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]. Pseudo-random numbers do not generate true randomness, and in order to heighten security, other methods of generating random numbers are required. Current random number generators (*RNG*) are usually implemented in computer programs, 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 hypothetical attacker 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 [13], 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 fluctuation signals from any given source, for instance an optical signal, the inherent natural unpredictability of said signal can be harnessed in order to produce a random number from a state that is nigh impossible to reproduce accurately. Clason [2] presents a device that generates a fluctuating analogue signal utilizing this method.

The optical signal output by the device described in [2] needs to be converted to a stream of random, raw bits via an Analog to Digital Converter (*ADC*). Some post-processing of the said raw bits has to be performed in order to ensure that the bits are workable, and to remove potential deterministic patterns from the data. One method for this post-processing we will explore in this work is Toeplitz extraction [15], typically performed on the host computer utilizing the randomly generated numbers. The post-processing finally yields a random number. This relationship can be seen in Figure 1.

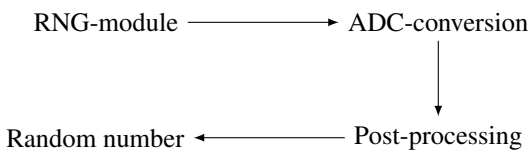

Figure 1: An abstract of the steps required for OQRNG.

Clason proposes a simpler and cheaper way to achieve QRNG [2] than what has been done in previous research. These systems have generally been large, bulky and expensive, whereas the method proposed is portable enough to theoretically be installed in a portable device – for instance a USB-thumbstick. In keeping with this, utilizing microcontrollers rather than a host computer for processing the raw bits extracted allows this system to be self-contained and portable, and further helps to keep the costs low and the solution reasonably complex.

However, due to the limited processing power of the average microcontroller, any implementation of Toeplitz extraction needs to work quickly and efficiently despite this hardware constraint. Experimenting with efficient implementations of this well-defined algorithm is the main focus of this thesis, and the key research areas explored in this work are as follows:

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 has been optimized quite well, and previous research can be utilized to address this research question. 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

[1] https://blog.cloudflare.com/randomness-101-lavarand-in-production/

to find the most optimal implementation with the least amount of effective processing time spent on the algorithm.

Research area 2 (*RA2*): Can we ensure that the output of random numbers is not primarily limited by our implementation, but rather limited by the processing power or the USB transfer speed of the microcontroller, alternatively by 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 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 and implementation strategy, as well as some limitations imposed on our work. Finally, section 7 will present the results of our experimentation.

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 the intrinsically random properties of a quantum process. Stefanov et. al. [12] 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. [14] 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. [12], but instead using shot noise. As described by Niemczuk [9], 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. [10] 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. [11]. 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. [10] and Singh et. al. [11] 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 [10] or the system is built on bespoke hardware with limitations in processing power which prevents a fully integrated system [11]. 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 studied, and in so doing a prototype device was constructed. This prototype utilized an LED soldered millimeters apart from a photodiode, which in turn outputs a variable analog signal from the device. In this section, we introduce the remaining 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 primary concern for our study is the inherently random, analog voltage current produced by the system.

3.2 ADC converter

This analog current is not suitable to operate on without further processing. As mentioned in Section 1, the signal needs to pass through an ADC to be converted into raw bits. 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 modern microcontrollers (*including both MCUs used in this study*) furthermore come equipped with internal ADCs that can be utilized. 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 two million samples per second (*MSPS*). This ADC provides a 12 bit sample size, providing roughly 24 Mbit/s of sampled data per second, derived by the following calculation.

$$\text{ADC Throughput} = \frac{2{,}000{,}000_{\text{MSPS}} \times 12}{1{,}000{,}000} \approx 24\,\text{Mbit/s} \qquad (1)$$

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 for embedded systems and real-time operations, and suitable as a processing unit for the purposes of this work. Unlike general CPUs, an MCU integrates a processor, memory and peripherals (*such as an ADC*) into a single chip. Furthermore, modern MCUs often feature advanced microarchitectural elements to enhance processing capabilities on single threads (*such as dual-issue superscalar architectures, allowing the MCU to run several instructions per CPU cycle*), making them suitable candidates for the post-processing required for OQRNG-data.

