View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repository of the Academy's Library

Real-time Processing System for a Quantum Random Number Generator

Balazs Solymos and Laszlo Bacsardi, *Member, IEEE*

***Abstract*—Quantum random number generators (QRNG) provide quality random numbers, which are essential for cryptography by utilizing the unpredictable nature of quantum mechanics. Advancements in quantum optics made multiple different architectures for these possible. As part of a project aiming to realize a QRNG service, we developed a system capable of providing real-time monitoring and long term data collection while still fulfilling regular processing duties for these devices. In most cases, hardware validation is done by simply running a battery of statistical tests on the final output. Our goal, however, was to create a system allowing more flexible use of these tests, realizing a tool that can also prove useful during the construction of our entropy source for detecting and correcting unique imperfections. We tested this flexibility and the system's ability to adequately perform the required tasks with simulated sources while further examining the usability of available verification tools within this new custom framework.**

***Index Terms*—quantum computing, quantum random number generation, statistical testing.**

I. Introduction

RANDOMNESS is used as a resource in a wide variety of applications nowadays. Numerical simulations, as well as several cryptographic use cases all, depend on quality random numbers for reliable operation [1], presenting a need for quality high-speed generation schemes. The inherently unpredictable nature of quantum mechanics poses an attractive foundation for potential solutions. While most quantum computing applications apart from quantum key distribution [2] are still in the experimental phase, quantum random number generation is already well established, with existing commercial products [3]. Advancements in quantum optics made many different theorized realizations feasible [4], possibly leading to new and better generation methods.

Under the framework of a Hungarian quantum technology project, our goal is to realize a quantum random number service which provides reliable random numbers. For this, a physical entropy source has to be built, paired with an adequate processing system to provide verified, quality output. While most processing systems mainly only consist of a single algorithm to extract a uniform output from the raw data coming from the hardware, our goal was to create one that can also realize real-time monitoring and collect long term statistics, allowing it to also aid the development of the physical architecture by providing a custom tool capable of detecting possible imperfections arising due to construction mistakes. Part of this project, there are multiple proposed physical entropy sources under construction currently. Our main goal was to create a system that is capable of supporting all of these. In this paper, we present one such system. Furthermore, due to the required flexibility, our solution can also be used with most other types of generators.

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

DOI: 10.36244/ICJ.2020.1.8

II. Random number generators

A. Generation approaches

Generally, random number generators can be categorized into two main groups: deterministically operating algorithm based pseudo-random number generators (PRNG) and true random number generators (TRNG), which utilize some physically proven nondeterministic phenomenon as their entropy source. PRNGs are usually easy to use and can provide on-demand high-speed bit generation, however, their deterministic operation presents an exploitable vulnerability to potential attackers [5][6]. Knowing a particular algorithm, it's inner state can be deducted after collecting a sufficiently large amount of output data. Once this state is known, the operation of the generator can be accurately simulated, all future and present outputs can be predicted, thereby rendering them useless for most use cases. To combat this, these generators can be used in conjunction with other, more easily accessible, albeit worse quality outside entropy sources [7], using their limited entropy to reinitialize their inner state. In this mode, they effectively function as randomness extractors (or more accurately randomness expanders) for weak entropy sources [8]. True random number generators, on the other hand, have no such inner state governing their operation. Each generated bit should be independent. Typically, these generators work by sampling some appropriate physical phenomena like radioactive decay, photoelectric effects or noises like avalanche, thermal or shot noise. The source of randomness for all of the mentioned cases can be traced back to the laws of quantum mechanics. This is good news, as quantum unpredictability has been experimentally verified numerous times [9][10][11]. The main challenge is the actual error-free construction and operation of these generators. No unwanted unknown outside bias or noise polluting the measurements is allowed. Another limiting factor for this approach can also be speed. The examined state of the underlying phenomena often can't change instantaneously. To avoid correlation between samples, adequate restrictions to sampling frequency have to be enforced. Their operation mode, therefore, is often referred to as "blocking" compared to the "non-blocking" nature of PRNGs, meaning that for each

batch of output data the user first has to wait for enough of the sampled physical events to occur, thereby presenting a speed limit independent of processing power. Detecting faulty operation states can present another potential problem. Due to the expected random nature, all possible output strings can occur during normal operation, making it impossible to tell with full certainty if a given output is the product of nominal or faulty operations. Fortunately, with the help of correctly designed statistical tests, some statements can still be made even in this case.

