Beyond the Seed: Randomness Redefined*

Arul Rhik Mazumder[1], Tyler Yang[1], Jean-Sébastien Pegon[2], and Bruno Huttner[2]

[1]Carnegie Mellon Quantum Computing Club
[2]ID Quantique

May 2025

Abstract

This challenge explores the fundamentals of pseudo-random random number generators (PRNG), true (TRNG), and quantum (QRNG), with an emphasis on their implementation, entropy characteristics, and practical limitations. The tasks will investigate the trade-offs between seed length, sample size, and entropy decay in PRNGs, the physical unpredictability of TRNGs, and the measurement-independent randomness offered by QRNGs. Special focus is given to benchmarking entropy across data scales, comparing correlation breakdown across methods, and identifying tasks where quantum randomness offers a measurable advantage.

Before You Begin:

This challenge consists of 4 tasks. The first three are exploratory tasks, meant to familiarize you with the importance of random number generation and the programming skills required to interface with IDQuantique's QRNG-as-a-service (QRNGaaS) API.

Task 4 is open-ended, allowing you to apply your skills in an innovative way using QRNGaaS. It is the most important task in the set. However, we strongly encourage you to complete Tasks 1-3, especially if you're not yet familiar with random number generation techniques or IDQuantique's QRNGaaS. In the event that we receive multiple high-quality submissions for Task 4 and are unable to determine a clear winner, we will first review the completion of Tasks 1-3, prioritizing the non-bonus tasks before considering the bonus ones.

Python code templates are provided for Tasks 1-3. The goal of these tasks is to familiarize you with different methods of random number generation, as well as explain the differences between PRNGs, classical TRNGs, and QRNGs.

We also want to get you thinking about entropy as a computational resource and how classical sources of entropy could be disrupted or compromised. We introduce the story of the Debian OpenSSL bug in the **Introduction** to show the effects of low-quality randomness. We also give several other examples of random number generation is critical for other computational processes. Hopefully, these provide some inspiration for your application of QRNG in Task 4.

And one last thing: the introduction and mathematical background may appear daunting at first. We have tried to combine mathematical rigor with intuition; if you ever get stuck, there is a ***Meaning*** section below each mathematical formula.

Please note: In the event of any discrepancies between this writeup and the challenge GitHub, this writeup takes precedence. The GitHub content—including descriptions and template code—is intended solely as a starting point and may be disregarded. Submissions will be evaluated based on their alignment with the guidelines provided in this writeup.

*Challenge GitHub repository: https://github.com/88Mangos/CMU-x-IDQuantique-Challenge

1 Introduction

Random numbers are the invisible backbone of modern computing. From simulating physical systems to training machine learning models, from generating game worlds to securing encrypted messages, they enable unpredictability and statistical variation. Random Number Generators (RNGs) are the tools that produce these numbers. There are three main types: Pseudo-Random Number Generators (PRNGs), which are fast, deterministic algorithms; True Random Number Generators (TRNGs), which extract entropy from physical processes like thermal or electrical noise; and Quantum Random Number Generators (QRNGs), which harness quantum indeterminacy to generate fundamentally unpredictable values.

But not all randomness is created equal. A good RNG should produce values that are uniformly distributed and statistically independent. Most importantly, in certain contexts like cryptography, an RNG must be unpredictable; even partial knowledge of its output or internal state should not help an attacker deduce anything about future values. For example, in cryptographic applications like generating a 256-bit key for AES encryption, a weak RNG could compromise the entire security system. In 2006, a bug in Debian's OpenSSL package reduced the entropy of its PRNG, enabling attackers to reproduce private keys and break into secure systems [1]. To prevent such vulnerabilities, RNGs are evaluated using statistical test suites like *TestU01*, *Dieharder*, and *NIST SP 800-22*, which assess the quality, uniformity, and entropy of output bitstreams [2] [3].

