Quantum Random Number Generator for One Time Password Generation

Vito Cucinelli, Alexandre Gautier, Gian Luca Schiano, Mirko Ciardo

April 2025

Abstract

In our project, we developed a quantum-circuit-based random number generator using Qiskit and the IBM Quantum platform. The objective is to demonstrate if quantum technology can be concretely applied to strengthen OTP generation, providing higher security guarantees compared to traditional methods based on pseudo-random algorithms. Moreover, our solution is entirely free and accessible to everyone, leveraging publicly available quantum computing resources. This approach promotes open research and allows a broader audience to experiment with quantum-enhanced security technologies without economic barriers.

1 Introduction to Random Number Generators (RNGs)

Random Number Generator is a computational or physical device designed to produce a sequence of numbers or symbols that, ideally, lacks any discernible pattern. Think of it as a system whose output is unpredictable. The core components of any RNG typically include a 'seed', which is an initial value used to kickstart the number generation process. Then there's the 'algorithm' or method itself, dictating precisely how subsequent numbers are derived. And finally, we have the 'output' – the sequence of random numbers that the generator produces.

The applications of RNGs are incredibly diverse and far-reaching. In the realm of cryptography, they are indispensable for creating secure keys and ensuring data protection. In the gaming industry, RNGs are what make experiences unpredictable and fair, from shuffling a deck of virtual cards to determining event outcomes. For simulations in science and engineering, they allow us to model complex systems with inherent randomness, like weather patterns or stock market fluctuations. And in statistical sampling, RNGs are crucial for selecting representative samples from larger populations, ensuring that research findings are unbiased and reliable. In essence, wherever there's a need for unpredictability or a fair, unbiased selection, RNGs play a vital role.

1.1 Comparison between PRNG and TRNG

It's important to distinguish between the two primary categories: Pseudo-Random Number Generators (PRNGs) and True Random Number Generators (TRNGs). While both aim to produce random sequences, their underlying principles and characteristics differ significantly. Pseudo-Random Number Generators (PRNGs) are algorithmic in nature, this means they use a mathematical formula or a pre-calculated list to produce a sequence of numbers that appears random. The sequence is initiated by a 'seed' value, and if you use the same seed, you will get the exact same sequence of numbers every single time. This makes PRNGs deterministic and reproducible. While this reproducibility can be useful for testing and debugging, it also means they are not truly random. PRNGs are generally fast to generate numbers and are relatively easy to implement in software, as they don't require specialized hardware.

On the other hand, we have True Random Number Generators (TRNGs). Unlike their pseudo-random counterparts, TRNGs derive their randomness from physical phenomena that are inherently unpredictable. These phenomena can include things like atmospheric noise, radioactive decay, or quantum mechanical processes. Because they are based on these unpredictable physical events, the sequences generated by

TRNGs are non-deterministic and non-reproducible. You cannot get the same sequence twice, even if you try to control the initial conditions. This makes them genuinely random. However, TRNGs are often slower in generating numbers compared to PRNGs, and they typically require specialized hardware to capture and process the physical phenomena, making them more complex to implement. The key takeaway is that TRNGs offer a higher quality of randomness, which is critical for applications demanding the utmost unpredictability, such as high-security cryptography.

1.2 Quantum Random Number Generators (QRNGs)

Having distinguished between Pseudo-Random Number Generators (PRNGs) and True Random Number Generators (TRNGs), we now turn our attention to TRNG: the Quantum Random Number Generator (QRNG). QRNGs exploit the principles of quantum mechanics to produce genuinely unpredictable random numbers.

One of the key quantum mechanical concepts that QRNGs exploit is *superposition*. In classical physics, a bit can either be a 0 or a 1. However, in the quantum world, a quantum bit, or *qubit*, can exist in a superposition of states. This means it can be both 0 and 1 simultaneously, with certain probabilities for each state. Mathematically, this is represented as:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$$

Here, $|0\rangle$ (read as "ket 0") and $|1\rangle$ (read as "ket 1") represent the two fundamental quantum states, analogous to the classical 0 and 1. The coefficients, in this case $\frac{1}{\sqrt{2}}$, determine the probability of the qubit collapsing into one of these states upon measurement.