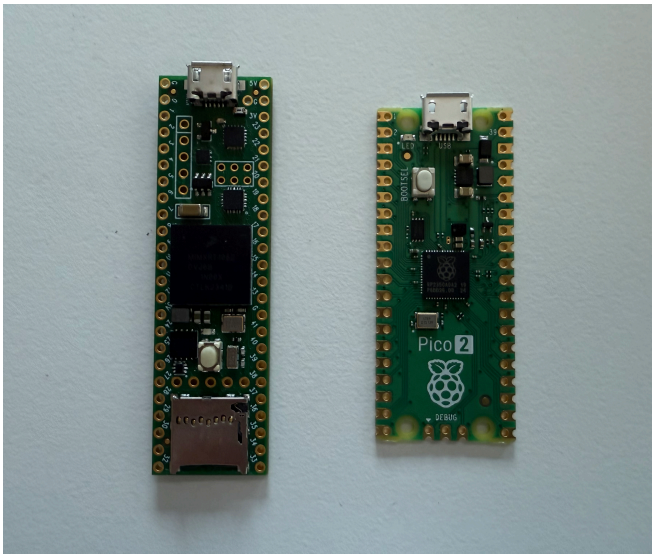

Figure 2: Teensy 4.1 on the left, compared to Raspberry Pi Pico 2 on the right.

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 pre-fetching 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 delays. Efficient memory management guarantees that randomness extraction can occur rapidly without major slowdowns in computing. Both approaches will be tested during development.

In our work, we intend to use Teensy 4.1[5] based on the ARM Cortex-M7 pictured to the right in Figure [**?**]. This MCU is especially suitable for computationally demanding tasks involving randomness extraction due to its dual-issue superscalar architecture and Digital Signal Processing (*DSP*) capabilities. The floating-point unit[6] (*FPU*) and Single Instruction, Multi-

[2] https://www.farnell.com/datasheets/1913106.pdf

[3] https://www.farnell.com/datasheets/1913106.pdf

[4] https://www.sciencedirect.com/topics/computer-science/direct-memory-access

[5] https://www.pjrc.com/store/teensy41.html

[6] https://www.sciencedirect.com/topics/computer-science/floating-point-unit

ple Data (*SIMD*) style DSP instructions improve how quickly it can perform bitwise and arithmetic tasks, which is crucial for quick Toeplitz extraction. SIMD-controlled DSP architectures, as described by Han et al. [4], leverage parallel vectorized computation to accelerate matrix operations – making them highly effective for Toeplitz matrix-vector multiplications.

In order to evaluate how efficent our implementation can become, our aim is to try our implementation on other MCU with different level of power and hardware support. Whereas Teensy 4.1, operating at `600 MHz`, is our primary development platform which we will evaluate closely, we aim to run our implementations on the Raspberry Pi Pico 2[7], operating at `150 MHz` (*pictured to the right in Figure [?]*). Due to the lower computational power of this MCU, there may be significant issues in utilizing this weaker model, yet it is significantly cheaper and easier to access. Testing will consist solely of running the implementation on these controllers and measuring execution speed and correctness of the output.

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 will 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.

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.

[7] https://datasheets.raspberrypi.com/pico/pico-2-product-brief.pdf

Algorithm 1 Toeplitz extraction

Require: $x[0..n-1]$ input bit array of length n
Require: $t[0..n+m-2]$ seed matrix of length $n+m-1$
Ensure: $y[0..m-1]$ output bit array of length m

```
for i = 0 to m − 1 do
    sum = 0
    for j = 0 to n − 1 do
        sum = sum + x[j]t[i + j]
    end for
    y[i] = sum mod 2
end for
return y
```

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 over 24 Mbit/s (*imposed by the ADC*) 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. [15]. 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 [15] 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, such as those developed by Chouhan et al. [1] and Zhang et al. [15], 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 Peripheral Component Interconnect Express (*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*) – reducing computational complexity from $O(n^2)$ to $O(n \log n)$. 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 sample several bitstreams from the OQRNG-device to use for our tests – each with varying degrees of repeated patterns that should be eliminated by the algorithm. All bitstrings tested are available as an appendix to this paper.

Our implementation of the Toeplitz extractor followed a structured, iterative approach divided into two phases. Phase one focused on exploring performance improvements through incremental algorithmic changes. Phase two then addressed architectural inefficiencies identified during phase one. Each phase then consisted of a number of individual iterations, in which performance was evaluated in terms of both correctness and execution time on the target hardware.

