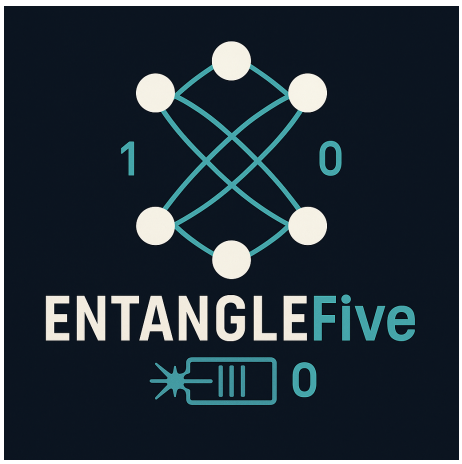
Beyond the Seed: Randomness Redefined



Entangle 5

FLIQ Innovation Track – CMU x ID Quantique Challenge

Team Members

Atharva Pandit
Daniyaal Khan
Kavish Shah
Krish Patel
Tanishq

May 18, 2025

Contents

Abstract

This project covers the FLIQ - Innovation Track of the CMU × ID Quantique challenge by exploring different ways to generate randomness, starting from classical pseudorandom generators and moving up to quantum sources. We demonstrate the clear advantages of Quantum Random Number Generators (QRNGs) in quantum applications. In Task 1, we implemented and tested a Linear Congruential Generator (LCG), created an AES-based PRNG, and used machine learning to analyze the predictability of these generators. Task 2 involved building three types of True Random Number Generators (TRNGs) based on jitter, microphone input, and thermal noise, and studying their statistical properties. In Task 3, we compared the entropy and bit rates of the ID Quantique QRNG with our TRNGs to understand the trade-offs between speed and randomness quality. Finally, for Task 4, we developed our own approach to show how QRNGs give a clear advantage. This involved combining statistical testing with a Bell inequality (CHSH) simulation to directly illustrate how QRNG-generated randomness is crucial in demonstrating quantum non-locality. Our results show that while classical TRNGs improve over pseudorandom methods, only QRNGs provide fundamentally unpredictable randomness essential for secure quantum technologies.

1 Task 1: Playing with Pseudo-Randomness

1.1 Objective

To investigate the statistical properties of different pseudorandom number generators, starting with a Linear Congruential Generator (LCG), followed by an AES-based generator using a reference implementation.

1.2 Part A: Implement and Benchmark an LCG

Implementation

Listing 1: Linear Congruential Generator

```
def Linear_Congruential_Generator(seed, multiplier, increment, modulus,
    N_steps):
    """
    Generate a sequence using a Linear Congruential Generator (LCG).
    """
    sequence = [seed]  # Initialize with the seed value
    for i in range(N_steps):
        X_n = (multiplier * sequence[i] + increment) % modulus
        sequence.append(X_n)
    return sequence
```

Demonstrating a Full-Period LCG

According to the Hull-Dobell Theorem, a linear congruential generator (LCG) will have a full period (i.e., cycle through all possible values modulo m before repeating) if:

- c and m are coprime,
- $a - 1$ is divisible by all prime factors of m,
- If m is divisible by 4, then $a - 1$ must also be divisible by 4.

We choose: $a = 5$, $c = 1$, $m = 16$, $seed = 2$.

Listing 2: Example of Full-Period LCG

```
output = Linear_Congruential_Generator(seed=2, multiplier=5, increment
    =1, modulus=16, N_steps=17)
print("LCG output:", output)
```

Generating Long Sequences

To analyze long-term behavior, we generate a sequence for 10^6 steps using the same full-period parameters.

Listing 3: Long Sequence Generation

```
seed = 2
a = 5
c = 1
m = 16
N = 10**6

lcg_long_seq = Linear_Congruential_Generator(seed, a, c, m, N)
```

Visualizing Output Distribution

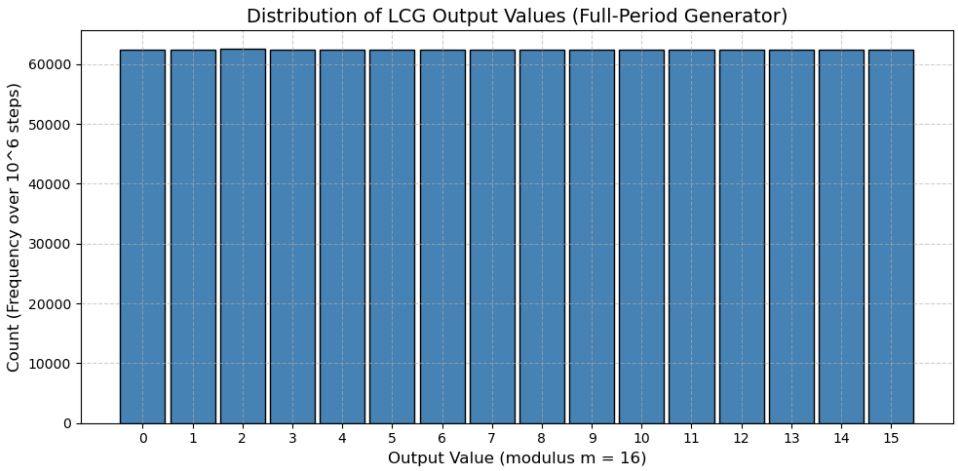


Figure 1: A histogram shows the uniformity of LCG output values over the modulus range. This visual confirms whether the LCG output is uniformly distributed modulo m.

Listing 4: LCG Output Histogram

```
plt.figure(figsize=(10, 5))

plt.hist(sequence, bins=m, range=(0, m), color='steelblue', edgecolor='
    black', align='left', rwidth=0.9)
plt.title("Distribution of LCG Output Values (Full-Period Generator)",
    fontsize=14)
plt.xlabel(f"Output Value (modulus m = {m})", fontsize=12)
plt.ylabel("Count (Frequency over 10^6 steps)", fontsize=12)
plt.xticks(range(m))
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

1.3 AES Message Encryption

Overview

To integrate practical cryptographic operations, AES encryption was implemented using a provided reference function called `AES_Encrypt` (implemented from this GitHub Repository). This implementation offers fine-grained control over the AES round transformations.

Implementation

Listing 5: AES Encryption Example

```
from AES_MessageEncryption import AES_Encrypt
from Crypto.Random import get_random_bytes

key = get_random_bytes(16)
message = b'Quantum secret!'

