QUANTUM CRYPTOGRAPHY REPORT

NEWER TESTING SUITES FOR QUANTUM RANDOM NUMBER GENERATOR

November 26, 2025

Shravar G Tanawde
M25IQT012
Indian Institute of Technology, Jodhpur
m25iqt012@iitj.ac.in

Contents

1 INTRODUCTION

Random number generation is a requirement in cryptography and secure communications. While classical pseudo-random number generators (PRNGs) derive randomness algorithmically from deterministic seeds, Quantum Random Number Generators (QRNGs) extract entropy from inherently unpredictable quantum processes such as radiation decay, squeezed light, or a beam splitter. Hence, QRNGs are expected to be truly random, in contrast to PRNGs, which are inherently predictable.

However, quantum processes may be theoretically unpredictable, but the physical devices used to produce these random strings are not. Detector inefficiencies, environmental fluctuations, temperature drift, component aging, and optical misalignment all reduce the effectual entropy output of a QRNG, introducing bias or long-range correlation into the stream of bits it produces. These imperfections accumulate over time, making QRNG testing necessary.

Compounding this problem is the risk of external manipulation. Research shows that attackers can deliberately bias QRNG hardware through methods such as controlled attenuation, wavelength-specific attacks, or injection locking, without creating detectable deviations in statistical test outputs. Similarly, extractors, which are the post-processing stage intended to amplify entropy, may inadvertently mask underlying flaws rather than reveal them, producing a statistically uniform but fundamentally faulty array of random strings.

The primary tools used to validate random number generators include statistical testing suites, such as NIST SP 800-22A, and the Dieharder testing suite. These suites analyze data streams for bias, structural patterns, and deviations from the expected distribution of truly random sequences. Though very useful, they are very limited. They were not designed for quantum hardware; they assume independent and identically distributed (i.i.d.) samples, and they can be defeated by carefully engineered sequences that appear statistically perfect yet are entirely predictable or compressible.

As QRNG technology develops and is deployed in high-security environments, testing methodologies must evolve. This has led to the emergence of modern frameworks that supplement classical statistical methods for better, more suitable results. These include:

- **Specious Randomness Testing Suites** that detect sequences that are too perfect statistically (e.g., excessively low chi-square values) or artificially uniform.
- **Machine-Learning Models**, such as transformer-based classifiers, trained to identify subtle non-randomness patterns missed by classical tests.
- **Hardware-aware Test Suites** like the TuRiNG framework, designed specifically for Physical Unclonable Functions (PUFs) and physically derived randomness, where spatial correlation or device-specific behavior must be considered.
- **Improved statistical testing suites** such as PractRand and TestU01, which extend detection capabilities for long-range correlations and rare structural anomalies.

Despite these advances, no single testing suite currently provides comprehensive coverage. Even cutting-edge approaches struggle with specious randomness, extraction-layer vulnerabilities, device manipulation, and "invisible attacks" that preserve classical statistics while degrading true entropy. Thus, the field lacks a unified framework capable of reliably validating QRNGs under realistic physical and adversarial conditions.

In this report, we will explore the aforementioned testing suites and how they improve upon classical statistical testing suites, specifically for quantum random number generation.

2 CLASSICAL TESTING SUITES

Classical statistical test suites were initially developed to evaluate the quality of algorithmic pseudorandom number generators. They assess whether a bitstream appears consistent with an ideal random sequence by applying a set of distribution- and pattern-based statistical tests. Although these suites remain widely used, they were not designed with quantum entropy sources in mind, and their applicability to QRNGs is therefore limited.

This section will review two of the most popular classical suites: NIST SP 800-22 and Dieharder Test, and analyze their methodologies and inherent weaknesses.

2.1 NIST SP 800-22A Statistical Test Suite

NIST SP 800-22 is one of the most popular randomness-testing suites. It consists of fifteen statistical tests that evaluate various aspects of uniformity and independence within the bitstream, such as:

- Frequency (Monobit) Test
- Block Frequency Test
- Runs Test
- Serial Tests
- Approximate Entropy Test
- Discrete Fourier Transform Test, etc

However, the NIST testing suite is very limited when we consider physical QRNGs. Device drift, detector imperfections, and environmental fluctuations cause issues that actively work against the fundamentals upon which NIST tests are built. The main limitations of NIST SP 800-22A are:

1. NIST SP 800-22A was first released in 2010 with the last revision in 2014. For more than a decade, the test has not undergone any updates to keep it current with the technological advances made during this period.
2. NIST testing relies on an assumption that the data is independent and identically distributed (i.i.d.).
3. Multiple tests in this suite require a very large bitstream, some even requiring 10^6 bits for functioning properly. This is usually not possible with QRNG.
4. Also, the suite focuses on detecting statistical imbalance, but it overlooks sequences that are artificially uniform or structured - also known as specious randomness

