TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Nikita Timokhin 184062IVSB

Replacing Pseudo-Random Numbers in Software with Hardware-Based True Random Number Generation

Bachelor's thesis

Supervisor: Tauseef Ahmed
PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Nikita Timokhin 184062IVSB

Pseudojuhuarvude asendamine tarkvaras riistvarapõhise pärisjuhuarvude genereerimisega

bakalaureusetöö

Juhendaja: Tauseef Ahmed
PhD

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Nikita Timokhin

15.05.2021

Abstract

The goal of this research work is to design and build a hardware-based true random number generator and compare its different aspects and capabilities to the modern of-shelf cryptographically useful software-based pseudorandom number generators. My theory is that hardware-based true random number generators can be, in fact, more efficient than the fully-software pseudorandom number generators in many ways, such as generation speed, trustworthiness of the randomness, repeat cycle – which, in theory, would get eliminated in case of the hardware solution – and so on.

During the research work, both theoretical analysis and practical experiment will be conducted. Theoretical analysis will include the analysis of existing software decisions, calculations of the limitations of the hardware being used, and such. Practical part will consist of results of different tests comparing the hardware and software generators, and an overview/analysis of them, which would form the conclusion.

The design of the generator consists of two parts: software and hardware. Software part includes writing the firmware for the ATMega328 AVR microcontroller for the device side, and a Python package to interface with it for the host (personal computer) side. The hardware part of the design falls out of the focus of the study and is mostly based on personal knowledge of electronics and engineering. The physical device consists of a white noise generator based on a transistor in reverse-breakdown mode, and an amplifier that converts it to a random binary stream.

The paper's results are the device designs and the conclusion of it being useful at all. The paper also covers a bit of history and an overview of the state of the art random generation technologies.

This thesis is written in English and is 75 pages long, including 8 chapters, 22 figures and 4 tables.

List of abbreviations and terms

AES	Advanced Encryption Standard
CBC	Cipher Block Chain
DRBG	Deterministic Random Bit Generator (synonymous to PRNG)
MT	Mersenne Twister
PRNG	Pseudo-Random Number Generator
RMSE	Root Mean Square Error
RNG	Random Number Generator
TRNG	True Random Number Generator
UC, uC	Microcontroller (µ for Micro)
XOR	Exclusive OR

Table of Contents

1 Introduction....................10
2 Background Information....................13
3 Description of the problem and formulation of the assignment....................14
4 Method and tools....................15
4.1 Overview of the method....................15
4.2 Overview of the tools....................19
4.2.1 Hardware....................19
4.2.2 Firmware and Software....................20
5 Theoretical analysis....................22
6 Practical results....................26
6.1 Diehard Tests....................26
6.2 Speed test....................27
6.3 Distribution uniformity test....................28
6.4 Visual representation....................31
7 Practical result analysis....................32
7.1 Diehard tests....................32
7.2 Speed test....................32
7.3 Distribution uniformity test....................34
7.4 Visual representation....................34
8 Summary....................36
References....................39
Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis....................41
Appendix 2 – Program code and notes....................42
Appendix 3 – Hardware circuit diagram and notes....................68
Appendix 4 – Data files, additional graphs....................71

List of Figures

Figure 1. Block-scheme of the TRNG workflow....19
Figure 2. RNG Diehard tests' sorted p-values....27
Figure 3. Distribution, 1000 samples....28
Figure 4. Distribution, 1 million samples....29
Figure 5. Distribution, 1 million samples, zoomed in....29
Figure 6. Distribution, 10 million samples....30
Figure 7. Distribution, 10 million samples, zoomed in....30
Figure 8. Visual representation....31
Figure 9. Hardware schematic....68
Figure 10. Photo: prototype of the noise generator on the breadboard....68
Figure 11. Photo: assembled device board being tested....69
Figure 12. Photo: device put into the enclosure and wired up....69
Figure 13. Photo: assembled device is connected and working....70
Figure 14. Photo: the device and its PSU....70
Figure 15. AES p-value distribution....71
Figure 16. MT p-value distribution....71
Figure 17. URAND p-value distribution....72
Figure 18. TRNG p-value distribution....72
Figure 19. AES visualisation, greyscale, 1000 values per side....73
Figure 20. MT visualisation, greyscale, 1000 values per side....74
Figure 21. URAND visualisation, greyscale, 1000 values per side....75
Figure 22. TRNG visualisation, greyscale, 1000 values per side....76

List of Tables

Table 1. Diehard tests RMSE values by RNG...26
Table 2. N values VS time of byte generation...27
Table 3. N values VS time of unsigned 4-byte long integer...27
Table 4. Bytes per second rate comparison..33

1 Introduction

Random numbers play an important role in many technical spheres. In computational physics, they are an essential part of Monte-Carlo simulations and other statistical research methods [3]. Such methods are employed by researchers in the many fields of science and play key roles in their research. Random number generation plays a critical role in many parts of cyber security and cryptography, too: key pair generators require reliable random number generators for symmetrical and asymmetrical encryption algorithms, Diffie-Hellman type algorithms. Cloud computing and cloud storage, one of the currently most popular and prosperous areas in IT, vastly rely on random numbers for data and connection security. Even entertainment, like video games and digital art of all sorts, employ random numbers in numerous creative ways [5]. Although not immediately evident, many aspects of the modern person's life rely on random number generation way more than it could seem. This, of course, makes the quality of the random numbers play a crucial role: weak data protection could lead to the leaking of people's personal data or the classified information of enterprises. That is why in fields where random numbers play a key role in security algorithms, and their quality is directly related to the overall safety of the process, it is essential to pay attention to how these numbers are generated.

Most web services and some security algorithms use pseudorandom number generators (PRNGs) to generate their random numbers. This makes such web services and algorithms vulnerable to the many PRNG attacks that are possible. Any PRNG is a deterministic software-based algorithm, and this is what makes them so vulnerable. On the other hand, a number of true random number generators (TRNGs) are not vulnerable to such attacks: they are non-deterministic and usually come in the form of an external hardware device that connects to the PC, server, or network. TRNG solutions have their upsides and downsides against PRNGs. The main upside being their true randomness property and the biggest downside their random value generation speed, which usually is significantly lower than that of PRNGs.

One of the most popular home computers of the 80s , the Commodore 64, had a built-in RND command in C64 BASIC, which selected a random number from a pre-determined sequence. Often, it resulted in patterns and predictable results from calling the function repeatedly. This is an extremely simple PRNG (pseudorandom number generator), which, of course, was not usable for any kind of secure or cryptological use, as for the computer itself. For such needs, hardware-based generators were used: there were many different approaches, such as generating the source noise out of overly amplified transistor or diode reverse breakdown thermal noise or using phase-locked loops and logic integral circuits to generate binary pseudorandom signals. There were not many even somewhat safe and high-entropy PRNGs back in those days. The most famous enterprise-grade algorithm was called RANDU [2] – it was used in economics to make predictions. Later, it has been proven that RANDU is vastly predictable and has patterns [2], and, thus, was making some of the predictions flawed.

Time went on, and technologies were developing further and further. The computational power of machines around the world was growing drastically year after year. Soon, relatively quick linear feedback shift register PRNG realizations were possible in software as well as many other methods of generating pseudorandom numbers. Everything became very complicated rather quickly, and right now, there is a whole plethora of PRNG software implementation of all kinds, shapes, and methods.

Most of these pseudorandom number generators have a hefty accent on the ‘pseudorandom’ part, and although they generate numbers quickly, there still may be patterns, repeats, and predictabilities [2]. There are, however, a few PRNGs that are considered safe: one of them is called Mersenne Twister [6] – it (and all its variants) use a rather complex logical setup to generate a set of random numbers, one after the other, with a massive repetition cycle.

Another interesting “safe” algorithm common for most Linux distribution is called URAND (also known as /dev/urand, or urand) [4]. It uses the so-called “128 bits of entropy” as the first state of the generator: they are collected from the electrical noise of the processor and are considered to be completely, absolutely random. However, the further generation is also relatively slow and not random at all. So, if one knows the generator's current state, they may predict all the future iterations of the generator. [7]

Finally, one of the widely approved crypto-safe noise generators uses Advanced Encryption Standard (AES) encryption in Cipher Block Chain (CBC) mode as a feedback loop. It generates a reliable pseudorandom stream in a relatively short time and is used in many high-end cryptological operations [4]. However, similarly to Mersenne Twister (MT), the AES CBC requires initial values. But while MT requires just one seed value, this method would require three unique sets of reliably random 128 bits.

The studies show that these three generation algorithms are supposedly the most widely used "default" crypto-usable algorithms available "off-shelf". And what is especially interesting is that they are not random and take way more processing power and computer resources than their less sophisticated relatives. Hardware random number generation was in use in the 1990s in the form of PCI extension cards for servers [8]. Their primary purpose was to speed up the encryption and decryption of data traffic and all the tasks alike. They were highly expensive and hence not affordable for any enthusiast, and in modern-day computation, they are obsolete. However, the idea of handing off most of the generation work to analog hardware and leave only the task of "harvesting" the results for the software made me think if this could be more efficient than the two afore-mentioned pseudorandom generation algorithms. And in addition, the analog circuit could be tuned to flat white noise response, making the numbers acquired from the device purely random by default every time. This work will investigate this problem and the construction of such a device (which would be a TRNG, or a true random number generator) and compare it with the three existing off-shelf crypto-safe PRNGs – MT, URAND, and AES CBC.

2 Background Information

A pseudorandom number generator (PRNG), also known as a deterministic random bit generator (DRBG), is an algorithm for generating a sequence of numbers with properties approximate to the properties of sequences of random numbers [4]. The sequence generated by the PRNG never is truly random, as an initial value, called the seed, entirely determines it [7]. While PRNGs are not truly random, as opposed to hardware true random number generators (TRNGs), they have a significant upside of generation speed, as well as reproducibility, that is, in some cases, important and useful.

PRNGs are essential for Monte Carlo method simulations (and other computational physics needs), procedural generation in computer games and digital art, and cryptography [3, 5]. To be called cryptographically safe, PRNGs (CSPRNGs) need to have an output unpredictable from its earlier outputs. They are also expected to have more complicated algorithms, which do not behave linearly, and big repetition cycles. [9]

On the contrary, a TRNG is a device that generates random numbers based on a physical process rather than utilizing a software-implemented algorithm. TRNGs are often based on microscopic phenomena that generate low-level, statistically random "noise" signals, such as thermal noise, the photoelectric effect, etc. [1, 10]. More elaborate TRNGs may use quantum effects[1] for even more unpredictable outcomes. These stochastic processes are, in theory, completely unpredictable. The actual randomness of such processes can be evaluated only through an experimental approach.

Typically, a hardware PRNG measures some sort of unpredictable physical phenomenon directly converted to an electric signal (in the case of this work – Zener breakdown shot noise of a transistor [11]). This unpredictable electric signal is then further amplified to measurable levels. Afterwards, it is converted to a digital signal utilizing additional circuitry: usually, the output is a simple stream of zeroes and ones. They are then repeatedly sampled, and so, N bit long random values are generated.

3 Description of the problem and formulation of the assignment

This paper focuses on the question of the possibility of a hardware-based true random number generator being faster and more reliably random than a state-of-the-art off-shelf software-based pseudorandom number generator. This question, however, unfolds into a number of different questions, such as – is there any kind of patterns to occur in the TRNG unit built for testing? If so, are they more or less evident than those in the case of software PRNGs? How fast will random numbers be generated on the TRNG box relative to the software-based pseudorandom number generators? Additionally, if the TRNG unit turns out to be random enough from all sides, its feature of true randomness could be demonstrated by building some applications that rely on it, such as a one-time-pad key generator utility. A true one-time-pad cipher requires a purely random key in order to be unbreakable. Therefore, a TRNG would play a key role in its successful operation.

The problem statements can be formally stated as:

- will the hardware-based TRNG have a more even dispersion or values than the software-based PRNGs and have better quality random output?
- will the hardware-based TRNG compare in speed with the software-based PRNGs?
- If the solution turns out to be effective, what use cases could it have?

4 Method and tools

4.1 Overview of the method

The main part is split into three subparts:

1) Theoretical analysis

2) Practical experiment

3) Result analysis

All parts are going to compare the capabilities, limitations, and vulnerabilities of the hardware-based self-designed TRNG, with those of the selected software PRNGs. Research on modern state-of-the-art safe and cryptologically useful PRNG solutions has been conducted. It showed that most such built-in methods in all programming languages are not suitable, are breakable, and very evidently pseudorandom. Three state-of-the-art up-to-date algorithms that are considered crypto-safe and readily available for many languages were selected for testing and comparison.

The first is the URAND algorithm. It gets its initial seed from actual electric noise and can be considered safer than most other algorithms [4]. It is built into most Linux distributives and other Unix-like systems.

The second one is the Mersenne Twister (MT): it is one of the most complex and compound PRNGs with an enormous repeat cycle, which is also considered random enough for cryptological usage and relatively lightweight in terms of computing time and power consumption [6]. MT does not get its seed from truly random sources and needs a seed input that can be done either by human, by preset numbers, via a true random number generation device, or in any other way. It uses a set of complex logic operations to get the next values based on the previous while being generally unpredictable [6].

The third selected algorithm is AES in cipher blockchain mode. It is not available off-shelf as a PRNG per se, however setting it up is easy with a looping algorithm. The next block uses an exclusive or (XOR) of the plaintext and ciphertext of the previous block as its input [12]. The AES pseudorandom number generation method is considered very uniform, highly unpredictable, and cryptographically safe [4, 9]. These three PRNGs are among the very few Cryptographically Safe PRNGs (CSPRNGs) available as ready to implement solutions. All three are very complicated algorithms, as a CSPRNG has to be as unpredictable and random as possible. The additional complexity at the cost of marginal randomness is an overhead compared to the simplistic methods used by a hardware random number generator. The hardware method is based on a transistor reverse-breakdown shot noise; as long as the microscopic structure of the transistor is kept safe and the device itself is physically secured, there is no way to influence or predict the next state of it from the previous values.

In the theoretical analysis part, the vulnerabilities of PRNGS and the TRNG have been studied and compared. It is important to see if a TRNG is possibly more vulnerable than a PRNG before any further in-depth studies. The limitations and capabilities of the hardware are studied, and value generation speed and its limit are estimated.

There are many methods described in theory to find the true randomness of a given bit sequence. One of the most famous ones is the Kolmogorov complexity and Kolmogorov randomness. The longer the function in a given programming language to print the given bit string is, the more complex (thus, more random) it is. Furthermore, the bit string is considered random if and only if the program that can produce such string is equal or longer than the string itself. [13]

However, this approach (and many other theoretical approaches to the question of randomness) is purely theoretical. It is not exactly clear what is considered program size and how to compare it to the bit string size in this case. An algorithm that searches for such a function that would print the random string generated by either a TRNG or a PRNG would consume immense amounts of machine time and power. Therefore, this study will leave the problem of proving the randomness to the more practical yet mathematically backed method of statistical testing. In the next section, statistical testing methods are described.

For the practical part of the work, each of the generators (three PRNGs and one TRNG) will get their output sampled into text files by a proprietary program (Appendix 2). Then, two blocks of tests are conducted on the generators and their sampled output data.