When we measure a qubit in such a superposition, its quantum state collapses to one of the definite classical states — either $|0\rangle$ or $|1\rangle$. For the specific superposition state $|\psi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$, there is a 50% probability of measuring a 0 and a 50% probability of measuring a 1. Crucially, according to the principles of quantum mechanics, the outcome of an individual measurement is fundamentally probabilistic and cannot be predicted with certainty beforehand.

It is this inherent, irreducible randomness at the quantum level that makes QRNGs a source of true random numbers. Each measurement effectively provides a random bit, and by performing many such measurements, we can generate a sequence of truly random numbers. This is why QRNGs are considered the gold standard for True Random Number Generation.

1.3 IBM Quantum Platform and Qiskit

To work with these quantum phenomena and develop applications like our QRNG, we need specialized tools and platforms. Two fundamental resources in this domain, which are relevant to our project, are the IBM Quantum Platform and Qiskit.

The **IBM Quantum Platform** is an initiative by IBM that provides access to actual quantum computers via the cloud. This is a remarkable resource because it allows researchers, developers, and enthusiasts like us to design and run quantum experiments on real quantum hardware, not just simulations. This hands-on access is invaluable for understanding the nuances of quantum computation and for testing the practical performance of quantum algorithms, including those used for random number generation. The platform typically offers a graphical interface called the IBM Quantum Composer for visually building quantum circuits, as well as programmatic access.

Complementing the hardware access is **Qiskit** (pronounced "kiss-kit"). Qiskit is an open-source software development kit (SDK) for working with quantum computers at the level of pulses, circuits, and application modules. It's primarily Python-based, making it accessible to a broad community of developers. Qiskit provides tools to create quantum circuits, simulate them on classical computers, and, importantly, run them on the real quantum devices available through the IBM Quantum Platform or other providers. It abstracts away much of the low-level complexity, allowing us to focus on designing and analyzing quantum algorithms. For our project, Qiskit is the framework we'll be using to implement and test our quantum random number generation circuits, both in simulation and potentially on IBM's quantum hardware. Together, the IBM

Quantum Platform and Qiskit provide a powerful ecosystem for exploring and harnessing the potential of quantum computing.

1.4 Application of QRNGs to Cybersecurity: One Time Passwords

True randomness is crucial for cryptographic applications: the generation of secure keys, encryption of communications, and user authentication all depend critically on the quality and unpredictability of random numbers.

One concrete and widely used application of random numbers is the generation of **One Time Passwords** (OTP). OTPs are temporary codes that are valid for only a single authentication session or for a very short period of time. Their main purpose is to strengthen the security of user authentication, minimizing the risks associated with static passwords.

Traditional passwords, even when complex, suffer from several vulnerabilities:

- They can be stolen through phishing attacks, database leaks, key-loggers, or simple guesswork.
- Once a static password is compromised, an attacker can reuse it indefinitely.
- Users often reuse passwords across multiple services, multiplying the risk of disclosure.

In contrast, OTPs significantly reduce these risks because:

- Even if an attacker intercepts the OTP, it becomes useless after a few seconds.
- OTPs are often combined with other factors (something the user knows or owns), enhancing overall security (two-factor authentication, or 2FA).

However, they are still vulnerable to Man in the Middle attacks, which means that the verifier has to be authenticated and in order to be secure, they have to contain random characters which makes the insertion difficult for the user.

Even if the different password can be written on a sheet of paper and shared between the user and the verifier, we usually use automatically ones computed by ad-hoc applications on smartphone, tablets or laptops.

There is two types of OTPs: either they change according to the time, or according to some events such as pressing a button. For the time-based OTP (TOTP), for each period of time, usually 30-60s, corresponds a password. The client has to perform local computation and clock synchronization is needed. To prevent the invalidation of the OTP if sent at the moment of changing frame, the verifier has to check for the previous and the next window. The other type of OTP, called Event-based OTP (HOTP) uses a monotonic integer counter to keep track of the number of occured events : for each login, the user presses a button and the next OTP is displayed. In this case, if an attacker accesses the devices used to display the different OTPs, the security is compromised.

