Efficient Exctraction of True Random Numbers from Quantum System on Resource Constrained Hardware

Love Arreborn
lovar063@student.liu.se

Nadim Lakrouz
nadla777@student.liu.se

1 INTRODUCTION

In computer science, there are many applications for randomly generated numbers. From generating keys for cryptography, salting password hashes, load balancing in distributed systems, memory addressing and so much more. However, the process of producing these random numbers tends to be pseudo-random, e.g. utilizing the current states of various modules [3]. These numbers do not generate true randomness, and in order to heighten security other methods of generating these are required. Current random number generators (*RNG*) are usually implemented in code, using certain states of the host machine as a starting point before running a predetermined algorithm [3]. This pseudo-random number generation (*PRNG*) comes with the drawback that the result is always deterministic, provided that the initial state is known.

Imagine, then, if a malicious attacker somehow manages to ascertain the state a computer was in when it generated a random number, for instance to produce an SSH-key. This attacker then has the opportunity to accurately reproduce the exact, deterministic state that produced said random number, in essence removing the safety that randomness brings. While it may sound unrealistic, the exponential increase in processing power and the burgeoning field of quantum computing does introduce the possibility that one day, what we perceive as random is nothing more than a simple algorithm to crack.

True random numbers, then, cannot be produced solely through code. These systems require some input that is neither replicable nor reproducible. One method that can be realistically used is the inherently random movement of lava lamps [14], which is used as a backup source of randomness for Cloudflare[1]. Another proposed solution for this is quantum random number generation (*QRNG*) [5]. By reading quantum fluctuations from any given source, for instance an optical signal, the inherent natural unpredictability of said source can be harnessed in order to produce a random number from a state that is nigh impossible to reproduce accurately.

In this project, we will be writing firmware for a quantum number generator, which generates true randomness by reading shot noise from an optical signal. Further details about how this signal is produced will be introduced in section 2 and 3, and builds on the work of Clason [2]. This optical signal will be converted to a stream of random, raw bits via an Analog to Digital Converter (*ADC*). In turn, these random bits will be processed via Toeplitz extraction [16] in order to process these bits into random numbers. Some processing has to be done on the microcontrollers themselves in order to ensure that the data is workable, and Toeplitz extraction is a tried and tested method to accomplish this. These random numbers will then be output from the microcontroller to the host computer via USB. This thesis will aim to answer one key research question: How can sampled vacuum fluctuations be processed efficiently in order to output QRNG?

In producing this firmware, several key considerations have to be made in order for this system to be usable in a production environment. The vision for the end product is a simple USB-stick that can be connected to a host, and produce true random numbers from ambient quantum fluctuations. While this system could very well be implemented on physically larger hardware, thus avoiding the constraints that limited hardware introduces, Clason proposes a simpler and cheaper way to achieve QRNG [2]. In keeping with this, utilizing microcontrollers for processing the raw bits extracted further helps to keep the costs low and the solution reasonably complex. Further details regarding the microcontrollers used in our work will be outlined in section 3. However, due to this portability constraint, our implementation needs to work quickly and efficiently on resource constrained hardware. As such, our main question is broken down into two concrete research areas:

Research area 1 (*RA1*): How can Toeplitz extraction be implemented as effectively as possible on resource constrained hardware in order to process raw bits into a workable random number?

Toeplitz extraction been optimized quite well, and previous research can be utilized for this. However, there are still considerations when implementing the firmware for the microcontroller in order to optimize the code. Our goal is to attempt several implementations in order to find the most optimal implementation with the least amount of effective processing time spent on the algorithm.

Research area 2 (*RA2*): How can we ensure that the output of random numbers is not limited by our firmware, but rather only limited by the USB transfer speed of the microcontroller or the ADC?

There will unequivocally be a bottleneck for the processing speed. For instance, the speed at which the ADC can process the optical signal into raw bits as well as the speed that the

[1]Cloudflare.com, accessed 2025-03-10

USB output can transfer processed random number to the host computer will be limiting factors. Further details on the limitations of the ADC will be outlined in section 3. The slowest of these bottlenecks will inevitably be the limiting factor for any implementation. Our research aims to ensure that our implementation of Toeplitz extraction does not become the limiting factor, but rather processing data fast enough to match or exceed the speed of the hardware.

Section 2 of this article will introduce the theory that allows for QRNG, and how this will be utilized in our works. Section 3 delves further into the hardware and algorithms our work will use, with related works in optimizing Toeplitz extraction listed under section 4. Section 5 will present our methodology as well as some limitations imposed on our work. **More sections to follow as we finish the article**.