Firstly, the classic original Diehard Test Battery processed the results of random generator sample data files (100 million sampled values written to a file). The Diehard test battery consists of 15 tests that produce more than two hundred p-values. In general, a p-value would represent how random the random data fed to the test is. It always lays in [0;1) range, 0.5 meaning the generator is perfectly random. However, it is impossible to conduct one single test once – the tests of the battery use different methods and often yield completely different p values. Hence, all the p-values from the 15 tests are expected to be uniformly distributed: an easy way to graphically check this is to sort all the values in order and compare them to an $x = y$ graph. Calculating the root mean square error of the set and comparing it to the other sets would give the numerical estimate of the randomness: the random value set with a lower error value is hence generated by a "more random" random generator. The tests themselves will be run with the original Diehard Test Suite released by G. Marsaglia in 1995 [14] – those are DOS programs, and they run fine on modern distributions of windows, and the definition of an unsigned 32-bit integer (which it uses as its working unit) did not change since then. The job of processing the test output, plotting the graphs, and calculating the root mean square error will be handled by a python program by me (code available in Appendix 2). The Diehard test suite has 15 statistical tests, which use highly complex mathematical base that falls out of the scope of this thesis work. However, the data on the nature of the tests is available from many sources, and many of these tests are not Diehard battery exclusive.

Secondly, proprietary testing is conducted. Apart from Diehard tests, additional custom tests will be run, too. They do not test the random qualities of the generator per se but instead focus on other qualities not evaluated by the Diehard tests. The first test does not employ the data gathered in the previous step and focuses on generation speed. Such a test aims to compare the TRNG device to the PRNG algorithms in terms of speed.

Test 1.1: Generate 1000 (thousand), 1000000 (million), 10000000 (ten million) random bytes with the setup and the selected software algorithms and compare the elapsed time.

Test 1.2: Generate 1000 (thousand), 1000000 (million), 10000000 (ten million) random unsigned 32-bit-long integers with the setup and the selected software algorithms and compare the elapsed time.

For the second test, the goal is to visualize the RNG outputs to check them for patterns, deviations, and uniformity of the value distribution. For this, two different types of random value set processing and visualizing Python scripts were created.

Test 2.1: Select 1000 (thousand), 1000000 (million), 10000000 (ten million) random bytes from each file generated by the TRNG and the selected software algorithms. Then, plot an X-Y graph of byte value (0-255) versus normalized times of the value occurring (0.00 – 1.00, 1.00 being the most rolled value) and compare the patterns and deviations.

Test 2.2: For each RNG, make a square image based on the list of values generated by it. Predefined picture width/height is to be the same for all four RNGs. From top right, line by line, fill in each pixel with a greyscale color (#000000 ... #FFFFFF) linearly related to the value from the list (0 … 255). Also, make black/white versions of the images by treating any value below 127 as black and other values as white. Such images will be easy and evident proofs of the presence or absence of obvious patterning in the datasets.

From tests 1.1 and 1.2, it is going to be easy to conclude the speed of the device in comparison to the software solutions. It will also become evident if it is more helpful in creating a random byte stream or for getting different types of values, such as integers, floating-point values, etc. The third test will show if any patterns appear in either the software solutions' output or in the hardware test unit's output. The output is compared, in terms of deviations and patterns, making it apparent if the TRNG's output value distribution is even.

4.2 Overview of the tools

4.2.1 Hardware

The main idea is to exploit a bipolar junction transistor in reverse breakdown mode to generate some thermal noise as the initial source of unpredictable voltages. Next, this noise is amplified with another transistor and sent to a comparator integral circuit. Its threshold can be finely adjusted with a trim resistor – a special device that one can turn with a screwdriver until the device is considered calibrated. It is to be adjusted, so the distribution of logic high and low states ("ones" and "zeroes") is even between the two at all times.

The binary uncertainty is then sampled by a microcontroller, ATMega328, which is the chip that is used in Arduino Nano boards. It is used in this project because of its simplicity and ease of prototyping. However, in the actual product, a bare microcontroller can be used, and the whole setup could be put on a small circuit board of the size of an average flash memory stick from the 2010s.

Figure 1 shows the block scheme of the hardware device's insides and its communication with the computer:

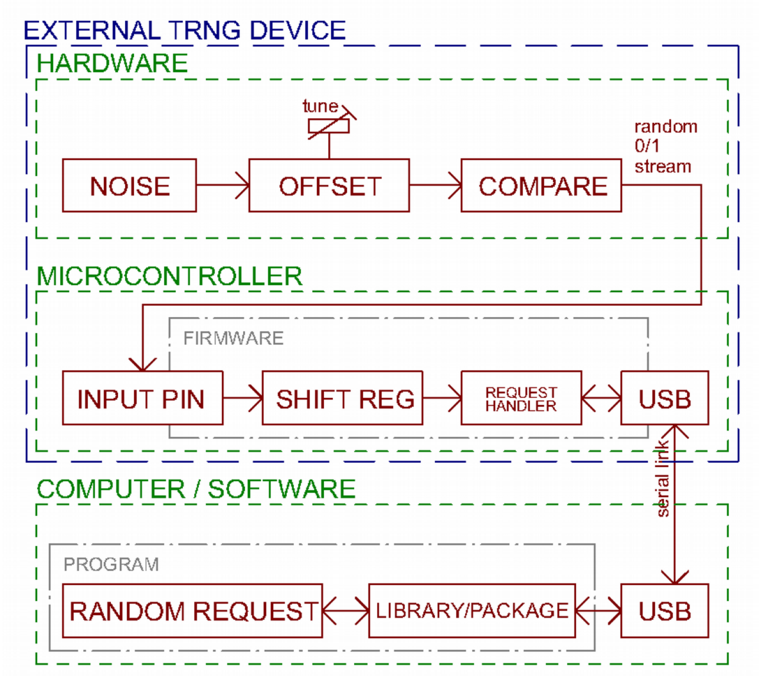

Figure 1. Block-scheme of the TRNG workflow

4.2.2 Firmware and Software

The uC (microcontroller) runs custom firmware. It does the tasks related to communication with its host and samples the binary stream from the hardware. Communication is carried out via serial port at 115200 Baud. The device expects a single character to be sent over the serial port by the host to set the mode of operation, such as:

- "B" - get one **b**yte. The uC then carries out a single procedure of getting a random byte and sends it back via the serial interface.
- "S" - **s**tream bytes. This command puts the microcontroller into a loop of endlessly generating values and sending them over the serial.
- "E" - **e**nd streaming bytes. This command breaks the stream loop and puts the microcontroller back into await command mode.

More commands may be available. For complete firmware C code reference, see Appendix 2.

The software part of the project is implemented in Python programming language, which is best suitable for prototyping. For the product development, C or C++ can be used due to their efficiency.

Firstly, a Python module called "trnglib" is developed (Appendix 2). This module depends on a python package called "pySerial" for serial communication with Arduino [15]. "trnglib" allows the user to operate the TRNG device without needing to know the specific command coding and other lower-level aspects of the process. It provides dedicated semantically understandable functions to connect and disconnect the device and to acquire values in different formats. It can also enable and disable the device's stream mode. The only detail the user has to know to connect are the details of the USB port the device is linked to. Many useful programs can then be implemented using trnglib as an interface between the high-level Python programs and low-level communication with the TRNG.

Additionally, a toolkit of proprietary utility programs for data gathering and analysis is written. It includes scripts for gathering sample data, different analysis scripts, and conversion scripts to interface the generated data with the Diehard testing program. All the Python scripts, as well as any other code, can be found in Appendix 2.

5 Theoretical analysis

Mersenne Twister (MT) and the Unix-common /dev/urand (URAND) are available as ready-made implementations in many languages, including Python. Setting up the AES CBC to work as a noise generator requires some proprietary coding. It is a feedback loop which “encrypts” itself, with the XOR of the message and the encryption result of the previous block is the plaintext for the next one, producing 128-bit noise blocks at the output. But since the AES itself is also available as a ready-to-use Python package. Therefore, Python is chosen for the implementation of all three (MT, URAND, AES) PRNGs and the TRNG interfacing library.

The TRNG is based on white noise, a non-deterministic process, while a PRNG is a deterministic device. Some attacks that are possible should be considered first.

Python implementations of MT and AES CBC noise require initial values. For MT, it is a seed (an integer number), and AES CBC noise requires three distinct sets of 16-byte random byte strings to start off. This makes these two generators vulnerable to seed attacks. Hijacking their seed will make the device highly predictable and, thus, insecure [7]. Another possible variant of this attack could be to flush the source entropy out of the system and replace it with a known seed. Although harder to execute, this sort of attack is possible [7]. URAND is less vulnerable to seed attacks: it still requires a seed, however, the algorithm collects it from the computer processor’s electric noise, a high entropy random bit stream source. A weak seed for MT or AES CBC noise may also increase the vulnerability of the PRNG. A general requirement for a PRNG seed is high entropy (“very random”), which excludes human-selected seeds, seeds generated by other PRNGs, seeds generated based on date and time, and so on.

The second type of possible attack is the PRNG attack – more specifically, a direct cryptanalytic attack. This is an attack that aims to predict future states of a PRNG if enough past states of it are known [7]. For example, MT is very vulnerable to such an attack. By knowing just 624 of its previous states, it is possible to reverse-engineer all of its previous and future states at once [16]. Such a vulnerability is present in any deterministic random byte generator (DRBG, synonymous to PRNG with an accent on being a deterministic device). For some algorithms, it is easy to conduct such an attack, for others it is complex. However, a deterministic device is always possible to predict given enough data.

“Hardware-based RNGs make use of a physical process to generate randomness. Because classical physics is deterministic, RNGs that rely on phenomena described within a purely classical noise model, such as thermal noise, can only be proved random under the assumption that the microscopic details of the system are in- accessible. This assumption is usually hard to justify physically. For example, processes in resistors and Zener diodes have memory effects. Hence, someone who is able to gather information about the microscopic state of the device — or even influence it — could predict its future behaviour.” [1]

Theoretically, an external TRNG may not be vulnerable to any of these attacks. As mentioned previously, there are ways to influence or predict its state, given access to the microscopic level of physical components of the randomness source – the noise generator in this case. However, in this study, it is assumed that TRNG hardware is kept securely. Moreover, realistically speaking, studying the TRNG and its internal components at a microscopic level while the device is in operation can be impractical.

When the proper measures are applied, the TRNG is cryptographically more safe and trustworthy than any of the PRNGs. However, it has limitations in generation speed.

There are state-of-the-art TRNG devices in the form of e.g. flash dongles, PCIe cards, devices with a serial connector, which all use different algorithms, connection protocols, run at different clock speeds, etc. [1, 10]. However, these devices or the data on them cannot be obtained due to budget limitations and the unavailability of their documentation. Therefore, in this study, a TRNG is designed and studied in comparison to the PRNGs in fields where they are comparable.

The setup is based around an ATMEGA328 microcontroller as a part of an Arduino Nano board. It communicates with the computer via USB Serial port. The microcontroller's clock is 16 MHz. It means if the whole routine of gathering and sending one byte took just one tick, the speed cap would be 16 million values per second – which would outrun most PRNGs for good. However, gathering one random byte from the TRNG takes the microcontroller several hundred instructions.

The device's serial port transmits to and receives information from the host computer at a 115200 baud rate – which essentially means 115200 symbols per second. It suffices to say "symbols" are equal to "bytes" in this case, because the setup only communicates one standard char (or byte) at a time both ways. Suppose a random symbol is ready to go at any moment of time inside the microcontroller. In that case, the nominal baud rate still should be halved for the speed estimation, as after each data transmit (no matter how long in bytes), the serial connection protocol sends a STOP byte. Hence, the absolute cap of values per second for the microcontroller would be 57600 bytes per second. Even if gathering one single random byte took the microcontroller an entire lot of 250 instructions, at a 16MHz rate, it would mean 16000000/250 = 64000 random bytes per second, which is still quite a bit more than our maximum baud rate.

Taking this as a starting point, the "best case" TRNG speed could be estimated to be up to 57600 random bytes per second. Then, generating one million values would take about 17.36 seconds, which is slow, although still tolerable for all the true randomness factors. However, in practice, it may tend to be less efficient.

The original Diehard test battery was selected as a mean to estimate the randomness of each RNG. It contains 15 tests, each employing a different, often very complicated statistical mean to calculate one or more p-values 0. In essence, a p-value represents the behaviour of an RNG and falls in the [0,1) range. Getting a p=0.5 would mean the generator may be ideal. The more this value deviates from the middle, the less random the generator appears to be. But since the tests are statistical and employ sampling of value groups, the p values will come in great quantities and differ widely. The expectation from an ideal RNG is a uniform distribution of p-values between 0 and 1. This implies, if all the P values obtained were to be sorted in ascending order and plotted on an X-Y graph (x for the P-value number in the list and y for the value itself), fit to [0,1) by the X coordinate, they would go along the $x = y$ function in an ideal case. In practice, though, there will be deviations. In this paper, these deviations are used to estimate the quality of the random generation numerically. For that, the standard root mean square error (RMSE) formula is used:

$$\sigma=\sqrt{\frac{\sum_{i=1}^{n}(P_I \cdot O_i)^2}{n}}$$

Sigma (σ) is the RMSE value in this formula. P and O are calculated ideal line and actual p-value for that index. The result of this formula will estimate the actual device's deviation from the curve of an ideal case, which in its turn shows how much the RNG's output deviates from an ideal case. Along with this, a visual graph of p-values versus their position in the sorted list is rendered for each RNG for more intuitive visual representation of the case.

The device is estimated to produce much higher quality random than the PRNGs. Hence, its RMSE is expected to be at least one decimal order less than that of any examined PRNG. The TRNG's p-value graph is also expected to lay flatter towards the ideal case reference.

6 Practical results

The test unit is assembled following the schematic in Appendix 3. The device requires an external 12v power supply unit. A Python interface package is written, as well as the microcontroller firmware in C. The device is calibrated for the most even distribution of values with a single trimmer while running a graphical software utility on the computer. All the tests mentioned in section 4.1 are conducted, and the results presented.

6.1 Diehard Tests

100000000 (100 million) one-byte samples are gathered from the TRNG and the PRNGs – MT, URAND and AES. The following has been done for each dataset individually: it is converted to a file format expected by George Marsagila's original 1995 implementation for DOS/Windows using a self-made python script and processed by the quick (less informative and more 'bare p-values') variant of the Diehard tests battery. The output text is then re-converted by means of yet another script to a sorted list of p-values, which is then 1) plotted on a graph next to an $x = y$ line and 2) used along with calculated values of $x = y$ to calculate the RMSE by the standard formula. The table of resulting values is provided.

A big main graph with all four RNGs and the ideal case is presented (Figure 2). Individual graphs for each RNG are rendered as well. The individual graphs can be found along with other additional materials in Appendix 4.