But in any case, the OTPs generators need what we call a *seed* to instantiate. From this seed, each OTP is generated: that means that if an attacker had access to this seed, he could be able to reconstruct all the passwords and the generator should be reinitialized in order to stop the attacks. That's why they require a reliable source of randomness to produce unpredictable, secure seeds. If the randomness behind an OTP generation process were to be compromised, attackers could potentially predict valid codes and gain unauthorized access.

That is why we want to use QRNGs to generate the seeds for the OTPs generators. Since we focus on the generation of the seed in this project, we assume secure the storage of the seed or the devices in charge of displaying the current OTPs.

2 Quantum Environment

2.1 Quantum randomness

To understand why a quantum computer is well-suited to reach true randomness, we have to understand how its elementary units of information work. To build a qubit, we usually use quantum particles, that are the smallest known building blocks of the physical universe. Thanks to the particularity of the quantum world, these particles can be represented by a wavefunction which is a superposition between two states that we usually define as $|0\rangle$ and $|1\rangle$. From this waveform, we can derive the probability of the particle to be in one of the possible states, however it is impossible to know exactly these probabilities.

These probabilities will collapse, along with the whole wavefunction, when we measure the state of the qubit : we force the qubit in a definite state. Of course, the final value of the qubit will follow the probabilities we mentioned before; however, this process can bereally random and unpredictable, due to the nature of quantum mechanics. The measurement is crucial if we want to transform the quantum information into classical ones. Thanks to this randomness, we might be able to construct true random numbers.

2.2 IBM Quantum Platform

Our project made use of the IBM Quantum platform to access real quantum computers via the cloud, this is an important service because allow everyone to produce quantum circuits. IBM Quantum Composer is used to grapichally design a quantum circuit, simulating the behavior and analyze the resulting state probabilities. IBM provides also a python library called Qiskit to access from code IBM Quantum Platform.
Qiskit is an open-source software development kit (SDK) for programming quantum computers, primarily designed for IBM quantum hardware. It provides tools for creating, manipulating, and running quantum circuits and programs, enabling developers to explore and utilize the potential of quantum computing.

Through Qiskit Runtime, a part of Qiskit, we sent computation jobs to the IBM Quantum platform, specifying for each a 16-qubit circuit and the number of shots (runs) to perform. This accessibility is crucial for experimentation, education, and research, making cutting-edge quantum computing tools available to a broad audience.

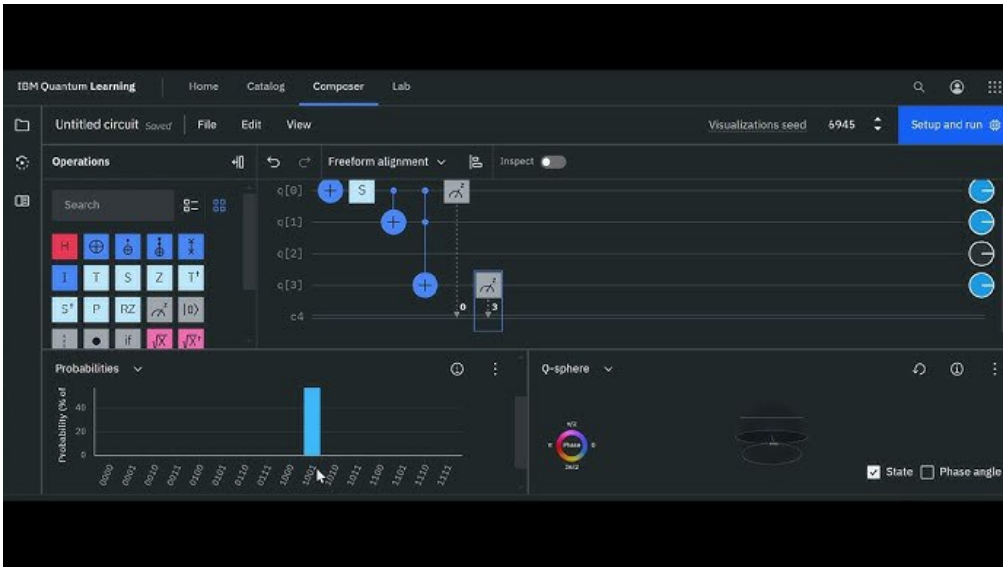


Figure 1: IBM Quantum Platform

2.3 Description of Quantum Gates

Quantum circuits operate using quantum gates, which are the equivalent of classical logic gates but follow quantum mechanical principles. Here, we detail the theoretical and mathematical properties of the gates used in the QRNG circuits.