2 THEORY

A majority of the research around this topic stems from physics, with implementations of the technology frequently being published and studied by physicists. As such, a brief introduction to the concepts used in previous research as well as an introduction to previous implementations of this technology will be presented in this section.

The idea of an optical QRNG (*OQRNG*) is not a novel one. The basis of the theory is that intrinsically random properties of a quantum process. Stefanov et. al. [13] proposes using the random choice of a photon between two output signals to generate a random stream of bits, however the theory behind it can be applied to other quantum processes as well. This particular theorem has been implemented by Wayne et. al. [15] to create a quantum number generator. While this article proves the efficacy of OQRNG, it utilizes a slightly different method.

2.1 Shot noise quantum fluctuations

Our work revolves around the measurement of shot noise of vacuum states rather than measuring arrival times of photons. Essentially, this is another quantum process with the same inherently random properties as described by Stefanov et. al. [13], but instead using shot noise. As described by Niemczuk [10], shot noise is minor fluctuations in an electrical current, which is inherently random. Reading this property, then, gives us an intrinsically random source from which to generate a random output, which in turn can be processed into a random number.

Implementations of this theory exist, however with significant drawbacks. Shen et. al. [11] presents an implementation using a fairly complex setup, in which a continuous-wave fiber laser is the optical source. They conclude that sampling the shot noise is, indeed, suitable for OQRNG. However, the implementation requires expensive and complex hardware, and the sheer size of the system prohibits it from being portable and easily reproducible in small-scale tests.

A more recent implementation of OQRNG in a smaller scale has been presented by Singh et. al. [12]. This particular implementation uses a bespoke circuit board where all components are present on a single board – e.g., this experimental setup contains an integrated ADC, post-processor, entropy controller and entropy generator. While this article cements the viability of OQRNG using shot noise (despite the article not being confirmed as peer reviewed), the bespoke nature of the circuit board makes this experiment difficult to reproduce. As our thesis will use commercially available ADCs and microcontrollers, the only bespoke component is the shot noise generator itself. Furthermore, the Toeplitz extraction is not run on the microcontroller itself in these experiments – instead, the hashing of these raw bits is done on the receiving computer as this bespoke circuit board featured a relatively weak processor.

In summary, previous research has proven that OQRNG can generate true randomness, and more specifically, Shen et. al. [11] and Singh et. al. [12] both implement OQRNG through readings of shot noise. However, there are limitations in both of these works. Either the system that generates the shot noise is large and complex [11] or the system is built on bespoke hardware with limitations in processing power which prevents a fully integrated system [12]. Furthermore, to the best of our knowledge, most of the work in this field is from the perspective of physicists, and there appears to be little research on this subject in the domain of computer science. Our work aims to bridge this gap by using commercially available hardware (other than the bespoke shot noise generator [2]) and focuses on implementing Toeplitz extraction directly on the microcontroller. Rather than focusing on the intricacies of quantum fluctuations, we will instead approach this problem from a computer science perspective.

3 BACKGROUND

Our work is a practical continuation of the work of Clason [2]. In this work, quantum shot noise originating from photodiodes was studies, and in so doing a prototype device was constructed. This device read from an optical source, outputting analog voltage from the data “seen” by the diodes. A prototype was constructed, in which an LED is read by a photodiode soldered millimeters apart. In this section, we introduce the hardware used for our implementations as well as the considerations taken in order to shift the focus from physics to computer science.

3.1 Optical RNG module

The one bespoke piece of hardware used in this study is the prototype designed by Clason [2] as a part of his masters thesis. This device produces the optical shot noise which will be the source of randomness in our work. Moving forward in this article, we will refer to this as the OQRNG-device.

As described in Clasons work [2], the OQRNG-device is an electro-optical system which measures optical shot noise, generating quantum randomness. The device has an LED and a photodiode positioned a few millimeters apart, ensuring efficient light coupling. The photodiode detects light from the LED, and converts the light into a current signal, which is sent to a transimpedance amplifier to convert it into measurable voltage. In order to minimize disruptions by other external lights, the system is enclosed in a shielded measurment box.

Whereas the exact quantum mechanisms that ensure that this system ensures randomness and further details regarding the

OQRNG-device is better derived directly from Clasons work [2], the end result as it correlates to our study is an inherently random, analog voltage current.

