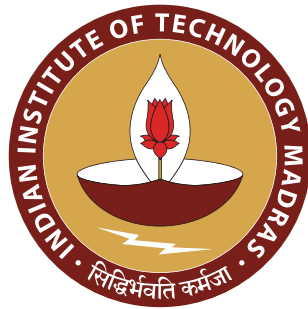


Internship Report

Name : Akash Chandra Behera

Registration no: EE22SFP0313



Department of Electrical Engineering

15th July 2022

Internship Report

Department of Electrical Engineering, IIT Madras

- 1. Name of the Student** : Akash Chandra Behera
- 2. Registration Number** : EE22SFP0313
- 3. Registered for** : Internship
- 4. Name of the Supervisor** : Professor Anil Prabhakar
- 5. Date of Joining** : 23/05/2022
- 6. Date of Conclusion** : 15/07/2022

- 7. Area of Research** : Device-Independent Randomness Amplification
QRNG and QKD protocols

Contents

Introduction	3
Objectives	3
Plesch-Pivoluska Protocol	4
Outline	4
Determining optimal CF	5
Task 1	5
Task 2	5
Tests for Randomness	6
Measure for Quality of Randomness	7
Simulation	8
Preparing GHZ states	8
Preparing Circuits	8
Processing and I/O	10
Results	12
Determining Optimal CF	12
Task 1	13
Task 2	14
Conclusion	15
Task 1	15
Task 2	15
Miscellaneous	16
QKD Protocols	16
Single Photon Detectors	16
Arduino	16
Power Loss in Optical Fibre	17

Introduction

Since Peter Shor proposed the algorithm that bears his name, it is only a matter of time before someone comes up with a powerful enough Quantum computer and renders the widely used RSA cryptography protocol useless. With new claims of Quantum supremacy every year, it is imperative to develop and implement Quantum Encryption protocols. Sources of certifiable random numbers are an essential component of any such protocol. The only information-theoretically certifiable source of Random numbers are Quantum Random Number Generators or QRNGs. QRNGs are devices that use the inherent randomness of quantum mechanics properties to produce random numbers.

Ideally, the output of any QRNG should be truly random (from the term “truly random”, we mean the output can be information-theoretically certified as random). But real-life implementation of QRNGs (like everything else in real-life) is not ideal. Given the need for mass production of QRNGs, it is essential to look for post-processing algorithms for QRNGs. More specifically, what we need is an algorithm that has the following properties :

- Can reduce the bias introduced by the imperfections of the QRNG.
- Cheap to implement.
- Preferable is Device-Independent.

Regarding performance, the last point is optional but is essential for the commercial success of QRNGs, as Device-Independence guarantees the user’s trust irrespective of the manufacturer’s reputation.

This project simulates one such post-processing protocol and discusses the performance of the chosen protocol concerning the criterion mentioned earlier.

Objectives

The objectives of my Internship project were to :

1. Test the efficacy and viability of a DI-Randomness Protocol.
2. To check how much fault the protocol can correct.
3. Try to distinguish between the output of a True QRNG and a PRNG using the Randomness amplification protocol.

The protocol of choice here is the DI-Randomness amplification protocol by Plesch and Pivoluska[1].

Plesch-Pivoluska Protocol

Plesch-Pivoluska Protocol is Randomness Amplification Protocol that is based on measuring Tripartite GHZ states[1]. One can easily simulate this protocol using IBM Qiskit. In the following lines, we briefly describe the step-by-step procedure to implement this protocol.

1. Consider a $2n$ -bit string denoted by $s_1 s_2 \dots s_{2n-1} s_{2n}$.
2. Take 2-bits say $s_i s_{i+1}$. Let xyz be labels for measurement settings of channel A, B and C respectively. Now, $s_i s_{i+1} \in \{00, 01, 10, 11\}$, map $s_i s_{i+1}$ to xyz , where $xyz \in \{100, 010, 001, 111\}$.
3. Now, if $x = 0$ measurement on channel A is done in Y -basis; else measurement is done in X -basis. Similarly for channel B and C.
4. Denote the output as $A = (a_1, a_2, \dots, a_i)$, $B = (b_1, b_2, \dots, b_i)$ and $C = (c_1, c_2, \dots, c_i)$.
5. Output $O = \text{Ext}(A, B)$, where Ext denotes some suitable extractor.

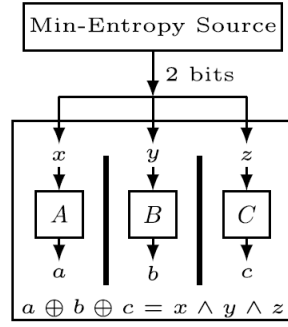


Figure 1: Schematic of the GHZ measurement settings

Reasons for choosing this protocol over others are as follows :

- Can be implemented using a single device, unlike competing protocols that require arbitrarily many devices[2] or arbitrarily many measurement settings[3].
- Unlike most competing protocols, this protocol is Device-Independent.