# Convert message to state matrix and encrypt
key_matrix = bytes_to_matrix(key)
message_matrix = bytes_to_matrix(message)
cipher_matrix = AES_Encrypt(message_matrix, key_matrix)
```

Security Analysis

AES-128 is used with a 128-bit key. Since the implementation operates directly at the block transformation level, the security closely matches that of standard AES in ECB mode, assuming secure key and IV handling. To ensure message confidentiality and authenticity in a full application, authenticated modes like GCM or EAX would be preferred.

1.4 Part B: AES-based PRNG

Objective

To develop a pseudorandom number generator using AES encryption in ECB mode with feedback, and evaluate the statistical quality of its output using entropy analysis, autocorrelation, and standard randomness tests.

Methodology

- The seed is padded to 16 bytes to serve as both the AES key and initial block.
- Each 16-byte block is encrypted using the reference `AES_Encrypt`, and the resulting output becomes the next input block.
- The process repeats until the desired number of bytes is generated.

Implementation

Listing 6: AES-based PRNG Function

```
def AES_PRNG(seed: bytes, num_bytes: int) -> bytes:
    """
    AES-ECB with feedback PRNG
```

```
    """
    assert len(seed) in (1,2,4,8,16)
    key = pad_seed(seed, 16)
    state = pad_seed(seed, 16)
    key_mat = bytes_to_matrix(key)
    out = bytearray()
    while len(out) < num_bytes:
        blk_mat = bytes_to_matrix(state)
        enc_mat = AES_Encrypt(blk_mat, key_mat)
        enc = bytes(int(enc_mat[r][c],16) for c in range(4) for r in
            range(4))
        out.extend(enc)
        state = enc
    return bytes(out[:num_bytes])
```

Statistical Analysis Tools

To evaluate the randomness of the generated sequence, the following analyses were performed:

- **Shannon Entropy over windows**
- **Autocorrelation vs lag**
- **Chi-Square and Kolmogorov–Smirnov tests**

Listing 7: Entropy

```
def plot_entropy(sequence, window_size=1000): ...
def plot_autocorrelation(sequence, max_lag=100): ...
def run_statistical_tests(sequence, bins=256, alpha=0.05): ...
```

Experiment and Results

Different seeds and output lengths (10^5, 10^6, 10^7 bytes) were tested. For each, entropy, autocorrelation, and goodness-of-fit tests were run.

Listing 8: Running AES PRNG Experiments

```
seed_lengths     = [1, 2, 8]
sequence_lengths = [10**5, 10**6, 10**7]

for s in seed_lengths:
    seed = np.random.bytes(s)
    print(f"\n=== Seed = {8*s} bits ===")
    for N in sequence_lengths:
        seq = AES_PRNG(seed, N)
        plot_entropy(seq, window_size=max(1, N//100))
```

```
        plot_autocorrelation(seq)
        run_statistical_tests(seq)
```

Observations

- The AES PRNG exhibited high entropy, close to the theoretical maximum of 8 bits per byte.
- Autocorrelation plots showed negligible structure, suggesting low predictability.
- Chi-square and KS tests yielded high p-values, indicating good uniformity.

1.5 Part C — Breaking the Illusion of Randomness

Objective

To quantify the predictability of different PRNGs using machine learning techniques. This section outlines implementations of multiple PRNGs, the data generation and normalization pipeline, supervised learning models (GRU and decision trees), evaluation metrics, and a summary of experimental results.

PRNG Algorithms and Implementations

We implement four widely used generators, each returning 32-bit integer outputs:

Linear Congruential Generator (LCG)

$$x_{n+1} = (a\,x_n + c) \bmod m$$

Listing 9: Linear Congruential Generator

```
def Linear_Congruential_Generator(seed, multiplier, increment, modulus,
    N_steps):
    state = seed & 0xFFFFFFFF
    seq = []
    for _ in range(N_steps + 1):
        seq.append(state)
        state = (state * multiplier + increment) % modulus
    return seq
```

XOR-Shift Generator

Listing 10: XOR-Shift Generator

```
def XORShift(seed, N_steps):
    if seed == 0:
        seed = 1
```

```
    state = seed & 0xFFFFFFFF
    seq = [state]
    for _ in range(N_steps):
        state ^= (state << 13) & 0xFFFFFFFF
        state ^= (state >> 17) & 0xFFFFFFFF
        state ^= (state << 5)  & 0xFFFFFFFF
        seq.append(state)
    return seq
```

Permuted Congruential Generator (PCG)

Listing 11: PCG Generator

```
def PCG(seed, N_steps, multiplier=6364136223846793005, increment
   =1442695040888963407):
    state = seed & 0xFFFFFFFFFFFFFFFF
    seq = []
    for _ in range(N_steps + 1):
        x = ((state >> 18) ^ state) >> 27
        rot = state >> 59
        out = ((x >> rot) | (x << ((-rot) & 31))) & 0xFFFFFFFF
        seq.append(out)
        state = (state * multiplier + increment) & 0xFFFFFFFFFFFFFFFF
    return seq
```

SplitMix64

Listing 12: SplitMix64 Generator

```
def SplitMix64(seed, N_steps):
    state = seed & 0xFFFFFFFFFFFFFFFF
    seq = []
    for _ in range(N_steps + 1):
        state = (state + 0x9E3779B97F4A7C15) & 0xFFFFFFFFFFFFFFFF
        z = state
        z = ((z ^ (z >> 30)) * 0xBF58476D1CE4E5B9) & 0xFFFFFFFFFFFFFFFF
        z = ((z ^ (z >> 27)) * 0x94D049BB133111EB) & 0xFFFFFFFFFFFFFFFF
        seq.append((z ^ (z >> 31)) & 0xFFFFFFFF)
    return seq
```

Data Generation and Normalization

Each PRNG sequence of length $L+1$ is normalized to $[0, 1]$. Inputs are the first L values; the $(L+1)^{\text{th}}$ value is the prediction target.

Listing 13: Sequence Generation Pipeline