B. Planned generator architectures

Improvements in optical technologies have led to the emergence of several possible quantum random number generator architectures [4]. Most QRNGs today, as well as the proposed entropy sources our system needs to be able to support, rely on quantum optics as the quantum nature of states of light allows for many different implementations. For the processing system, information about the most probable error cases associated with these sources is relevant since these errors are expected to be detected. Another important factor is the maximum generation rate with which the system needs to be able to keep up. We briefly examine these for the proposed sources our system is designed to work with in the future.

1) Branching path generator: The first proposed architecture is a variant of one of the earliest quantum optical solutions [12]. It puts a single photon into path superposition using a beam splitter. Assuming this splitter has an ideal 50:50 split ratio, with detectors for each possible path, the resulting which-path information provides uniform randomness. With the introduction of some delay on only one of the paths and by rejoining them later, the difference of the possible paths can be seen from their arrival times. This allows for using only one detector, which is advantageous because bias coming from the difference in detectors becomes a non-factor. Notable possible error and bias sources in this construction are:

- Imperfection of the beam splitter can introduce bias.
- The additional delaying element in one of the paths means a higher chance of photon loss, leading to slightly reduced detection rates from that path, causing bias.
- Dark count rate of the detector.
- Source producing multiple or no photons.
- Losses in other parts.

Imperfections that lead to no detection or multiple detections (last two points) only affect output rate, not quality. Therefore, only effects that introduce unwanted bias need to be detected and corrected. Typical achievable bit rates for these generators are in the Mbps range.

2) Photon counting generator: Using a continuous light source, one can follow an approach similar to radioactive decay based generators [13]. The expected number of photons that arrive from the source during a given window of time follows a Poisson distribution. This is not the sought after uniform distribution, but various methods exist to transform it into that [14]. Notable possible processing challenges for this case can be the following:

- Poisson instead of uniform distribution.

TABLE I
POSSIBLE OUTCOMES OF STATISTICAL TESTS

Reality	Conclusion	
	Accept H_0	Accept H_a
Data is random	No error	Type I error
Data is not random	Type II error	No error

- Other bias originating from imperfections, like less detection due to losses, detector efficiency, and dark count rate, etc...

These generators can reach speeds of 50 Mbps or more depending on used post-processing.

3) Time of arrival generator: Measuring the time difference between each detection instead of the number of photons, a similar scheme to the photon counting case can be realized. This statistic is also exponential, which is not the ideal uniform distribution, however, similarly to the previous case, there are options to remedy this. One way is to compare the time between the detection of the first and second, and second and third photons, then assign our output bit the result of this operation. Notable possible error sources are the following:

- Detector dead time.
- Accurate time measurement.
- Other bias originating from imperfections.

Generation speeds of these architectures can reach up to more than 100 Mbps [15], setting the highest expectation for real time testing in the processing system.

III. STATISTICAL TESTS

Defining randomness is more of a philosophical problem due to the very nature of it. Deciding with certainty if a given output is indeed random or not, is therefore quite problematic. Consequently, in most practical approaches, we settle for less than absolutely certain, but most probable. Being able to state for a given output that it's much more probably a result of faulty operations than not, is good enough.

Randomness is a probabilistic property, so we can use statistical tests to make these statements. Each test examines a statistical property and decides whether it's within our expectations or not. Since there are infinitely many ways a series can be non-random and one test only looks for one of these, usually multiple tests are run in parallel until we say the results are good enough. Another interesting fact is that for a truly random output the tests are expected to fail from time to time. The decision about a given bit string is generally made according to hypothesis testing. Our null hypothesis (H_0) is that the output originated from a perfectly random source. Our alternate hypothesis (H_a) is that it did not. We gather some evidence trying to decide between our competing hypotheses, typically by investigating some probabilistic value with known theoretical distribution calculated from our string (calculate a p-value), then check if it exceeds some a priori defined critical value. If it does we reject our null hypothesis (H_0) and accept the alternative (H_a). This is the general principle behind the widely used NIST tests [16] too.