For this project, we will use Hadamard Extractor i.e.

$$\text{Ext}_{\text{Hadamard}}(A, B) = \bigoplus_{i=1}^n (A_i \wedge B_i)$$

Hadamard Extractor is simple to implement and has been proved to be secure. Unlike alternatives (e.g. Trevisan Extractor), Hadamard extractor can be implemented while maintaining Device-Independence[4].

Note that the Hadamard extractor compresses the output by a factor n , where n is the number of bits in A ; we shall refer to this factor as the Compression factor (or CF in short). Usually, the higher the CF, the more secure the output is, but unnecessarily high CF can lead to low bit-rate, as the output file size is $\frac{\text{Input Size}}{\text{CF}}$. Note, the CF is always a multiple of 2.

Outline

For Task 1, we need to generate output data from an δ -Santha-Vazirani (SV) source.

A source is called δ -SV source with respect to E if

$$\frac{1}{2} - \delta \leq \Pr(Z_i = 0 \mid E, Z_1, \dots, Z_{i-1}) \leq \frac{1}{2} + \delta, \text{ where } \delta \in [0, \frac{1}{2}]$$

For our project, it suffices to remember the following table of probabilities for every δ value. Here, $\Pr(i)$ refers to probability of getting i .

δ	$\Pr(0)$	$\Pr(1)$
0.1	60%	40%
0.2	70%	30%
0.3	80%	20%
0.4	90%	10%

Table 1: Output characteristics of SV sources

We describe the outline of the project in the following subsections.

Determining optimal CF

Goal is to determine the optimal CF value that balances output file-size with quality of randomness.

We proceed as follows :

1. Generate data from SV source with $\delta = 0.1$
2. Process this RAW dataset for CF= 4, CF= 8, CF= 10, CF= 12, CF= 14, CF= 16 and CF= 32.
3. Subject all processed dataset to NIST Test Suite and compare the result.

Task 1

This task aims to check how much bias the protocol can manage to correct when the Compression Factor is optimal.

1. Generate biased data using `numpy.random` module. This data simulates a defective QRNG. We shall generate data from SV sources with δ values mentioned in Table 1.
2. Subject the biased RAW data to NIST Tests and save the results.
3. Simulate the circuitry of the [Plesch-Pivoluska Protocol](#) using IBM Qiskit's `AerSimulator` and input the biased random number to it. The output is the processed data.
4. Subject processed data to NIST Tests. Compare the results of NIST Tests for RAW and Processed data.

Task 2

The aim of this task is to check if the [Plesch-Pivoluska Protocol](#) be used to distinguish between a TRNG and a PRNG, using degree of randomness amplification as a distinguishing factor.

1. Take RAW data generated by a TRNG and PRNG.
2. Use `statmodels` module to measure the Serial Correlation, Mean and Entropy of the RAW data.
3. Process the RAW data using the code developed for Task 1.
4. Measure the Serial Correlation, Mean and Entropy of the processed data; and compare results for RAW and Processed data.


Note, we shall use the data from the two-entropy source QRNG developed at IIT Madras as TRNG data, as the data has passed both NIST and DieHarder Tests[5]. For PRNG, we shall use the in-built PRNG of the NumPy module, which is based on Mersenne Twister[6].

Tests for Randomness

Definition of randomness is a topic of debate; however, there are some universally recognized tests, chief among them are NIST Test Suite and DieHarder Tests. For our project, though, we shall use the NIST tests as the minimum file size required for the DieHarder test is quite high. Let's have a quick overview of the NIST Test Suite. The NIST Test Suite was introduced by the National Institute of Standards and Technology (NIST), US[6]. The test suite consists of 15 tests described below.

1. Frequency (Monobit) Test : This checks if $\text{freq}(1s) \approx \text{freq}(0s)$.
2. Frequency Test within M-bit Blocks.
3. Runs Test: Checks the total number of runs (uninterrupted sequence of identical bits) in the sequence.
4. Longest Runs Test : Checks for the Longest Run of Ones in M-bit Blocks.
5. Binary Matrix Rank Test: Checks for linear dependence of M-bit sub-strings of the original sequence.
6. Discrete Fourier Transform (Spectral) Test: Checks for periodic features using DFT.
7. Non-overlapping Template Matching Test: Detects generators that produce too many occurrences of a given aperiodic pattern.
8. Overlapping Template Matching Test: The focus of the Overlapping Template Matching test is the number of occurrences of pre-specified target strings.
9. Maurer's "Universal Statistical" Test: Checks the lossless compressibility of sequence. Lossless compressibility can be used as a measure of non-randomness.

10. Linear Complexity Test: Checks if the sequence is complex enough to be considered random.
11. Serial Test: Checks for overlapping patterns within m-bit sub-blocks of an n-bit block.
12. CumSum (Cumulative Sum) Test: Assign -1 to 0 and +1 to 1, then check the sum of corresponding random walk w.r.t Standard Normal Distribution.
13. Random Excursions Test: Determines the number of cycles having exactly K visits in a cumulative sum random walk.
14. Random Excursions Variant Test: Checks the total number of times that a particular state is visited in a random walk.

Note, as we don't have access to a Linux PC, we shall use the python port of the Linux-based NIST Test Suite .