5.1 Phase one

Initially, we require a "naive" version designed to prioritize correctness over speed. This version was first executed on a separate computer to generate reference output for various input bitstrings, which were later used as accuracy baselines. The naive implementation was then flashed onto the Teensy 4.1 microcontroller, where execution time was measured in microseconds. Each subsequent iteration introduced controlled modifications aimed at improving throughput.

The initial implementation followed the pseudocode described in Algorithm 1, using matrix multiplication over the raw input and seed data. It relied on `std::vector<int>` for storing the raw data, and utilized nested loops to compute each output bit. From this point, this implementation was adapted over the coming iterations, and each new version was then verified in the same manner as the initial version as outlined in Section 5.3.

Iteration 1 - Data structures: This iteration kept the same algorithmic logic as the initial implementation, but experimented with data structures such as raw pointers and hash maps to enhance the use of `std::vector`. The goal was to reduce or eliminate the reliance on nested loops.

Iteration 2 - Bitshifting: Basic bitwise operation was introduced to replace arithmetic operations whenever possible. Multiplication was replaced with logical `AND &`, and modulo operations with bit masking `& 1`. The goal was to reduce the number of instructions and improve per-bit processing speed.

Iteration 3 - Batching and Hardware optimization: This iteration focused on optimizing performance through batching and the use of ARM-native instructions, beginning by testing batching alone. This was followed by isolated use of ARM instructions such as `__builtin_popcountll()` (*which counts the number of set bits in an unsigned integer*). After establishing their individual effect, we combined both techniques, Multiple batch sizes were tested to determine their impact. More details and benchmarks for each configuration can be found in the section 6.

5.2 Phase two

Phase two focused on addressing inefficiencies and design issues that were unintentionally introduced or not addressed during earlier iterations. Rather than continuing with new algorithmic ideas, this phase aimed to identify and fix structural problems. Several assumptions from phase one were re-evaluated – such as the benefits of certain data structures or abstractions.

Iteration 4 - Loop unrolling: This iteration focused on reducing the number of loops in the extractor by manually unrolling repeated operations. The goal was to decrease overhead created by loops. Whereas this operation is commonly done by compiler optimization, manually unrolling the loops guarantees the optimization rather than leaving it to the compiler.

Iteration 5 - Removal of vector usage: This iteration removed `std::vector` in favor of fixed-size types like `uint32_t` and `uint64_t` to reduce the overhead introduced by creating and populating this complex data structure.

Iteration 6 - Data type exploration: Following the removal of vectors, this iteration explored alternative static data types such as: `array`, `unordered_map` and `bitset` to determine the most efficient structure for storing input and seed data.

5.3 Evaluation

To evaluate the correctness of each implementation, a baseline was generated as discussed in Section 5. Using the naive, initial implementation to process bits and saving for later evaluation gave us a source of truth against which to compare following iterations. To verify that the algorithm successfully removed the patterns it should, we verified the measured en-

tropy score with the command line utility `ent`[8]. This tool verifies the statistical randomness of any given file, and does several calculations to provide an estimate of how random any given binary file is.

The way `ent` calculates the entropy of a given binary file is by checking the frequency of certain byte values. If some byte values are more frequent, the binary file has a low entropy – essentially singifying that there are predictable patterns. Conversely, if all byte values appear roughly equally, the binary file has a high entropy – meaning that it is more random, and less predictable. The maximum entropy of any given file is $\log_2(256) = 8$ bits per byte, meaning that a perfectly random binary file has an entropy score of 8. For the purposes of our work, we consider values greater than 7.99 to be acceptable.

Furthermore, `ent` also provides a measurement of the possible compression ratio of the file, e.g. how much more the file can be theoretically compressed. This is calculated as $1 - \frac{\text{entropy}}{\text{maximum entropy}}$, signifying that the file cannot be compressed further – that the data is already as compressed as its random distribution allows. Essentially, this informs us that the file has no more detectable statistical patterns or compressable regularities remaining. We consider any file where `ent` reports a compression ratio of 0% to be acceptable.

In regards to execution time, 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 the execution speed of an algorithm. 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).

In our implementation, execution time for each extraction was measured using Arduino's built-in `micros()`[9] function. This function offers a convenient and platform-independent method for recording elapsed time with microsecond precision, enabling consistent performance evaluation across different iterations.

To isolate the execution speed of the algorithm as precisely as possible, we only intend to measure the execution speed of the algorithm. Whereas there is some other code present in each iteration (*primarily packing the output data to be sent to the host computer in a workable format*), the brunt of the work should reasonably be performed by the algorithm. As such, we expect the time spent on executing other code to be negligible. Furthermore, this code performs significantly less work per cycle, and does not exponentially increase the larger the input and output requirements increase.