Table I shows the possible resulting cases regarding our decisions and their relations to reality. Type I error happens when we decide a random sequence to be non-random. For real applications, this may result in speed loss, as detected error states are not allowed to reach the user, and execution of other unnecessary actions associated with error states. Since even during nominal operations some of these errors are expected, overreaction to these states should be avoided because dropping everything that is not in our predefined critical range can lead to a skewed distribution. Type II error is the failure to detect real errors. It is much more dangerous than Type I because it potentially means a vulnerability unknown to the operator, so no action is taken to correct it, thereby propagating this vulnerability to everything that uses the generated randomness later. This type of error is the one we strive to minimize. The probability of Type I and Type II errors share a relation to each other and the length of the examined string. A commonly chosen value for Type I error probability is 0.01, as recommended by the NIST Statistical Test Suite (STS).

The two most widely used statistical test collections nowadays are the NIST Statistical Test Suite (containing 15 tests) and the Dieharder [17] test collection, which is an extension of the 1995 Diehard [18] tests, containing more than a hundred different tests, including the ones already in the STS too. Running tests is computationally expensive, so when choosing a set of them to use, an additional goal is to have tests orthogonal to each other, meaning that each of them investigates independent properties from one another. This is explicitly stated in the STS documentation, while the Dieharder pack aims to provide the most all-round and deep examination possible. Other new, less adapted approaches for testing [19] also exist, but the common main problems with currently available solutions in our case are that they don't support flexible, real-time monitoring and are computationally expensive.

Tests are good for finding unexpected error sources, for reliable system validation, however, they cannot be used. Proper analysis of the built architecture should never be skipped. The NIST recommendations for generator design [20], [21], [22] aim to provide a reliable baseline for this topic. In them, they specify certain expectations to be met for architectures. The general layout of a generator according to this can be seen in Figure 1. These recommendations specify different tests to be run at the start of the system, during operation, and occasionally when requested from the outside, while the three main error states expected to be detected are: entropy decrease, source error, implementation error. Our system realizes all the steps in Figure 1 after digitization while fulfilling these recommendations.

IV. SYSTEM ARCHITECTURE

A. Specification

The processing system should take care of everything from digitization to providing the final output to the outside world. The following are the main points to fulfill and consider:

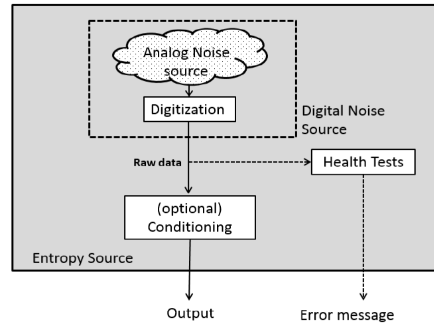

Fig. 1. General layout of a random number generator according to the NIST recommendation.

- Real-time monitoring: The system needs to detect potential errors when they happen during operation, and if they do, block the output. This requires real-time monitoring and testing of the bitstream coming from the hardware.
- Correct the potential bias of the input and perform the required processing to be able to provide a uniformly distributed output.
- Test the uniform output for processing errors.
- Long term validation of the whole system (hardware and processing): Store the required data for long term statistics and run specialized tests on them.
- Ability to easily change the processing system according to the specific needs of different hardware architectures. Enabling custom solutions tailored to the specifics of each physical source would also be beneficial, so flexibility is a plus.

Considering these goals, we focused on creating a system mainly to solve the real-time monitoring challenge while allowing it to be adapted to the needs of all three planned possible hardware layouts.