Measure for Quality of Randomness

To distinguish between processed and unprocessed data of a TRNG or PRNG, one needs a measure of the quality of randomness. Now, most randomness tests like NIST and DieHarder, use p -value to decide if the given data is random or not, where the definition of p -value is as follows:

$$P\text{-value} = \Pr(\text{Observed Data} | \text{Given sequence is random})$$

whereas what we need is a measure of the quality of randomness, say \tilde{P} -value, such that

$$\tilde{P}\text{-value} = \Pr(\text{Given sequence is random} | \text{Observed Data})$$

and in general

$$\tilde{P}\text{-value} \neq P\text{-value}$$

Thus we cannot use NIST Test Suite or DieHarder Tests to evaluate the increase/decrease in quality of randomness of a sequence; though; we may use the following criterion.

1. Entropy Test: Here, Entropy refers to the information density of the file's contents expressed as a number of bits per character.
2. Arithmetic Mean: We find the mean of all bits and compare it with the theoretical value of 0.5.
3. Serial Correlation Coefficient: Measures the Correlation between each byte in the file. For better randomness, this should be close to 0.

Simulation

We need to do three things to simulate the [Plesch-Pivoluska Protocol](#) using Qiskit and Python.

1. Make a circuit to prepare the GHZ state.
2. Based on the aforementioned circuit, build four circuits, one corresponding to each measurement setting.
3. Write code that
 - Sequentially reads the input file.
 - Fires the appropriate circuit and reads and saves the output.
 - Applies the Hadamard Extractor with desired Compression Factor (CF).

Preparing GHZ states

We begin by writing the code to create Tripartite GHZ states. Now a GHZ state is one with three particles in the state

$$\frac{|000\rangle + |111\rangle}{\sqrt{2}}$$

Say, state of Channel i is denoted by $|\psi_i\rangle$. To create this state, we begin by importing `qiskit` and other necessary modules. Next we initial state of all channels to $|0\rangle$.

We put a Hadamard operator on channel. We then apply CNOT to Channel 1 with reference to Channel 0. This way we get the first Bell state i.e.,

$$|\psi_0\psi_1\rangle = \frac{|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle}{\sqrt{2}} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Applying CNOT to Channel 2 with respect to Channel 0, we get

$$|\psi_0\psi_1\psi_2\rangle = \frac{|00\rangle \otimes |0\rangle + |11\rangle \otimes |1\rangle}{\sqrt{2}} = \frac{|000\rangle + |111\rangle}{\sqrt{2}}$$

Thus, the needed circuit is as shown below.

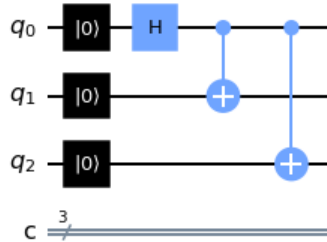


Figure 2: Circuit to prepare GHZ state

Preparing Circuits

Next, we create four circuits, corresponding to input bits, $s_i s_{i+1} = 11, 10, 01, 00$. For every combination of $s_i s_{i+1}$, we assign a unique xyz , where $xyz \in \{100, 010, 001, 111\}$. In particular we shall use the following map.

$s_i s_{i+1}$	xyz
01	001
10	010
00	100
11	111

Table 2: Measurement Setting Map

Thus, the circuits are as follows :

1. $s_i s_{i+1} = 01$, we measure Channel 2 in X -basis whereas Channel 0 and Channel 1 are measured in Y -basis.

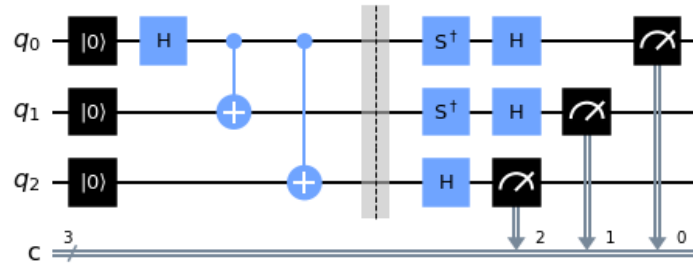


Figure 3: Circuit with measurement setting 001

2. $s_i s_{i+1} = 10$, we measure Channel 1 in X -basis whereas Channel 0 and Channel 2 are measured in Y -basis.

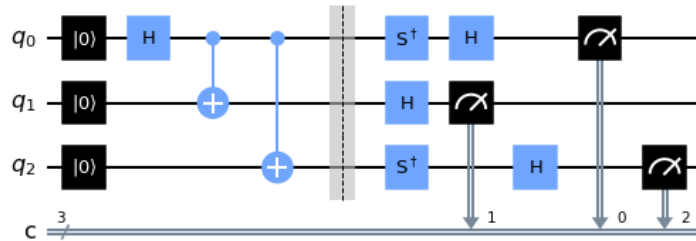


Figure 4: Circuit with measurement setting 010

3. $s_i s_{i+1} = 00$, we measure Channel 0 in X -basis whereas Channel 1 and Channel 2 are measured in Y -basis.