As we will use fixed-size bitstrings for evaluation, we can then derive the throughput of the algorithm in $Mbit/s$ as follows:

$$Throughput_{\text{Mbit/s}} = \frac{DataSize_{\text{bits}}}{ExecutionTime_{\text{ms}}} \times 10^{-3} \quad (2)$$

This measurement allows us to place the throughput of our algorithm soundly in the bounds imposed on us by the hardware. Plugging in the $24Mbit/s$ bound imposed by the ADC with an arbitrarily chosen 64-bit sample size, we can derive the average execution speed in microseconds:

$$\frac{64}{24} \times 10^{-3} \text{ ms} \approx 2.667 \times 10^{-3} \text{ ms} = 2.667\ \mu\text{s}. \quad (3)$$

In section 6, this calculation will be used to derive the execution speed of the various iterations.

5.4 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.

Whereas our implementation can be controlled at the algorithmic and structural level, compiler optimization are managed outside of our influence. The Arduino framework automatically sets compiler flags and optimization levels specifically chosen for the hardware – for instance, specific flags are selected when compiling code for Teensy, applying aggressive settings for performance. However, we cannot determine exactly how these flags influence our iterations. Consequently, this makes it impossible to isolate the impact of compiler optimizations from the performance gains of our implementations. Therefore, any performance differences observed in our experiments are considered to stem solely from our implementation, as the compiler behavior remains opaque and unchangeable in this context.

6 RESULTS

We created a script to facilitate easier testing, which is attached to this paper. Using this script, all bitstrings used for testing can be evaluated against the baseline, ensuring that the output from the new iteration matches the baseline exactly. Furthermore, the script also provides the average execution time of only the Toeplitz extraction in microseconds. A brief overview of the architecture of this test script can be seen in Figure 3.

6.1 Phase one

Bit size	Teensy (μs)	Pico (μs)
64	13.1564	106.3914
512	788.3139	5302.4979
1024	3124.0580	21111.2163

Table 1: Iteration 0 - Naive implementation

[8] https://manpages.ubuntu.com/manpages/trusty/man1/ent.1.html

[9] https://docs.arduino.cc/language-reference/en/functions/time/micros/

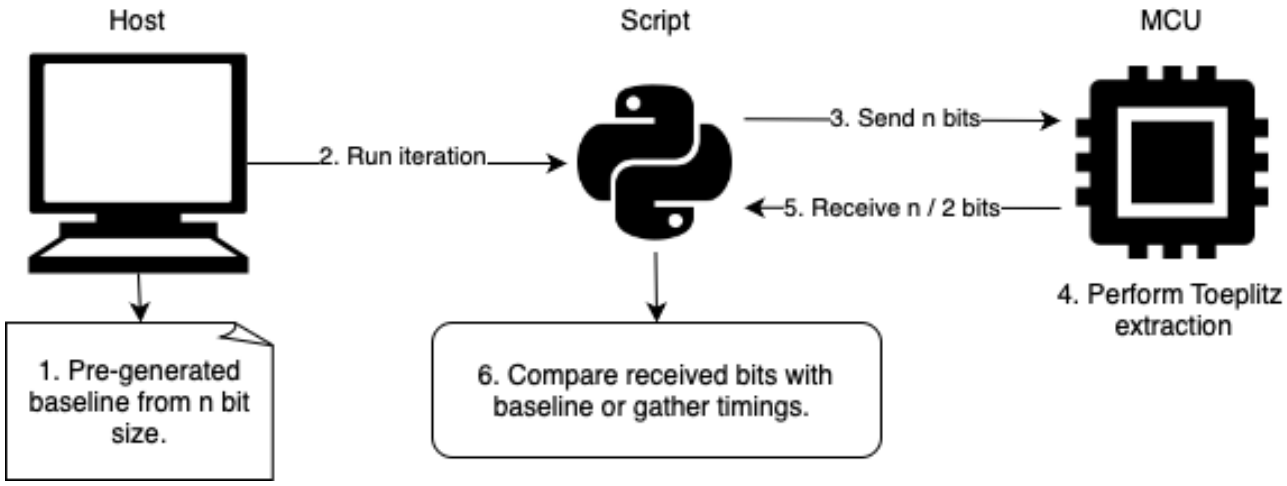

Figure 3: A brief overview of the testscript architecture.