Table 1. Diehard tests RMSE values by RNG

	TRNG	MT	URAND	AES
RMSE, 1	0.01763	0.03857	0.03210	0.04906

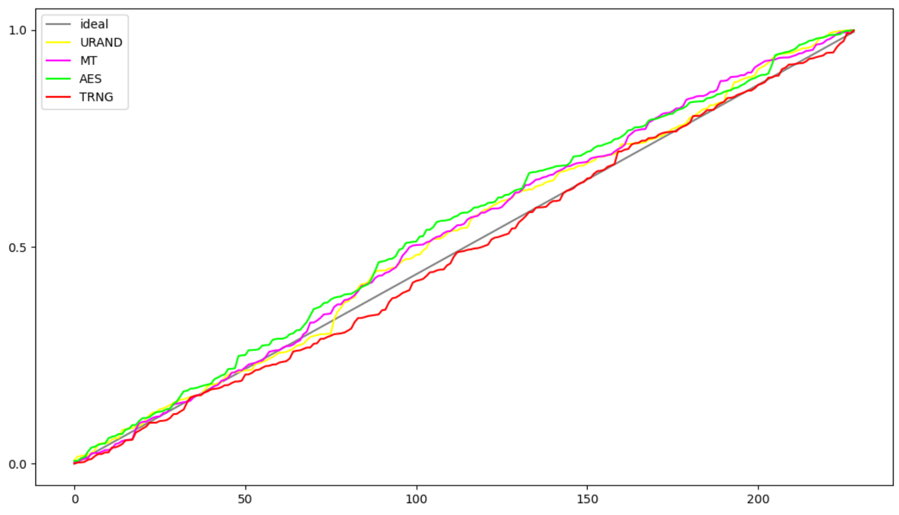

Figure 2. RNG Diehard tests' sorted p-values

6.2 Speed test

Tables 2 and 3 contain the results of speed testing the four RNGs. Times of generating different quantities of bytes are presented in Table 2, whereas the results for 32-bit unsigned integers are presented in Table 3.

Table 2. N values VS time of byte generation.

	TRNG	**MT**	**URAND**	**AES**
1000 bytes	0.6821s	0.0016s	0.0019s	0.0005s
1000000 bytes	665s (11min)	1.952s	2.181s	0.474s
10000000 bytes	6648s (1.8hr)	19.439s	21.263s	4.478s

Table 3. N values VS time of unsigned 4-byte long integer.

	TRNG	**MT**	**URAND**	**AES**
1000 uint32s	2.6627s	0.0017s	0.0022s	0.0016s
1000000 uint32s	2661 (44min)	2.082s	2.559s	1.820s
10000000 uint32s	~7.3hr[1]	22.288s	25.206s	18.781s

1 This particular test case was actually not conducted – it is the 1 million result multiplied by 10. Seconds do not matter at the scale of things this test achieved.

6.3 Distribution uniformity test

Figures 3 to 7 are graphs of the RNGs' value distributions for different sample counts. Figures 5 and 7 are zoomed-in versions of figures 4 and 6, respectively. They are provided to depict the test results better.

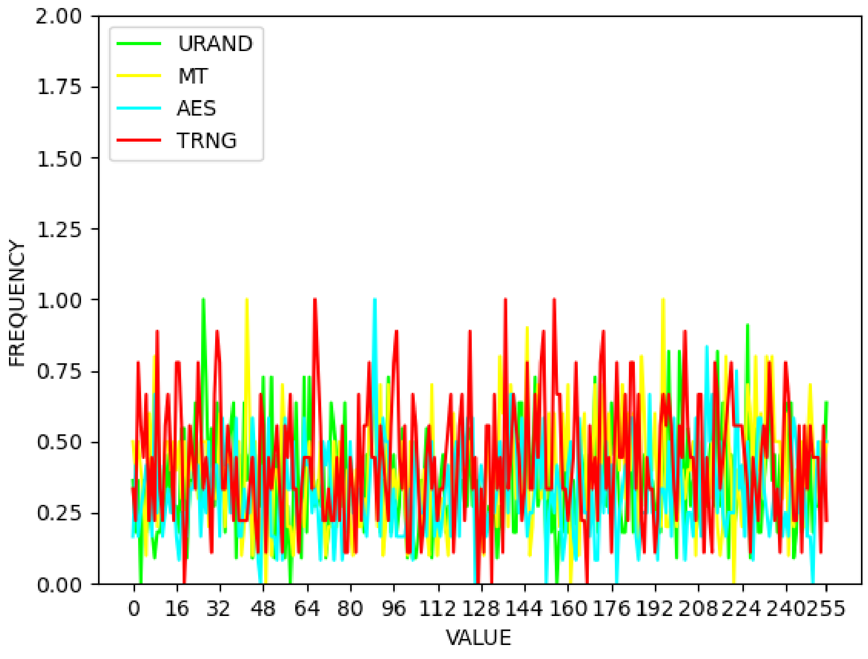

Figure 3. Distribution, 1000 samples

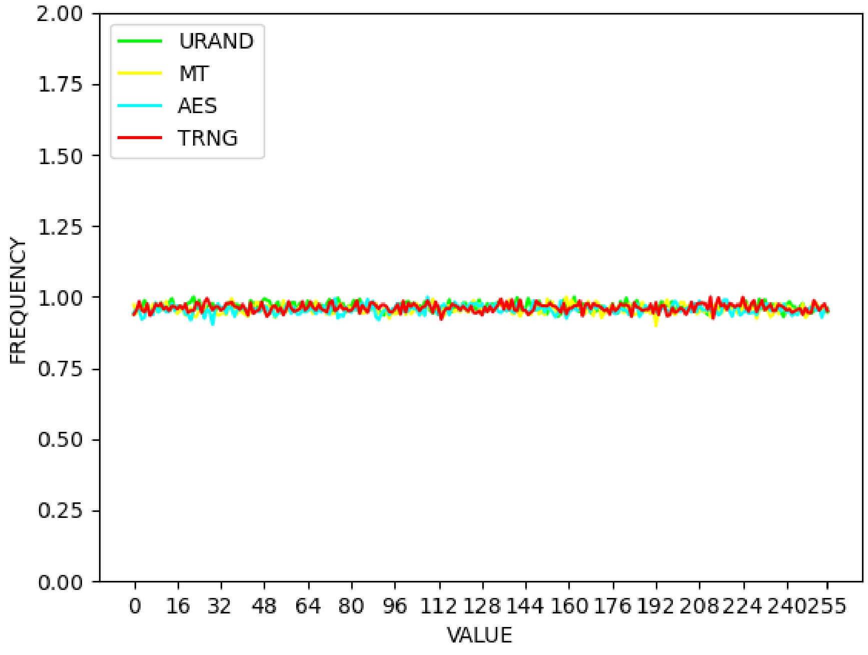

Figure 4. Distribution, 1 million samples

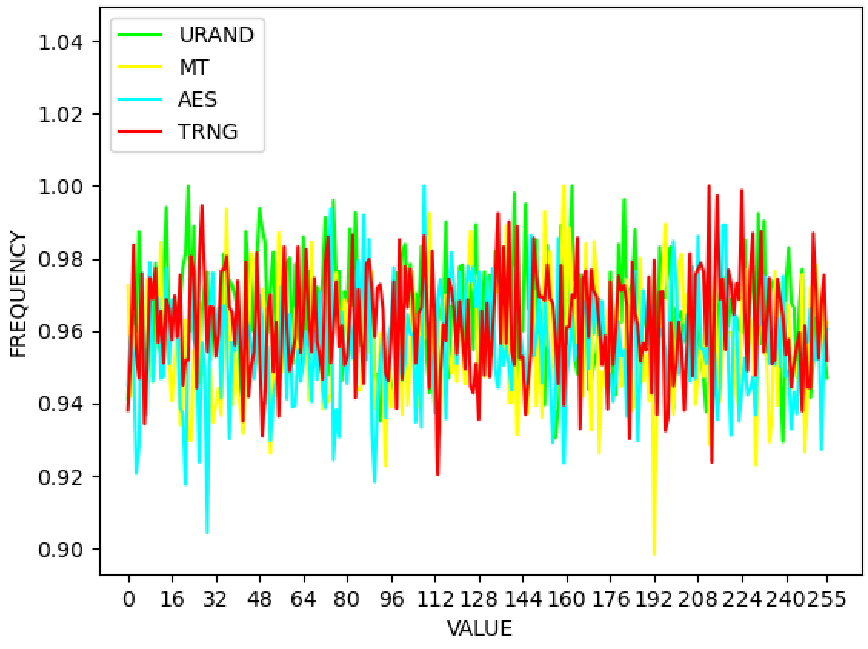

Figure 5. Distribution, 1 million samples, zoomed in

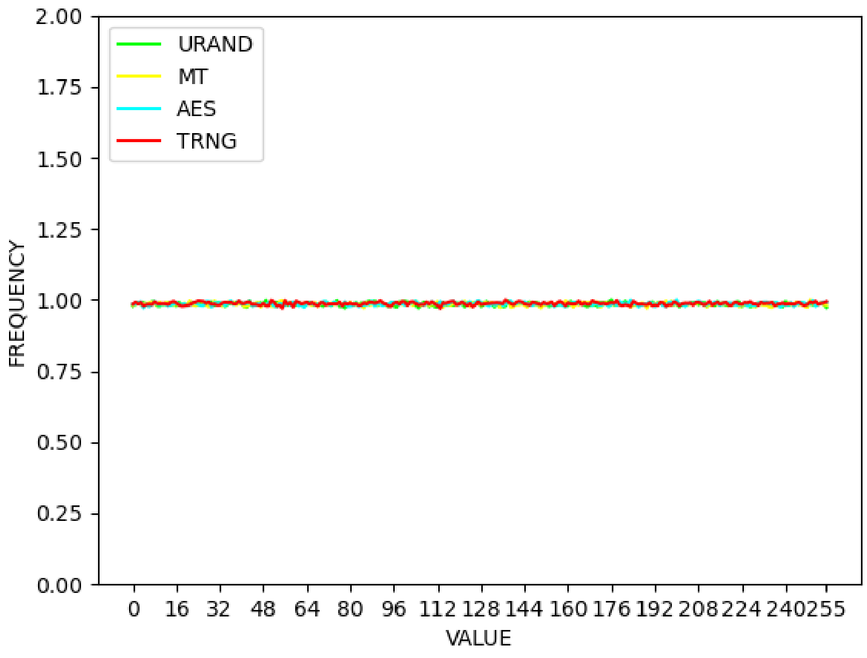

Figure 6. Distribution, 10 million samples

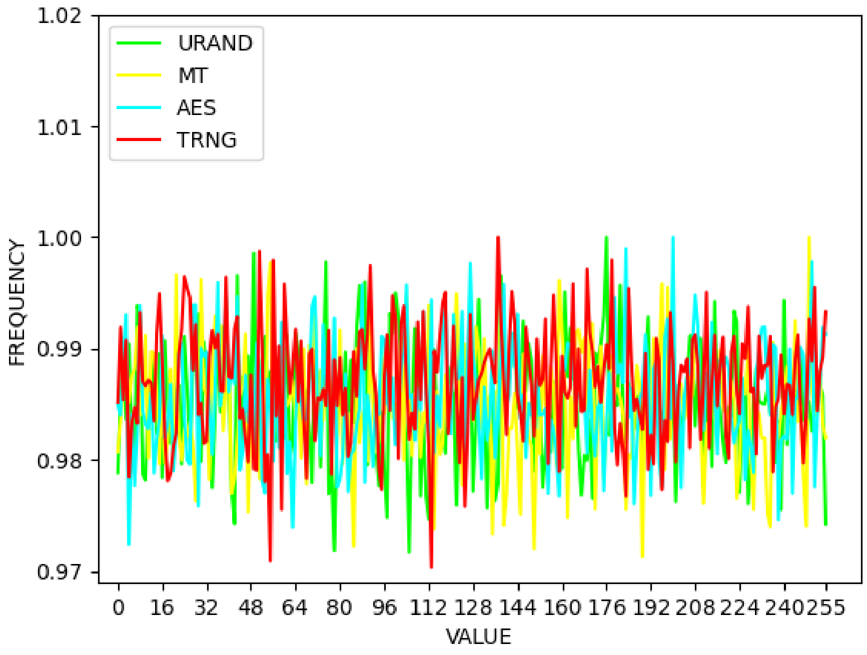

Figure 7. Distribution, 10 million samples, zoomed in

6.4 Visual representation

Figure 8 shows four images that are the visualisations of each of the RNGs' datasets. Values greater than 127 are converted to white pixels, whereas values equal to or lower than 127 are interpreted as black pixels. The visualisation program can be found in Appendix 2. More pictures are available in Appendix 4.

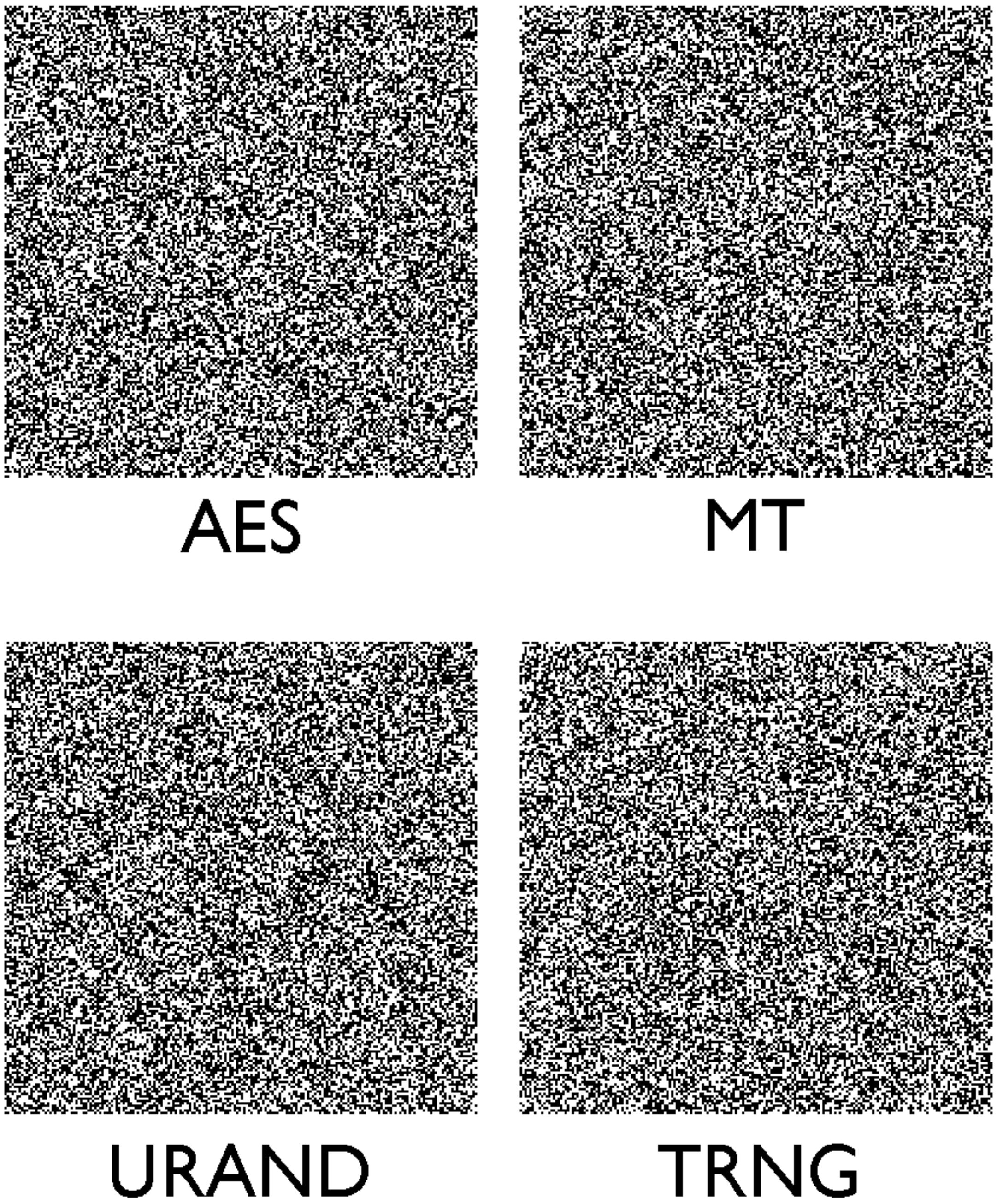

Figure 8. Visual representation

7 Practical result analysis

7.1 Diehard tests

The final TRNG's RMSE value is 2.2 times smaller than the average of the PRNG's RMSE values. It bests URAND by 1.8 times, MT by 2.2 times, and URAND by 2.8 times. The graphs in Figure 2 also evidently show that although all the experimentally obtained curves have nonlinearities and deviations relative to the ideal $x = y$ relation, the TRNG's curve overall lays flatter towards it. Figure 2 graphs and the numerical RMSE values from Table 1 show the superior quality of the true random number generator's output.