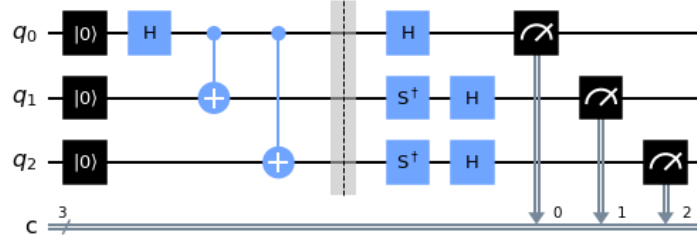


Figure 5: Circuit with measurement setting 100

4. $s_i s_{i+1} = 11$, we measure all Channels in X -basis.

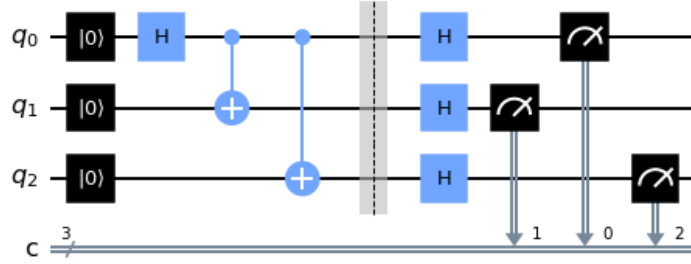



Figure 6: Circuit with measurement setting 111

Processing and I/O

Next we write a code to read input file in chunks using `pandas.read_csv`. For every chunk of file having n zeroes and ones, we fire each of the four circuits once with `shot=n` and `memory=True`; this is to optimize performance as firing Qiskit circuits is the slowest part of the simulation.

Next, we read each chunk in groups of $2k$, where $2k$ is the Compression Factor (CF). We read two of these $2k$ bits at a time, fire the appropriate circuit, then sequentially retrieve the results from RAM and append output bits to A and B. Once we have A and B, we apply the Hadamard extractor to A and B to get output bit $O = Ext(A, B)$.

On the next page, I have given a sketch of how the I/O part of the code works. Complete code for the simulation can be found [here](#) .

```


1 bit_len=16 ##### This our CF
2 chunk_size=10**7 ##### This our chunk size
3
4 for chunk in pd.read_csv('Input_Data.dat',index_col=False,chunksize=chunk_size):
5     input_=chunk.iloc[:,0]
6
7     #####
8     # Set all counters
9     k1=0 # and so on
10    #####
11
12    #####
13    # fire qiskit circuits once with shots=(chunk_size)
14    job1=simulator.run(compiled_circuit,shots=(chunk_size),memory=True)
15    result1=job1.result()
16    rez1=np.array(result1.get_memory(compiled_circuit)) #and so on
17    #####
18
19    #####
20    ####This part divides data into n-bit groups, where n=CF
21    for i in np.arange(length):
22        n=int(bit_len*i)
23        n_=int(bit_len*(i+1))
24        byte=np.array(input_[n:n_])
25        A=[]
26        B=[]
27
28        #####
29        ####This part divides data into 2-bit groups and processes them
30        for k in np.arange(bit_len/2):
31            ind=int(2*k)
32            ind_=int(2*(k+1))
33            sub_bit=np.array(byte[ind:ind_])
34            a=sub_bit[0]
35            b=sub_bit[1]
36
37            #####
38            ####This part reads each 2-bit group
39            ## and retrives data of appropriate circuit
40            if a==0 and b==1: #This is for xyz=001 and so on for other cases
41                A.append(int(rez1[k1][0]))
42                B.append(int(rez1[k1][1]))
43                k1=k1+1
44            #####
45
46            #####
47            # This part performs the Hadamard extractor
48            k_=myAND(A,B) # This a custom numpy ufunc that does AND
49            jj=reduce(lambda x, y: x ^ y, k_) # This performs the XOR
50            Out.append(jj)
51            #####
52            # This part outputs the data to specified file
53            df=pd.DataFrame(Out)
54            df.to_csv('Output.dat',mode='a',index=False,header=False)

```

Sketch of code for Processing and I/O

Results

Determining Optimal CF

Before we apply the [Plesch-Pivoluska Protocol](#) to any given dataset, we first need to determine an optimal Compression Factor (CF), using which we may perform our analysis. To do this, I prepared a data file using the 0.1-SV source and processed it for increasing CF values. The dataset and detailed NIST Test Suite Results can be found [here](#) . The results are as follows.