Beyond cryptography, RNGs are critical in a wide range of applications:

Simulation Bias: In simulations, especially those using Monte Carlo methods, random numbers are crucial for sampling outcomes and modeling complex systems. Poor-quality randomness can introduce biases, distorting statistical results, and cause uncertainties to be underestimated. For example, a faulty RNG used in a physics simulation could yield incorrect predictions about particle behavior, leading to unrealistic results that could mislead researchers or engineers.

Pattern Emergence in Games and Gambling: In gaming and gambling software, predictable PRNGs undermine fairness and trust. Predictable randomness allows players or attackers to exploit patterns, leading to potential financial losses and breaches of fairness. A famous case occurred in online gambling, where flaws in the RNG used for slot machines were reverse-engineered, allowing players to predict outcomes and win disproportionately, costing casinos millions of dollars [4].

Machine Learning and Data Sampling Failures: In machine learning, randomness plays an essential role in data shuffling and initialization. Poor randomness in dataset shuffling can lead to data leakage or non-independent and identically distributed (i.i.d.) sampling, which compromises the generalization of models. Similarly, random initialization of neural network weights can cause biased model performance or poor convergence. For instance, a weak RNG used for shuffling training and testing datasets could result in overfitting, where the model performs well on training data but poorly on unseen data.

In all these cases, ensuring high-quality randomness is crucial to the integrity of simulations, fairness in games, and robustness in machine learning models. In this project, we explore the design and implications of PRNGs, TRNGs, and QRNGs. We'll benchmark their performance, evaluate their entropy quality, and test their robustness through real-world applications like key generation. Ultimately, our goal is to understand how different sources of randomness stack up, and how they can make or break the systems we depend on.

2 Mathematical and Statistical Background

What Makes a Good RNG?

A random number generator (RNG)—whether pseudo-random (PRNG), true-random (TRNG), or quantum (QRNG)—should produce sequences of numbers that behave like true random samples. That means they should satisfy the following key properties:

- **Unpredictability**:

$$P(X_{n+1} \mid X_1, \ldots, X_n) = P(X_{n+1})$$

Variables: X_i is the i-th random value in the sequence.
Meaning: The probability of the next number, X_{n+1}, should not change even if you know all the previous numbers. This means the future values are not predictable from the past.

- **Correct Distribution**:

$$P(X_i \leq x) = F_X(x)$$

Variables: X_i is a random value; $F_X(x)$ is the target cumulative distribution function (CDF).
Meaning: The numbers should follow a certain distribution (like uniform or normal). For example, in a uniform distribution between 0 and 1, the chance that $X_i \leq 0.5$ should be exactly 0.5.

- **Independence and Identical Distribution (i.i.d.)**:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i)$$

Variables: $P(X_1, \ldots, X_n)$ is the joint probability of seeing the full sequence; $P(X_i)$ is the individual probability of each value.
Meaning: Each value should come from the same distribution (identical), and knowing one value shouldn't help you guess another (independent).

For practical use, we also want:

Entropy Measures

Entropy tells us how "random" or "uncertain" a variable is.

- **Shannon Entropy**:

$$H(X) = -\sum_{x} P_X(x) \log P_X(x)$$

Variables: $P_X(x)$ is the probability that X equals x.
Meaning: Measures the average amount of uncertainty or information you gain when observing the outcome of a random variable X. Higher entropy means more unpredictability and randomness in the values that X can take.

- **Min-Entropy**:

$$H_{\min}(X) = -\log \max_{x} P_X(x)$$

Variables: $\max_x P_X(x)$ is the highest probability of any single value.
Meaning: This is a conservative measure of randomness, focusing on the most predictable outcome. If one value is way more likely than others, min-entropy is low.

- **Mutual Information**:

$$I(X;Y) = H(X) + H(Y) - H(X,Y)$$

Variables: $H(X)$ and $H(Y)$ are the individual entropies; $H(X,Y)$ is the joint entropy.
Meaning: This measures how much information X and Y share. If they're totally unrelated, $I(X;Y) = 0$.