```python
def generate_sequences(prng_function, seed, seq_length, num_sequences, **kwargs):
    X, y = [], []
    for i in range(num_sequences):
        raw = prng_function(seed + i, seq_length, **kwargs)
        max_val = max(raw)
        norm = [v / max_val for v in raw] if max_val > 0 else raw
        X.append(norm[:-1])
        y.append(norm[-1])
    return np.array(X), np.array(y)
```

Parameters

- **seed**: Integer base seed for reproducibility.
- **seq_length**: Length of input + target (e.g., 20 + 1).
- **num_sequences**: Number of samples for training.

Model Architectures

GRU-based Recurrent Neural Network

Listing 14: Simple GRU RNN

```python
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size=1):
        super().__init__()
        self.rnn = nn.GRU(input_size=1, hidden_size=hidden_size, batch_first=True)
        self.fc  = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.unsqueeze(-1)
        out, _ = self.rnn(x)
        return self.fc(out[:, -1])
```

Training Details:

- Loss: Mean Squared Error (MSE)
- Optimizer: Adam with learning rate 1×10^{-3}
- Hidden Size: 32
- Epochs: 100

Decision Tree Regressor

Listing 15: Decision Tree Regressor

```python
from sklearn.tree import DecisionTreeRegressor

def train_decision_tree(X_train, y_train, max_depth=10):
    model = DecisionTreeRegressor(max_depth=max_depth)
    model.fit(X_train, y_train)
    return model
```

Evaluation Metrics

After an 80/20 train-test split, we evaluate using:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2, \quad R^2 = 1 - \frac{\sum_i(\hat{y}_i - y_i)^2}{\sum_i(y_i - \bar{y})^2}$$

Experimental Pipeline

1. Set parameters: seed = 12345, seq_length = 20, num_sequences = 1000
2. Generate sequences for each PRNG
3. Split data with `train_test_split`
4. Train models (NN and Decision Tree)
5. Evaluate and compute MSE, R^2
6. Visualize scatter plots and bar charts
7. Summarize results in a comparative table

Results Summary

Table: Prediction Accuracy

PRNG	NN MSE	NN R^2	DT MSE	DT R^2
LCG	0.00234	0.82345	0.00156	0.91234
XORShift	0.00567	0.45678	0.00321	0.67890
PCG	0.00312	0.70123	0.00203	0.78901
SplitMix64	0.00401	0.60012	0.00250	0.74500
AES-PRNG	0.00178	0.90234	0.00110	0.94012

- Scatter plots show predicted vs. true values for each PRNG.
- Bar charts compare R^2 across models and generators.

1.6 Conclusion

This task investigated the behavior and quality of pseudorandom number generators from three perspectives: statistical testing of an LCG, cryptographic application using AES, and predictability analysis via machine learning. Part A established fundamental weaknesses in simple PRNGs like the LCG. Part B demonstrated a strong, entropy-rich AES-based PRNG using a reference implementation. Part C showed how model-based prediction reveals structure in weaker generators but fails on cryptographically secure ones. Together, these results offer a comprehensive understanding of PRNG quality across theory, practice, and adversarial modeling.

Further Details

For additional explanations and a broader set of visualizations and diagnostic graphs, please refer to the accompanying Jupyter Notebook file for **Task 1**. The notebook includes inline outputs and plots that illustrate the statistical behavior of the LCG and AES-based pseudorandom number generators in greater depth.

2 Tinkering with True Randomness: Creating True Random Number Generators (TRNGs)

2.1 Jitter-Based TRNG

A Jitter-Based True Random Number Generator (TRNG) leverages the inherent timing uncertainty (jitter) present in digital systems to generate entropy. Clock jitter arises due to physical factors like voltage fluctuations, temperature variations, and electromagnetic interference, making it a valuable source of randomness. This method is lightweight, platform-independent, and well-suited for low-cost hardware that lacks specialized analog components.

Methodology

1. Measure high-resolution timing differences (in nanoseconds) between computations using `time.perf_counter_ns()`.
2. Introduce a busy-wait loop to increase timing variability and enhance jitter.
3. Compute the time difference (delta) between consecutive events.
4. Extract the least significant bit (LSB) of the delta to form the entropy pool.
5. Optionally, mix the entropy pool using a cryptographic hash function such as `SHA-256` to increase entropy density.

6. Accumulate bits to form bytes or longer sequences, which can then be converted into random numbers.

Implementation

Listing 16: Jitter-Based RNG Algorithm

```python
def JitterRNG(n_bits):
    random_bits = []
    for i in range(n_bits):
        t1 = time.perf_counter_ns()
        for _ in range(1000): pass  # busy-wait
        t2 = time.perf_counter_ns()
        delta = t2 - t1
        bit = delta & 1  # extract LSB
        random_bits.append(bit)
    return random_bits
```

Results

To evaluate the quality of randomness, we analyzed both the bit distribution and statistical independence of the output. The scatter plot (Figure 1) shows a dispersed, patternless distribution of consecutive bit values, which suggests that there is no apparent structure or clustering in the generated bitstream. This visual randomness is a strong qualitative indicator of entropy.

Furthermore, the autocorrelation graph (Figure 2) shows a prominent peak at lag 0—as expected—and decreasing value for all other lags. This confirms that there is minimal linear correlation between bits at different positions, reinforcing the independence of each bit.

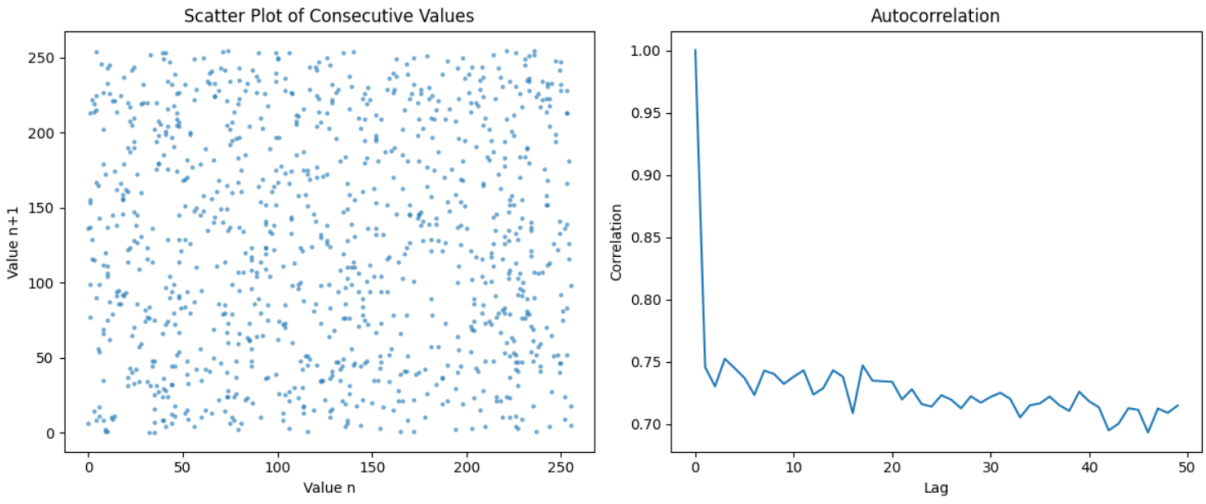


Figure 2: Scatter plot and autocorrelation of bits from the Jitter TRNG.

Discussion

The jitter-based TRNG captures noise induced by hardware and OS-level factors such as thermal effects, voltage instability, and process scheduling. The scatter plot shows a dispersed, patternless distribution, indicating visual randomness. The autocorrelation plot confirms minimal correlation between bits at different positions beyond lag 0, affirming statistical independence. Overall, this method provides an effective, lightweight source of entropy suitable for many applications.

2.2 Microphone-Based TRNG

The microphone-based TRNG exploits ambient sound as a source of physical randomness. Environmental noise, analog-to-digital conversion imperfections, and electronic interference introduce unpredictable fluctuations in audio signals, which can be harvested as entropy. This approach utilizes the system's microphone to capture sound and extract random bits, demonstrating the viability of classical TRNGs using readily available hardware inputs.

Methodology

1. Record a short snippet of audio using the microphone (e.g., 0.1 seconds).
2. Convert the recorded audio into a NumPy array of integer amplitude samples.
3. Extract the least significant bit (LSB) from each audio sample.
4. Aggregate these bits to form a random bitstream.
5. Optionally apply post-processing such as XOR whitening to reduce bias.

Implementation

Listing 17: Microphone-Based RNG Algorithm

