_str* | F | F | F | 0.361 |
| *diehard_count_1s_byt* | 0.196 | F | F | 0.406 |
| *diehard_parking_lot* | 0.061 | F | F | 0.011 |
| *diehard_2dsphere* | 0.872 | F | F | 0.564 |
| *diehard_3dsphere* | 0.844 | 0.097 | F | 0.823 |
| *diehard_squeeze* | F | F | F | 0.954 |
| *diehard_runs* | F | 0.475 | 0.522 | 0.198 |
| *diehard_craps* | F | F | F | F |
| *sts_monobit* | F | F | F | 0.062 |
| *sts_runs* | F | F | F | 0.322 |
| *sts_serial* | F | F | F | 0.563 |

TABLE II
THE RESULTS FOR THE GENERATOR BASED ON AMPLIFIED SPONTANEOUS EMISSION PART 2.

| Name of the test | Original | XOR | Von Neumann | Iterating Von Neumann | 4 bit Von Neumann |
|---|---|---|---|---|---|
| *diehard_birthdays* | F | F | F | F | F |
| *diehard_operm5* | F | 0.080 | F | F | 0.078 |
| *diehard_rank_32x32* | 0.565 | 0.042 | 0.036 | 0.479 | 0.215 |
| *diehard_rank_6x8* | F | F | F | F | F |
| *diehard_bitstream* | F | F | F | F | F |
| *diehard_opso* | F | F | F | F | F |
| *diehard_oqso* | F | F | F | F | F |
| *diehard_dna* | F | F | F | F | F |
| *diehard_count_1s_str* | F | F | F | F | F |
| *diehard_count_1s_byt* | F | F | F | F | F |
| *diehard_parking_lot* | F | F | F | F | F |
| *diehard_2dsphere* | F | F | F | F | F |
| *diehard_3dsphere* | F | F | F | F | F |
| *diehard_squeeze* | F | F | F | F | F |
| *diehard_runs* | F | 0.631 | 0.767 | 0.363 | 0.617 |
| *diehard_craps* | F | F | F | F | F |
| *sts_monobit* | F | F | F | F | F |
| *sts_runs* | F | F | F | F | F |
| *sts_serial* | F | F | F | F | F |

TABLE III
THE RESULTS FOR THE GENERATOR BASED ON THE ARRIVAL TIMES OF PHOTONS PART 1.

| Name of the test | Original | XOR | Von Neumann | Iterating Von Neumann | 4 bit Von Neumann |
|---|---|---|---|---|---|
| *diehard_birthdays* | 0.564 | 0.810 | 0.536 | 0.100 | 0.853 |
| *diehard_operm5* | 0.952 | 0.163 | 0.066 | 0.463 | 0.360 |
| *diehard_rank_32x32* | 0.313 | 0.948 | 0.576 | 0.327 | 0.917 |
| *diehard_rank_6x8* | 0.121 | 0.817 | 0.606 | 0.884 | F |
| *diehard_bitstream* | 0.305 | 0.121 | 0.524 | 0.297 | F |
| *diehard_opso* | F | 0.448 | 0.227 | 0.068 | F |
| *diehard_oqso* | 0.927 | 0.691 | 0.987 | 0.134 | F |
| *diehard_dna* | 0.549 | 0.602 | 0.972 | 0.533 | F |
| *diehard_count_1s_str* | 0.927 | 0.976 | 0.935 | 0.540 | F |
| *diehard_count_1s_byt* | 0.941 | 0.875 | 0.674 | 0.821 | F |
| *diehard_parking_lot* | 0.863 | 0.100 | 0.273 | 0.012 | F |
| *diehard_2dsphere* | 0.576 | 0.336 | 0.754 | F | F |
| *diehard_3dsphere* | 0.574 | 0.982 | 0.575 | 0.031 | 0.246 |
| *diehard_squeeze* | 0.114 | 0.498 | 0.043 | 0.013 | F |
| *diehard_runs* | 0.176 | 0.805 | 0.684 | 0.284 | 0.419 |
| *diehard_craps* | 0.307 | 0.711 | F | 0.737 | F |
| *sts_monobit* | 0.360 | F | 0.762 | 0.249 | F |
| *sts_runs* | F | 0.892 | F | F | F |
| *sts_serial* | 0.570 | 0.459 | 0.548 | 0.453 | F |