Statistical Testing of Randomness

We use tests to see if a sequence "looks" random. Here are a few:

- **Chi-Square Test**:

$$\chi^2 = \sum_i \frac{(O_i - E_i)^2}{E_i}$$

Variables: O_i = observed frequency of outcome i, E_i is the expected frequency under a target distribution.
Meaning: This test checks if outcomes appear as often as they should. Big differences mean the data might not be random.

- **Kolmogorov–Smirnov (K–S) Test**:

$$D_n = \sup_x |F_n(x) - F(x)|$$

Variables: $F_n(x)$ = empirical cumulative distribution function (ECDF) of the sample, $F(x)$ = cumulative distribution function (CDF) of the expected distribution (e.g., uniform), D_n = maximum distance between the two.
Meaning: Measures how well the sample distribution matches the expected distribution. A large D_n suggests the data deviates from the target distribution (e.g., uniform), indicating potential non-randomness.

- **Autocorrelation**:

$$\rho_k = \frac{\mathrm{Cov}(X_t, X_{t+k})}{\mathrm{Var}(X)} = \frac{\sum_{t=1}^{N-k}(x_t - \bar{x})(x_{t+k} - \bar{x})}{\sum_{t=1}^{N}(x_t - \bar{x})^2}$$

Variables: ρ_k = autocorrelation at lag k; $\mathrm{Cov}(X_t, X_{t+k})$ = covariance between values at time t and $t+k$; $\mathrm{Var}(X)$ = variance of the series.
Meaning: This measures the correlation between values k steps apart. In a truly random sequence, values at different lags should not show significant correlation. A high autocorrelation at a specific lag indicates some dependency between values at that lag.

- **Dieharder & NIST SP800-22**: These are suites of many randomness tests (like those above) applied in a rigorous way. They check for patterns, frequency issues, dependence, and other flaws in RNGs.

3 Challenge

Task 1: Playing with Pseudo-Randomness

In this task, you'll investigate how pseudo-random number generators (PRNGs) work in practice, starting from simple models and progressing to cryptographically secure methods. A PRNG is a deterministic algorithm designed to generate sequences that mimic true randomness. While the output may appear random, it is fully determined by an initial value called the *seed*. All PRNGs consist of three main components:

- **Seed (Initial State):** A fixed-size input—typically an integer—that determines the entire output sequence. The same seed always produces the same sequence, which makes PRNGs reproducible.
- **State Transition Function:** A rule that updates the internal state in each step, often using a recurrence relation or cipher operation. This function is deterministic, even if the output appears random.
- **Output Function:** A function that converts the internal state into usable outputs—e.g., numbers in $[0, 1)$ or bitstrings.

Despite their determinism, good PRNGs can produce sequences that pass many statistical tests for randomness. Two important properties of PRNGs are:

- **Period:** The number of steps before the sequence repeats. A high-quality PRNG should have a very long period.
- **Reproducibility:** The ability to regenerate the same sequence by reusing the same seed. This is essential for debugging, simulations, and scientific reproducibility.

In this challenge, you'll begin with a simple PRNG—the **Linear Congruential Generator (LCG)**—and later explore a more realistic method based on the **Advanced Encryption Standard (AES)**.

Part A: Implement and Benchmark an LCG

The LCG is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m$$

where:

- X_0 is the seed,
- a is the multiplier,
- c is the increment,
- m is the modulus,
- $\{X_n\}$ is the resulting pseudo-random sequence.

To ensure a full period (i.e., the sequence covers all values before repeating), choose parameters that satisfy the **Hull–Dobell Theorem**:

1. $\gcd(c, m) = 1$,
2. $a - 1$ is divisible by all prime factors of m,
3. If m is divisible by 4, then $a - 1$ must also be divisible by 4.

Tasks:

1. Implement an LCG using valid parameters satisfying the conditions above.
2. Generate at least 10^6 outputs.
3. Visualize the results:
   - Plot a histogram of the outputs to assess uniformity.
   - Plot the entropy (e.g., Shannon or Min entropy) every 1,000 samples.

Note: LCGs are not secure for cryptographic use. They are predictable and easy to reverse-engineer. We use them here to introduce PRNG concepts.