2.3.1 Hadamard Gate

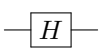
The Hadamard gate (H) generates superposition states from the computational basis states $|0\rangle$ and $|1\rangle$. The mathematical representation is:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

and its action on the basis states is given by:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

In a quantum circuit, it is represented by the symbol:

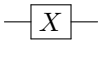


2.3.2 NOT Gate (X Gate)

The NOT gate, or X gate, flips the state of the qubit from $|0\rangle$ to $|1\rangle$ and vice versa. It is mathematically described by:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

In a quantum circuit, it is represented by the symbol:



2.3.3 Measurement Operation

Measurement gates collapse the superposition of a qubit into one of its basis states. For a general qubit state:

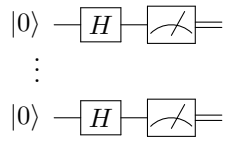
$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

the measurement yields state $|0\rangle$ with probability $|\alpha|^2$ and state $|1\rangle$ with probability $|\beta|^2$, ensuring that $|\alpha|^2 + |\beta|^2 = 1$. In a quantum circuit, it is represented by the symbol:

3 Description of Proposed Circuits

3.1 Basic QRNG

The Basic QRNG circuit involves initializing a different qubits, applying a Hadamard gate to place them into an equal superposition of $|0\rangle$ and $|1\rangle$, followed by a measurement.



If we have n qubits, their state before the measuremet is:

$$H^{\otimes n}\,|0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{i=0}^{2^n-1}|i\rangle = \frac{1}{\sqrt{2^n}}(|0\rangle + |1\rangle + |2\rangle + |3\rangle + \ldots + |2^n-1\rangle)$$

where $|m\rangle$ denotes the decimal representation of the state of the n qubits, considering the top qubit in the circuit as the most significant digit.

Advantages:

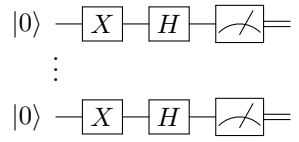
- Simple design for easy implementation.
- Produces genuine quantum randomness.

Drawbacks:

- Sensitive to initialization bias, potentially skewing the randomness.
- Imperfections in gate operations can introduce biases.

3.2 QRNG Type 1

QRNG Type 1 introduces an additional NOT gate before the Hadamard gate to invert the initial state from $|0\rangle$ to $|1\rangle$. This setup tests biases originating from initialization.



If we have n qubits, their state before the measurent is:

$$H^{\otimes n}\,|1\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{i=0}^{2^n-1}(-1)^{|i|}\,|i\rangle = \frac{1}{\sqrt{2^n}}(|0\rangle - |1\rangle - |2\rangle + |3\rangle + \ldots + (-1)^n\,|2^n-1\rangle)$$

where $|i| = i_1 + i_2 + \cdots + i_n$ denotes the Hamming weight of i, i.e., the number of ones in its binary representation $(i_1, i_2, \ldots, i_n)$.

Advantages:

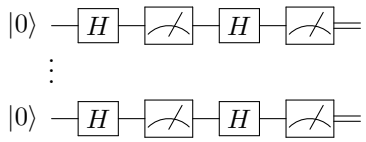
- With Basic QRNG, it detects and characterizes initialization bias clearly.

Drawbacks:

- Additional gate introduces further potential sources of error.

3.3 QRNG Type 2

QRNG Type 2 circuit tests the robustness of a quantum gate's ability to recreate superposition after measurement. A qubit undergoes initial Hadamard transformation, measurement (without storing results), followed by another Hadamard gate and final measurement.



The final state depends on the outcome of the first measurement.

Advantages:

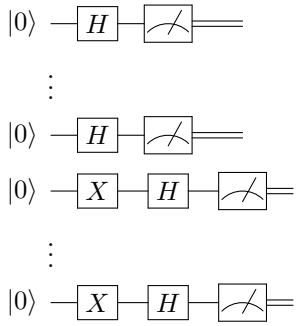
- Evaluates gate-measurement correlations.

Drawbacks:

- Extremely sensitive to decoherence and measurement-induced errors.