TABLE IV
THE RESULTS FOR THE GENERATOR BASED ON THE ARRIVAL TIMES OF PHOTONS PART 2

| Name of the test | H function | S-box | Toeplitz-matrix | SHA256 |
|---|---|---|---|---|
| *diehard_birthdays* | 0.307 | 0.732 | 0.991 | 0.065 |
| *diehard_operm5* | 0.359 | 0.912 | 0.336 | 0.241 |
| *diehard_rank_32x32* | 0.243 | 0.383 | 0.313 | 0.007 |
| *diehard_rank_6x8* | 0.304 | F | 0.121 | 0.438 |
| *diehard_bitstream* | 0.115 | F | 0.145 | 0.120 |
| *diehard_opso* | 0.880 | F | 0.219 | 0.354 |
| *diehard_oqso* | 0.858 | F | 0.524 | 0.412 |
| *diehard_dna* | 0.482 | 0.351 | 0.368 | 0.734 |
| *diehard_count_1s_str* | 0.575 | F | 0.280 | 0.578 |
| *diehard_count_1s_byt* | 0.037 | F | 0.520 | 0.280 |
| *diehard_parking_lot* | 0.547 | 0.109 | 0.576 | 0.818 |
| *diehard_2dsphere* | 0.784 | 0.126 | 0.304 | 0.792 |
| *diehard_3dsphere* | 0.877 | 0.524 | 0.565 | 0.501 |
| *diehard_squeeze* | 0.905 | F | 0.348 | 0.274 |
| *diehard_runs* | 0.780 | 0.555 | 0.463 | 0.518 |
| *diehard_craps* | 0.644 | 0.392 | 0.733 | 0.384 |
| *sts_monobit* | 0.898 | 0.041 | 0.403 | 0.989 |
| *sts_runs* | 0.632 | F | F | 0.520 |
| *sts_serial* | 0.576 | F | 0.468 | 0.565 |

## REFERENCES

[1] Sara El Gaily, Sándor Imre, "Quantum Optimization of Resource Distribution Management for Multi-Task, Multi-Subtasks", Infocommunications Journal, Vol. XI, No 4, December 2019, pp. 47-53. DOI: 10.36244/ICJ.2019.4.7

[2] Sandor Imre, Laszlo Gyongyosi. "Advanced quantum communications: an engineering approach", Wiley, 2012, ISBN: 978-1-118-00236-0, DOI: 10.1002/9781118337462

[3] Laszlo Gyongyosi, Sandor Imre, Hung Viet Nguyen: "A Survey on Quantum Channel Capacities", IEEE [11] Communications Surveys and Tutorials, IEEE, DOI: 10.1109/COMST.2017.2786748, 2018.

[4] Laszlo Gyongyosi, Laszlo Bacsardi and Sandor Imre, "A Survey on Quantum Key Distribution", Infocommunications Journal, Vol. XI, No 2, June 2019, pp. 14-21. DOI: 10.36244/ICJ.2019.2.2

[5] Mario Stipcevic, "Quantum random number generators and their applications in cryptography", Proc. SPIE 8375, Advanced Photon Counting Techniques VI, 837504 (2012); DOI: 10.1117/12.919920

[6] Herrero-Collantes, Miguel, and Juan Carlos Garcia- Escartin, "Quantum random number generators." Reviews of Modern Physics 89.1 (2017), DOI: 10.1103/revmodphys.89.015004

[7] Ma, Xiongfeng, et al. "Post-processing for quantum random-number generators: Entropy evaluation and randomness extraction." Physical Review A 87.6 (2013), DOI: 10.1103/physreva.87.062327.

[8] Qi, Bing. "True randomness from an incoherent source." Review of Scientific Instruments 88.11 (2017), DOI: 10.1063/1.4986048

[9] Zhang, Xiao-Guang, et al. "Note: Fully integrated 3.2 Gbps quantum random number generator with real-time extraction." Review of Scientific Instruments 87.7 (2016), DOI: 10.1063/1.4958663

[10] Shakhovoy, Roman, et al. "Quantum noise extraction from the interference of laser pulses in an optical quantum random number generator." Optics express 28.5 (2020), DOI: 10.1364/oe.380156

[11] Martínez, Aldo C., et al. "Advanced statistical testing of quantum random number generators." Entropy 20.11 (2018), DOI: 10.3390/e20110886

[12] Kelsey, John, et al. "Cryptanalytic attacks on pseudorandom number generators." International workshop on fast software encryption. Springer, Berlin, Heidelberg, (1998) DOI: 10.1007/3-540-69710-1_12

[13] Gras, Gaëtan, et al. "Quantum entropy model of an integrated QRNG chip." (2020) arXiv preprint arXiv:2011.14129

[14] Michael A. Wayne, Evan R. Jeffrey, Gleb M. Akselrod and Paul G. Kwiat: "Photon arrival time quantum random number generation", Journal of Modern Optics Vol. 56, No. 4, 20 February 2009, 516–522, DOI: 10.1080/09500340802553244

[15] Jie Yang, Fan Fan, Jinlu Liu, Qi Su, Yang Li, Wei Huang, and Bingjie Xu, "Randomness Quantification for Quantum Random Number Generation Based on Detection of Amplified Spontaneous Emission Noise" Quantum Science and Technology, Vol. VI, No. 1, 2020, DOI: 10.1088/2058-9565/abbd80

[16] Ádám Marosits, Ágoston Schranz and Eszter Udvary, "Amplified spontaneous emission based quantum random number generator", Infocommunications Journal, Vol. XII, No 2, July 2020, pp. 12-17. DOI: 10.36244/ICJ.2020.2.2

[17] "NIST SP 800-22: Documentation and Software" https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software, (Last visit: 23 Feb 2021).