Part B: Implement and Benchmark AES-Based PRNGs

Next, you'll implement a more secure PRNG based on the **Advanced Encryption Standard (AES)**. AES is a block cipher that encrypts 128-bit blocks using a 128-bit key. It consists of several rounds of transformation steps designed to thoroughly mix the input data.

Key transformations include:

- **SubBytes:** Byte-wise substitution using a non-linear S-box.
- **ShiftRows:** Row-wise shifting to spread bytes across columns.
- **MixColumns:** Mixing column bytes using finite field arithmetic.
- **AddRoundKey:** XOR with round-specific subkeys derived from the original key.

Implementing AES from scratch is non-trivial, so we recommend using a reference implementation[1].

This part focuses on how the **entropy of the seed (key)** affects the randomness quality of the generated sequences.

Tasks:

1. Modify your AES-based PRNG to support variable-length seeds (e.g., 8, 16, 32, 64, and 128 bits).
   Since AES requires a 128-bit key, shorter seeds must be padded or extended to fit this size. To keep things simple and avoid introducing additional entropy, you may choose one consistent method, such as:
   - Padding the seed with a fixed bit pattern (e.g., all 0s or all 1s),
   - Repeating the seed as many times as needed to reach 128 bits (e.g., repeat a 16-bit seed 8 times),
   - Or another natural method of your choice, clearly explained in your comments.
2. For at least three seed lengths, generate sequences of increasing length (from 10^5 to 10^7).
3. For each combination of seed and sequence length:
   - Plot entropy over iterations.
   - Plot autocorrelation as a function of lag k.
   - Apply statistical tests (Chi-squared or Kolmogorov–Smirnov) to assess deviation from a uniform distribution.

Expectation: With shorter seeds, the output should begin to lose randomness in longer sequences. A 128-bit seed should maintain randomness much longer, modeling how limited initial entropy spreads across the output.

[1] Here is one such AES implementation on GitHub

(BONUS) Part C: Breaking the Illusion of Randomness

Train a simple model (e.g., a neural network or decision tree) to predict the next output in sequences from your PRNGs. Start with your LCG and AES PRNGs.

Then, try the same with one or more of the following more advanced PRNGs:

- XOR-Shift Generator [5]
- Permuted Congruential Generator (PCG) [6]
- SplitMix64 [7]

Your model should succeed in predicting values from the LCG, but fail on AES and modern PRNGs—demonstrating their improved unpredictability.

Task 2: Tinkering with True Randomness

In Unix/Unix-like operating systems, you might be familiar with the special files `/dev/random` and `/dev/urandom`, which are used to generate random bits. These devices rely on **Pseudo-random Number Generators (PRNGs)**, but they seed the randomness from environmental noise collected from various system sources, such as thermal noise in hardware or device driver events. While these sources are physical, the randomness generated by the PRNG algorithm itself is deterministic, meaning that once the seed is known, the generated sequence can be predicted. Thus, while the initial entropy for these systems comes from true physical randomness, the randomness produced by the underlying algorithm does not qualify as "true" randomness.

This brings us to **True Random Number Generators (TRNGs)**, which generate random numbers based entirely on physical processes. Unlike PRNGs, which follow deterministic algorithms, TRNGs rely on unpredictable, chaotic physical phenomena to generate entropy. These processes might include **thermal noise** (the random fluctuations of electrical charge in resistive components), **jitter** (small, random timing variations in system processes), or even **ambient noise** from microphones or other environmental factors.

For example, the entropy source for `/dev/random` involves physical phenomena like thermal fluctuations in hardware components. While the seed is sourced from true randomness, the subsequent random numbers are generated via a deterministic PRNG. A **TRNG**, on the other hand, would directly convert these physical fluctuations into random numbers without applying an algorithm that could potentially introduce patterns or biases.