7.2 Speed test

Any of the PRNGs easily beat the TRNG in a speed test. The order of N of seconds it takes a TRNG and either of the three PRNGs to generate the same number of values are orders apart, especially for unsigned 32-bit integers: TRNG has to do four times the work of when it generates bytes, which is evident by its times for ints being rough multiples of its times for bytes.

To estimate the difference in numbers, consider Table 3. In it, calculations of bytes per second for each case, an average of each method's speed rate, and relations of each PRNG's speed to the TRNG's average speed are added to Table 2.

Table 4. Bytes per second rate comparison

	TRNG		**MT**		**URAND**		**AES**	
N bytes	Time, s	Byte/s	Time, s	Byte/s	Time, s	Byte/s	Time, s	Byte/s
1000	0.6821	1466	0.0016s	625000	0.0019	526315	0.0005	2000000
1000000	665	1503	1.952s	512295	2.181s	458505	0.474	2109704
10000000	6648	1504	19.439	514429	21.263	470300	4.478	2233139
Avg. Byte/s	■	1491	■	550574	■	485040	■	2114281
Times faster than TRNG:			369.26		325.31		1418.03	

As evident, any of the PRNGs beat the test TRNG unit by at least 300 times, and AES, stacking whole 128 bits, or 16 bytes, of random values per its cycle, outruns both the TRNG by more than an entire thousand times, as well as its digital relatives by more than three times.

Such a significant difference happened because of an imperfect noise generator. Due to budget constraints, high-frequency components for this study are not available, and so, the noise generator's frequency range is significantly lower than the microcontroller's. To compensate, 128 assembly NOPs after each read cycle are added, bringing the sampling frequency down to desired ranges. However, this workaround wastes 16*128=2048 cycles. Since the uC runs on 16MHz (16000000 cycles per second), (1/16000000)*2048=0.128 *milliseconds* of time after each read cycle are wasted, which is a tremendous loss of time. When generating one million values, this accumulates 128 seconds of wasted time. Generating one million random bytes with the TRNG took 665 seconds in total, so (128/625)*100=20.48% of this time is wasted by NOPs.

Due to the unavailable components, the shift register has been implemented in the microcontroller firmware, while it could be done in hardware. Leaving this task to hardware and reading whole bytes instead of reading 0/1 one by one eight times for two bytes and XORing them in software could speed the process up by up to 20 times.

It is also worth noting how URAND and MT have almost the same speed efficiency in bytes and integers, while AES does not show such a quality and slows down roughly four times for ints, same as the TRNG (Tables 2 and 3). This tells about their completely different principles of operation.

7.3 Distribution uniformity test

The ideal RNG byte value distribution graph over an infinite amount of samples should look like a flat line. This would mean that every value got rolled the same amount of times, which, in its turn, would mean that this RNG is fair and evenly distributed. The obtained datasets are processed with a script that counts how many times each of the 256 values got rolled. Then the results are normalised to be floats between 0 to 1. Figures 3 to 7 show the percentage of the times the value got rolled. In Figure 3, nothing is evident because the number of samples is so low (1000 samples) there simply is almost no chance each of the 256 possible states got rolled about four times so far. As more samples are acquired, the value distribution graphs become more like the ideal flat line. As shown in figures 4 and 5, with 1 million samples, the distribution is relatively flat for all the PRNGs and the TRNG. Every value is selected at least 90% of the time, as with other values. When compared with PRNGs, the TRNG value distribution is flatter over one million or more samples.

7.4 Visual representation

This was an additional step that has been conducted to further study the characteristics of the four randomly generated sets. It was a point of curiosity to visually represent different noise and try searching for visual pattern. The repetition cycle of all the selected PRNGs must be tremendous, and – it seems – no weak seeds got rolled, so the images presented in Figure 8, and other result images, are quite uniform noise. The TRNG visual noise profile looks no different from PRNGs in the sense of how non-patterned it is. It cannot be concluded from these images on which of the RNGs produce more patterns: the PRNGs' repetition cycle lengths are immense, and no patterns emerged to oppose the TRNG's patternless results. It was not possible to determine which of the images in Figure 8 is which RNG, or which one is the TRNG's result without the name captions. In short, despite not being truly random, modern CSPRNGs may deliver noise of satisfactory quality. The code that converts data generated by the custom data collection scripts into images is accessible in Appendix 2.

8 Summary

A true random number generator – a dedicated hardware device developed as a result of this study – has pros and cons compared to the conventional crypto-safe pseudorandom number generators. The quality of the random generation turns out to be higher for the true random number generator, as estimated by statistical testing of a hundred million single-byte samples. Furthermore, the byte value dispersion for all samples is visibly flatter for the hardware generator unit than for the software algorithms, as is shown in the figures in 6.3.

However, this particular TRNG unit and more advanced TRNGs can have a potential drawback. The estimated quality of the test unit's random is about 2.2 times better. However, the speed of the generation is at very least about 350 times slower than that of the software RNG implementations. This ratio could be improved with a different type of connection and dedicated hardware that could perform the tasks currently managed by microcontroller firmware.

As previously discussed, the software-based PRNGs have two main weaknesses due to their deterministic nature. The hardware device which is based on a reverse-brokedown transistor's shot noise, makes it invulnerable to such attacks. The generator has no seed, and predicting its values by building its model would take tremendous effort and examination of the particular device on microscopic levels. There are, however, vulnerabilities related to the hardware nature of this device: if the environmental temperature is shifted significantly, the pre-calibrated threshold level could also shift, or the parameters of the generated noise could be changed. The values would become biased, and the distribution would not be even anymore. However, it would require one to have physical access to the device for such a significant change. The physical security of facilities the TRNG is deployed at is out of the scope of this thesis work.

To conclude the initial research questions of this study:

- will the hardware-based TRNG have a more even distribution or values than the software-based PRNGs and have better quality random output?

Yes, the true random number generator has a notable advantage in value distribution, and shows better results under statistical testing. Combined with the essentially unpredictable nature of shot noise, this makes this random number generator very secure.

- will the hardware-based TRNG compare in speed with the software-based PRNGs?

The value generation speed of the device developed over the course of this work is far slower than any of the pseudorandom number generators that are tested. This is due to the hardware limitations of this particular unit: in theory, the device could be upgraded to work at significantly faster speeds.

- If the solution turns out to be effective, what use cases would it have?

Despite being slow, the device has many upsides – especially in terms of security – and has the potential to be upgraded. However, the speed limit excludes the use of this particular RNG for applications that require tremendous amounts of random data in a limited time to be generated, such as complicated Monte Carlo method simulations. There also are PRNGs suitable for those. However, such a device could be a great security feature for companies that run web services and have to deal with tasks such as generating tokens, keys, and keychains, and other data that must be random, does not require vast amounts of data per second to be generated. Each piece of data has to be completely separate and independent from each other - something a PRNG can never achieve because of its deterministic nature. Another possible application could be tasks such as fair random generation for multiplayer online games.

Another possibility to use this device to further increase the security of pretty much any system that uses PRNG is to use the TRNG to generate PRNG seeds and reset PRNGs after a relatively short time of being created. This is a more abstract idea, and it could be applied in many fields where many random values are required in a short time, and they have to be more secure and unpredictable.

References

[1] Frauchiger D., Renner R., Troyer M., True randomness from realistic quantum devices, Institute for Theoretical Physics, ETH Zurich, Switzerland

[2] Lewis, Peter A. W. Graphical analysis of some pseudorandom number generators, Monterey, California. Naval Postgraduate School

[3] Bauke H., Mertens S., Random numbers for large scale distributed Monte Carlo simulations, Otto Von Guericke Universität, Magdeburg

[4] Röck A., Pseudorandom Number Generators for Cryptographic Applications, Paris-Lodron-Universität, Salzburg, March 2005

[5] Hades review, Polygon, December 2020 https://www.polygon.com/22167819/hades-game-of-the-year-2020

[6] Matsumoto M., Nishimura T., Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator, Keio University and the Max-Planck-Institut für Mathematik

[7] Kelsey J., Schneier B., Wagner D., Hall C., Cryptanalytic Attacks on Pseudorandom Number Generators, University of California, Berkeley

[8] Du Preez A., Johnson M.G.B., Leist A., Hawick K.A., Performance and Quality of Random Number Generators, Computer Science, Institute for Information and Mathematical Sciences, New Zealand, April 2011

[9] Babaei M., Farhadi M., Introduction to Secure PRNGs, Shahrood University of Technology, Iran, September 2011

[10] Yu F., Li L., Tang Q., Cai S., Song Y., Xu Q., A Survey on True Random Number Generators Based on Chaos, School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha, China, December 2019

[11] Sanchez D., White Noise Generator Circuitry and Analysis, University of Texas, USA, April 2008

[12] Prescott T., Random Number Generation Using AES, Automotive Compilation Vol. 8, ATMEL Corporation, 2011

[13] Gammerman A., Vovk V., Kolmogorov Complexity: Sources, Theory and Applications, Computer Learning Reseach Centre and Department of Computer Science, Royal Holloway, University of London, Egham, UK

[14] Alani M.M., Testing Randomness in Ciphertext of Block-Ciphers Using Diehard Tests, College of Computer Engineering and Sciences, Gulf University, Kingdom of Bahrain, 2010

[15] Imreh G., Python in a Physics Lab, Institute of Atomic and Molecular Sciences Academia Sinica, Taiwan

[16] Jagannatam A., Mersenne Twister – A Pseudo Random Number Generator and its Variants, George Mason University, 2008