3.4 QRNG Type 3

QRNG Type 3 employs multiple qubits simultaneously. One half of the qubits follows the Basic QRNG procedure, while the other half follows QRNG Type 1, potentially counterbalancing biases and enriching data analysis possibilities.



The first half follows the same state as the basic QRNG circuit, while the second half follows the same state as the QRNG Type 1 circuit.

Advantages:

- Mitigates initialization biases through complementary setups.

Drawbacks:

- Increased complexity, raising susceptibility to error propagation and decoherence.

3.5 Effect of Decoherence

Decoherence represents the degradation of quantum coherence due to interaction with the environment. Two main decoherence mechanisms significantly influence QRNG circuit performance:

- **Relaxation Time (T_1)**: Represents the characteristic time it takes for a qubit initially in excited state $|1\rangle$ to spontaneously decay to the ground state $|0\rangle$. A short T_1 directly impacts QRNG Type 1 and Type 3 circuits by systematically biasing results towards state $|0\rangle$.
- **Dephasing Time (T_2)**: Characterizes the time scale over which qubits lose their phase coherence. A shorter T_2 significantly impacts all circuits, especially Type 2, which relies on maintaining and repeatedly reconstructing coherent superposition states. It reduces randomness and introduces predictable patterns or biases.

Managing decoherence involves optimizing circuit designs, reducing operation times, and enhancing environmental stability to ensure high-quality random number generation.

4 NIST Statistical Suite

The NIST Statistical Test Suite is a collection of statistical tests developed by the U.S. National Institute of Standards and Technology (**NIST**) aimed to evaluate the quality of sequences of binary digits (0s and 1s) produced by either hardware-based Random Number Generators (RNGs) or software-based Pseudorandom Number Generators (PRNGs). It aims to determine whether the sequence possesses specific statistical characteristics expected of a truly random sequence. This is crucial for applications where randomness quality is critical, such as:

- Cryptography (generating keys, nonces, etc.),
- Scientific simulations (e.g., Monte Carlo methods).

The process to perform NIST statistical Suite is the following one:

1. **Input**: The suite takes a sequence (or multiple sequences) of binary digits as input,
2. **Battery of Tests**: It applies a series of different statistical tests (currently 15 main tests in SP 800-22 Rev 1a, some with sub-tests). Each test is designed to detect specific types of non-random patterns or deviations from expected random behavior, such as:
    - Bias towards 0s or 1s (Frequency tests),
    - Unusual streaks of identical bits (Runs tests),
    - Periodicity (Discrete Fourier Transform test),
    - Compressibility or predictability (Maurer's Universal test, Linear Complexity test, Entropy tests),
    - Uniformity of block patterns (Serial tests, Template Matching tests).
3. **Output (p-value)**: For each test applied to a sequence, the suite calculates a p-value. The p-value is the probability of observing a sequence that is "less random" (according to the specific metric of that test) than the sequence being tested, assuming the generator is perfectly random*.
4. **Interpretation**: A significance level, denoted by α, is chosen beforehand, typically $\alpha = 0.01$.
    - If the calculated p-value $\geq \alpha$, the sequence is considered to have passed that specific test,
    - If the calculated p-value $< \alpha$, the sequence is considered to have failed that test. A very low p-value (especially 0, within the limits of calculation precision) indicates a strong failure.

In our experiment, we choose to use a **lenght** n of the sequence equal to 1 million of bits, this was done because the NIST suggest to use this lengh to obtain a reliable results(statistically meaningfull). The **sample size** was chosen following the formula $1/\alpha$, that with $\alpha = 0.01$ in our case is equal to 100 even if the NIST suggest to use 1000 as sample size, we were not able to produce such large amount of numbers, due to limits on IBM platform and on the long execution tasks. The **block size** used is the one used by the default by the library, because tailored on 1 million bit numbers and respect the NIST guidelines to be not larger than $\lfloor log_2 n \rfloor = 19$

No single test can definitively prove randomness and the strength of the NIST suite lies in its diversity. An RNG is generally considered good if producted sequences consistently pass all or nearly all of the tests. When testing multiple sequences, one expects a small proportion (around α, e.g., 1%) of sequences that fail tests just due to statistical chance. However, consistent failures on specific tests, very low p-values, or failures across a wide range of tests are strong indicators that the RNG is biased and should not be trusted for applications requiring high-quality randomness. In essence, the NIST STS provides a standardized benchmark to gain statistical confidence (or detect deficiencies) in the randomness of number generators.