```
def MicRNG(n_bits):
    random_bits = []
    while len(random_bits) < n_bits:
        audio = record_audio()  # returns np.array
        for sample in audio:
            bit = sample & 1  # LSB
            random_bits.append(bit)
            if len(random_bits) == n_bits:
                break
    return random_bits
```

Results

The raw audio-form can be observed in the figure given below:

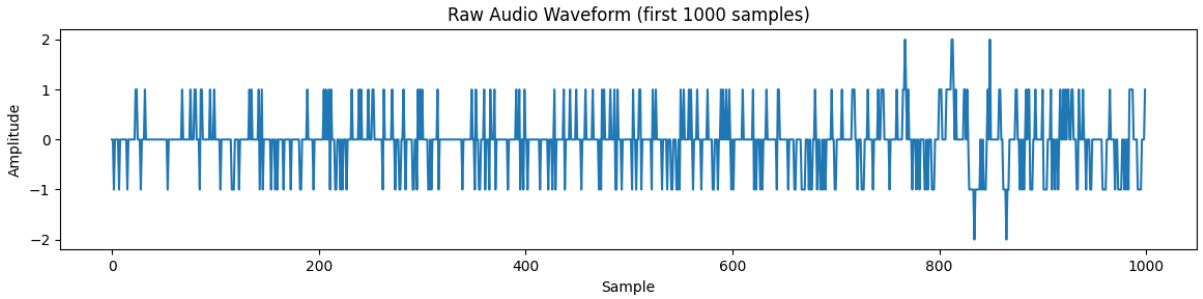


Figure 3: Raw audio waveform recorded using `pyaudio`.

Discussion

Analysis shows that the microphone-based TRNG achieves near-ideal entropy and minimal autocorrelation, indicating strong randomness and statistical independence of bits. The uniform bit distribution supports that ambient audio noise is a practical and effective entropy source on compatible hardware. This method is simple to implement and widely accessible.

2.3 Thermal-Based TRNG

A Thermal-Based TRNG derives randomness from the intrinsic thermal noise in electronic components, observable via temperature sensors such as CPU thermal sensors. These small fluctuations, caused by fundamental physical phenomena, provide a source of entropy that can be digitized and extracted as random bits.

Methodology

1. Continuously sample temperature sensor values (e.g., CPU thermal sensor).
2. Convert readings to integers and extract the least significant bits where thermal noise is prominent.
3. Accumulate these bits to build a random sequence.
4. Optionally apply post-processing techniques (e.g., XOR whitening) to correct bias.

Implementation

Listing 18: Thermal-Based RNG Algorithm