B. Realization

For the various expectations to be met, multiple tests and other processes need to be executed during various stages of processing. Two main approaches can be chosen here: all the needed components work together as part of a larger program, or all the components form smaller independent individual programs that can communicate and work together to get the same result. We chose the latter option as this allows for more flexibility. Furthermore, in the case of some software malfunction, only the corresponding smaller part is affected, leading to higher redundancy. The whole system is realized as a Linux virtual machine. This allows for relatively easy testing and development. Smaller processing parts can be run as individual daemons, permitting the use of tried and tested management tools offered by the system for communication and resource management. The block diagram of the relations between processes can be seen in Figure 2. Input data is read from the network via UDP. We assume that a safe local network is shared with the hardware so no additional safety measures are needed in this step. Should the need arise, different, more secure methods can easily be implemented.

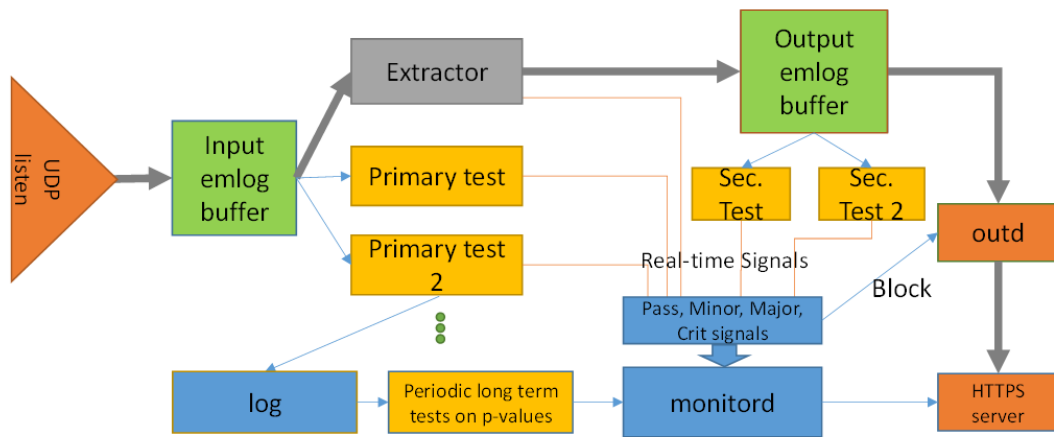

Fig. 2. Layout of the system and relations of the processes within. The thicker arrows show the path of the input data.

The data then gets read into a special cyclic emlog device buffer [23]. The cyclic nature here is important, as this allows for the independent operation of tests giving each process a separate file descriptor, thus allowing multiple different tests with different speeds to operate on the same buffer. (Solving the problem of synchronizing tests with different processing requirements and speed, leading to efficient resource utilization.) The size of this buffer can be chosen to be as big as 1 MB, meaning that even slower tests can always access sizeable continuous data. The processed stream is written to another cyclic buffer, from which the dedicated *outd* daemon can provide output to the outside world through https. With these two buffers, testing both the incoming raw data and processed output is possible. Components of the system are written in C/C++ to allow for easy adaptation of tests used in major open-source suites, which are mainly written in C. These components communicate with the *monitord* daemon responsible for monitoring the system via real-time signals while also logging their activities to their respective log files. A real-time status report is constructed in *monitord* with configurable parameters for deciding when to allow or block output. The log files can later be used for long term analysis of the whole system.

V. Tools available in the system

A. Extractors

Removing unwanted bias from the output is the main goal of entropy extractor algorithms. Several different of these exist, some even specially recommended for use with quantum generators [24]. During operation, these algorithms take in a chunk of bits and output a usually smaller chunk with better properties. They are mostly designed for correcting weaker entropy sources and are computationally expensive. When working with a specific good quality source some custom options are possible [25], but for our simulated case, more general solutions are preferred first. We used the SHA hash (recommended by NIST) for this purpose because hardware acceleration is supported for it in many modern processors.

B. Statistical tests

In the system, each statistical test can be done with a single function call, following a mostly standardized format. Each test gets the bits to test (also defining test length if there are more variants available), the critical threshold value for failure, and in some cases, other optional parameters, then returns its decision and the calculated p-value used for said decision. This permits us to easily define multiple tests to be run sequentially in our processes while also simplifying the code needed for singular test cases. For this to work and better support continuous operation, existing statistical tests had to be reimplemented using a custom bit container class. Readily available sources and sufficient existing documentation made this task straightforward. All 15 tests contained in the NIST STS have already been adapted this way to our architecture.