The interpretation of empirical results can be conducted in any number of ways. Two approaches NIST has adopted include:

1. The examination of the proportion of sequences that pass a statistical test,
2. The distribution of P-values to check for uniformity.

4.1 Proportion of Sequences Passing a Test

Given the empirical results for a particular statistical test, compute the proportion of sequences that pass. The range of acceptable proportions is determined using the confidence interval defined as, $\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$,m where $p^{-1} = 1 - \alpha$, and m is the sample size. If the proportion falls outside of this interval, then there is evidence that the data is non-random, so the overall test fail.

4.2 Uniform distribution of p-values

The distribution of P-values is examined to ensure uniformity. This may be visually illustrated using a histogram , whereby, the interval between 0 and 1 is divided into 10 sub-intervals, and the P-values that lie within each sub-interval are count and displayed. The histogram is produced for each statistical tests inside the suite.

Uniformity may also be determined via an application of a χ^2 test and the determination of a P-value corresponding to the Goodness-of-Fit Distributional Test on the P-values obtained for an arbitrary statistical test(i.e., a P-value of the P-values). This is accomplished by computing $\chi^2 = \sum_{i=1}^{10} \frac{(F_i - \frac{s}{10})^2}{\frac{s}{10}}$, where F_i is the number of P-values in sub-interval i, and s is the sample size. A P-value is calculated such that $P_{\text{value},T} = \text{igamc}\left(\frac{9}{2}, \frac{\chi^2}{2}\right)$. If $P_{\text{value},T} \geq 0.0001$, then the sequences can be considered to be uniformly distributed:

- If $P_{\text{value},T} \geq 0.0001$: The hypothesis of uniformity is not rejected. It is concluded that the p-value distribution is compatible with a uniform distribution (the uniformity test is passed).
- If $P_{\text{value},T} < 0.0001$: The hypothesis of uniformity is rejected. It is concluded that the p-value distribution is not uniform (the uniformity test has failed), suggesting a problem either with the generator or with the primary test being analyzed.

5 Results

In this section, we will find out if the QRNG produce numbers with enough randomness by exploiting NIST Statistical test, if not we dive into possible problems. The circuit that generate quantum randomness is the **Type 3** for two reasons. The first is one is that is really fast to execute on quantum computer and the second one is related to limitation on the usage of QPU by IBM Quantum Platform(10 minutes). With this circuit we are able to generate around 30 1 million bit numbers in about 10 minutes. The Quantum Computer chosen is **IBM sheerbroke**.

As seen in 3.5, quantum circuits are present different errors due to the nature of quantum particles: **T1(Relaxation Time)** and **T2(Dephasing time)**. So the effect of decoherence can cause the quantum state to degrade and deviate from the ideal one before the measurement even occurs.

Table 1: Proportion of passed sequences

Test Name	**Pass Rate**	**Verdict**
Frequency (Monobit)	0.292 857	FAIL
Frequency within Block	0.592 857	FAIL
Runs	0.278 571	FAIL
Longest Runs in Block	0.728 571	FAIL
Matrix Rank	0.992 857	PASS
Discrete Fourier Transform	0.971 429	PASS
Overlapping Template Matching	0.964 286	FAIL
Maurer's Universal	1.000 000	PASS
Linear Complexity	0.978 571	PASS
Serial	0.996 429	PASS
Approximate Entropy	1.000 000	PASS
Cumulative Sums (Cusum)	0.257 143	FAIL
Random Excursions	0.894 643	FAIL
Random Excursions Variant	0.963 492	FAIL