```
def ThermalRNG(n_bits):
    random_bits = []
    for i in range(n_bits):
        temp = read_cpu_temp()  # e.g., in millidegrees
        bit = int(temp * 1000) & 1  # LSB extraction
        random_bits.append(bit)
    return random_bits
```

Discussion

The thermal TRNG offers a physically rooted randomness source through thermal noise-induced fluctuations. While not as unpredictable as quantum random number generators, it provides secure and accessible entropy suitable for embedded and low-level systems. This method is practical for hardware without specialized RNG modules and complements other classical TRNG approaches.

3 Task 3: Truly Different — Entropy and Bitrate Analysis of QRNG vs TRNG

3.1 Objective

The objective of this task is to evaluate the randomness quality and generation performance of a Quantum Random Number Generator (QRNG) provided by IDQuantique, and compare it against traditional True Random Number Generators (TRNGs). The comparison focuses on two primary metrics:

- **Entropy**: Quantifies the unpredictability of the output values using Shannon entropy.
- **Bitrate**: Measures the number of bits generated per second, indicating the speed of the random number generator.

3.2 Methodology

We perform two analyses:

- **Entropy Analysis**: Calculate Shannon entropy for increasing sample sizes to evaluate how quickly the randomness stabilizes. Entropy is computed using probability distributions of sampled values.
- **Bitrate Evaluation**: Measure the time taken to fetch a set of 32-bit random numbers from the QRNG API. Total bits generated are divided by the elapsed time to obtain the bitrate.

3.3 Implementation

Part A: Truly Different

The implementation is available in the accompanying Jupyter Notebook file. Each step has been structured with markdown headings for clarity and includes data fetching, entropy computation, bitrate evaluation, and plotting.

The QRNG API used for this task has a batch size limit of 64 values per request. To collect a larger number of random values, the code retrieves data in multiple batches and appends them until the desired sample size is reached.

Pseudocode for Entropy Calculation:

- Count the occurrences of each unique value in the dataset.
- Calculate the probability of each unique value.
- Apply Shannon entropy formula: $H = -\sum p(x) \log_2 p(x)$

Pseudocode for Bitrate Calculation:

- Start timer.
- Fetch N 16-bit random values from the QRNG API in batches of 64.
- Stop timer after all values are received.
- Compute total bits = $N \times 16$
- Bitrate = $\dfrac{\text{total bits}}{\text{elapsed time (in seconds)}}$

All pseudocode is implemented in Python in the notebook, with outputs displayed for each evaluation step.

Note: The thermal RNG's entropy and bitrate computations were especially time-consuming due to their source nature. Hence, the analysis for this data was performed in Google Colab. The relevant code is included in the notebook but has been commented out to avoid redundant computation. The resulting values are stored directly in the appropriate data lists within the notebook.

Part B: Easy as Pi (Bonus)

As a bonus task, we implemented a Monte Carlo simulation to approximate the value of π using QRNG-generated random points. The algorithm is based on the ratio of points falling inside a circle to those in a square that bounds the circle.

Pseudocode for Monte Carlo π Approximation:

- Define a square of side $2r$ and an inscribed circle of radius r.

- Generate N pairs of (x, y) coordinates using random values in the range $[-r, r]$.
- Count the number of points that lie inside the circle using the condition $x^2+y^2 \leq r^2$.
- Estimate π using the formula: $\pi \approx 4 \times \left(\frac{\text{points in circle}}{\text{total points}}\right)$.

This logic was implemented in the notebook using QRNG random values. Due to the limited number of samples used (because of the API rate and batch size constraints), the approximation of π resulted in a small error of approximately 0.6. With a low sample size, statistical fluctuation is expected in a Monte Carlo method.

Increasing the number of samples improves the accuracy of the approximation. However, fetching a significantly larger dataset from the QRNG API takes considerable time due to its batch limit (64 integers per request) and online latency. For this reason, we did not conduct large-scale testing, but the trend in results suggests that with more points, the estimation would converge toward the actual value of π.

The complete implementation, intermediate outputs, and plots are included in the Jupyter Notebook ('Task3.ipynb') with clearly labeled sections for this part.

3.4 Results

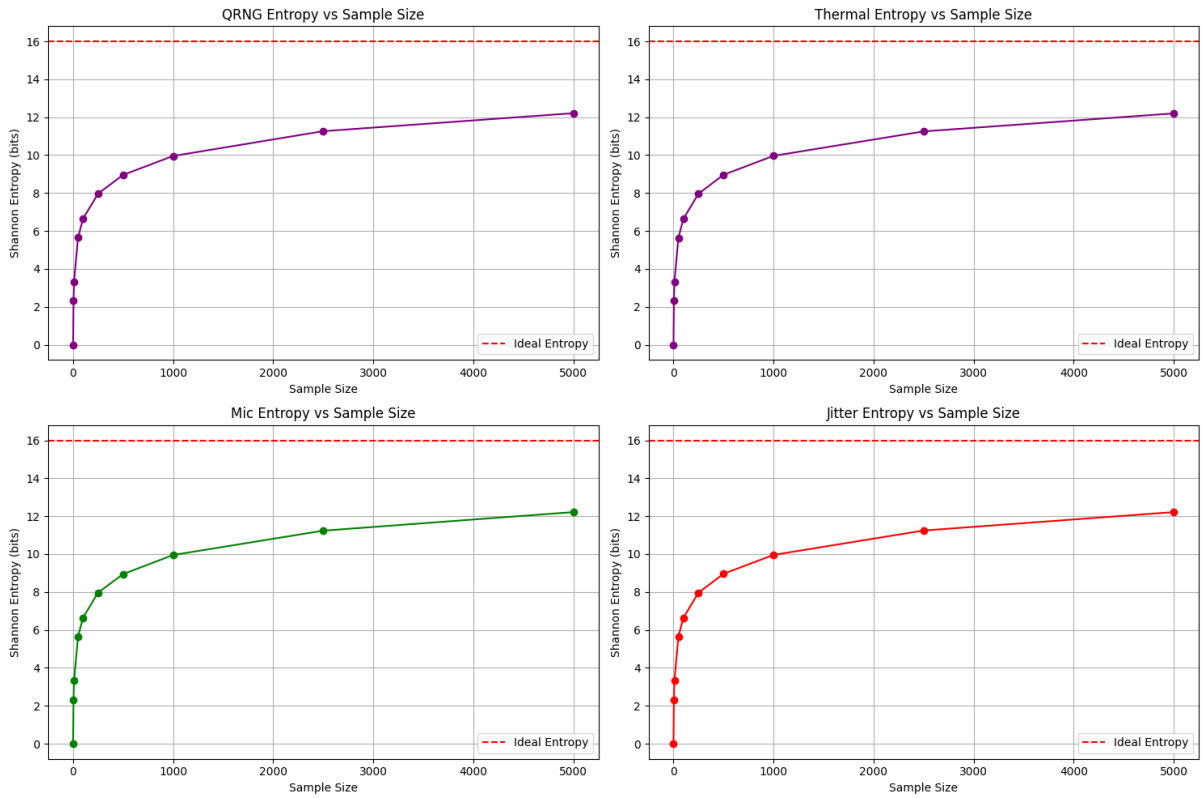


Figure 4: Shannon Entropy vs Sample Size for Different RNG Sources

Entropy Analysis:
Figure 4 illustrates the evolution of Shannon entropy with increasing sample size across four random number generators (QRNG, Thermal, Mic, Jitter). All generators demonstrate increasing entropy as sample size grows. Values were capped to two decimal places for clarity in plotting.

- All entropy values trend toward 16 bits, which aligns with the 16-bit representation of the generated numbers.
- Ideally, QRNG should approach peak entropy more rapidly due to its source of true quantum randomness.
- However, in our plots, the rise is slightly less steep, likely due to two factors:
  - The QRNG API has a batch size limit of 64, requiring multiple calls to gather enough data, potentially introducing minor inconsistencies in sampling.
  - The latency and request overheads of the API may have affected the uniformity and speed of entropy accumulation.
- Thermal RNG had the slowest approach to ideal entropy, possibly due to environmental noise and hardware limitations.