Different tests look for different vulnerabilities, but generally, they follow the same structure: first they transform the data into a more suitable format, then calculate a statistical value, which is later compared to a reference distribution resulting in a p-value. If this p-value is below some critical threshold, the test fails. By changing the reference distribution or the way the statistical value is calculated, tests can be created for non-uniform distributions too. This can be especially useful when our hardware has some mathematically describable known bias, making testing the raw, unprocessed, biased input possible. To demonstrate this, a generalized monobit test (expected ratio of ones and zeros can differ from 50:50) is also implemented in our system.

C. Long term statistics

Long term statistics can be calculated using relevant information collected from the processes, which is stored in the form of log files. Whenever a test is executed, the resulting p-value and the length of the tested string is saved. In some cases, other test specific auxiliary values are collected too. This can allow for the construction of tests similar to the one from which the data is collected, but spanning a wider range of data, effectively realizing an expanded version of the original. In the case of the monobit test, for example, it

means the following: The test uses an internal sum calculated from the difference between the number of ones and zeros in the examined sequence, adding +1 to it for each bit which is one, and -1 for each zero. The p-value is then calculated from the normalization of this sum by the square root of the sequence length. By saving this sum and the number of examined bits, an extended monobit test can be constructed for the whole log file, aggregating the information from all the smaller tests. Similar extended versions can be made from other tests that use easily extractable metrics for calculating their results. From the 15 STS tests these are:

- Runs test: sequence length, ratio of ones, number of runs.
- Test for the longest run of ones in a block: length and number of tested blocks, counts in internal cells.
- Binary matrix rank test: number of blocks, number of full rank and full rank-1 matrices.
- Cumulative sums (cusums) test: sequence length, largest excursion, excursion at the end of the sequence.
- Random excursions test: sequence length, internal excursion statistics, excursion at the end of the sequence.

Another more generalized way to summarize information is the use of KS tests [26][27]. The expected distribution of p-values is uniform, so statistical tests can be run to verify this. The KS test is one such test, examining the deviation of the actual results from this expectation and producing a new p-value accordingly. When enough of these new values are collected the test can be run on them again, resulting in p-values representing more and more tested data. Theoretically, for non-random sources, the results of these repeated tests tend to zero over time. Keeping track of how many bits of data each p-value represents and only testing values representing the same amount of data together, a hierarchical structure can be followed for aggregation. Ideally with this method limitless data can be summarized, however, due to limited computational precision and errors adding up more and more, this is not possible for practical use cases as the results get skewed by these factors. These operations are also computationally quite expensive, so depending on generator speed and available resources, suitable compromises for choosing logged data to be tested this way might have to be made.

VI. TESTING THE SYSTEM

Since the proposed hardware architectures in Section II-B are still under construction, we tested our system with available software PRNGs. For most test cases we used AES_RNG with the Crypto++ C++ library [28]. The main expectation towards the test generator is good enough short term behavior to pass initial randomness testing, which it satisfies, allowing us to test error detection capabilities. (The nominal operation of the generator must produce good enough randomness to not be seen as errors to the system.) In long-term statistics, the weaknesses of this generator can be seen however, as the results from KS testing start to tend to zero after some hours of operation. This is most probably due to the combination of using a deterministic algorithm with the relatively weak internal entropy source of the test computer. Interestingly, for the log data of short monobit tests, the resulting aggregated p-values reached zero for all KS tests (using 10000 samples each), while in other weak cases only a tendency towards zero is observed. This implies that this phenomenon is not due to the weakness of the generator. One possible explanation is that a monobit test processing an n bit long sequence can only have n different outputs as p-values, resulting in only n different samples for the KS tests, leading to a non-uniform distribution. The NIST input size recommendation for the monobit test is minimum 100 bits, however, results from our longer 1024 bit variant still exhibit this when used directly for long term statistics calculations. Using the special, extended version of the monobit test utilizing auxiliary logged information, creating p-values for bigger chunks of data before KS testing can solve this problem.