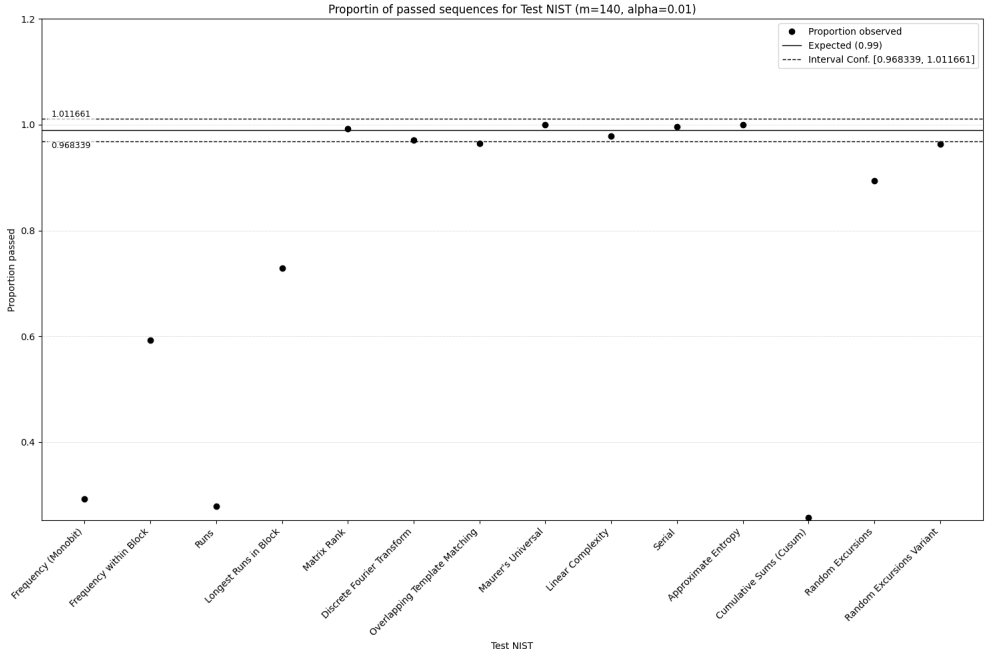
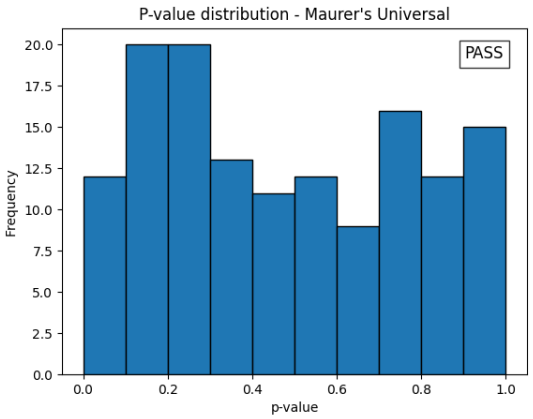


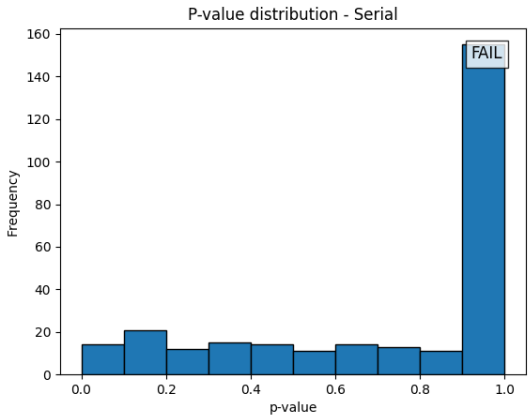
Figure 2: Proportion of passed sequences

For the **Proportion of Passed Sequences**, Figure 2 shows that these biases causes the failure of the **Frequency (Monobit) test**, as the overall count of '0's and '1's is unbalanced. The same bias causes the cumulative sum in the **Cusum test** to progressively and constantly drift away from zero, leading to failure. The bias also alters the expected length and frequency of sequences of identical bits, negatively affecting the **Runs test**. The failure of these fundamental tests often also reflects in the results of more complex tests. For example, comparing the **Frequency (Monobit) test** with the **Frequency within Block test**: while the first one fails severely due to the bias accumulated over the entire sequence, the latter (which analyzes the distribution of frequencies over smaller blocks) might fail less severely or even pass. This happens because a constant bias affects all blocks similarly, but does not necessarily drastically alter the distribution of frequencies among the blocks (which is what the Frequency within Block test primarily evaluates).