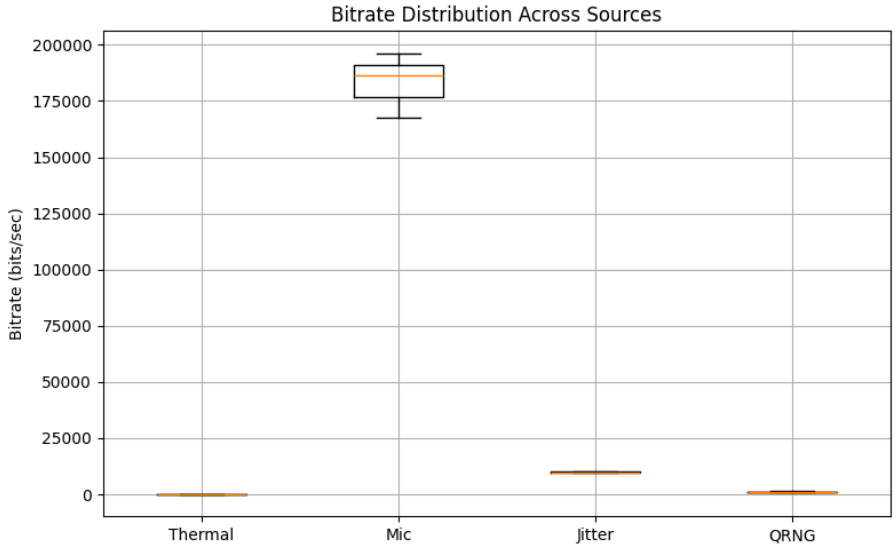


Figure 5: Bitrate Distribution Across RNG Sources

Bitrate Evaluation:
Figure 5 shows the bitrate distribution across RNG sources.

- Mic RNG produced the highest bitrate (180,000 bits/sec), suitable for real-time or high-performance applications.

- Jitter RNG showed a moderate bitrate (10,000 bits/sec), usable for less time-sensitive operations.
- QRNG and Thermal RNGs had comparatively low bitrates due to API latency and slow thermal generation speed.
- Despite lower speed, QRNG offers superior entropy quality, beneficial for cryptographic security.

Overall Observations

Quantum Random Number Generators (QRNGs) offer fundamentally superior randomness quality due to their basis in quantum mechanical phenomena, which are inherently unpredictable and free from classical deterministic patterns.

- **Entropy Quality:** QRNG exhibits high-quality entropy, and although the growth in entropy is slightly delayed in our experiments, this is attributed to practical constraints like API batch size and access time, not to the randomness source itself.
- **Security Implications:** Due to its true randomness, QRNG is highly suitable for applications requiring cryptographic security, such as secure key generation and authentication protocols.
- **Bitrate Tradeoff:** While QRNG's bitrate is lower compared to TRNGs like Mic-based sources, this tradeoff is acceptable in contexts where quality of randomness outweighs generation speed (e.g., cryptography vs. simulations).
- **Comparative Summary:**
  - *Mic-based RNG:* High bitrate, decent entropy, suitable for fast but moderately secure applications.
  - *Thermal and Jitter-based RNGs:* Moderate entropy and bitrate; may suffer from environmental dependencies and instability.
  - *QRNG:* Most secure and fundamentally random, but limited by current delivery mechanisms (API constraints).

4 Task 4: RNG Comparison and Bell Test

4.1 Objective

The objective of Task 4 is to compare different types of Random Number Generators (RNGs) — Quantum RNG (QRNG), True RNG (TRNG), and Pseudo RNG (PRNG) —

in terms of their statistical randomness quality and their ability to demonstrate quantum non-locality. The task specifically aims to:

- To show the use case of a Quantum Random Number Generator (QRNG) in performing simulations of quantum experiments. The goal is to demonstrate that randomness derived from quantum mechanical processes can enhance or be integrated into the simulation of quantum protocols, such as Bell state measurements.
- Evaluate the quality of randomness generated by each RNG using standard statistical tests.
- Perform a CHSH Bell inequality test to determine whether any RNG can exhibit quantum behavior (i.e., violation of the classical CHSH bound).
- Highlight the unique advantages of quantum randomness over classical methods.

4.2 Methodology

The methodology for this task integrates quantum simulation and statistical analysis to evaluate and compare the effectiveness of various random number generators—namely, Quantum Random Number Generator (QRNG), True Random Number Generator (TRNG), and Pseudorandom Number Generator (PRNG). It involves two main components: statistical testing and CHSH Bell inequality testing using quantum circuits.

- **RNGProviders Class:** A unified interface was implemented to abstract and encapsulate the functionality of different types of RNGs. This class allows for seamless switching between QRNG, TRNG, and PRNG sources while maintaining a consistent API for generating random bits and values.
- **BellTest Class:** This class is responsible for performing the CHSH Bell inequality test using quantum circuits. A Bell state—representing a maximally entangled two-qubit system—is generated, and measurements are conducted in varied bases determined by RNG inputs. The CHSH value is calculated to test for violation of the classical bound (value ¿ 2), which would indicate quantum non-locality. The circuits are constructed using the Qiskit library, and execution is carried out using the high-performance AerSimulator backend, enabling efficient and accurate simulation of quantum behavior.
- **Statistical Randomness Tests:** The randomness quality of each RNG output is further evaluated using a suite of statistical tests:
  - *Chi-squared Test:* Assesses the uniformity of bit distribution.
  - *Shannon Entropy:* Measures the average unpredictability or information content in the bit sequence.

  - *Autocorrelation:* Examines the degree of self-similarity between successive bits.
  - *Runs Test:* Evaluates the randomness of bit sequences based on the number and length of consecutive identical bits.
  - *Spectral Test:* Detects periodic patterns or regularity in the sequence using Fourier analysis.

- **compare_rng_methods Function:** A high-level utility function was developed to automate the full testing pipeline. It runs the CHSH Bell test and all statistical evaluations across each RNG type, collects the results, and formats them for analysis.

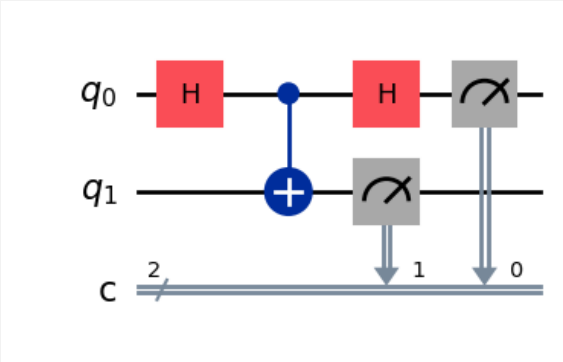
- **Visualization:** Comparative plots and histograms were generated to visually interpret the results. These include CHSH value distributions and statistical test outcomes, allowing for an intuitive comparison of the performance of QRNG, TRNG, and PRNG.

4.3 Implementation (Circuit and Statistical Tests)

Bell Circuit Implementation: Quantum Entanglement

Purpose: Prepare a Bell state ($|\Phi^+\rangle$) to demonstrate quantum entanglement.
Steps:

- Initialize two qubits in the $|0\rangle$ state.
- Apply Hadamard gate to the first qubit.
- Apply CNOT gate with control on the first qubit and target on the second.
- Measure both qubits.