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Program code and notes

```
'''
	TRNG PROJECT - NOISE DATA VISUALISER
	usage: python3 analyse_dataToImage.py targetDataTxt.txt <values per side, int>
<scale factor, int>
	scale is optional! defaults to 1
'''

import sys
from PIL import Image

def test():
	img = Image.new( 'RGB', (255,255), "black") # Create a new black image
	pixels = img.load() # Create the pixel map
	for i in range(img.size[0]):    # For every pixel:
	    for j in range(img.size[1]):
	        pixels[i,j] = (i, j, 100) # Set the colour accordingly
	img.show()

def get_thresh(data, side, name, scale=1, thresh=128):
	counter = 0
	img = Image.new( 'RGB', (side,side), "black") # Create a new black image
	pixels = img.load() # Create the pixel map
	for i in range(img.size[0]):    # For every pixel:
		for j in range(img.size[1]):
			val = data[counter]
			if (val > thresh):
				val = 255
			else:
				val = 0
			pixels[i,j] = (val,val,val) # Set the colour accordingly
			counter += 1

	newsize = (side*scale, side*scale)
	img = img.resize(newsize, Image.NEAREST)
	img.save(oname + "-threshold.png")

def get_greyscale(data, side, name, scale=1):
	counter = 0
	img = Image.new( 'RGB', (side,side), "black") # Create a new black image
	pixels = img.load() # Create the pixel map
	for i in range(img.size[0]):    # For every pixel:
		for j in range(img.size[1]):
			pixels[i,j] = (data[counter], data[counter], data[counter]) # Set
the colour accordingly
			counter += 1
	newsize = (side*scale, side*scale)
	img = img.resize(newsize, Image.NEAREST)
	img.save(oname + "-greyscale.png")


if (__name__ == "__main__"):
	# get args
	try:
		fname = sys.argv[1]
	except Exception as e:
		raise ValueError("please provide name of the target txt file with values as
first argument!")
	try:
		f = open(fname, "r")
	except Exception as e:
```

```
	raise ValueError("unable to open txt file!")
try:
	side = sys.argv[2]
except Exception as e:
	raise ValueError("please provide picture side size as second argument!")
try:
	side = int(side)
except Exception as e:
	raise ValueError("picture side size should be an integer!")
try:
	scale = sys.argv[3]
except Exception as e:
	scale = 1
try:
	scale = int(scale)
except Exception as e:
	raise ValueError("scale should be an integer!")

# read data

data = []
lines_read = 0
for i in f:
	data.append(int(i))
	if (lines_read == side*side+1):
		break
	lines_read += 1

# process
oname = (fname.split("/")[-1]).split(".")[0]
get_greyscale(data, side, oname, scale)
get_thresh(data, side, oname, scale)
```

```
'''
	TRNG PROJECT - MULTIPLE FILES DHARD RESULT PLOTTER
	usage: python3 analyse_dhard_plot_all.py
	output: RMSE values to the console, and a graph.
	essentialy it is analyse_dhard_plot.py but without options
	and with hard-coded filenames.
'''

import sys
import matplotlib.pyplot as plt
from os.path import join
import convert_getDiehardPvals as diehard_conv
import analyse_dhard_plot as plotter

if (__name__ == "__main__"):

	'''
		CAREFULLY CHECK THESE!!!
		this script does not take any external input -
		the filenames and folder with the filenames are hardcoded.

		The script expects a bunch of DIEQUICK.exe results
		(original Diehard tests battery, quick version)

	'''

	# path to the folder with quick dhard test results
	folder = "../../results/diehard_analysis/"

	# names of the files. these can be text files, binary files or whatever.
	names = [
		"URAND",
		"MT",
		"AES",
		"TRNG"
	]

	'''
		this is the rest of the program. should do just fine without
		your interventions.
	'''
	fnames = []
	for x in names:
		fnames.append(join(folder, x))

	# this is for the lines do be of different colours
	colData = [1,1,0,1,0,0,1,0,0,1,0,0]
	counter = 0

	# parse and plot each result file
	for f in fnames:
		d = diehard_conv.convert(f)
		if (f == fnames[0]):
			plotter.plot_ideal(d[0])
		print("RMSE for", names[counter],"=", d[1])
		plotter.plot(d[0], f,
col=(colData[counter],colData[counter+1],colData[counter+2]))
		counter += 1
	plt.legend(loc="upper left")
	plt.show()
```

```
'''
       TRNG PROJECT - DIEHARD QUICK TEST RESULTS PLOTTER
       usage: python3 analyse_dhard_plot.py <pathtofile>
       output: RMSE value and a graph
'''

import sys
import matplotlib.pyplot as plt
from math import sqrt
import convert_getDiehardPvals as diehard_conv

'''
       plots the given list of pvalues
       will plot absolute garbage if given anything else
       other than the diehard_conv result.
'''
def plot(pvalues, fname, col="red"):
       idealvalues = []
       for x in range(len(pvalues)):
              idealvalues.append(round(x/len(pvalues), 4))
       label = fname.split("/")[-1]
       plt.plot(pvalues, color=col,label=label)

'''
       plots the ideal x=y line to compare to
'''
def plot_ideal(pvalues, col="grey"):
       plt.yticks([0,0.5,1])
       idealvalues = []
       for x in range(len(pvalues)):
              idealvalues.append(round(x/len(pvalues), 4))
       plt.plot(idealvalues, col,label="ideal")

if (__name__ == "__main__"):
       # read args
       try:
              fname = sys.argv[1]
       except Exception as e:
              raise ValueError("please provide target DIEFAST.exe analysis textfile as 
first argument!")
       # convert and plot
       fname = fname.strip()
       d = diehard_conv.convert(fname)
       print("RMSE: ", d[1])
       plot(d[0], fname)
       plot_ideal(d[0])
       plt.legend(loc="upper left")
       plt.show() #display the graph
```

```
'''
        TRNG PROJECT - TAKE NOISE DATA FILES AND DRAW THEIR VALUE DISTRIBUTION
        ON THE SAME GRAPH
        usage: python3 analyse_ndDistributionPlot-ALL.py <nsamples> <ntiers>
        ntiers is an int between 1 and 256 and is optional (default 256)

        makes sense to set it to a multiple of 8 or 16
        nsamples is the maximum numble of sample values to take into account
        is also optional (default 1000000 - one million)

        essentialy it is analyse_ndDistributionPlot.py but with a nsamples arg
        and with hard-coded filenames.
'''

import sys
import matplotlib.pyplot as plt
import numpy
from os.path import join

import analyse_ndDistributionPlot as plotter

if (__name__ == "__main__"):

        # read options
        try:
                nsamples = sys.argv[1]
        except Exception as e:
                nsamples = 1000000

        try:
                nsamples = int(nsamples)
        except Exception as e:
                raise ValueError("N of samples should be an integer! you provided:",
nsamples, type(nsamples))

        try:
                tiers = sys.argv[2]
        except Exception as e:
                tiers = 256

        try:
                tiers = int(tiers)
        except Exception as e:
                raise ValueError("N of tiers should be an integer! you provided:", tiers,
type(tiers))

        '''
                CAREFULLY CHECK THESE!!!
                this script does not take any external input -
                the filenames and folder with the filenames are hardcoded.

                The script expects results of prng_getNoiseData or trng_getNoiseData
        '''
        # folder with target files
        folder = "../../results/noisedata/"
        # filenames
        names = [
                "URAND.txt",
                "MT.txt",
                "AES.txt",
                "TRNG.txt"

        ]
        # make filenames
        fnames = []
        for x in names:
                fnames.append(join(folder, x))

        # setup the plot
        colors=["lime","yellow","cyan","red"]
        ticks = [0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240,
255]
```

```
plt.xticks(ticks)
plt.ylim(0, 2)
plt.ylabel('FREQUENCY')
plt.xlabel('VALUE')
# plot all noise data files' distributions
fcount = 0
for fname in fnames:
        plotter.draw(fname, tiers, nsamples, colors[fcount])
        fcount += 1
plt.legend(loc="upper left")
print("rendering...")
plt.show()
```

```
'''
	TRNG PROJECT - TAKE A NOISE DATA FILE AND DRAW ITS VALUE DISTRIBUTION GRAPH
	usage: python3 analyse_ndDistributionPlot.py <filepath> <ntiers>
	ntiers is an int between 1 and 256 and is optional (default 256)
	makes sense to set it to a multiple of 8 or 16
'''

import sys
import matplotlib.pyplot as plt
import convert_noiseDataToDistribution as conv_noise

'''
	function for drawing one file's contents as a value distribution graph
	inputs are filename (path to the noise data file), tiers (nubmer of value
	groups) and graph colour
'''
def draw(fname, tiers=256, limit=1000000, col="red"):
	print("plotting by:", fname)
	data = conv_noise.convert(fname, tiers, limit)
	plt.plot(data, color=(col),label=(fname.split("/")[-1]).split(".")[0])

# main run
if (__name__ == "__main__"):
	# read filename arg
	try:
		fname = sys.argv[1]
	except Exception as e:
		raise ValueError("please provide target data textfile as first argument!")
	# read and parse value limit arg
	try:
		nsamples = sys.argv[2]
	except Exception as e:
		nsamples = 1000000
	try:
		nsamples = int(nsamples)
	except Exception as e:
		raise ValueError("N of samples should be an integer! you provided:", 
nsamples, type(nsamples))
	# readn and parse tier
	try:
		tiers = sys.argv[3]
	except Exception as e:
		tiers = 256
	try:
		tiers = int(tiers)
	except Exception as e:
		raise ValueError("N of tiers should be an integer! you provided:", tiers, 
type(tiers))

	# plot graph
	ticks = [0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 
255]
	plt.xticks(ticks) #set the tick frequency on x-axis
	# set other stuff
	plt.ylim(0, 2)
	plt.ylabel('FREQUENCY') #set the label for y axis
	plt.xlabel('VALUE') #set the label for x-axis
	draw(fname, tiers, nsamples)
	plt.legend(loc="upper left")
	print("rendering...")
	plt.show() #display the graph
```

```
'''
        TRNG PROJECT - CONVERT NOISE DATASET TO DIEHARD-READABLE FORMAT
        usage: python3 convert_dataFileToDiehardASCII.py <filename>
'''
import sys

'''
        main convert function
        takes in filename as argument
'''
def convert(fname):
        # open file
        try:
                f = open(fname, "r")
        except Exception as e:
                raise ValueError("no target file!")

        # read line by line
        data = []
        for i in f:
          data.append(int(i))
        f.close()

        # convert to new file
        newname = fname.split(".")[0]+"-diehard.txt"
        f = open(newname, "a")
        f.truncate(0)
        # iterate over each 40 bytes (10 ints per line)
        for x in range(0, len(data), 40):
                if (x+40>len(data)):
                        break
                for y in range(0, 40):
                        d = hex(data[x+y])[2:].upper()
                        if (len(d) == 1):
                                d = "0"+d
                        f.write(d)
                f.write("\n")

if (__name__ == "__main__"):
        try:
                fname = sys.argv[1]
        except Exception as e:
                raise ValueError("please provide target data textfile as first argument!")

        convert(fname)
```

```
'''
	TRNG PROJECT - QUICK DIEHARD TEXT (DIEQUICK.EXE) RESULT PARSER
	this script converts the output of DIEQUICK.exe (essentialy text files)
	into useful format.
	Intended to be used by other scripts as a module, has no main function.

	2021
'''


'''
	calculates the root mean square error of a given p-values list
	relatively to an x=y line.
'''
def calc_rmse(pred, act):
	if not (len(pred) == len(act)):
		raise ValueError("something went wrong")
	error = 0
	for x in range(len(pred)):
		error += (pred[x]-act[x])*(pred[x]-act[x])
	error /= len(pred)
	error = sqrt(error)
	return error

'''
	converts results of DIEQUICK.exe into useful form
	returns [0] the list of p-values found in the file
	and [1] the root mean square error for it relatively to x=y
'''
def convert(fname):
	# open file
	try:
		f = open(fname, "r")
	except Exception as e:
		raise ValueError("no target file:", fname)

	# read line by line
	data = ["zero"]
	for i in f:
		data.append(i)
	f.close()

	# these are hard-coded numbers of lines that contain p-values of the test
	# do. not. change them. if you use the same release i did.
	# might need adjustments if you use a different test suite, though!
	pstrings = []
	pstrings += data[2:11]+[data[17]]+[data[19]]+[data[29]]+[data[38]]
	pstrings += data[40:65]+data[77:97]+data[98:180]+data[185:187]
	pstrings += data[191:216]+data[220:230]+data[238:258]+data[263:283]
	pstrings += data[298:299]+data[301:311]+data[327:328]

	# parse p-values out of file strings
	pvalues = []
	for s in pstrings:
		pvalues.append(float("0."+s.split(".")[-1][0:4]))
	pvalues = sorted(pvalues)

	# generate ideal values, 4 digits past the dot
	idealvalues = []
	for x in range(len(pvalues)):
		idealvalues.append(round(x/len(pvalues), 4))

	# return a tuple with [0] being pvalues list and [1] being the RMSE
	return pvalues, calc_rmse(idealvalues, pvalues)
```

```
'''
	TRNG PROJECT - NOISE DATA TO DISTRIBUTION GRAPH CONVERTER
	this is a utility file for other scripts to use and it has no main run.
	it expects a .txt file with one byte value written down as a decimal number per 
line
	these are generated by trng_getNoiseData and prng_getNoiseData

	the "tiers" setting sets the number of groups the values will be grouped into
	256 (default) tiers means one tier per value, so the graph would show
	the result for every value
	a setting of 16 would group values into 16 ranges of (256/16) values
	(0...15, 16...31, 32...47, ... ...)
	this is sometimes needed to see if the general graph direction is flat or not
'''
def convert(fname, tiers=256, limit=1000000):
	# attempt to open the noise data file
	try:
		f = open(fname, "r")
	except Exception as e:
		raise ValueError("no target file!")
	data_unnorm = []
	data = []
	# make the unnormalized output array
	for x in range(tiers):
		data_unnorm.append(0)
	# parse the file
	lines_read = 0
	for i in f:
		data_unnorm[int(int(i)/(256/tiers))] += 1
		lines_read += 1
		if (lines_read == limit):
			print("hit user-set value number limit for", fname)
			break
	# make the normalized [0:1] range output array
	for t in data_unnorm:
		data.append(t/max(data_unnorm))
	return data
```

```
'''
    TRNG PROJECT - DEMO PROGRAM: RANDOM PRIME NUMBER GENERATOR
'''

import random
import math
from trnglib import trng

# checks if a number is prime
def checkPrime(n, trng, length_bytes=16, k=128):
    # detect obvious cases
    if n == 2 or n == 3:
        return True
    if n <= 1 or n % 2 == 0:
        return False
    # find r and s
    s = 0
    r = n - 1
    while r & 1 == 0:
        s += 1
        r //= 2
    # do k tests
    for _ in range(k):
        # a = randrange(2, n - 1)
        a = int.from_bytes((trng.getRandomBytesArray(length_bytes)), 'big')
        if (a > n):
            a -= n
        if (a < 2):
            a = 2
        x = pow(a, r, n)
        if x != 1 and x != n - 1:
            j = 1
            while ((j < s) and (x != n - 1)):
                x = pow(x, 2, n)
                if x == 1:
                    return False
                j += 1
            if x != n - 1:
                return False
    return True



# returns a prime number
def generatePrime(trng, length=128):
    if not (length % 8 == 0):
        raise ValueError("bad key length!")
    length_bytes = int(length/8)
    p = 4
    while not checkPrime(p, trng, length_bytes, 128):
        p = int.from_bytes((trng.getRandomBytesArray(length_bytes)), 'big')
        p |= (1 << length - 1)
        p |= 1
        print("tried for prime", p)
    print("\nfound prime:", p, "\n\n")
    return p

# main run
if (__name__ == "__main__"):

    device = trng(portname="/dev/cu.wchusbserialfd120")
    device.connect()

    primes = []
    for x in range(4):
        primes.append(generatePrime(device))

    device.disconnect()

    print("final list of primes is:\n")
    for p in primes:
        print(p)
```

```
'''
	TRNG PROJECT - DEMO PROGRAM: SIMPLE TEST
	usage: python3 demo_simpletest.py

	connects to the trng
	gets one random byte, one random uint32 and a stream of 16 random bytes
	disconnects from the trng

	2021
'''
from trnglib import trng
from time import sleep

# setup and connect the TRNG
print("")
device = trng(portname='/dev/cu.wchusbserialfd120')
device.connect()
sleep(1)

# byte read
print("\ngetting a random byte from TRNG")
v = device.getRandomByte()
print("got value:", v)
sleep(1)

# int read
print("\ngetting a random 32-bit integer from the TRNG")