TABLE II
TEST RESULTS FOR THE 49:51 UNEVEN INPUT DISTRIBUTION

Test	Pass	Fail	Percent
cusums	109	3	2.7%
dft	3	0	0%
longest_runblock	108	4	3.6%
monobit	441	7	1.6%
monobit_long	6	9	60.0%
monobitblock	111	1	0.9%
runs	15	0	0.0%
serial	27	1	3.6%

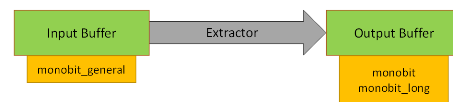

Fig. 3. Configuration for testing modified monobit test and error correction with extractor

To achieve effective real-time monitoring, possible hardware faults have to be detected as they occur. One such fault may be the sudden shift of the ratio of ones and zeros. This is easily simulated and can be a likely error type, especially for the first proposed hardware architecture. (For example, by some unexpected photon loss in one of the paths.) We simulated two error cases: one with the ratio of 45:55 and another with 49:51. In the first simulation, the difference from uniformity is so big that most tests fail (which is expected). The results from the second simulation can be seen in Table II.

Only faster tests (low latency) are shown in the table. We expect the monobit test to be the most sensitive to this error type, so we ran two instances of it. One with 1024 bit sequence length and another with 32768. The results show the longer version of the monobit test to be the most sensitive, while other test statistics are getting much closer to normal, surprisingly including its shorter version too. This demonstrates that for detecting even finer differences, longer, higher latency tests are needed, thereby calling for a compromise between detection speed and sensitivity.

With this simulated error, the generalized, modified for this non-uniformity variant of the monobit test can also be examined. A block diagram of this can be seen in Figure 3.

This layout also shows the correct operation of the imple-

mented extractor. We used the 45:55 ratio error case as it represents a bigger deviation to be fixed. The results can be seen in Table III.

TABLE III
RESULTS FROM THE TEST CASE CORRESPONDING TO FIGURE 3

Test	Pass	Fail	Percent
monobit	6473	63	1.0%
monobit_long	203	1	0.5%
monobit_general	46020	565	1.2%

The results meet our expectations: the failure rate is around 1% and the speed loss caused by the extractor is also apparent, as the monobit and monobit_general tests have the same sequence length, effectively showing the reduced bit rate as a difference in total runs. (Since the main goal is to test the operation of the system, a rather "safe" extraction method is used, not prioritizing extraction efficiency)

We also investigated the computational need of each test by measuring execution time inside the processes. These results are not hardware-independent but can be used to roughly compare the tests to each other. Results using a virtual machine utilizing 4 threads of a 5-year-old Intel i7-4710HQ laptop processor can be seen in Table IV. This benchmark can serve

TABLE IV
EXECUTION TIME OF TESTS AND CORRESPONDING PROCESSING SPEEDS.

Test	avg. time	min. time	avg. kbit/s	max. kbit/s
approxentropy	3737ns	3482ns	2140	2297
cusums	958ns	240ns	8348	33000
dft	838ns	507ns	9543	15770
excursions	562ns	527ns	14220	15770
linearcomplex	122433ns	121778ns	65	65
longest_runblock	406ns	173ns	19667	46022
matrix	1183ns	1119ns	6758	7143
mauer	1140ns	1052ns	7016	7599
monobit	107ns	18ns	74147	442810
monobit_long	135ns	18ns	58982	442810
monobitblock	948ns	384ns	8431	20791
runs	315ns	55ns	25378	143091
serial	4111ns	1978ns	1945	4043
template	88539	87741	90	91

as a starting point when choosing tests we want to run in real-time, as our limited computational resources may not be able to run certain tests at the necessary speeds to keep up with the incoming data stream. Simpler tests tend to run faster, potentially reaching speeds up to 400 Mbits/second even in this suboptimal environment, while for other, more complicated ones it can be said with high confidence that we won't be able to run them real-time with a decent entropy source.

VII. CONCLUSION

In this paper, we presented a system capable of realizing output based real-time monitoring for random number generators while providing possibilities for analyzing long term behavior too. In the absence of the planned entropy sources, we utilized readily available pseudo-random number generators to simulate probable situations the system needs to be able to handle, validating our design. While doing so, we also examined the future applicability of the tools available in this framework.