Table 1 presents the average execution time of iteration 1 on both Teensy 4.1 and Raspberry Pico Pi 2 across the input sizes. The Teensy consistently outperforms the Pico, with the gap widening as the bit size increases. These results will be used as baseline for future comparison.

Bit size	**Teensy (μs)**
pointers	13.7999
hash map	68.8099

Table 2: Iteration 1 - Data Structures

Table 2 presents the average execution time of iteration 1 on Teensy 4.1. This iteration introduced additional complexity without yielding significant gains and in some cases led to worse performance. Although this iteration did not achieve the desired performance gains, it provided valuable insight into the limitations of certain data structure choices within the context of this study.

Notably, as the execution speed of the more powerful Teensy 4.1 increased with this iteration, we omit the limited results of executing the same code on Pico 2. During the remainder of this phase as well as the next, test results of any given iteration of Pico 2 are only reported if there are significant enough improvements in execution speeds for Teensy to potentially lead to gains for the weaker MCU.

Bit size	**Teensy (μs)**
64	16.4689
512	1006.6255
1024	3996.5972

Table 3: Iteration 2 - Bitshifting

Table 3 shows the execution times for iteration 2 on the Teensy 4.1, which utilizes bitshifting instead of the original data structure approach. A clear increase in executiom time compared to iteration 1 is observed.

Additionally, the isolated effect of applying a single bitmask operation `& 1` was evaluated. This resulted in a slight reduction in average execution time, from `13.1564` μs to `13.1000` μs, although this specific result is not included in the tables above.

Bit size	**Teensy (μs)**
64/64	43.0760
512/512	2663.8194
1024/1024	10513.1767

Table 4: Iteration 3 - Batching and Hardware optimization

Table 4 presents the average execution time of iteration 3 on the Teensy 4.1 for varying batching and bit sizes. Additionally, when performing the same 1024/1024-bit operation, the Raspberry Pi Pico 2 showed average execution time of `80806.006` μs indicating a substantially lower throughput compared to the Teensy.

6.2 Phase two

Bit size	**Teensy (μs)**	**Pico (μs)**
64	9.7017	70.7402
512	551.6358	3978.1928
1024	2195.1979	15830.8784

Table 5: Iteration 4 - Loop unrolling

Table 5 presents the execution time of iteration 4, which utilizes a 4x unrolled bit-processing loop. Compared to erlier iterations, this approach yields a substantial performance improvments on both the Teensy and the Raspberry Pi Pico Mcu's.

In addition to the results presented in Table 5, a further tests was conducted on the Teensy using a single-loop unrolled implementation for the 64-bit size, resulting in an average execution time of `6.6626` μs. An additional fully unrolled variant,

where loops were entirely eliminated, produced a measured execution time of `0.0491` μs. However, this measurement was later determined to be invalid due to packaging error.

Bit size	Teensy (μs)	Pico (μs)
64	0.0501	0.2175

Table 6: Iteration 5 - Removal of vector usage

Table 6 presents the results of iteration 5, in which vector structure were removed in favor of fixed-size integer types `uint32_t` and `uint64_t`. This iteration led to an extremely efficient implementation that approaches the physical execution limits of the Teensy 4.1. Due to the removal of `std::vector`, this implementation is limited by the available integer sizes of the MCU, and could as such only be tested with at most 64 bits. The ramifications of this will be discussed in more detail in Section 7.

Data structure	Teensy (μs)
array	0.4284
unordered_map	31.5090
bitset	0.0474

Table 7: Iteration 6 - Data type exploration

Table 7 presents the results of iteration 6 using 64 bits input size, which was introduced to address the structural limitations encountered in iteration 5 (*specifically, the fixed-width output constrains imposed by the removal of* `std::vector`). In this iteration, another set of alternative data structures were evaluated, the vector-based approach from iteration 1 serving as a baseline for comparison.

As the bitset appeared to provide gains in execution speed, we attempted to increase the input bit size to 128 bits in order to investigate whether this data structure could allow for larger outputs. However, this substantially increased the execution time to `179.8712` μs.