While PRNGs are typically faster and more efficient, generating large sequences of random numbers quickly, they rely on deterministic algorithms and can be predicted if the seed is known. This makes PRNGs well-suited for applications requiring high throughput but where security and unpredictability are less critical, such as simulations or non-secure random sampling. TRNGs, however, provide higher-quality randomness because they draw from physical processes that are inherently unpredictable. This makes them ideal for secure cryptographic applications where predictability could have dire consequences, such as key generation or digital signatures. The trade-off is that TRNGs tend to be slower and often require specialized hardware, making them more expensive and complex to implement in comparison to PRNGs.

In this task, you'll explore the physical randomness that underpins these systems. By generating random numbers based on natural processes like jitter, sound, or thermal fluctuations, you'll gain insight into the foundations of true randomness before the advent of **Quantum Random Number Generators (QRNGs)**, which take randomness to a new level by leveraging the unpredictable nature of quantum mechanics.

Part A: Jitter-bug!

In this part of the challenge, you will create a true random number generator using the natural timing variations (jitter) in your computer's processes.

Part B: What's That Sound?

In this part of the challenge, you will create a true random number generator using the ambient noise from your system microphone.

Part C: Things Are Heating Up!

In this part of the challenge, you will create a true random number generator using thermal noise from your system sensors.

Task 3: Quest for Quantum Randomness

Although TRNGs are significantly harder to predict than PRNGs—since they generate data from chaotic physical systems—they are not truly unpredictable. These systems, despite their complexity, are governed by classical physics and thus have deterministic underlying behavior. What appears as randomness is merely a reflection of our ignorance about the system's full state. In fact, Kolmogorov complexity theory shows it's undecidable to prove whether any sequence is truly random [8]. Classical TRNGs can therefore never guarantee absolute unpredictability: in theory, an adversary with enough scientific insight and computational power could reconstruct the entropy source through passive monitoring, malicious modifications, or signal injection.

Quantum Random Number Generators (QRNGs) address these limitations by relying on quantum entropy sources. Unlike classical systems, quantum systems are inherently probabilistic—measurement outcomes are truly random and cannot be predicted, only described by probability amplitudes. QRNGs exploit this fundamental randomness by measuring properties of quantum states, such as photon polarization or arrival times. A common setup sends individual photons through a beam splitter, where each has a 50/50 chance of being transmitted or reflected.

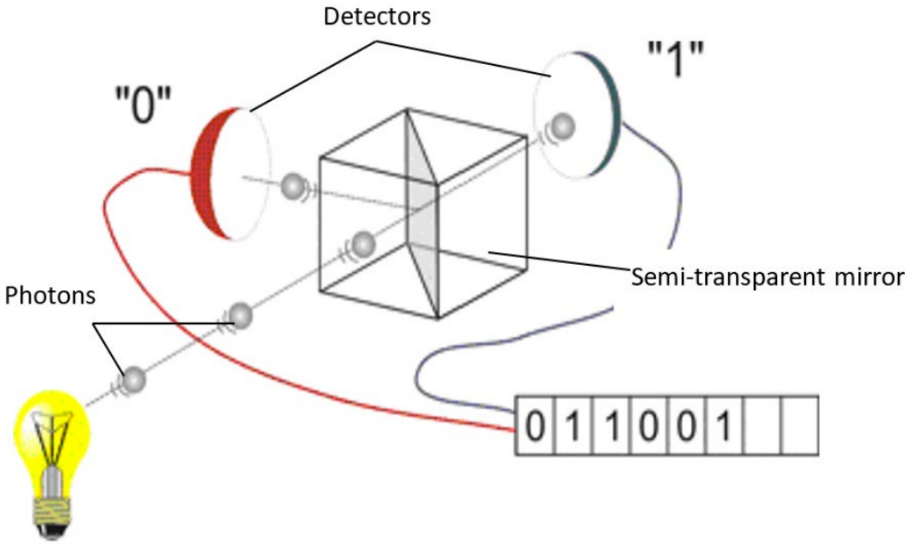

Figure 1: Experiment with laser, semi-transparent mirror and single photon detector [9]

Importantly, this randomness is not due to unknown variables or experimental imprecision: if we send identical photons into the same setup under identical conditions, there is still no hidden mechanism or classical cause that determines the outcome. The photon remains in a superposition until measured, and the act of detection itself is what irreducibly "creates" the outcome—reflected or transmitted. Because the outcome is not determined by any prior state of the system, the resulting bits are fundamentally random. No classical noise model or algorithm can replicate this, in fact one that could would would violate Bell's Theorem [10], making QRNGs uniquely suited for random number generation applications requiring provable unpredictability.