3.2 ADC converter

This analog current isn't suitable to operate on without further processing. As mentioned in section 1, the signal needs to pass through an ADC and converted into raw bits in order for it to be usable. In his thesis, Clason [2] suggests a discrete ADC chip capable of analyzing frequencies higher than 25 MHz, as this is the highest frequency studied in his work. However, in the interest of keeping the implementation light and cheap, we will be using ADCs that provide less samples per second and lower frequencies. This is done mainly for ease of development and access to this hardware.

Many microcontrollers furthermore come equipped with internal ADCs that can be utilized, and while these provide a lower sample size (often around 1 MSPS), the ease of development may be prudent to utilize for this proof-of-concept. While our initial ADC has a fairly low throughput, this can always be upgraded if it ends up becoming too limiting.

Should these internal ADCs prove too limiting, we propose utilizing MAX11102AUB[2] with an effective sample rate of 2 million samples per second (MSPS). This ADC provides a 12 bit sample size, providing roughly 2.86 MB/s of sampled data per second, derived by the following calculation.

$$DataRate = \frac{2,000,000_{MSPS} \times 12}{8 \times 1,048,576} \approx 2.86MB/s \qquad (1)$$

The final output from the ADC, whether built into the microcontroller or an external one, will be a stream of raw bits, as the analog signal from the OQRNG-device is processed.

3.3 Microcontroller

Microcontrollers (MCUs) are compact and low-power computing devices designed primary for embedded systems and real-time applications. Unlike general purpose CPUs, MCUs integrates a processor, memory, and peripherals into a single chip. Thus, enabling efficient and autonomous operation in constrained environments. Various development frameworks and tools are available for programming MCUs, including the Arduino Framework, manufacturer specific SDKs and raw C++, among others. Each framework presents different trade-offs in terms of performance overhead, efficiency, and ease of development.

Modern microcontrollers feature advanced microarchitectural elements to enhance processing capabilities. Some of high performance MCUs utilize dual-issue superscalar architectures, enabling them to run several instructions in one cycle. This type of design greatly improves single-threaded capabilities, which makes these MCUs ideal for tasks that need real-time data processing. The use of dual-issue architectures in embedded devices has demonstrated high execution speeds, along with the added benefit of improved energy efficiency. Thus, making them a great option for applications that require low power but high performance. Research conducted by Lozano et al. [9] specifically explores energy efficient dual-issue processors designs. This study shows that optimizing instruction throughput and power consumption can significantly improve processing efficiency in embedded systems. While energy efficiency is not the primary goal for this study, ensuring that the resulting USB-device does not consume too much power is tangentially relevant for real-world usage.

Since MCUs often function under strict timing requirements, it is critical to have effective ways to access memory and transfer data for processing in real time. High-performance MCUs enhance memory usage in various ways. Some of them use Tightly Coupled Memory[3] (TCM), which gives fast SRAM with specific access routes for important data, avoiding cache misses and guaranteeing consistent performance. Moreover, instruction and data caching techniques, including instruction prefetching and branch prediction, help minimize execution delays in computationally intensive real-time applications. Another important aspect is Direct Memory Access[4] (DMA), which enables data transfer between peripherals such as the ADC and RAM, without CPU intervention. This offloading reduces processing overhead, allowing the MCU to manage fast data transfers effectively. These improvements are especially significant for Toeplitz extraction, where large amount of random data needs to be processed and sent quickly with low delay. Efficient memory management guarantees that randomness extraction can occur rapidly without major slowdowns in computing. Both approaches will be tested during development.

The Teensy 4.1[5], based on the ARM Cortex-M7 is especially good for computationally demanding tasks involving randomness extraction due to its dual-issue superscalar architecture and DSP capabilities. The floating-point unit[6] (FPU) and SIMD-style DSP instructions improves how quickly it can perform bitwise and arithmetic tasks, which are crucial for quick Toeplitz extraction. SIMD-controlled DSP architectures, as described by Han et al. [4], leverage parallel vectorized computation to accelerate matrix operations. Thus, making them highly effective for Toeplitz matrix-vector multiplications. The Cortex-M7 does not have special features for cryptography, its ability to execute multiple instructions quickly, combined with fast memory access and an efficient pipeline design, allows it to effectively manage and process randomness extraction. Moreover, its USB High-Speed (480 Mbit/sec) interface, supporting both USB device and USB host modes, allows rapid transmission of extracted random data to an external system, ensuring minimal bottlenecks in high-rate randomness generation.