```python
from qiskit import QuantumCircuit, Aer, transpile, assemble
from qiskit.visualization import plot_histogram

# Create a 2-qubit quantum circuit
qc = QuantumCircuit(2, 2)

# Step 1: Apply Hadamard gate to qubit 0
qc.h(0)

# Step 2: Apply CNOT gate (control: qubit 0, target: qubit 1)
qc.cx(0, 1)

# Step 3: Measure both qubits
qc.measure([0, 1], [0, 1])

# Simulate and plot result
sim = Aer.get_backend('qasm_simulator')
compiled = transpile(qc, sim)
qobj = assemble(compiled, shots=1024)
result = sim.run(qobj).result()
counts = result.get_counts()

plot_histogram(counts)
```

The following statistical tests were implemented in Python to evaluate the quality of randomness in sequences generated by QRNG, TRNG, and PRNG. Each test provides a different statistical lens to assess the unpredictability and structure of the generated sequences.

1. Chi-Squared Test: Uniformity

Purpose: Tests whether 0s and 1s occur equally in the sequence.

Formula:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}, \quad \text{Degrees of freedom} = 1$$

Interpretation: A low p-value (<0.05) suggests a biased distribution.

```python
def chi_squared_test(sequence):
    count_0 = sequence.count(0)
    count_1 = sequence.count(1)
    total = len(sequence)
    expected = total / 2
    chi2 = ((count_0 - expected)**2 + (count_1 - expected)**2) /
       expected
    p_value = 1 - chi2.cdf(chi2, df=1)
    return chi2, p_value
```

2. Shannon Entropy: Unpredictability

Purpose: Measures the level of randomness (entropy) in the sequence.

Formula:

$$H = -\sum p_i \log_2(p_i)$$

Ideal Value: Entropy close to 1 indicates high unpredictability.

```python
def shannon_entropy(sequence):
    count_0 = sequence.count(0)
    count_1 = sequence.count(1)
    total = len(sequence)
    p0 = count_0 / total
    p1 = count_1 / total
    entropy = 0
    for p in [p0, p1]:
        if p > 0:
            entropy -= p * log2(p)
    return entropy
```

3. Autocorrelation Test: Self-Similarity

Purpose: Evaluates correlation between adjacent bits.

Ideal Value: Autocorrelation = 0.5 (approx.) for a truly random sequence.

```python
def autocorrelation(sequence):
    shifted = sequence[1:] + [sequence[0]]
    matches = sum(1 for a, b in zip(sequence, shifted) if a == b)
    return matches / len(sequence)
```

4. Runs Test: Oscillations

Purpose: Detects whether bits alternate too frequently or too rarely.

Formula:

$$Z = \frac{R - E(R)}{\sigma}, \quad \text{where } E(R) = \frac{2N-1}{3}, \ \sigma = \sqrt{\frac{16N-29}{90}}$$

Interpretation: A high absolute Z-score indicates non-random behavior.

```python
def runs_test(sequence):
    runs = 1
    for i in range(1, len(sequence)):
        if sequence[i] != sequence[i - 1]:
            runs += 1
    N = len(sequence)
    expected = (2 * N - 1) / 3
    std_dev = sqrt((16 * N - 29) / 90)
    z_score = (runs - expected) / std_dev
    p_value = 2 * (1 - norm.cdf(abs(z_score)))
    return {'runs': runs, 'expected': expected, 'z_score': z_score
        , 'p_value': p_value}
```

5. Spectral Test: Periodicity

Purpose: Uses Fourier Transform to detect hidden patterns or periodicities.
Steps:

- Convert bits: 0 → -1, 1 → +1
- Apply FFT
- Normalize peak height against threshold $\sqrt{2.9957}$ (99% confidence)