The next logical step is to pair the system with the proposed hardware as construction reaches a prototype state and optimize the monitoring tools used for this specific use case. Although the main goal was adequately supporting particular architectures, since the input of the system is only the raw data stream, it can be paired with other generators too. The flexible modular nature allows for the analysis of the tools within given a known, etalon generator. To effectively realize this potential in the future, however, some additional development aimed at easier usability for potential users not yet familiar with the code might still be needed.

ACKNOWLEDGMENT

The authors would like to additionally thank Attila Iván Gyula for his contributions and insight to the development of the main system architecture.

REFERENCES

[1] Y. Dodis, S. J. Ong, M. Prabhakaran, and A. Sahai, "On the (im) possibility of cryptography with imperfect randomness," in *45th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2004, pp. 196–205, DOI: 10.1109/FOCS.2004.44.

[2] L. Gyongyosi, L. Bacsardi, and S. Imre, "A survey on quantum key distribution," I*nfocommunications Journal*, vol. 11, no. 2, pp. 14–21, 2019.

[3] "QUANTIS random number generator by ID Quantique," https://www.idquantique.com/random-number-generation/products/quantis-random-number-generator/, (Last visit: Dec 5, 2019).

[4] M. Herrero-Collantes and J. C. Garcia-Escartin, "Quantum random number generators," *Reviews of Modern Physics*, vol. 89, no. 1, p. 015004, 2017, DOI: 10.1103/RevModPhys.89.015004.

[5] I. Goldberg and D. Wagner, "Randomness and the netscape browser," *Dr Dobb's Journal-Software Tools for the Professional Programmer*, vol. 21, no. 1, pp. 66–71, 1996.

[6] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your ps and qs: Detection of widespread weak keys in network devices," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 205–220, DOI: 10.5555/2362793.2362828.

[7] M. Santha and U. V. Vazirani, "Generating quasi-random sequences from slightly-random sources," in 2*5th Annual Symposium onFoundations of Computer Science, 1984*. IEEE, 1984, pp. 434–440, DOI: 10.1109/SFCS.1984.715945.

[8] R. Raz, "Extractors with weak random seeds," in *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. ACM, 2005, pp. 11–20, DOI: 10.1145/1060590.1060593.

[9] S. J. Freedman and J. F. Clauser, "Experimental test of local hiddenvariable theories," *Physical Review Letters*, vol. 28, no. 14, p. 938, 1972, DOI: 10.1103/PhysRevLett.28.938.

[10] A. Aspect, J. Dalibard, and G. Roger, "Experimental test of bell's inequalities using time-varying analyzers," *Physical review letters*, vol. 49, no. 25, p. 1804, 1982, DOI: 10.1103/PhysRevLett.49.1804.

[11] M. Giustina, M. A. Versteegh, S. Wengerowsky, J. Handsteiner, A. Hochrainer, K. Phelan, F. Steinlechner, J. Kofler, J.-A° . Larsson, C. Abell´an et al., "Significant-loophole-free test of bell's theorem with entangled photons," *Physical review letters*, vol. 115, no. 25, p. 250401, 2015, DOI: 10.1103/PhysRevLett.115.250401.

[12] T. Jennewein, U. Achleitner, G. Weihs, H. Weinfurter, and A. Zeilinger, "A fast and compact quantum random number generator," *Review of Scientific Instruments*, vol. 71, no. 4, pp. 1675–1680, 2000, DOI: 10.1063/1.1150518.

[13] H. Schmidt, “Quantum-mechanical random-number generator,” *Journal of Applied Physics*, vol. 41, no. 2, pp. 462–468, 1970.

[14] H. F¨urst, H. Weier, S. Nauerth, D. G. Marangon, C. Kurtsiefer, and H. Weinfurter, “High speed optical quantum random number generation,” *Optics express*, vol. 18, no. 12, pp. 13 029–13 037, 2010, DOI: 10.1364/OE.18.013029.