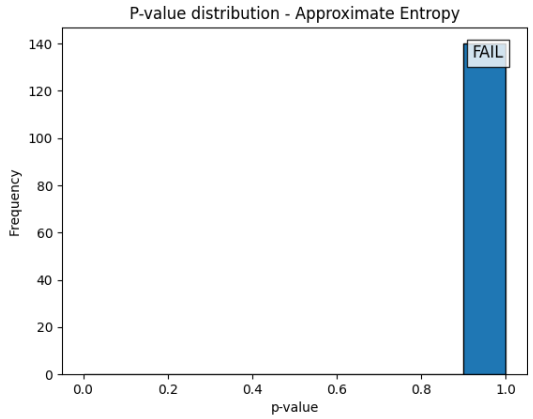
This observation reinforces the hypothesis that a systematic hardware error is present: a small but constant bias is amplified (or rather, its cumulative effect becomes statistically significant) over long sequences, whether they are generated in a single long process or obtained by concatenating shorter sequences.



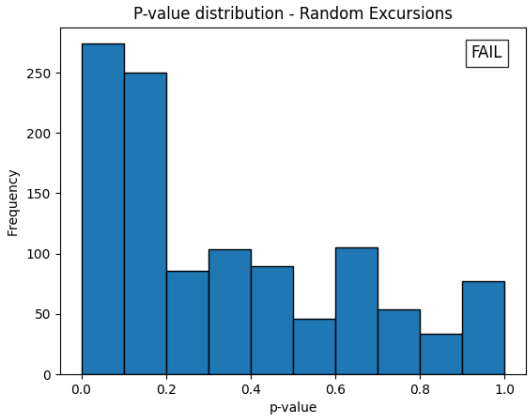
(a) Maurer's Universal



(b) Serial



(c) Approximate Entropy



(d) Random Excursions

Figure 3: Uniformity Test

The **Uniformity Test** does not directly evaluate the randomness of the original bit sequence, but is applied to the results obtained from one of the primary statistical tests when that primary test has been performed on a large number of different sequences generated by the same generator. The purpose is to verify whether the collection of p-values, obtained from these multiple executions of the primary test, is

uniformly distributed in the interval [0, 1]. If the random number generator under examination is good (i.e., produces truly random sequences), then the p-values resulting from that test should themselves behave as random variables uniformly distributed between 0 and 1.

In Figure 3 are depicted histograms for all the tests and the verdict for each test. In our experiment, specifically in Figure 3b and Figure 3c, we find out that some tests pass the Proportion Test, but fail the Uniformity Test. This fact highlights that the QRNG, even when producing sequences that pass the primary test, does it in a statistically suspicious manner because produce a lot of p-values greater than 0.01 but are not distribuited uniformly and is present a peak in count of low p-values. So the p-values are not distributed as they should be if they came from truly random data analyzed with a valid test.

6 Conclusion

Current quantum computers are not perfect random number generators 'out-of-the-box'. Their raw output reflects the device's noise and imperfections due to physical errors, particularly measurement bias and imperfect state preparation. To obtain high-quality random numbers from quantum hardware, steps involving error mitigation and/or randomness extraction (post-processing) could be future improvement on this project.

References

[1] Honno. *sts-pylib: Python interface to the NIST statistical tests for randomness*. Accessed: 2025-05-04. 2020. URL: https://github.com/honno/sts-pylib.

[2] Lawrence E. Bassham III et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800-22 Rev 1a. Gaithersburg, MD: National Institute of Standards and Technology, Apr. 2010. DOI: 10.6028/NIST.SP.800-22r1a. URL: https://doi.org/10.6028/NIST.SP.800-22r1a.

[3] R. B. Prajapati and S. D. Panchal. "Enhanced Approach To Generate One Time Password (OTP) Using Quantum True Random Number Generator (QTRNG)". In: *International Journal of Computing and Digital Systems* 15.1 (2024), pp. 279–292. DOI: 10.12785/ijcds/150122. URL: https://doi.org/10.12785/ijcds/150122.

[4] Owen Root and Maria Becker. "Does True Randomness Exist? Efficacy Testing IBM Quantum Computers via Statistical Randomness". In: *arXiv preprint arXiv:2401.12250* (Jan. 2024). 12 pages, 6 figures. DOI: 10.48550/arXiv.2401.12250. URL: https://arxiv.org/abs/2401.12250.