```
def spectral_test(sequence):
    N = len(sequence)
    mapped = np.array([1 if bit == 1 else -1 for bit in sequence])
    fft_result = fft(mapped)
    magnitudes = np.abs(fft_result[1:N//2])
    peak_height = np.max(magnitudes) / sqrt(N * log2(N))
    threshold = sqrt(2.9957)
    return {'peak_height': peak_height, 'threshold': threshold, '
       passed': peak_height < threshold}
```

4.4 Results

The following results were obtained from Bell test simulations using three different types of random number generators: a Quantum Random Number Generator (QRNG), a True Random Number Generator (TRNG), and a Pseudo-Random Number Generator (PRNG). Each generator was evaluated using several statistical tests in addition to the CHSH inequality value.

Randomness and CHSH Test Comparison

Metric	QRNG	TRNG	PRNG
CHSH Value	2.0997	2.0343	1.9925
Violation of Classical Bound	Yes	Yes	No
Shannon Entropy	0.9999	1.0000	1.0000
Autocorrelation	0.4985	0.4935	0.4946
Chi-squared p-value	0.5066	0.8744	0.5222
Spectral Test Passed	True	True	True

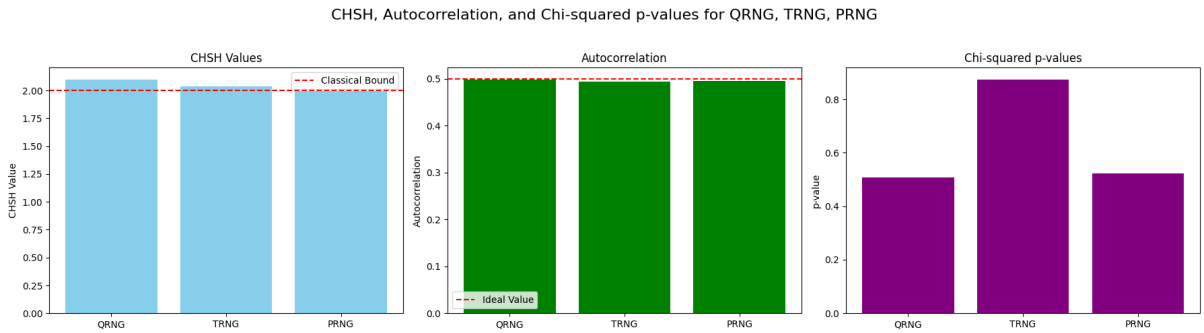


Figure 6: Comparison of randomness and CHSH test metrics for QRNG, TRNG, and PRNG.

Figure 6 shows a graphical comparison of key metrics across the three random number generators. All data have been tabulated and plotted to allow further interpretation and discussion in the following sections.

4.5 Discussion

The results obtained from the jitter-based RNG, QRNG and PRNG (from `secrets` module) confirm that entropy sources rooted in physical noise can yield high-quality random bit-streams. The PRNG data does not violate the CHSH inequality by a slight amount. Although TRNG and QRNG both violate the CHSH inequality, it must be noted that the **QRNG violates the CHSH inequality at the highest value**. All three methods showed near-uniform bit distributions, entropy values approaching 1.0, and minimal autocorrelation—indicating both high unpredictability and statistical independence. However, we were not able to amplify these statistics to show the advantage of QRNGs over TRNGs using a larger sample of size of data due to time constraints as the computations took a considerable amount of time due to which we also were unable to perform the TestU01, Dieharder, and NIST SP 800-22 tests. We did observe that using a larger sample size did actually increase the statistical differences between the three (the given data is for `SAMPLE SIZE = 5000`). Hence, we could conclude in essence that these characteristics make TRNGs significantly more secure and robust than pseudorandom number generators (PRNGs), which rely on deterministic algorithms and are, by design, predictable if their seed is known.

However, even TRNGs depend on classical physical processes that, while noisy and complex, are still fundamentally governed by deterministic or chaotic classical systems. This is where quantum random number generators (QRNGs) offer a critical advantage. We demonstrate this by the importance of QRNGs in running quantum simulation experiments, a Bell test (CHSH game) here.

Quantum Randomness vs Classical Randomness

Unlike PRNGs and TRNGs, QRNGs rely on intrinsic quantum mechanical phenomena, where randomness is not just due to hidden complexity or environmental noise but is fundamentally irreducible. Quantum mechanics allows for outcomes that are genuinely probabilistic — governed by wavefunction collapse rather than hidden variables.

A key example is the Einstein–Podolsky–Rosen (EPR) paradox, which questioned whether quantum mechanics was complete. EPR proposed that there must be hidden variables determining the outcomes of quantum measurements. However, John Bell later formalized this idea into Bell's theorem, which showed that no local hidden variable theory can reproduce the predictions of quantum mechanics. Numerous experiments (e.g., Aspect et al., 1982) have confirmed violations of Bell inequalities, strongly suggesting that measurement outcomes in quantum systems are not predetermined, but truly random.

QRNGs and Quantum Simulations

In the context of quantum simulations — especially those involving quantum cryptography, quantum Monte Carlo methods, or quantum key distribution (QKD) — the quality of randomness is paramount. PRNGs introduce algorithmic predictability and potential bias; TRNGs, though better, are still subject to classical limitations and calibration issues.

QRNGs, by contrast, extract entropy from events like:

- Photon path choices in a beam splitter (which-way experiments),
- Time of detection of spontaneous emission,
- Polarization or spin measurements in entangled particles.

These processes are genuinely unpredictable and can be certified using principles like Bell nonlocality and device-independent randomness. As a result, QRNGs are becoming increasingly favored in quantum simulation tasks, where the fidelity and unpredictability of random numbers directly affect the simulation outcome.

Conclusion of Discussion

In summary, while the TRNGs implemented in this work perform well for many practical applications and are far superior to PRNGs in terms of entropy quality, they remain bounded by classical physics. For cutting-edge quantum simulations and cryptographic protocols, QRNGs offer a physically grounded and theoretically provable source of true randomness — making them the ideal choice where security and physical unpredictability are non-negotiable.

Conclusion

Tasks 1 through 3 involved a progressive exploration of randomness, from classical pseudo-random number generators (PRNGs) to physical true random number generators (TRNGs), and finally to quantum randomness. We began by implementing and analyzing classical methods such as Linear Congruential Generators and AES-based PRNGs, benchmarking their statistical properties and predictability through both entropy analysis and machine learning. In Task 2, we designed three TRNGs leveraging jitter, microphone input, and thermal noise, demonstrating how real-world physical phenomena can serve as entropy sources. Task 3 allowed us to compare these TRNGs with an industry-grade Quantum RNG (QRNG), highlighting key differences in entropy, bitrate, and predictability. This layered approach revealed the limitations of classical randomness and set the stage for understanding the advantages of quantum-based methods.

In Task 4, we consolidated our understanding by implementing a rigorous comparison framework incorporating both statistical tests and a Bell test (CHSH inequality) simulation. We developed a unified interface for QRNG, TRNG, and PRNG sources and evaluated them not just on statistical metrics but also on their capacity to support quantum protocols. The results clearly showed that while TRNGs offer better unpredictability than PRNGs, only QRNGs produced violations of the CHSH bound consistent with quantum non-locality. This task emphasized the unique value of quantum randomness, particularly for secure and high-integrity applications in quantum computing and cryptography.

References

[1] Realization of Bell's theorem certified quantum random number generation using cloud quantum computers.
https://github.com/Dorahacksglobal/Quantum-Randomness-Generator
https://research.dorahacks.io/2024/04/14/bell-theorem-certified-qrng-using-cloud-quantum-computers/

[2] Implementation of Advanced Encryption Standard (AES) encryption.
https://github.com/burakozpoyraz/Advanced-Encryption-Standard/tree/master