Moreover, QRNGs are less vulnerable to environmental interference than classical TRNG. For example, thermal-noise-based TRNGs may malfunction in extreme temperatures, which is problematic for devices like smartphones. By contrast, the QRNG chip in the Samsung Galaxy Quantum 5, developed by IDQuantique, remains stable across wide temperature ranges [11]. Similarly, classical TRNGs can be disrupted by electromagnetic pulses (EMPs)—a concept popularized in games like *Need For Speed*. QRNGs, however, are much more resilient to such attacks thanks to their fundamentally different architecture [12].

Fun fact: in 2001, IDQuantique became the first company to commercialize a quantum random number generator. In this task, you'll use their QRNG-as-a-service API to generate real quantum randomness!

Part A: Truly Different

In this part of the challenge, you will become familiar with IDQuantique's QRNG-as-a-service API[2]. Your task is to compare the QRNG's outputs with the TRNGs from parts A, B, and C of Task 2. We primarily want you to measure and compare two metrics between TRNGs and QRNGs:

1. **Entropy Analysis**: As in **Task 1 Part B**, analyze how entropy evolves as you increase the number of generated outputs. Use either **Shannon entropy** or **Min-entropy** to quantify the randomness. Plot entropy as a function of sample size to observe how quickly your RNG approaches ideal randomness or where it begins to degrade.

2. **Bitrate Evaluation**: Measure the efficiency of your RNG by calculating its **bitrate**—the number of random bits it produces per second. Bitrate is critical: even a high-entropy generator is impractical if it's too slow for real-time use cases such as cryptographic protocols, secure communications, Monte Carlo simulations, or high-performance computing. To compute the bitrate of your RNG, follow these steps:

   (a) Measure the time required to generate N random values.

   (b) Estimate the number of random bits per value (e.g., assume 32 bits for a 32-bit integer).

   (c) Multiply the total number of values by the bits per value to get the total number of random bits.

   (d) Divide the total bits by the elapsed time to compute the bitrate (in bits per second).

(BONUS) Part B: Easy as Pi!

We aren't requiring this task because we would like you to focus on your innovative application of true randomness from QRNGs.

But if you're curious, this task will show you one example of a randomized algorithm that benefits from QRNG's true randomness. In this part of the challenge, you will implement a Monte Carlo algorithm that approximates π.

Below is the pseudocode for the algorithm: [13]

```
given radius r
create square of side length 2r
consider the inscribed circle of radius r
generate a large number of random points
```

$$\frac{area(circle)}{area(square)} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \approx \frac{\text{no. of points generated that fall inside the circle}}{\text{no. of points generated that fall inside the square}}$$

[2]You should use the following API Key: `aTo4BKRvnc49uRWDk034zaua87vGRXKk9TMLdfkI`
Note that the key ID is `xuh04u9rdl` and it will expire at the end of day on May 25th

Task 4: Demonstrating Quantum Advantage

Task 4 challenges you to apply your understanding of random number generation in a creative and technically rigorous way using IDQuantique's QRNG-as-a-Service (QRNGaaS). This task represents the culmination of the previous exercises and is your opportunity to design and implement an innovative application that highlights the unique advantages of quantum randomness.

Building on the lessons of Tasks 1–3, Task 4 asks you to think critically about where conventional PRNGs and classical TRNGs might fall short—whether due to determinism, bias, or environmental vulnerability—and to explore how quantum randomness can offer stronger guarantees of unpredictability and entropy. Your application could be in cryptography, simulation, secure communication, gaming, machine learning, or any domain where true randomness plays a vital role.