In order to evaluate just how much constraint our implementation can allow, our aim is to try our implementation on other MCUs with varying levels of power and hardware support. Whereas Teensy 4.1 is our primary development plat-

[2] Technical specification for MAX11102AUB, accessed 2025-03-13

[3] Technical specification for MAX11102AUB, accessed 2025-03-13

[4] ScienceDirect Journals & Books, accessed 2025-03-13

[5] Teensy developer documentation, accessed 2025-02-27

[6] ScienceDirect Journals & Books, accessed 2025-03-13

form which we will evaluate closely, we aim to run our implementations on Raspberry Pi Pico 2[7] as well as ESP32-S3[8]. Due to the lower computational power of these MCUs, there may be significant issues in utilizing these weaker models, yet they are significantly cheaper and easier to access. Testing of these will consist solely of running the implementation on these controllers and measuring execution speed.

3.4 Toeplitz extraction

The raw bits from the ADC can potentially have some deterministic patterns, and as such have to be processed somehow in order to remove these patterns. Several methods exist for this purpose, and for our work, we will perform this preprocessing via Toeplitz extraction. The main focus of this study is to implement this extraction algorithm as effectively as possible on resource constrained hardware.

A detailed account of the inner workings of Toeplitz extraction can be found in the work of Chouhan et al. [1]. This work focuses on implementing Toeplitz extraction on field-programmable gate-arrays (FPGA), but some specific details can be derived from their work. As these authors describe, Toeplitz extraction is a strong contender for our work due to a lower computational complexity than other alternatives, as well as a relatively easy algorithm to use. This extraction utilizes either matrix multiplication or hashing between a pseudo-random seed and the raw data provided from a high-entropy source of randomness – in our case, the OQRNG-device.

To summarize the theoretical working of Toeplitz extraction (as explained by Chouhan et al. [1]), a pre-determined seed matrix (T) is multiplied with the sampled raw bit matrix (K). The size of the seed is directly dependent on the size of the sampled data, and can be fixed or continually re-sampled as needed. To ensure high levels of entropy, our intuition is that re-sampling the seed from the OQRNG-device continually is prudent. The sample and seed with then be processed with matrix multiplication to remove deterministic patterns, and produce a bitstring that results in our randomly generated number. An example of how this extraction works can be seen below.

$$T = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \quad k = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$h(k) = T \cdot x \mod 2$$

We then perform XOR multiplication on the matrices:

$$h(k) = \begin{bmatrix} (1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1) \\ (0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1) \\ (1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0) \\ (1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0) \end{bmatrix}$$

Which finally simplifies to:

$$h(k) = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

This allows us to interpret the result as a random 4-bit integer:

$$h(k) = (1011)_2 = 11_{10}$$

Whereas this example is quite small in order to demonstrate how the algorithm operates, this can of course be scaled up significantly with larger values for both T and k respectively, granting larger integers. The main focus of this work is implementing this algorithm as efficiently as possible on our MCUs, and as such, several optimization efforts need to be taken into account during our experimentation.

3.5 Summary

With the assumption that the OQRNG-device produces a truly random analog signal, we can now clearly define the scope in which this thesis operates. Considering the maximum conversion speed from the ADC and the USB-output from the MCU, we have a clear bound between 2.86 MS/s (for the ADC) and 480 MB/s (for the USB output) in which Toeplitz extraction needs to be executed. Any speeds over 2.86 MB/s allows us to upgrade the ADC iteratively to continue increasing the output speed. Any implementation of Toeplitz extraction must then execute fast enough on any given microcontroller feasible for the proposed quantum RNG-thumbstick as to not be the decisive limiting factor.

4 RELATED WORKS

Most related works found tend to revolve around optimizing Toeplitz using more advanced hardware, and few works seem to delve into evaluating the implementation in code.

As mentioned in section 3.4, Chouhan et al. [1] utilized FPGA to implement Toeplitz extraction specifically for OQRNG, a work also studied by Zhang et al. [16]. Furthermore, Zhang et al. utilized a standardized min-entropy evaluation to ensure true, unbiased randomness in their result. Both of these implementations utilize powerful hardware where the matrix multiplication is offloaded to FPGA. These implementations provided extraction speeds of between 3.36 Gpbs [16] to 26.57 Gbps [1]. Neither of these implementations operate on constrained hardware, instead creating bespoke circuit boards for their works.