7 DISCUSSION

Observing the results, we can place them into the context of the limits imposed by the ADC, as discussed in Section 3.2. In algorithm (3) (*as seen in Section 3.3*), we calculate the average execution speed required to be $2.667\mu s$ for the ADC that requires soldering to the microcontroller. From Tables 6 and 7 we demonstrate execution speeds well below that, with iteration 5 as well as iteration 6 – specifically with bitsets – being the most prominent results to discuss. Here, our implementation achieved speeds of $0.0501\mu s$ and $0.0474\mu s$ respectively which within the bounds of the ADC listed in Section 3.2. The built in ADC on Teensy has an effective sample rate of 1 MSPS (*e.g. half of MAX11102AUB*), which would require an average execution speed of ~$2.667/2\mu s \approx 1.334\mu s$, making our implementation suitable even for that limited conversion speed.

The main culprit that led other iterations to not be viable was the inclusion of `std::vector`, which was used to ensure we could deliver variable lengths of our return data – e.g. either 32, 64 or 128 bits. The overhead introduced from initializing and populating the vector led to significant delays in execution speed, leading to all other attempts at optimization to be superfluous. In fact, any other complexities led to significantly worse results than the naive, original implementation.

However, it is important to note that both of these implementations has a hard limit of working on 64 bits input. Whereas larger bitstrings still yield results with high entropy, the execution speed increases far above the hardware limitations, as evident in 7. Iteration 5 is hard capped at using only 64 bits input, as we need to store the bits in fixed-size integers before running them through the algorithm, and on embedded machines we have no larger integer types available.

One issue still remains concerning the size of the input string and subsequent output string. The larger the size of the matrix provided to the Toeplitz function (*e.g. the larger our input*), the higher the potential overall entropy becomes. As our implementation is essentially capped at 64 bits input at this point in time (*without being limited by the hardware*), this may or may not prove to not be secure enough for use in real world applications. This, however, is out of scope for this particular thesis, but a consideration nonetheless.

Any requirement of output integers larger than 32 bits may prove problematic. Whereas the bits processed by the algorithm does show an entropy measure quite close to 8, e.g. reliably random, this does not necessarily mean that two sets of 32 bits can be combined on the host machine to produce a 64-bit integer. Combining these two sets may very well result in a less random result – in essence, an entropy score that is not entirely random. As quantum computing evolves, the process of bruteforcing algorithms with all possible numbers becomes less of an impossible task, which is why this process of generating random numbers is required. The purpose of using quantum flucuation for random number generation is to provide greater assurances of security not possible from traditional, pseudo-random RNG, and being so severely limited by the size of the output is a major flaw in this implementation worth considering.

8 CONCLUSION

This work set out to investigate the feasibility of implementing Toeplitz extraction on resource constrained hardware. Furthermore, we aimed to conclude whether the extraction algorithm could be executed fast enough to not be the limiting factor, e.g. that the speed of the extraction did not exceed the speed of the ADC conversion. Ostensibly, these investigations were both successful, considering that we managed to create an implementation that executes in roughly one half microsecond on average, far exceeding the limits imposed by the ADC. The optimization with the algorithm revolved around removing as much complexity as possible, avoiding anything that is not the simplest form of data structure for maximum performance gains.

As mentioned in Section 7, this comes at a price. The implementation that did indeed perform above expectation could only output a 32-bit integer, as we cannot take any larger input than 64 bits without any more complex data structure. How much of a flaw this is for a system utilizing this method of OQRNG is beyond the scope of this thesis to investigate. However, it is likely that a larger output is required for any application revolving around tight security. For less critical systems and less essential functionality, this method might, however, prove just secure enough. For instance, an IoT-device that requires a random number output for symmetric cryptography when connecting to a user device may be an ideal candidate for this method of OQRNG.

9 FUTURE WORK

The main limitation for producing output with a larger size is the lack of a suitable data type in the limited standard library available for MCUs. Furthermore, it appears as if generic containers (*such as* `std::vector`) that have already been severely optimized still impose a far greater performance penalty for this purpose. Future work may focus on implementing custom data types for larger bit sizes, for instance 128-bit and 256-bit integers. It may be possible that external libraries exist that alredy provide implementations of these integer sizes, but even then a slimmed down and customized implementation may prove the better alternative for this extreme performance constraint.

Finally, this proof-of-concept has not been tested when connected to the OQRNG-device itself, and has focused on a theoretical exploration of the capabilities of the MCUs as well as the limitations of Toeplitz extraction. Sampling raw, live data from the OQRNG-device and processing it utilizing the firmware developed in this work may reveal flaws not evident from our tests, as well as demonstrating the efficacy of our implementation in a real world scenario.

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] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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