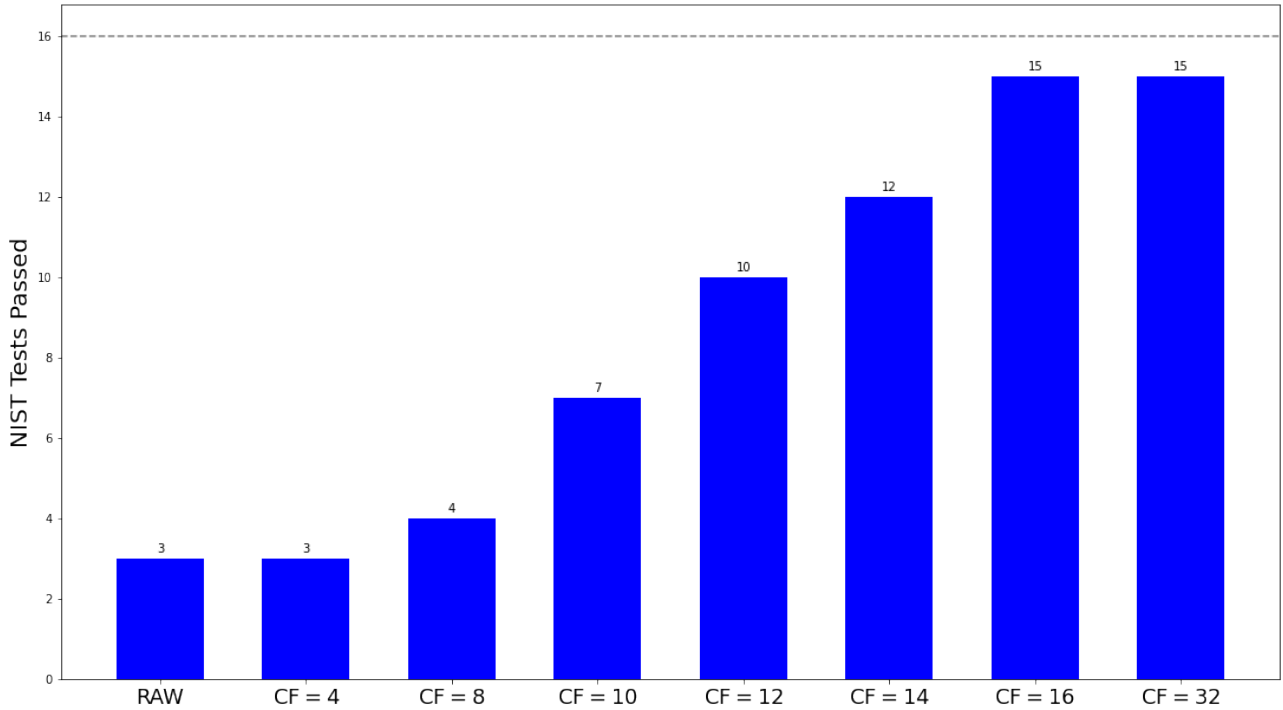



Figure 7: Number of NIST passed for varying CF

We see that the Optimal Compression Factor (CF) is about 16. Below this value, the failure rate is high, and beyond this value, there is no increase in success rate as the output file size gets increasingly smaller.

Note, even the CF= 16 and CF=32 files fail Maurier's Universal Test. This is expected as the output file size is reduced drastically for CF= 16 and CF= 32; Maurier's Universal Test also demands a comparatively large sample size[7]. Indeed processing a 60MB RAW input file, we get an Processed output file of about 3.6MB; the RAW file passes two of the NIST Tests, whereas the Processed file passes all the NIST Tests. The 60MB RAW and Processed file can be found [here](#).

Task 1

Prepared 4 dataset that simulate Santha-Vazirani sources with $\delta = 0.1$, $\delta = 0.2$, $\delta = 0.3$ and $\delta = 0.4$ respectively. I tested the RAW and Processed files for every dataset using the NIST Tests. The Processed and RAW files and detailed NIST Test results can be found [here](#) . The results are as follows.