Another important point that is often neglected in randomness extraction using Toeplitz matrices is how the seed key is handled when forming the Toeplitz matrix. Numerous systems,

[7] Raspberry Pi Pico 2 documentation, accessed 2025-03-13
[8] ESP32-S3 documentation, accessed 2025-03-13

such as those developed by Chouhan et al. [1] and Zhang et al. [16], depends on fixed seeds. Fixed seeds can create security risks over prolonged operation time. To tackle this problem, Lin et al [8] proposed a method for seed-renewable Toeplitz post-processing in QRNG. Their strategy incorporates a dynamic seed pool within the FPGA, where each instance of post-processing picks a new, randomly selected seed. Thus, minimizing temporal correlations between extractions. Furthermore, an external seed updating mechanism via PCIe ensures that seeds are refreshed whenever a certain security limit is reached. Compared to fixed-seed methods, this renewable approach enhances cryptographic robustness and ensures sustained high-security randomness extraction in real-world applications.

Efficient Toeplitz matrix-vector multiplication (TMVM) is critical for optimizing randomness extraction that relies on Toeplitz, particulary in constrained hardware environments. Liao et al. [7] showed that this process could be greatly accelerated using Fast Fourier Transform (FFT) and its inverse (IFFT). Thus, reducing computational complexity from $O(n^2)$ to $O(nlogn)$. Their implementation on FPGA utilized this approach for deep neural networks, resulting in a 28.7 times decrease in model size while still achieving fast inference speeds. By using FFT and IFFT acceleration, Toeplitz post-processing for randomness extraction could achieve higher throughput. Thus, potentially could improve performance and reduce latency.

5 METHODOLOGY

With the consideration that our work revolves around optimizing Toeplitz extraction in order to quickly process random bits into a random number, we will take an iterative approach. For our tests, we will use a pre-defined stream of raw bits which is sent to the microcontroller via USB, and run several different implementations of Toeplitz extraction to produce numbers. As we always use a pre-defined bitstream, the result will at this stage be deterministic, giving us a clear indication whether the algorithm works as intended.

However, in order to ensure the results work with varying data, we cannot limit ourselves to simply one stream of bits. The main point of the algorithm is to remove patterns in the bitstream that may lead to less randomized results. As such, we will manually produce several bitstreams to use for our tests – each with varying degrees of repeated patterns that should be eliminated by the algorithm. Furthermore, in order to see how well our implementation will work with realistic data from the OQRNG generator, we will sample streams of bits directly from the source.

5.1 Iterative approach

Implementation of Toeplitz extraction will begin by a naive implementation not optimized for speed, but rather for accuracy. This implementation will be executed on a separate computer in order to produce the correct random number for every provided bitstring. These will be used as our baseline for accuracy for future iterations.

This naive implementation will then be flashed to our microcontrollers, beginning with Teensy 4.1 as this is the more capable of the microcontrollers used for this experiment. Code to measure the execution speed in microseconds will be implemented and tested before we load the naive implementation on said microcontroller. We expect that several implementations may be too resource intensive or have a memory complexity far greater than our cheaper, less capable microcontroller are able to handle, and as such these may not be able to be tested until a few iterations of optimization has occurred.

Each iteration will consist of incremental improvements to the algorithm. Our initial investigation has shown several avenues for improving the throughput of the algorithm, and each planned iteration will be discussed briefly. Depending on the empirical results of these iterations, there may be a need for further optimization iterations other than those listed.

Iteration 1 - Naive implementation: The naive implementation consists of a Toeplitz matrix T and key k, acquired from fixed slices of the bitstream. Then, matrix-vector multiplication will be performed using T and k, thus performing Toeplitz extraction – which finally will eliminate jitter and produce a random number. The implementation of this in code will be quite simple, using nested loops to iterate over all values and multiply them. We expect that the throughput of this naive implementation will be far away from optimized iterations, and result in $O(n^2)$ complexity.

Iteration 2 - Efficient data structures: The naive implementation will not utilize efficient data structures for maximum speed, rather it will likely use fixed-size arrays which need to be iterated over for a time complexity of $O(n^2)$. Our hypothesis is that more efficient data structures may allow data to be processed in a far more efficient manner – however, this scenario poses a risk for greater memory complexity which may not be suitable for use on microcontrollers. Which data structures might be most feasible will be evaluated during this iteration and as such is not preordained.

Iteration 3 - Bitshifting: We expect that Toeplitz extraction can be significantly improved with bitwise XOR operations, thus reducing overhead and memory usage. Instead of explicitly constructing the Toeplitz matrix, a right-shift operation can be used to dynamically reconstruct matrix rows. Additionally, matrix-vector multiplication can be improved using XOR operations to extract bits more efficient, thus minimizing unnecessary computations.