v = device.getRandomInt()
print("got value:", v)
sleep(1)

# print stream of bytes
print("\nsetting the device to byte stream mode and printing 16 random bytes")
device.startStream()
for x in range(16):
	v = device.readStream()
	print("value #", x, ":", v)
device.stopStream()
sleep(1)

# disconnect
print("")
device.disconnect()
print("")
sleep(1)
```

```
'''
	TRNG PROJECT - SOFTWARE ALGORITHM OUTPUT DATA GATHERER
	gathers results of mt, urand and aes noise generators into text files
	one line - one byte value

	2021
'''

# imports

import serial
import time
from random import *
import os
import sys
from Crypto.Cipher import AES

# functions

def acquire_mt(n_samples = 1000):
	# do plot over n samples
	n_aliveMessage = n_samples / 8
	out_data = []
	for x in range(n_samples):
		out_data.append(randint(0, 255))
		if (x % n_aliveMessage == 0):
			print("i'm still alive. MT samples generated:", x)

	f = open("mtData.txt", "a")
	f.truncate(0)
	for i in out_data:
		f.write(str(i) + "\n")
	f.close()

def acquire_urand(n_samples = 1000):
	# do plot over n samples
	n_aliveMessage = n_samples / 8
	out_data = []
	for x in range(n_samples):
		out_data.append(randint(0, 255))
		if (x % n_aliveMessage == 0):
			print("i'm still alive. URAND samples generated:", x)
	f = open("urandData.txt", "a")
	f.truncate(0)
	for i in out_data:
		f.write(str(i) + "\n")
	f.close()

def acquire_aes(n_samples = 1000):
	# do plot over n samples
	# each cycle generates 16 bytes, so
	n_samples_orig = n_samples
	n_samples = round(n_samples / 16)+1
	n_aliveMessage = round(n_samples / 8)
	out_data = bytearray()
	key = os.urandom(AES.block_size)
	iv = os.urandom(AES.block_size)
	plaintext  = os.urandom(AES.block_size)
	encryptor = AES.new(key, AES.MODE_CBC, IV=iv)
	for x in range(n_samples):

		ciphertext = encryptor.encrypt(plaintext)
		out_data += ciphertext
		plaintext = bytes(a ^ b for (a, b) in zip(plaintext, ciphertext))
		if (x % n_aliveMessage == 0):
			print("i'm still alive. AES samples generated: ~", x*16)
	out_data = out_data[:n_samples_orig]
	f = open("aesData.txt", "a")
	f.truncate(0)
	for i in out_data:
		f.write(str(i) + "\n")
	f.close()
```

```
# main run

if (__name__ == "__main__"):
        try:
                n_samples = sys.argv[1]
        except Exception as e:
                raise ValueError("please provide N of samples as first argument!")

        try:
                n_samples = int(n_samples)
        except Exception as e:
                raise ValueError("N of samples should be an integer!")

        acquire_mt(n_samples)
        acquire_urand(n_samples)
        acquire_aes(n_samples)
```

```
'''
	TRNG PROJECT - SOFTWARE ALGORITHM WORK TIME MEASURER
	gathers times of N uint32's generation for mt, urand and aes noise
	prints it to the screen

	2021
'''

# imports

import sys
from random import *
import os
from Crypto.Cipher import AES
import time

# time calculators

def get_urand_time(n_samples = 1000):
	start_time = time.time()
	#result = os.urandom(n_samples)
	for x in range(n_samples):
		temp = os.urandom(4) # 4 bytes for uint32
	print("URAND: %s seconds" % (time.time() - start_time))

def get_mt_time(n_samples = 1000):

	start_time = time.time()
	for x in range(n_samples):
		temp = randint(0, 4294967295) # uint32 range
	print("MT:    %s seconds" % (time.time() - start_time))

def get_aes_time(n_samples = 1000):
	# each cycle generates 16 bytes, so 4 uints per cycle
	n_samples = round(n_samples / 4) + 1

	#start of the random engine action
	out_data = bytearray()
	start_time = time.time()
	key = os.urandom(AES.block_size)
	iv = os.urandom(AES.block_size)
	message  = os.urandom(AES.block_size)
	for x in range(n_samples):
		encryptor = AES.new(key, AES.MODE_CBC, IV=iv)
		message = encryptor.encrypt(message)
		iv, message = message, iv
		out_data += message
	print("AES:   %s seconds" % (time.time() - start_time))

# main

if (__name__ == "__main__"):
	try:
		n_samples = sys.argv[1]
	except Exception as e:
		raise ValueError("please provide N of samples as first argument!")

	try:
		n_samples = int(n_samples)
	except Exception as e:
		raise ValueError("N of samples should be an integer!")

	get_urand_time(n_samples)
	get_mt_time(n_samples)
	get_aes_time(n_samples)
```

```
'''
	TRNG PROJECT - SOFTWARE ALGORITHM WORK TIME MEASURER
	gathers times of N bytes generation for mt, urand and aes noise
	prints it to the screen

	2021
'''

# imports

import sys
from random import *
import os
from Crypto.Cipher import AES
import time

# time calculators

def get_urand_time(n_samples = 1000):
	start_time = time.time()
	#result = os.urandom(n_samples)
	for x in range(n_samples):
		temp = os.urandom(1)
	print("URAND: %s seconds" % (time.time() - start_time))

def get_mt_time(n_samples = 1000):

	start_time = time.time()
	for x in range(n_samples):
		temp = randint(0, 255)
	print("MT:    %s seconds" % (time.time() - start_time))

def get_aes_time(n_samples = 1000):
	# each cycle generates 16 bytes, so
	n_samples = round(n_samples / 16)+1

	#start of the random engine action
	out_data = bytearray()
	start_time = time.time()
	key = os.urandom(AES.block_size)
	iv = os.urandom(AES.block_size)
	message  = os.urandom(AES.block_size)
	for x in range(n_samples):
		encryptor = AES.new(key, AES.MODE_CBC, IV=iv)
		message = encryptor.encrypt(message)
		iv, message = message, iv
		out_data += message
	print("AES:   %s seconds" % (time.time() - start_time))

# main

if (__name__ == "__main__"):
	try:
		n_samples = sys.argv[1]
	except Exception as e:
		raise ValueError("please provide N of samples as first argument!")

	try:
		n_samples = int(n_samples)
	except Exception as e:
		raise ValueError("N of samples should be an integer!")

	get_urand_time(n_samples)
	get_mt_time(n_samples)
	get_aes_time(n_samples)
```

```
'''
	TRNG PROJECT - DEVICE TEST CALIBRATION UTILITY
	plots the distribution of things to the console as a rough ASCII graph
	used mainly for calibration and evenity testing

	2021
'''

from trnglib import trng
import time
from random import *
from trng_consolegraph import plotData

device = trng(portname='/dev/cu.wchusbserialfd120')
trng.connect()
trng.startStream()

# do plot over n samples
chunk_size = 4096*32
while True:
	data = []
	for x in range(chunk_size):
		data.append(int.from_bytes(trng.readStream(), "big"))
	plotData(data, tiers=64)
```

```
'''
	TRNG PROJECT - FAST CONSOLE PLOT
	plots the distribution of things to the console as a rough ASCII graph
	used mainly for calibration and evenity testing

	2021
'''

glob_block = "█"
glob_halfblock = "▒"
glob_empty = " "

from random import *
import os

def clearConsole():
    command = 'clear'
    if os.name in ('nt', 'dos'):  # If Machine is running on Windows, use cls
        command = 'cls'
    os.system(command)

def getTierData(data, values = 256, tiers = 16):
	tierSize = values / tiers
	tierQuantities = []
	for t in range(tiers):
		tierQuantities.append(0)

	for d in data:
		tierQuantities[int(d / tierSize)] += 1

	return tierQuantities

def printPlot(tierData):
	tiers = len(tierData)
	maxQuantity = max(tierData)
	normalizedTiers = []
	for t in tierData:
		normalizedTiers.append(t*10/maxQuantity)
	for x in range(9, 0, -1):
		str = ""
		for t in range(len(normalizedTiers)):
			if (normalizedTiers[t] >= x):
				str += glob_block
			else:
				str += glob_empty
		print(str)

def plotData(data, values = 256, tiers = 16):
	t = getTierData(data, values, tiers)
	clearConsole()
	printPlot(t)

def test():
	data = []
	for x in range(256):
		data.append(randint(0, 255))
	plotData(data)

if (__name__ == "__main__"):
	test()
```

```
'''
	TRNG PROJECT - DEVICE OUTPUT DATA GATHERER FOR N VALUES
	gathers N samples from the TRNG and puts them into a txt file
	called "deviceData.txt"
	the file is wiped before writing!!!
	if the program/the TRNG crash on execution, the file contents are
	saved up until the moment it crashed.

	2021
'''

import sys
from trnglib import trng


def getNumbers(n_samples):
	if (n_samples < 10000):
		n_aliveMessage = round(n_samples/100)
	else:
		n_aliveMessage = round(n_samples/1000)

	f = open("deviceData.txt", "a")
	f.truncate(0)

	device = trng(portname="/dev/cu.wchusbserialfd120")
	device.connect()
	device.startStream()

	for x in range(n_samples):
		f.write(str(device.readStream()) + "\n")
		if (x % n_aliveMessage == 0):
			print("i'm still alive. samples acquired:", x)

	device.stopStream()
	f.close()



if (__name__ == "__main__"):
	try:
		n_samples = sys.argv[1]
	except Exception as e:
		raise ValueError("please provide N of samples as first argument!")

	try:
		n_samples = int(n_samples)
	except Exception as e:
		raise ValueError("N of samples should be an integer!")

	getNumbers(n_samples)
```

```
'''
        TRNG PROJECT - DEVICE N BYTES GENERATION TIME MEASURE SCRIPT
        usage: python3 trng_timetestBytes.py <int n samples>
        2021
'''

import sys
from trnglib import trng
import time

def get_trng_time(n_samples = 1000):
        device = trng(portname="/dev/cu.wchusbserialfd120")
        device.connect()
        start_time = time.time()
        device.startStream()
        for x in range(n_samples):
                temp = device.readStream()
        device.stopStream()
        print("TRNG: %s seconds" % (time.time() - start_time))

if (__name__ == "__main__"):
        try:
                n_samples = sys.argv[1]
        except Exception as e:
                raise ValueError("please provide N of samples as first argument!")

        try:
                n_samples = int(n_samples)
        except Exception as e:
                raise ValueError("N of samples should be an integer!")

        get_trng_time(n_samples)
```

```
'''
	TRNG PROJECT - DEVICE N 32BIT UNSIGNED INTS GENERATION TIME MEASURE SCRIPT
	usage: python3 trng_timetestInts.py <int n samples>
	2021
'''

import sys
from trnglib import trng
import time

def get_trng_time(n_samples = 1000):
	device = trng(portname="/dev/cu.wchusbserialfd120")
	device.connect()
	start_time = time.time()
	device.startStream()
	for x in range(n_samples):
		temp = device.readStreamInt()
	device.stopStream()
	print("TRNG: %s seconds" % (time.time() - start_time))



if (__name__ == "__main__"):
	try:
		n_samples = sys.argv[1]
	except Exception as e:
		raise ValueError("please provide N of samples as first argument!")

	try:
		n_samples = int(n_samples)
	except Exception as e:
		raise ValueError("N of samples should be an integer!")

	get_trng_time(n_samples)
```

```
'''
	TRNG PROJECT - INTERFACE LIBRARY
	a library to facilitate the connection to the arduino-based TRNG

	2021
'''

# imports

import serial
from time import sleep

# internal vars

class trng:

	'''
	class instance initializer
	'''
	def __init__(self, portname, verbose=True):
		self.portname = portname
		self.verbose = verbose
		self.connection = None

	'''
	tries establishing connection to the port PORTNAME
	returns True on success and False (and a printed warning) on failure
	'''
	def connect(self):
		if (self.verbose):
			print("Trying to connect to the TRNG device...")
		try:
			self.connection = serial.Serial(
				port=self.portname,
				baudrate=115200,
			    bytesize=serial.EIGHTBITS,
			)

		except Exception as e:
			if (self.verbose):
				print("TRNG connection failed! error:", e)
			return False
		sleep(2)
		if not self.connection.is_open:
			self.connection.open()
		if (self.verbose):
			print("TRNG connection successful!")
		return True

	'''
	disconnects the device
	'''
	def disconnect(self):
		self.sendOption(b"D")
		print("Trying to disconnect to the TRNG device...")
		sleep(1)
		if (not self.connection == None):
			self.connection.close()
			if (self.verbose):
				print("device disconnected successfully.")
		else:
			if (self.verbose):
				print("device not connected!")

	'''
	checks if the connection is there
	'''
	def checkConnection(self):
		if (self.connection == None):
			raise ValueError("device not connected!")
			return False
		return True
```

```
'''
sends one character of message to the device
message is a string, only the first letter of it matters
ignores case
'''
def sendOption(self, message):
        if (self.checkConnection()):
                self.connection.write(message)
        else:
                if (self.verbose):
                        print("device not connected!")

'''
sets the TRNG into S(tream) mode
'''
def startStream(self):
        self.sendOption(b"S")
'''
stops the TRNG's S(tream) mode with the E(nough) command
'''
def stopStream(self):
        self.sendOption(b"E")


'''
gets one single random byte from the trng
returns it as an INTEGER
'''
def getRandomByte(self):
        self.sendOption(b"B")
        val = self.connection.read()
        return int.from_bytes(val, "big")


'''
gets N random bytes from the trng
'''
def getRandomBytesArray(self,n):
        if (self.checkConnection()):
                self.startStream()
                bytes = bytearray()
                for x in range(n):
                        val = self.connection.read()
                        bytes += val
                self.stopStream()
                return bytes




'''
gets a random unsigned 32bit integer
'''
def getRandomInt(self):
        if (self.checkConnection()):
                bytes = bytearray()
                for x in range(4):
                        self.sendOption(b"B")
                        val = self.connection.read()
                        bytes += val
                randint = int.from_bytes(bytes, "big")
                return randint

'''
reads whatever is on the serial
is to be used after startStream
use stopStream after you're done with it
returns its decimal value as a python integer
'''
def readStream(self):
        val = self.connection.read()
        return int.from_bytes(val, "big")


'''
same as above but as a single element python bytes array
'''
def readStreamAsBytes(self):
        if (self.checkConnection()):
                val = self.connection.read()
                return val

'''
reads whatever is on the serial 4 times
makes a 32bit uint out of that
is to be used after startStream
use stopStream after you're done with it
'''
def readStreamInt(self):
        if (self.checkConnection()):
                bytes = bytearray()
                for x in range(4):
                        val = self.connection.read()
                        bytes += val
                randint = int.from_bytes(bytes, "big")
                return randint
```

```
/*
  TRNG PROJECT - ARDUINO SOFTWARE
  2021
*/

void setup() {
  Serial.begin(115200);
  DDRC = 0x00; // reg C all input
  DDRB = 0xFF; // reg B all output, we will only use B5
               // for some debugging via on board LED
  // show that it's alive
  blinky(3);
  delay(120);
  blinky(3);

}

// blinks N times. is there for debug/demo purposes
void blinky(unsigned char n)
{
  for (unsigned char t = 0; t < n; t++)
 {
   PORTB = 0xFF;
   delay(60);
   PORTB = 0x00;
   delay(60);
 }
}

// resets the connection - this resolves the locked port bug
void softReset()
{
  blinky(1);
  delay(50);
  Serial.end();
  delay(500);
  Serial.begin(115200);
  blinky(4);
}

// function for accumulating 1 byte worth of uncertainty
char getByteData()
{
  char out = 0,
       temp = 0;
  for (unsigned char counter = 0; counter < 8; counter++)
  {
    out = out << 1;
    temp = (PINC >> 5) & 0x01;
    out |= temp;

    // slow noise compensation - to be removed when the circuitry gets upgraded
    for (int a = 0; a < 128; a++)
    {
     __asm__ __volatile__ ("nop\n\t");
    }
  }
  return out;
}

// generates a random byte by getting 2 uncertainty bytes and XORing them for
// even more advanced effect
char getRandomByte()
{
  char byte1 = getByteData();
  char byte2 = getByteData();
  char out = byte1 ^ byte2;
  return out;
}

// the 'awit for command' loop
char opt = '0';
```

```
void loop()
{
  // READ OPTION

  if (Serial.available() > 0)
  {
    opt = Serial.read();
  }

  // D(isconnects) the connection
  if (opt == 'D')
  {
    softReset();
  }

  // send a single random B(yte)
  else if (opt == 'B')
  {
    char out = getRandomByte();
    Serial.print(out);
  }

  // S(tream) UNTIL E(nough)
  else if (opt == 'S')
  {
    char temp = 0,
         byte1 = 0,
         byte2 = 0;

    while (true)
    {
      // E(nough) breakout sequence
      if (Serial.available() > 0)
      {
        if (Serial.read() == 'E')
          break;
      }
      char out = getRandomByte();
      Serial.print(out);
    }
  }

  // nullify the option if it was not zero
  if (opt != '0')
    opt = '0';
}
```