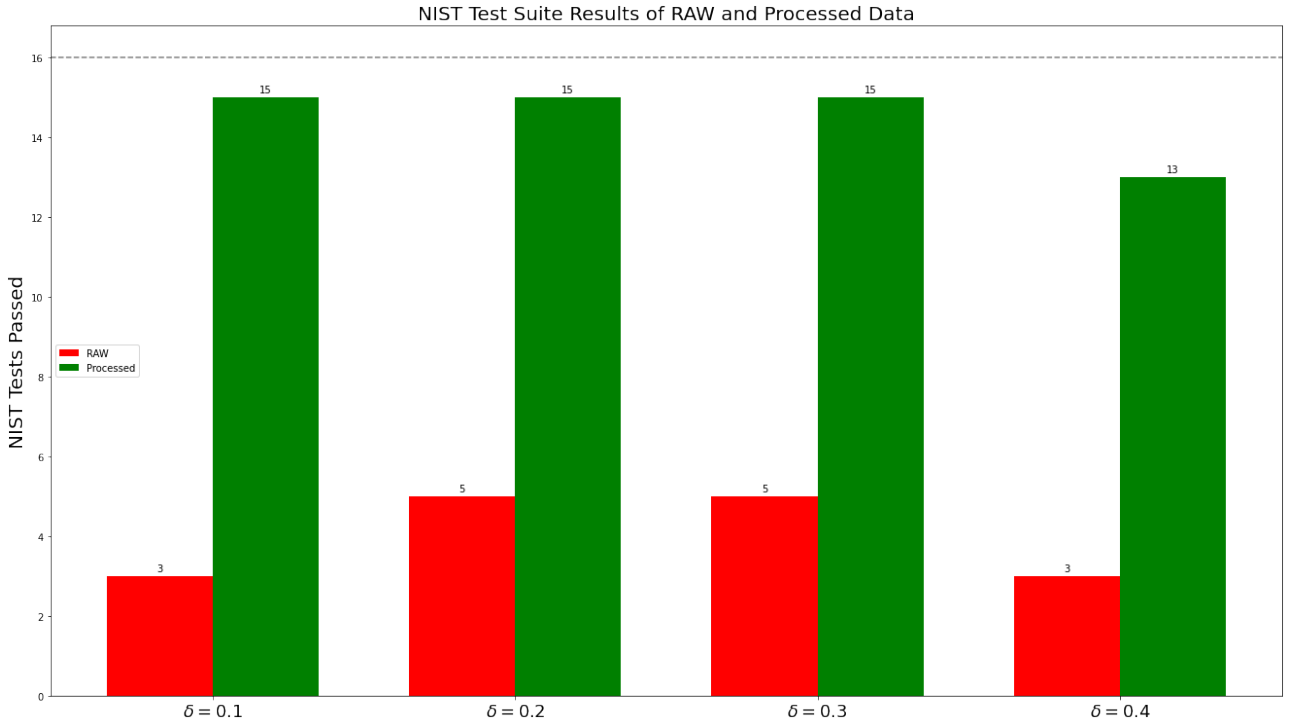


Figure 8: NIST Tests passed by RAW and Processed files from different SV sources

Note, $\delta = 0.4$ SV source, the processed file failed more NIST Tests compared to Processed file of other SV sources. This is understandable, as the higher the δ , the higher the bias, and thus a higher compression factor is needed, but then Processed file size decreases accordingly.

Note, all the processed datasets failed Maurier's Universal Test, but this was expected as explained [earlier](#).


Task 2

The change in mean and entropy for TRNG and PRNG was nearly the same. The change in correlation was significant (see fig. 9), mostly differing by about an order of magnitude. Note for set 2, the correlation of processed TRNG data increased instead of decreasing.

In the table above, $\Delta_{\text{Correlation}} = \frac{\text{Correlation RAW}}{\text{Correlation Processed}}$ and similarly for Δ_{Entropy} and Δ_{Mean} .

Dataset	$\Delta_{\text{Correlation}}$		Δ_{Entropy}		Δ_{Mean}	
	QRNG	PRNG	QRNG	PRNG	QRNG	PRNG
1	1.443	12.888	0.789	0.788	1.001	0.992
2	0.438	3.666	0.7885	0.7885	0.997	0.997
3	9.950	75.737	0.788	0.788	0.994	0.992
4	3.433	11.017	0.788	0.789	0.990	0.999

Table 3: Results (rounded to three decimal places)

To see the original values, click [here](#) .

The processed and RAW files and detailed NIST Test results can be found [here](#) .