Iteration 4 - Batching: Finally, we consider the concept of batching larger amounts of bits for processing. Consider that an input buffer of bits is read from the ADC and stored, waiting for processing. Rather than taking 64 bits, and shifting them one by one, we can take two batches of 64 bits and multiply them directly using XOR bitshifting – eliminating the need to process these bit-by-bit. As we will have a constant stream of bits from the ADC, we theorize that this method will allow for better performance in real-time processing over reading individual bits.

Iteration 5 - ARM Hardware instructions: Rather than performing bitshifting operations in the code itself, certain microcontrollers come equipped with a separate processor specifically for bitshift-operations. Offloading the shifting to these processors rather than running them on the main CPU may allow faster processing of the data than performing the shifting in the code itself. However, this operation isn't natively supported by Arduino, and will potentially lead to extreme rewrites of the code which may prove too time consuming to do for two microcontrollers (as the code will not be reusable between controllers). As such, this optimization may only be done for one controller or left for future work.

Each scenario will first be executed on a single thread, and multiple threads may theoretically be executed concurrently to further optimize the data. The feasibility of this highly depends on the performance of every individual implementation. Depending on the results during our experimentation, this may yield yet another iteration.

5.2 Evaluation

In order to evaluate the efficacy of each iteration, we will test the code by processing our pre-determined bitstreams, saving the processed bits in a binary file and evaluating this file with the command line utility `ent`. This utility runs a series of tests on the data to determine its level of entropy the file has – rather, how information dense the file is. A perfect result (*e.g. entirely random*) has an entropy value of 8.0, whereas acceptable entropy values for our tests are greater than 7.99, denoting a high level of randomness. Furthermore, `ent` can verify the arithmetic mean of the file, summing all the bits in the file and dividing it by the length of the file. A good distribution of randomness should have an arithmetic mean of 127.5. To ensure each iteration works as expected, we will compare the `ent` result from each iteration against the original baseline, ensuring that the resulting processed bitstring is identical between iterations. There are a few more measurements shown by `ent`, and more details regarding these can be found in the utilities manual page[9].

As discussed in section 3, the hardware used will impose a clear bound on how quickly our implementation needs to process the bits in order to match the speed of the ADC, as well as the output speed of the USB-port, both in MB/s. In order to verify accuracy, the output of random numbers should be identical for each sample bitstring tested. Certain iterations might, however, use larger parts of the bitstring, and thus produce a slightly variable result.

Hyncica et. al. [6] propose that measuring execution time of algorithms directly via the microcontrollers internal timers (while subtracting the interrupt overhead) provides adequate measurements of an algorithms execution speed. An additional advantage is that the same code can be used to measure execution speed on several different microcontrollers, rather than relying on counting CPU cycles (as the process for this may vary greatly between controllers). As we will use fixed-size bitstrings for evaluation, we can then derive the throughput of the algorithm in MB/s as follows:

$$Throughput * MB/s = \frac{DataSize * bits}{ExecutionTime_{ms}} \times \frac{1}{8} \times \frac{1000}{10^6} \quad (2)$$

[9] Manual page for `ent`, accessed 2025-04-23.

This measurement allows us to place the throughput of our algorithm soundly in the bounds imposed on us by the hardware. During implementation, measurements of time complexity may be discussed with regards to the code – but in the end, the performance of the code as it runs is what will be evaluated most thoroughly.

5.2 Limitations

Our proposed iterations all assume that the limited hardware will support it. Whereas we are confident that Teensy 4.1 will be able to handle each iteration step (even the naive implementation), the remaining microcontrollers with lower specifications might not be suitable for the first iterations. Testing the implementations on different microcontrollers could turn out to be unfeasible – however, this remains to be seen during the experimentation.

6 EXPERIMENTATION

Each iteration results in an implementation written in C++, all of which will be supplied as an appendix to this paper. Furthermore, to simplify the implementation process, we will use the Arduino framework to communicate with the hardware. Whereas this introduces some overhead, it drastically reduces the time spent per iteration, and is deemed to be a suitable starting point.

Initially, we created a script to simulate a stream of bits from a file, and pre-computed the expected results using a trusted version of Toeplitz extraction. With the expected results saved to a file on the host computer, each iteration could then be deployed to the MCU, and tested with the same data as the baseline. As the algorithm is fully deterministic with this uniform data, the exact same results in the same order should be produced by all iterations. The simulation script then simply asserts that the MCU produces the same random string of bits from the resulting iterations as with the baseline.