Appendix 3 – Hardware circuit diagram and notes

In this appendix, a circuit diagram of the device's hardware part is presented, as well as some photos of the build process and the final result.

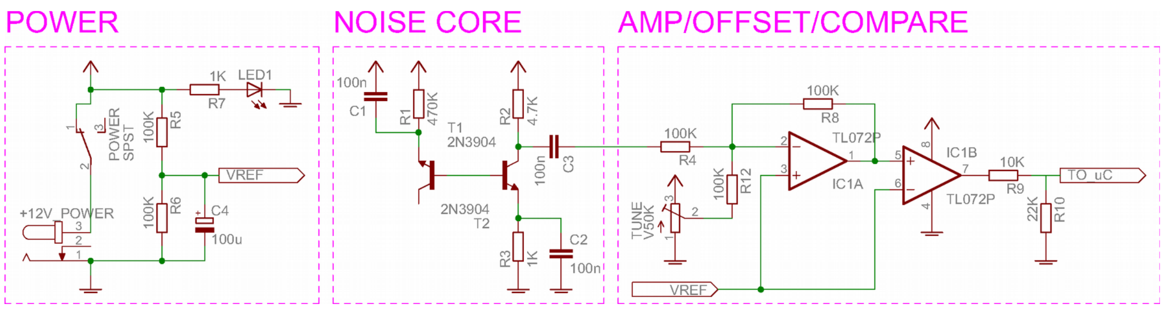

Figure 9. Hardware schematic

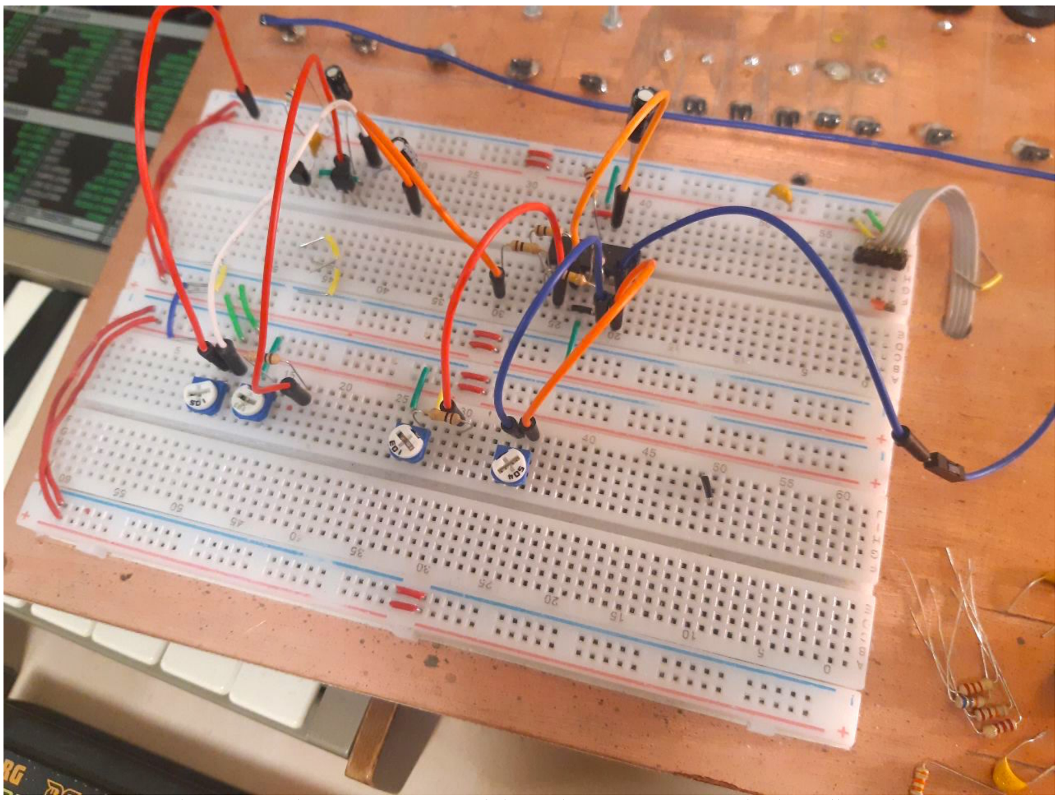

Figure 10. Photo: prototype of the noise generator on the breadboard

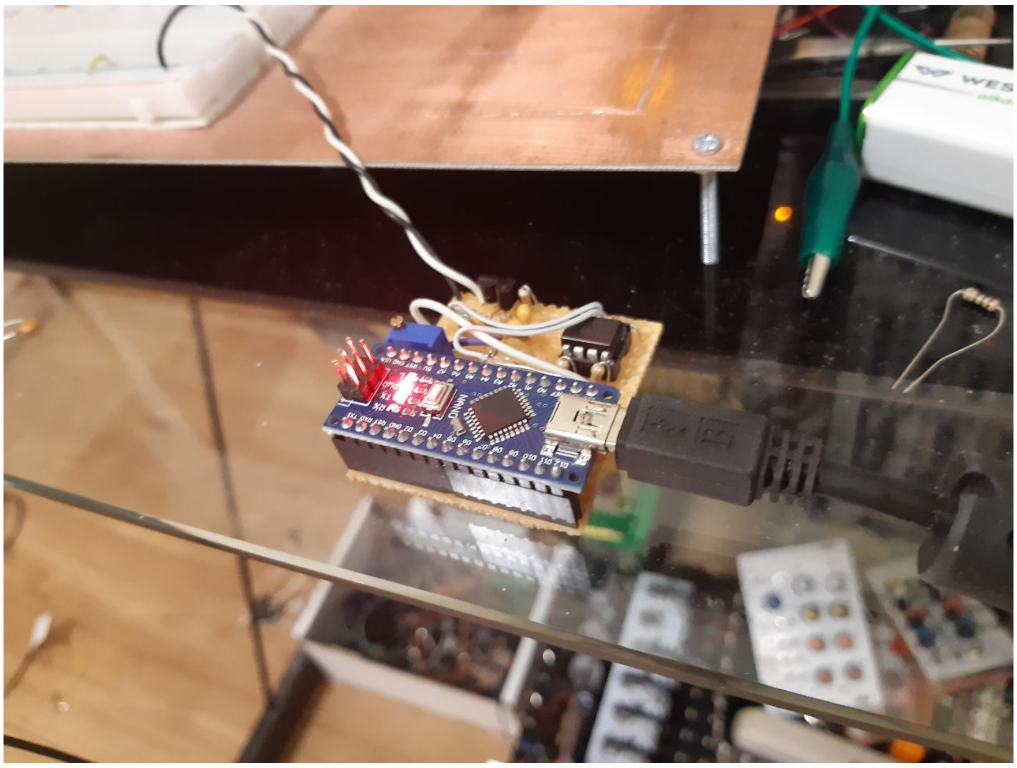
Figure 11. Photo: assembled device board being tested

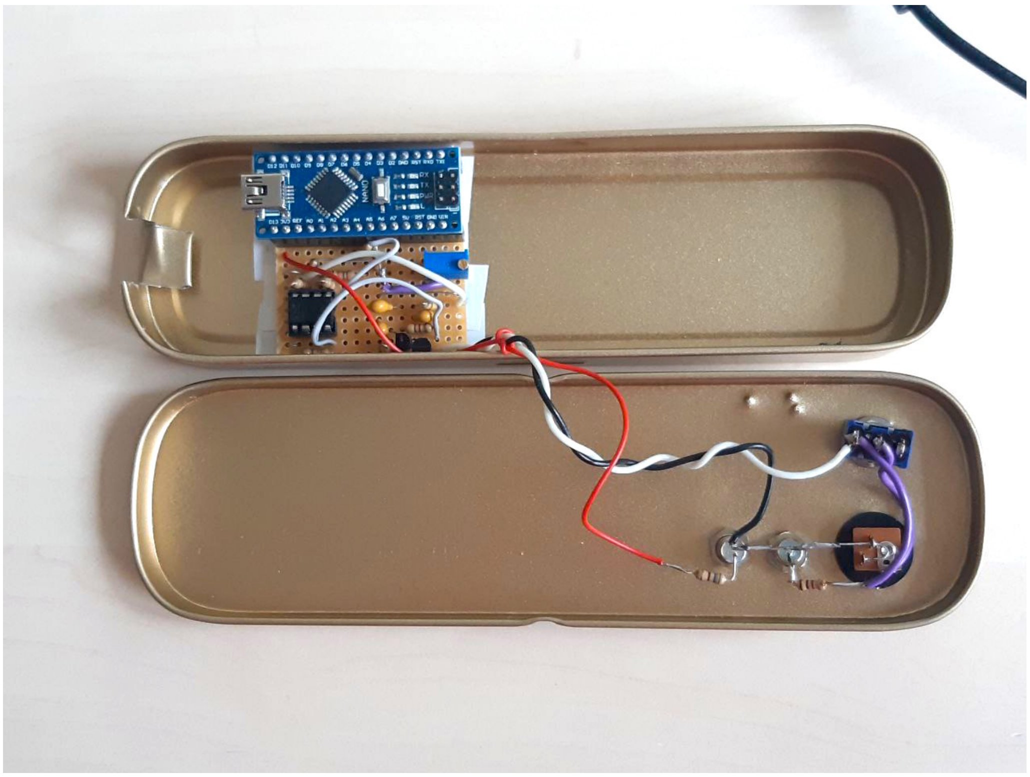
Figure 12. Photo: device put into the enclosure and wired up