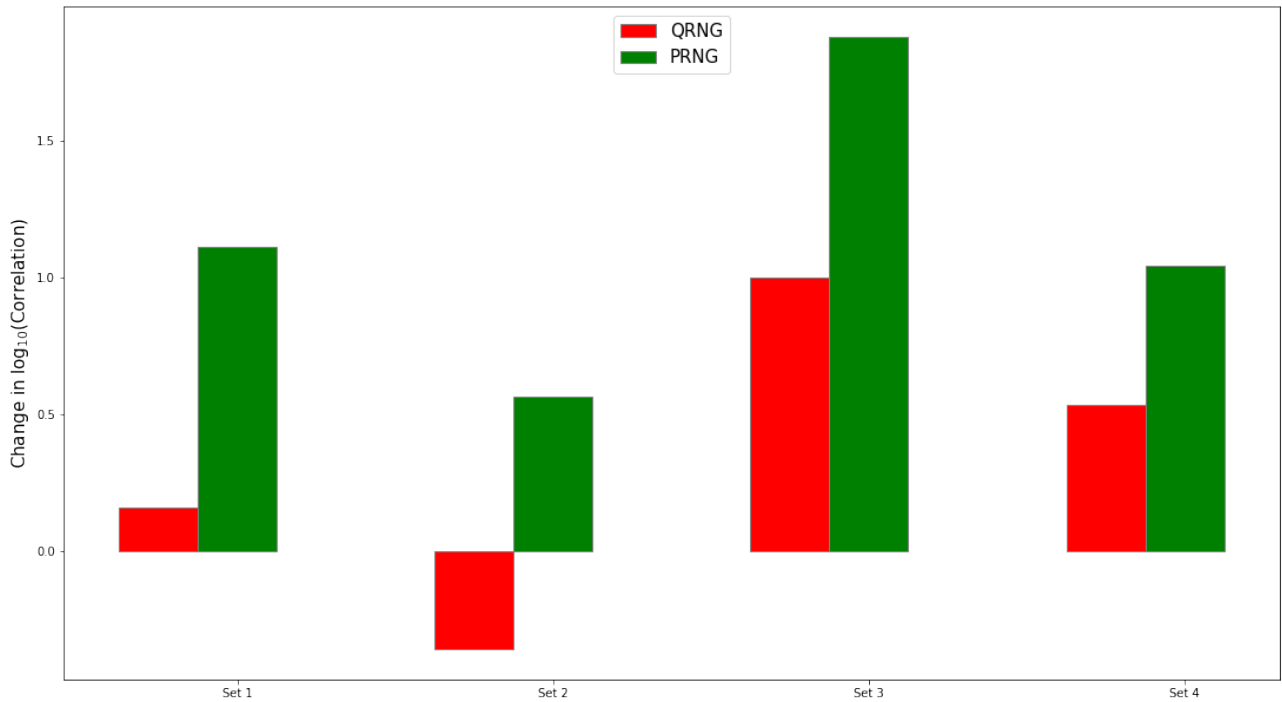


Figure 9: Plot showing change in $\log_{10}(\text{correlation})$

Conclusion

Task 1

Through the simulation, we note that [Plesch-Pivoluska Protocol](#) can indeed be used for amplifying the randomness of a biased entropy source. However, several issues hinder the practical implementation of this and other similar protocols. We discuss more pressing of such issues as follows.

- **Low efficiency**

We saw that the optimal compression factor is 16. So to get 1 GB of Random data, one needs about 16 GB of RAW data. Considering any practical implementation will also have other sources of loss, the efficiency of any practical implementation will be less than $\frac{1}{16}$, which is very impractical given the bit rate of current QRNGs. This low output bit rate issue can be partially solved by using better randomness extractors like Trevisan extractor[4], but then Device-independence will be sacrificed. Other extractors also impose stricter assumptions on the source.

- **GHZ State**

The protocol does not consider the faults inherent in any practical implementation in the GHZ state preparation and measurement. Cost is yet another factor; any apparatus for preparing and measuring GHZ states is neither easy to maintain nor cheap. It also adds bulk to the overall QRNG setup. Yet another strong argument is that a reasonably good apparatus to prepare and measure GHZ states can itself act as a QRNG. Indeed, this protocol is like coupling one entropy source (SV Source) with another entropy source (GHZ state measurement); but this has already been achieved successfully through much simpler means[5].

- **Speed**

One of the main problems faced during the simulation was that of speed. It was reading and firing appropriate circuits that slowed the simulation. In a practical scenario, it is reasonable to assume that this will be the slowest part of the machine, especially so when one considers the added complexity of accurately preparing and measuring GHZ states.

Task 2

Regarding the use of [Plesch-Pivoluska Randomness Amplification Protocol](#) as a test to differentiate PRNG and TRNG, we note one can not use change in mean or entropy as distinguishing criteria. Though $\Delta_{\text{Correlation}}$ seems to be a valid distinguishing candidate, however, to get any conclusive results, tests with larger datasets have to be performed. We also note that Qiskit's AerSimulator uses Mersenne Twister PRNG; while this was not a problem for Task 1, it certainly is for Task 2. Using IBM's real quantum computers will improve the accuracy of results; however, to use those, the code needs further optimization.

Miscellaneous

QKD Protocols

I learnt about Quantum Key Distribution Protocols, from BB84 to S13. In particular, I had a detailed overview of the protocols implemented at IITM, namely DPS-QKD and COW-QKD. One can broadly classify QKD protocols into two types based on working principles: Super-position based protocols and Entanglement based protocols[8]. BB84, SSP, B92, SARG04 and S13 are Super-position based protocols, whereas E91, BBM92, DPS and COW are Entanglement-based protocols.

1. **Differential Phase Shift (DPS)** [9]: Alice puts the photon in a 3-time bin superposition using 3-arm DLI (Delay Line Interferometer); she then randomly applies a differential phase of 0 or π . Bob recombines these using DLI and notes the time-bin of photon detection. He then informs Alice of the time-bins; from this, and the knowledge of random phase applies, they get a sifted key. The presence of Eve is determined by monitoring the Quantum Bit Error rate (QBER).
2. **Coherent One Way (COW)**[10]: Learnt the basics of Coherent-One Way QKD. Alice generates Laser pulses, these pulses pass through a variable attenuator to give the following logical bits.

$$\begin{aligned} |0\rangle &= |\alpha\rangle_{2k-1} |0\rangle_{2k} \\ |1\rangle &= |0\rangle_{2k-1} |\alpha\rangle_{2k} \end{aligned} \tag{1}$$


where $|\alpha\rangle$ denotes the weakly coherent state, and the subscripts denote the time-stamp. Bob informs Alice which items he got; from this and the record of states sent, Alice and Bob establish a sifted key. Alice also sends $|\alpha\rangle_{2k-1} |\alpha\rangle_{2k}$ decoy states; thus, Eve is noticed if she tries a PNS (Photon Number Splitting, basically capturing a photon) attack due to the missing photon. If Eve makes a coherent attack, she will break the coherence of the CW photon stream and get noticed.

Single Photon Detectors

Single Photon Detectors, or SPDs in short, are essential for Discrete Variable QKD and QRNG protocols. We describe a few characteristics essential for their practical deployment[11].

1. **Dead-time (τ)**: Sets a maximum on the detector counter rate, but for experiments, we usually have $\mu\tau \ll 1$, where μ is the mean photon number.
2. **Spectral Range**: Depends on the constituent materials and design.
3. **Dark counts**: Refers to False detections. Gating (activating the detector only for short intervals) can minimize dark counts.
4. **Detector efficiency**: For practical implementation, one has to consider all possible inefficiencies of the detector.
5. **Time Jitter**: Refers to the time interval between photon absorption and generation of output electric pulse. Time Jitter and Photon Resolution determine the maximum clock rate of photon counting experiments. Usually, Jitter is the dominant determining factor.