[15] Y.-Q. Nie, H.-F. Zhang, Z. Zhang, J. Wang, X. Ma, J. Zhang, and J.-W. Pan, “Practical and fast quantum random number generation based on photon arrival time relative to external reference,” *Applied Physics Letters*, vol. 104, no. 5, p. 051110, 2014, DOI: 10.1063/1.4863224.

[16] “NIST SP 800-22: Documentation and Software,” https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software, (Last visit: Dec 5, 2019).

[17] “dieharder by Robert G. Brown, Duke University Physics Department, Durham, NC 27708-0305 Copyright Robert G. Brown, 2019,” https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[18] G. Marsaglia, “”the marsaglia random number cdrom including the diehard battery of tests of randomness”. florida state university. 1995. archived from the original on 2016-01-25.” https://web.archive.org/web/20160125103112/http:/stat.fsu.edu/pub/diehard/.

[19] K. Kubícek, J. Novotný, P. Švenda, and M. Ukrop, “New results on reduced-round tiny encryption algorithm using genetic programming,” *Infocommunications Journal*, vol. 8, no. 1, pp. 2–9, 2016.

[20] “SP 800-90A Rev. 1 Recommendation for Random Number Generation Using Deterministic Random Bit Generators,” https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final, (Last visit: Dec 5, 2019).

[21] “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: Dec 5, 2019).

[22] “SP 800-90C (DRAFT) Recommendation for Random Bit Generator (RBG) Constructions,” https://csrc.nist.gov/publications/detail/sp/800-90c/draft,, (Last visit: Dec 5, 2019).

[23] “emlog – the embedded-system log-device, version 0.70, 10 july 2018,” https://github.com/nicupavel/emlog, (Last visit: Dec 5, 2019).

[24] A. De, C. Portmann, T. Vidick, and R. Renner, “Trevisan’s extractor in the presence of quantum side information,” *SIAM Journal on Computing*, vol. 41, no. 4, pp. 915–940, 2012,
DOI: 10.1137/100813683.

[25] X. Ma, F. Xu, H. Xu, X. Tan, B. Qi, and H.-K. Lo, “Postprocessing for quantum random-number generators: Entropy evaluation and randomness extraction,” *Physical Review A*, vol. 87, no. 6, p. 062327, 2013, DOI: 10.1103/PhysRevA.87.062327.

[26] A. Kolmogorov, “Sulla determinazione empirica di una lgge di distribuzione,” *Inst. Ital. Attuari, Giorn.*, vol. 4, pp. 83–91, 1933.

[27] N. Smirnov, “Table for estimating the goodness of fit of empirical distributions,” *The annals of mathematical statistics*, vol. 19, no. 2, pp. 279–281, 1948, DOI: 10.1214/aoms/1177730256.

[28] “Crypto++ library 8.2, a free c++ class library of cryptographic schemes,” https://www.cryptopp.com, (Last visit: Dec 5, 2019)..

Balazs Solymos received his B.Sc. degree in 2018, followed by his his M.Sc. degree in early 2020 in Electrical Engineering from the Budapest University of Technology and Economics (BME). He recently started pursuing his PhD at the Department of Networked Systems and Services, BME. He is currently involved in a research project aiming to establish a quantum random generator service on campus. His current research interests are quantum communications and quantum computing.

Laszlo Bacsardi (M’07) received his M.Sc. degree in 2006 in Computer Engineering from the Budapest University of Technology and Economics (BME). He wrote his PhD thesis on the possible connection between space communications and quantum communications at the BME Department of Telecommunications in 2012. Between 2009 and 2020, he worked at the University of Sopron, Hungary (formerly known as University of West Hungary) in various positions including Head of Institute of Informatics and Economics, University of Sopron. Since 2020, he is associate professor at the Department of Networked Systems and Services, BME. His current research interests are quantum computing, quantum communications and ICT solutions developed for Industry 4.0. He is the Vice President of the Hungarian Astronautical Society (MANT), which is the oldest Hungarian non-profit space association founded in 1956. He is corresponding member of the International Academy of Astronautics (IAA). Furthermore, he is member of IEEE, AIAA and the 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.