The goal is not just to demonstrate technical functionality, but to design something meaningful: an application where the use of QRNG is essential, not just ornamental. **Why does quantum randomness matter here? What would go wrong with lower-quality randomness? How does your system take advantage of the statistical and physical properties of QRNGs?** Ideally, your solution will not only use the QRNGaaS API but also provide benchmarking or analysis to justify its use over traditional methods.

If you're still looking for inspiration, think back to the introduction: the Debian OpenSSL bug, simulation bias, unfair gaming systems, and faulty model training pipelines—all show how things can go wrong when randomness fails. This task is your opportunity to show how quantum randomness can help them go right. Some directions that you could explore (not including ones we mentioned earlier) are:

- An oracle mechanism for smart contracts (e.g., NFT minting, lotteries) that uses QRNG to prevent front-running or manipulation.
- A scientific experiment (e.g., Bell test) simulation where QRNG is used to ensure loophole-free measurements.
- A QRNG-seeded secure boot process or entropy pool generator for embedded systems with limited resources.
- A secure multi-party computation (SMPC) system that depends on unbiased shared randomness via QRNG.
- A randomized compiler or ASLR (Address Space Layout Randomization) mechanism that improves unpredictability using QRNG.

Of course, also feel free to choose an topic that was not mentioned above. Whatever direction you choose, be sure to clearly articulate your design, implementation, and justification. Submissions will be judged primarily on technical soundness and originality, but we will also consider clarity of explanation, relevance to the topic of random number generation, and the depth of engagement with quantum randomness as a computational resource.

Justifying the quantum advantage in random number generation is a nontrivial task—one that even researchers at leading companies like ID Quantique continue to investigate. So, if you find it challenging, you're not alone. Most arguments for QRNGs are grounded in theoretical physics, relying on the fundamental unpredictability of quantum measurements. However, if you want to strengthen your case in practice, consider empirical validation using statistical test suites such as *TestU01*, *Dieharder*, and *NIST SP 800-22*. Alternatively, you can design and demonstrate a protocol or cryptographic scheme where a QRNG offers measurable resilience or security benefits over traditional PRNGs or TRNGs.

References

[1] Constantin, L. *Debian's OpenSSL bug still haunts the internet.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166. (2008).

[2] Information Technology Laboratory. *Advanced Encryption Standard (AES).* Tech. rep. FIPS 197. National Institute of Standards and Technology (2001).

[3] L'Ecuyer, P. and Simard, R. "TestU01: A C library for empirical testing of random number generators." *ACM Transactions on Mathematical Software (TOMS)* **33** (2007), 1–40.

[4] Geeting, D. B. "Russians Engineer a Brilliant Slot Machine Cheat—And Casinos Have No Fix." *WIRED* (2017). Accessed: 2025-05-07.

[5] Marsaglia, G. *Xorshift RNGs.* Online: http://www.jstatsoft.org/v08/i14/paper. Posted to comp.lang.c Usenet group. (2003).

[6] O'Neill, M. E. "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation." In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014).

[7] Vigna, S. *An experimental exploration of Marsaglia's xorshift generators, scrambled.* arXiv preprint arXiv:1402.6246. (2015).

[8] Li, M. and Vitányi, P. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer (2008).

[9] ID Quantique. *Quantum versus Classical Random Number Generators.* Tech. rep. Accessed: 2025-05-08: ID Quantique SA (2023). URL: https://marketing.idquantique.com/acton/attachment/11868/f-64900ef6-6e7e-4b4c-a9f9-c912a2cfde59/1/-/-/-/-/Classical%20RNG%20Vs%20QRNG_White%20Paper.pdf.

[10] Bell, J. S. "On the Einstein Podolsky Rosen Paradox." *Physics Physique Fizika* **1** (1964), 195–200.

[11] Maundrill, M. *SK Telecom unveils the Samsung Galaxy Quantum 5.* (2024).

[12] IDQuantique. *Random Number Generation White Paper.* (2023).

[13] GeeksforGeeks. *Estimating the value of pi using Monte Carlo.* (2022).