Arduino

Arduino is an integrated information processing platform equipped with a programmable microprocessor. With Arduino IDE and a little knowledge of C++, one can easily program an Arduino microprocessor to perform any desired task; this, combined with the cross-platform abilities of Arduino, makes it an integral part of any electronics project. Now, the SPD sends out an electrical pulse for each detection event; we need a way to count the number of pulses (and thus detection events) every second. To perform this task, I wrote a C++ based Arduino program using Interrupt function. The Interrupt function runs when there is a change in voltage of the specified target pin; one can specify if it is the rising edge or the falling edge, or the level change that triggers the Interrupt function. The code can be found [here](#) .

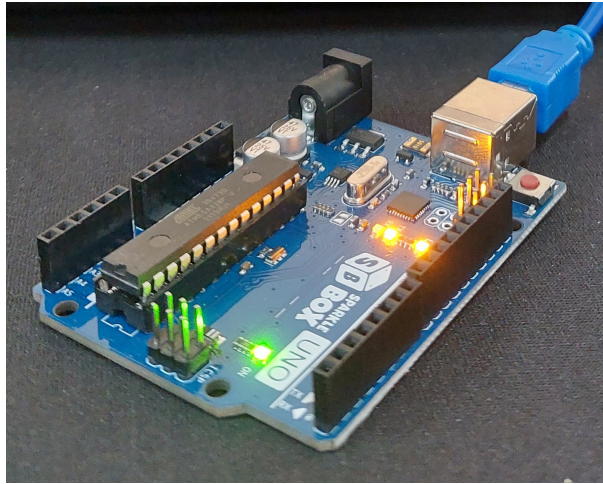


Figure 10: Arduino Uno

Power Loss in Optical Fibre

The QKD project needs an optical link between 5G Lab, ERNET IITM Research park and SETS Chennai. I learnt how to measure the power loss in an optical fibre. The procedure is as follows :

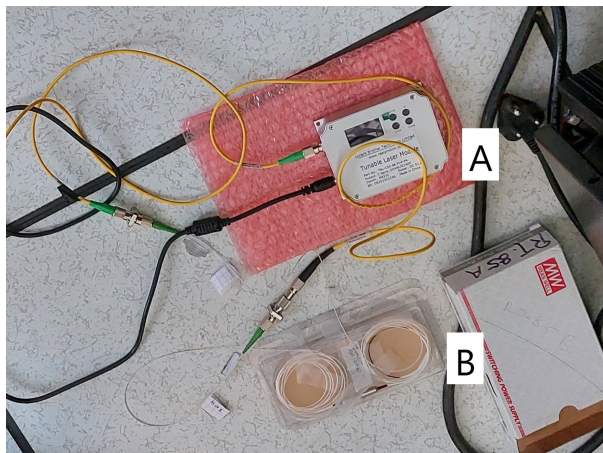


Figure 11: A labels the Laser module, B labels the Isolator

1. Connect the Laser to the Isolator module; this ensures no back-scattered (which can light damage the Laser).
2. Connect the other end of the Isolator to the optical fibre link connector at ERNET.
3. Power on the Laser and set the wavelength to $1550nm$ (this is the wavelength of operation for the QKD system).



Figure 12: A labels the Laser module, B labels the Isolator

4. Measure the output at the other end of the cable using a handheld Optical power meter.

References

- (1) Plesch, M.; Pivoluska, M. *Physics Letters A* **2014**, 378, 2938–2944.
- (2) Brandão, F. G.; Ramanathan, R.; Grudka, A.; Horodecki, K.; Horodecki, M.; Horodecki, P.; Szarek, T.; Wojewódka, H. *Nature communications* **2016**, 7, 1–6.
- (3) Colbeck, R.; Renner, R. *Nature Physics* **2012**, 8, 450–453.
- (4) Ma, X.; Xu, F.; Xu, H.; Tan, X.; Qi, B.; Lo, H.-K. *Physical Review A* **2013**, 87, 062327.
- (5) Shaw, G.; Sivaram, S.; Prabhakar, A. **2019**, 1–4.
- (6) Bassham III, L. E.; Rukhin, A. L.; Soto, J.; Nechvatal, J. R.; Smid, M. E.; Barker, E. B.; Leigh, S. D.; Levenson, M.; Vangel, M.; Banks, D. L. **2010**.
- (7) Coron, J.-S.; Naccache, D. **1998**, 57–71.
- (8) Nurhadi, A. I.; Syambas, N. R. **2018**, 1–5.
- (9) Inoue, K.; Waks, E.; Yamamoto, Y. *Phys. Rev. Lett.* **2002**, 89, 037902.
- (10) Gisin, N.; Ribordy, G.; Zbinden, H.; Stucki, D.; Brunner, N.; Scarani, V. **2004**.
- (11) Hadfield, R. H. *Nature photonics* **2009**, 3, 696–705.