As our system does not rely on interrupts in any capacity, measuring the execution speed of the implemented Toeplitz extraction function can be done using the built in `micros()`-function provided by Arduino. This provides us with an accurate measurement of the execution time per function call. The average execution time per test set over each iteration will be presented in **`TABLE 1`**, along with the fastest and slowest executions per iteration respectively. Furthermore, the entropy value on the processed bitstream will be noted in the table to demonstrate that the important measurement, e.g. the randomness, still holds true between iterations.

For our tests, we used six files with varying degrees of patterns in the bitstring, ranging from quite disastrous to acceptable. We constructed a simple test script to facilitate easier testing of each iteration. This script allowed us to easily test each implementation by automating several tasks – generating a baseline binary file with data processed by a good, proven implementation of Toeplitz extraction, giving us a way to ensure each iteration still produces the correct result. Furthermore, this script allows us to define different sample sizes for the MCU to utilize, and can extract the amount of time the algorithm takes to execute (*see table 1*) per each sample size.

Below we'll detail the results for each iteration.

RESULTS

All iterations was tested on 6 different input datasets, with some results varying less than 0.1 microseconds. The results with such slight variations have been omitted from the table. All the tables shown below has been tested on the dataset 'biased_bits_bad' which can be found as an appendix.

Due to compatibility issues, Pi Pico 2 was tested on different machine, creating further variation in the results – about 1 microsecond.

Iteration 1

The first iteration brought few surprises, as it ran quite well on Teensy. To our surprise, it executed well enough on Pi Pico 2 as well, albeit with $8\times$ slower execution speed.

[]@ >p() * 0.1000 >p() * 0.1583 >p() * 0.1583 >p() * 0.1583 >p() * 0.1417 >p() * 0.1417 >p() * 0.1417@

Bit size	**Teensy Min (ţs)**	**Teensy Max (ţs)**	**Teensy Avg (ţs)**	**Pico Min (ţs)**	**Pico Max (ţs)**	**Pico Avg (ţs)**
64	13	14	13.1564	103	117	106.3914
512	788	791	788.3139	5296	5311	5302.4979
1024	3124	3130	3124.0580	21108	21119	21111.2163

Iteration 2

Although the updated version included enhanced data structures, theoretical and practical analysis indicated that these modifications do not provide advantages for this project. Due to the small size of the input data (64, 512, 1024 bits) - using plain, contiguous memory layouts such as `std::vector<int>` proves to be very efficient in terms of access speed and memory usage. Current C++ compilers can effectively optimize these simple structures, often generating faster and more compact machine code than manually optimised alternatives. Introducing additional data structures or indirection would only add overhead without improving performance.

From an algorithmic perspective, Toeplitz extraction inherently requires a double summation loop, which leads to a time complexity of $O(n^2)$ no matter the optimization applied internally. Thus, making attempts to change it would not result in lower asymptotic cost.

In summary, given the size and type of this system, a straightforward and minimal approach is not just adequate, but also recommended.

Iteration 3

Coming soon!

CHANGELOG

2025-02-14: Added background section, smaller reviews to introduction.

2025-02-28: Template adjusted, added methodology. Started review of background and theory to add stronger correlation to computer science. Not yet finished due to review of articles as well as some additional information required from the project owner. The update to theory and background should be considered a heavy work in progress at this stage.

2025-03-10: Moved evaluation down in the methodology in order to provide a better flow. Elaborated further on Toeplitz extraction and ADC converters, as well as motivating the selection of these. Some additional information added in introduction as motivation for the work.

2025-03-13: Elaborated on background, as well as adding more details regarding hardware. Note that the hardware selected is subject to change over time. Further elaborated on related works in optimizing Toeplitz extraction.

2025-04-22: Begun including details regarding initial experimentation, updating details regarding experiments that had to change (e.g. no baseline on separate hardware).

2025-04-27: Added first table with test data.

REFERENCES

[1] Shubham Chouhan, K. S. V. Anurag, G. Raghavan, and P Kanaka Raju. 2024. FPGA-based Toeplitz Strong Extractor for Quantum Random Number Generators. In *2024 IEEE 5th India Council International Subsections Conference (INDISCON)*. 1–5. DOI: http://dx.doi.org/10.1109/INDISCON62179.2024.10744392

[2] Martin Clason. 2023. *Development of a QRNG front-end for shot noise measurement: analysis of quantum shot noise originating from photodiodes*. Independent thesis Advanced level (degree of Master (Two Years)). Linköping University, Department of Electrical Engineering, Information Coding, Linköping, Sweden. Available from: 2023-12-22.