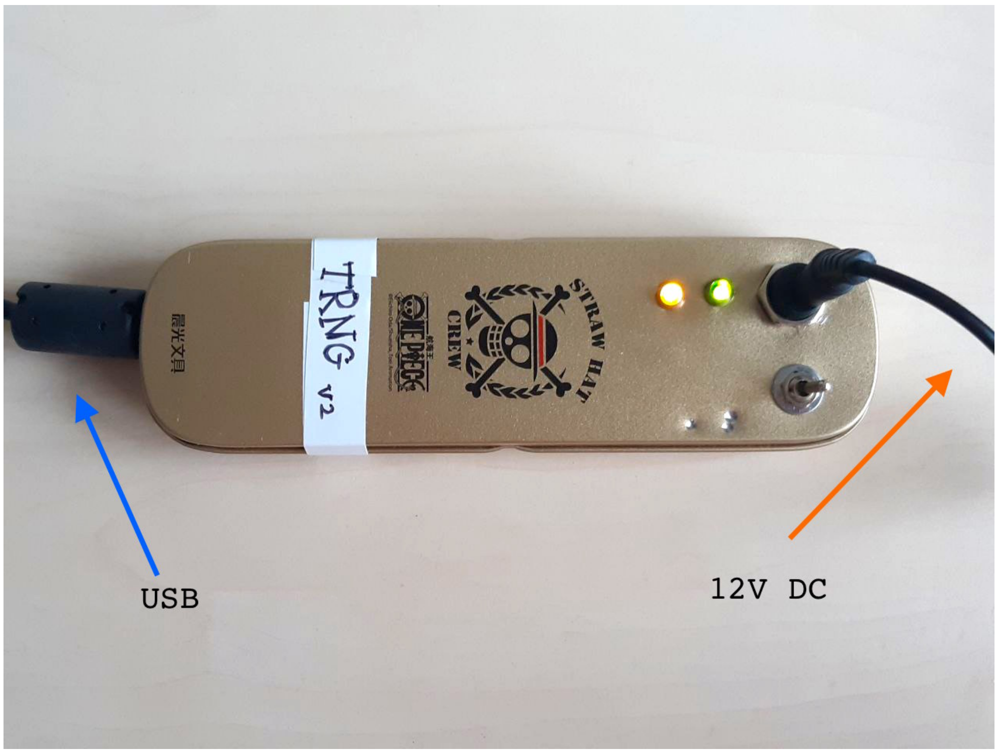

Figure 13. Photo: assembled device is connected and working

Figure 14. Photo: the device and its PSU

Appendix 4 – Data files, additional graphs

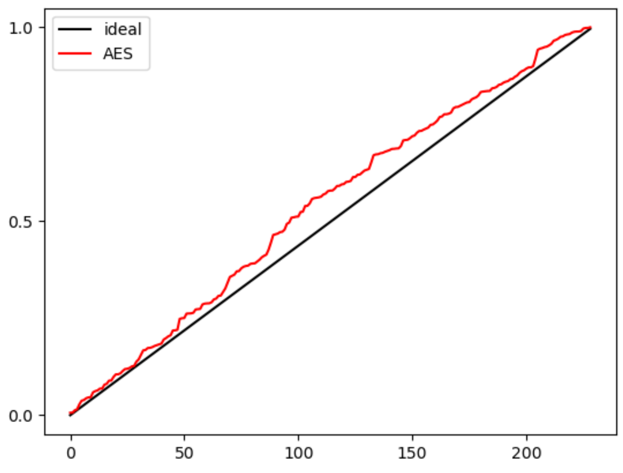

Figure 15. AES p-value distribution

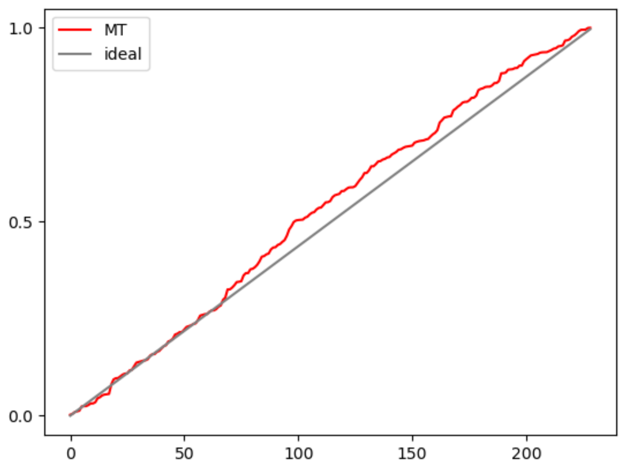

Figure 16. MT p-value distribution

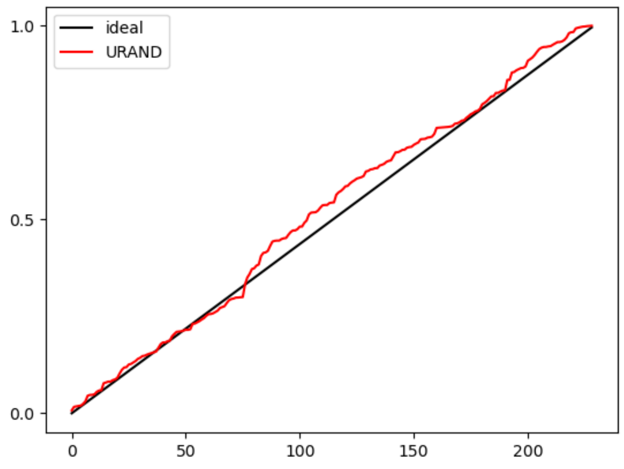

Figure 17. URAND p-value distribution

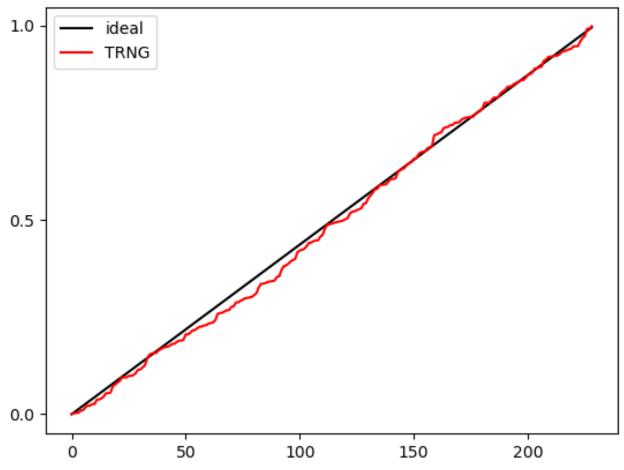

Figure 18. TRNG p-value distribution

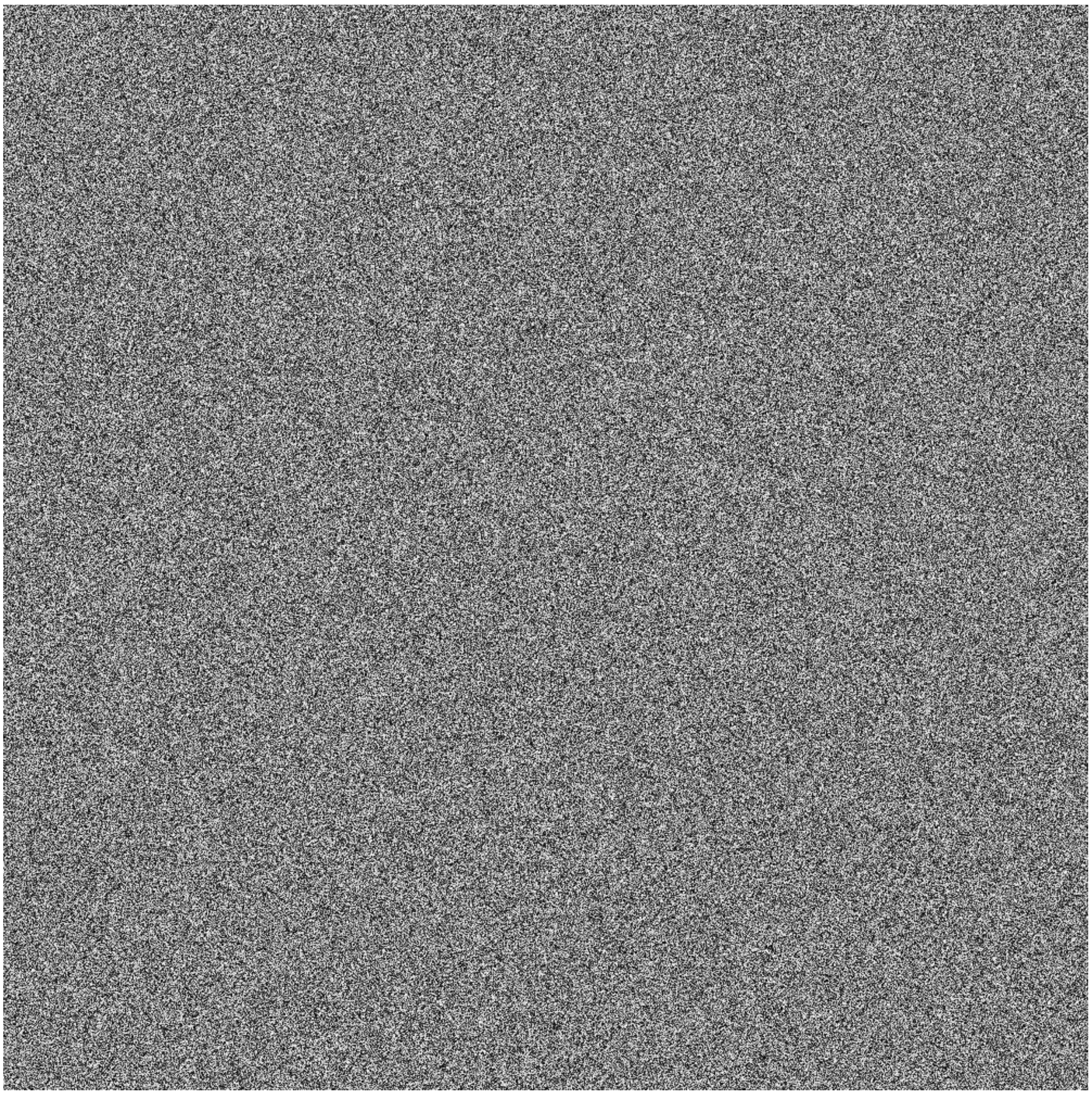
Figure 19. AES visualisation, greyscale, 1000 values per side

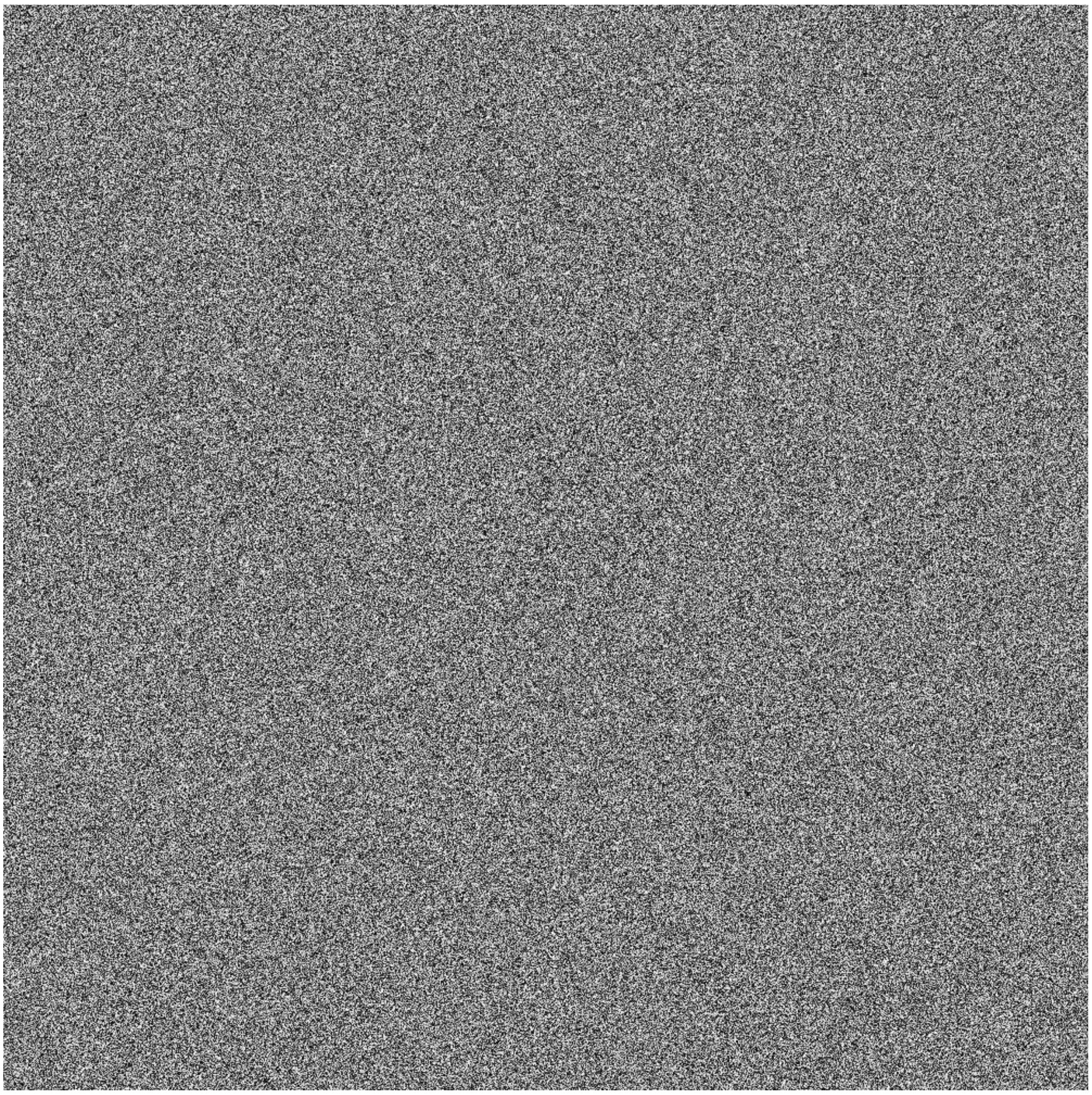
Figure 20. MT visualisation, greyscale, 1000 values per side

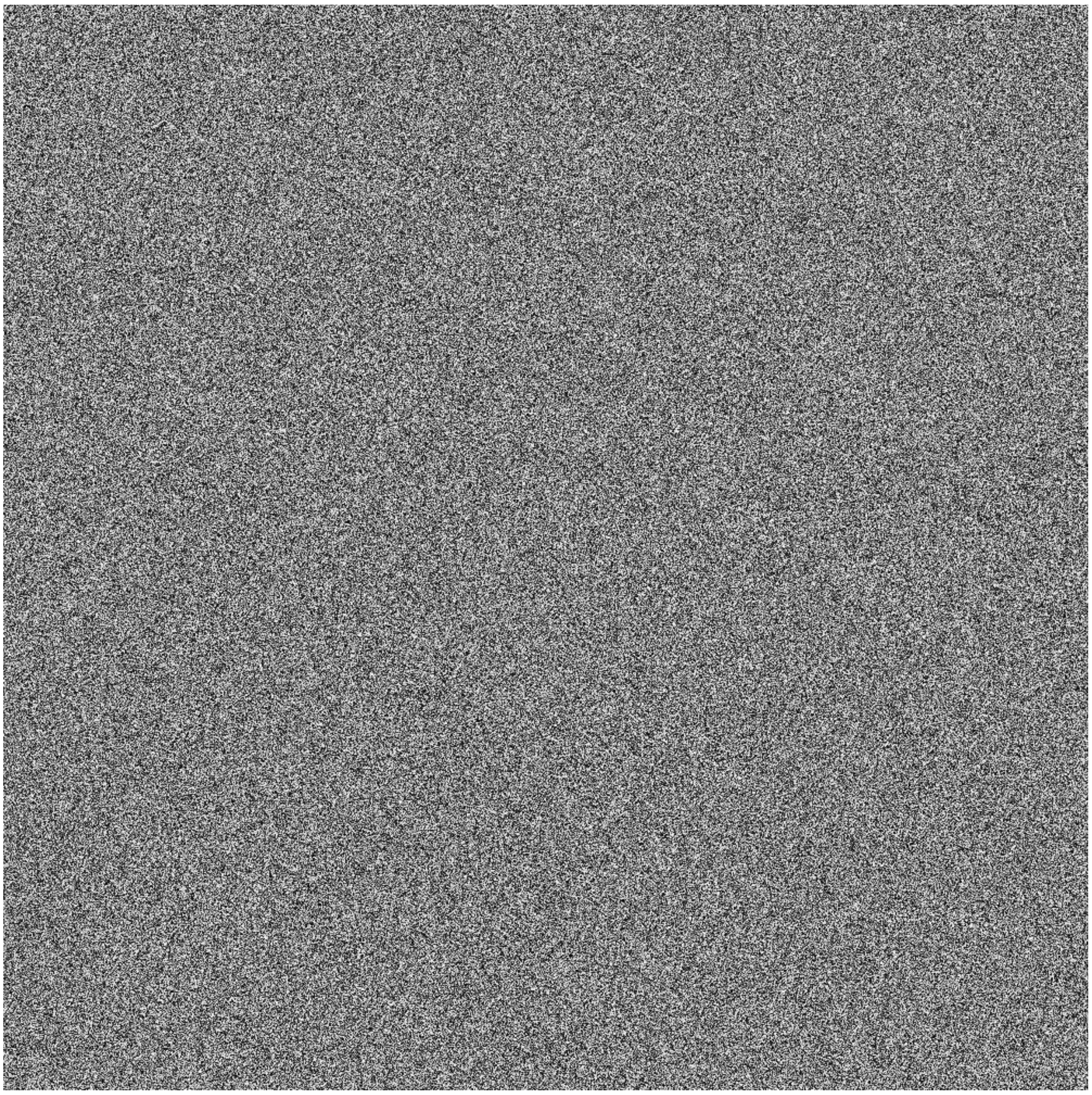
Figure 21. URAND visualisation, greyscale, 1000 values per side

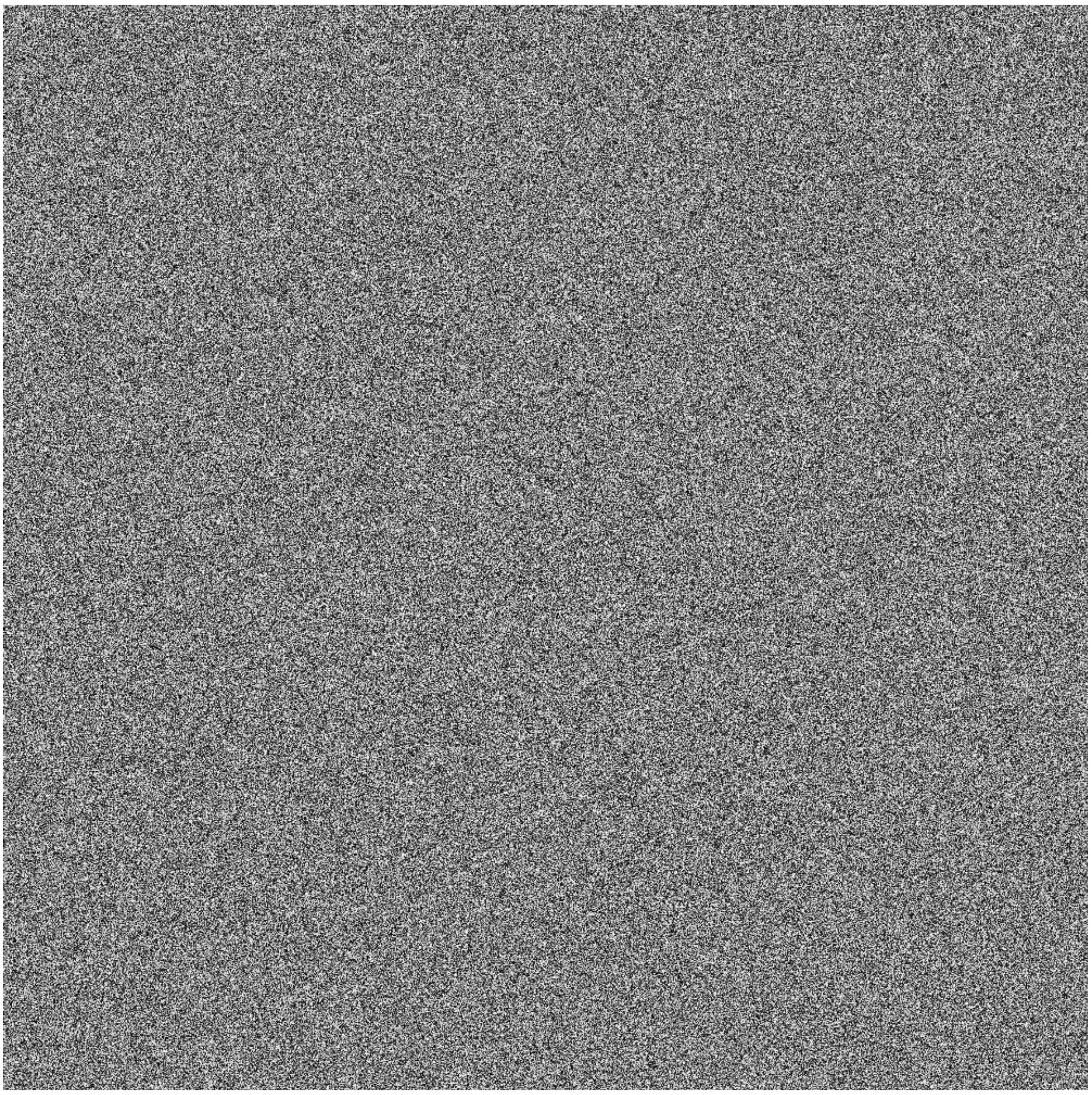
Figure 22. TRNG visualisation, greyscale, 1000 values per side

If one wishes to get the test datasets obtained through testing, the software kit, or any additional data, please, request it via nitimo@ttu.ee.