[18] G. Marsaglia, "The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness". Florida State University. 1995. archived on 2016-01-25." https://web.archive.org/web/20160125103112/http:/stat.fsu.edu/pub/diehard/.sdfsdf

[19] Robert G. Brown. "Robert G. Brown's General Tools Page", https://webhome.phy.duke.edu/~rgb/General/dieharder.php, (Last visit: 23 Feb 2021.)

[20] NIST/SEMATECH e-Handbook of Statistical Methods, "Chi-Square Goodness-of-Fit Test" https://itl.nist.gov/div898/handbook/eda/section3/eda35f.htm (Last visit: 14 May 2021)

[21] Balazs Solymos, Laszlo Bacsardi, "Real-time Processing System for a Quantum Random Number Generator", Infocommunications Journal, Vol. XII, No 1, March 2020, pp. 53-59. DOI: 10.36244/ICJ.2020.1.8

[22] "SP 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation," https://csrc.nist.gov/publications/detail/sp/800-90b/final, (Last visit: 23 Feb 2021)

[23] Salil P. Vadhan, "Pseudorandomness", Foundations and Trends in Theoretical Computer Science: Vol. 7: No. 1–3, 2012, pp. 1-336.

[24] R. B. Davies, "Exclusive OR (XOR) and hardware random number generators", Tech. Rep., 2002. [Online], Available: http://www.robertnz.net/pdf/xor2.pdf (Last visit: 23 Feb 2021)

[25] John Von Neumann, "Various techniques used in connection with random digits", National Bureau of Standards Applied Math Series 12, pp. 36–38., 1951

[26] Y. Peres, "Iterating Von Neumann's Procedure for Extracting Random Bits", Ann. Statist., pp. 590–597., 1992, DOI: 10.1214/aos/1176348543

[27] P. Elias, "The efficient construction of an unbiased random sequence" Ann. Math.Statist., pp. 865–870., 1972, DOI: 10.1214/aoms/1177692552

[28] Ruilin Zhang, Sijia Chen, Chao Wan, Hirofumi Shinohara, "High-Throughput Von Neumann Post- Processing for Random Number Generator", 201850 International Symposium on VLSI Design Automation and Test (VLSI-DAT), pp.1-4, 2018, DOI: 10.1109/vlsi-dat.2018.8373253

[29] Markus Dichtl, "Bad and Good Ways of Post-Processing Biased Physical Random Numbers" 14th International Workshop, FSE2007, Luxembourg, Luxembourg, March 26-28, 2007, DOI: 10.1007/978-3-540-74619-5_9

[30] Wegman, Mark N., and J. Lawrence Carter, "New hash functions and their use in authentication and set equality." Journal of computer and system sciences 22.3 (1981): 265-279., DOI: 10.1016/0022-0000(81)90033-7

[31] Krawczyk H., "New Hash Functions for Message Authentication.", Advances in Cryptology, EUROCRYPT (1995) Lecture Notes in Computer Science, vol 921. Springer, Berlin, Heidelberg. DOI: 10.1007/3-540-49264-X_24

[32] "Data Encryption Standard (DES)" https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25 (Last visit: 23 Feb 2021)

[33] AVAROĞLU, ERDİNÇ, and Taner Tuncer. "A novel S-box-based post-processing method for true random number generation." Turkish Journal of Electrical Engineering & Computer Sciences 28.1 (2020): 288- 301., DOI: 10.3906/elk-1906-194

[34] "dieharder - Linux man page" https://linux.die.net/man/1/dieharder (Last visit: 23 Feb 2021)

**Botond L. Márton** received his B.Sc. degree in computer engineering from Budapest University of Technology and Economics (BME) in early 2021. He is currently pursuing his M.Sc. at BME. Currently he is involved in a quantum key distribution project at the university. His research interests are quantum computing and quantum communications.

**Dóra Istenes** received her B.Sc. degree in the beginning of 2021 in Computer Science Engineering from the Budapest University of Technology and Economics (BME). She started her M.Sc. studies at BME in the same year. Her current research interests are quantum computing and communications.

**László Bacsárdi** (M'07) received his MSc degree in 2006 in Computer Engineering from the Budapest University of Technology and Economics (BME) and his PhD in 2012. He is corresponding member of the International Academy of Astronautics (IAA). Between 2009 and 2020, he worked at the University of Sopron, Hungary in various positions including Head of Institute of Informatics and Economics. Since 2020, he is associate professor at the Department of Networked Systems and Services, BME and head of Mobile Communications and Quantum Technologies Laboratory. His current research interests are quantum computing, quantum communications and ICT solutions developed for Industry 4.0. He is chair of the Telecommunications Chapter of the Hungarian Scientific Association for Infocommunications (HTE), Vice President of the Hungarian Astronautical Society (MANT). Furthermore, he is member of AIAA, IEEE and HTE as well as alumni member of the UN established Space Generation Advisory Council (SGAC). In 2017, he won the IAF Young Space Leadership Award from the International Astronautical Federation.