[3] R. Gennaro. 2006. Randomness in cryptography. *IEEE Security Privacy* 4, 2 (2006), 64–67. DOI: http://dx.doi.org/10.1109/MSP.2006.49

[4] Liang Han, Jie Chen, Chaoxian Zhou, Ying Li, Xin Zhang, Zhibi Liu, Xiaoyun Wei, and Baofeng Li. 2004. An embedded reconfigurable SIMD DSP with capability of dimension-controllable vector processing. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* 446–451. DOI: http://dx.doi.org/10.1109/ICCD.2004.1347960

[5] Miguel Herrero-Collantes and Juan Carlos Garcia-Escartin. 2017. Quantum random number generators. *Rev. Mod. Phys.* 89, Article 015004 (Feb 2017), 48 pages. Issue 1. DOI: http://dx.doi.org/10.1103/RevModPhys.89.015004

[6] Ondrej Hyncica, Pavel Kucera, Petr Honzik, and Petr Fiedler. 2011. Performance evaluation of symmetric cryptography in embedded systems. In *Proceedings of the 6th IEEE International Conference on Intelligent*

Data Acquisition and Advanced Computing Systems, Vol. 1. 277–282. DOI: http://dx.doi.org/10.1109/IDAACS.2011.6072756

[7] Siyu Liao, Ashkan Samiee, Chunhua Deng, Yu Bai, and Bo Yuan. 2019. Compressing Deep Neural Networks Using Toeplitz Matrix: Algorithm Design and Fpga Implementation. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1443–1447. DOI: http://dx.doi.org/10.1109/ICASSP.2019.8683556

[8] Fading Lin, Wenbo Ge, Zhijie Song, Xinxuan Cui, Yanqiang Guo, Xiaomin Guo, and Liantuan Xiao. 2024. Seed Renewable Parallel and Real-Time Toeplitz Post-Processing for QRNG. *Journal of Lightwave Technology* 42, 24 (2024), 8606–8615. DOI: http://dx.doi.org/10.1109/JLT.2024.3445327

[9] Hanni B. Lozano and Mabo Ito. 2015. Energy efficient dual-issue processor for embedded applications. In *2015 IEEE 12th International Conference on Networking, Sensing and Control*. 544–549. DOI: http://dx.doi.org/10.1109/ICNSC.2015.7116095

[10] Jakub Niemczuk. 2020. Shot noise-based quantum random number generator. In *Quantum Technologies 2020*, Eleni Diamanti, Sara Ducci, Nicolas Treps, and Shannon Whitlock (Eds.), Vol. 11347. International Society for Optics and Photonics, SPIE, France, Article 1134717, 6 pages. DOI: http://dx.doi.org/10.1117/12.2554898

[11] Yong Shen, Liang Tian, and Hongxin Zou. 2010. Practical quantum random number generator based on measuring the shot noise of vacuum states. *Phys. Rev. A* 81, Article 063814 (Jun 2010), 5 pages. Issue 6. DOI: http://dx.doi.org/10.1103/PhysRevA.81.063814

[12] Jaideep Singh, Rodrigo Piera, Yury Kurochkin, and James A. Grieve. 2024. A Compact Quantum Random Number Generator Based on Balanced Detection of Shot Noise. (2024). https://arxiv.org/abs/2409.20515

[13] André Stefanov, Nicolas Gisin, Olivier Guinnard, Laurent Guinnard, and Hugo Zbinden. 2000. Optical quantum random number generator. *Journal of Modern Optics* 47, 4 (2000), 595–598. DOI: http://dx.doi.org/10.1080/09500340008233380

[14] Greg Taylor and George Cox. 2011. Digital randomness. *IEEE Spectrum* 48, 9 (2011), 32–58. DOI: http://dx.doi.org/10.1109/MSPEC.2011.5995897

[15] Michael A. Wayne, Evan R. Jeffrey, Gleb M. Akselrod, and Paul G. Kwiat. 2009. Photon arrival time quantum random number generation. *Journal of Modern Optics* 56, 4 (2009), 516–522. DOI: http://dx.doi.org/10.1080/09500340802553244

[16] Xiaoguang Zhang, You-Qi Nie, Hao Liang, and Jun Zhang. 2016. FPGA implementation of Toeplitz hashing extractor for real time post-processing of raw random numbers. In *2016 IEEE-NPSS Real Time Conference (RT)*. IEEE, USA, 1–5. DOI: http://dx.doi.org/10.1109/RTC.2016.7543094