2.2 Dieharder Test Suite

Dieharder tests originate from the original Diehard testing suite. It includes thirty-one statistical checks covering pattern counts, permutations, bit correlations, and various forms of distribution analysis. In comparison to NIST, it offers a much wider variety of tests, but it suffers from similar limitations due to being a statistical testing suite. As a result, Dieharder is unable to detect systematic issues that arise from hardware manipulation, environmental drift, or extractor-induced masking of non-uniformity. These limitations are:

1. Dieharder originates from Diehard Test, which was first proposed in 1995, and has not seen an update in its test since 2008.
2. It assumes the input is i.i.d. and needs a sufficiently long bitstream.
3. It evaluates only statistical properties of the output, without incorporating any information about the physical entropy source.
4. Like NIST, it can be misled by deterministic sequences engineered to satisfy expected statistical patterns.

These limitations motivate the need for more advanced testing frameworks that incorporate structural, physical, and machine–learning based analysis, which will be discussed in the next section.

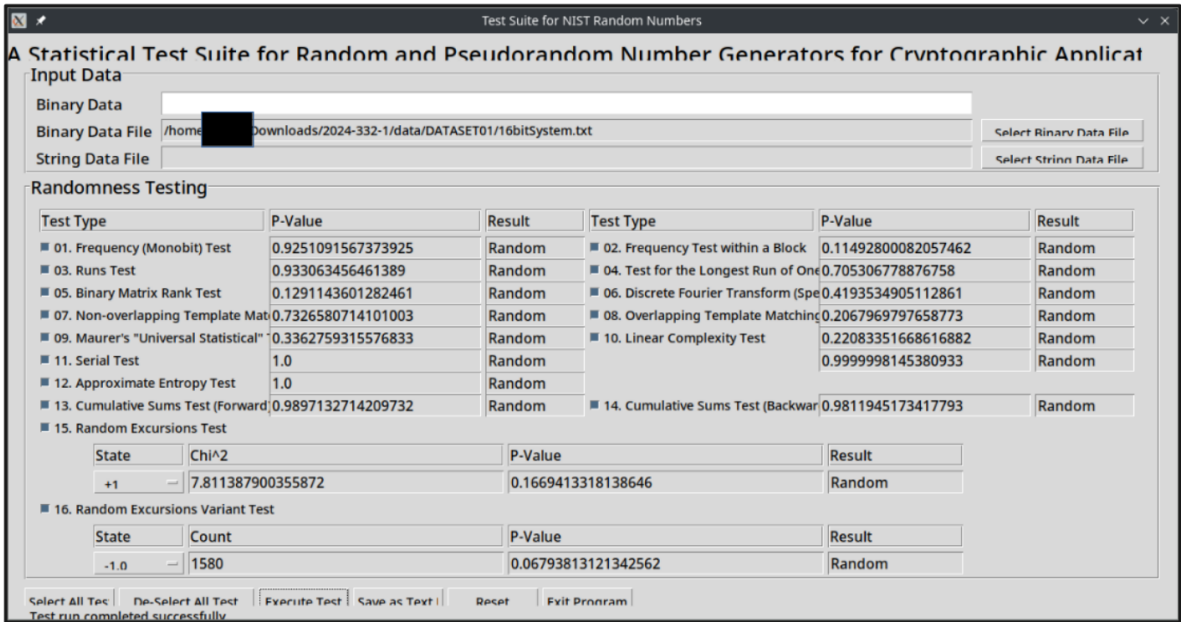

Figure 1: 16-bit Specious Randomness Passing NIST Tests

3 NEWER TESTING SUITES

Newer QRNG testing suites were necessary due to limitations found in classical statistical suites. These newer methods extend beyond simple output-based tests and incorporate deeper structural analysis, compressibility considerations, machine-learning detection, hardware awareness, or long-range statistical evaluation. Together, they represent an attempt to identify subtle or adversarial deviations that traditional suites cannot detect.

3.1 Specious Randomness Testing Suite

Specious randomness refers to sequences that perfectly mimic the statistical properties expected of random data yet are entirely predictable because deterministic rules generate them. These sequences can pass classical tests despite their high compressibility rate. The method of constructing such a sequence involves constructing a balanced n-gram sequence. Deterministic sequences can be engineered such that every possible substring of length n appears with exactly equal frequency. This produces distributions that appear "ideal" under frequency-based tests, yet they are not truly random. Since this sequence is created by using permutations of all possible substrings of length n, we can quickly find predictable patterns forming. Using various compression methods, such as modified Huffman coding, it is easy to make these patterns compressible. A fundamental property of true randomness is that it cannot be compressed (Kolmogorov complexity). The predictability of these patterns makes it an inferior choice for security, but the uniformity of these strings means that they easily pass statistical testing suites.

Chi-Squared Test or χ^2 Test is a standard statistical test that denotes "distance from expectation". Traditional χ^2 tests only check for results that deviate above expectation (i.e., overly uneven distributions). Modern approaches additionally flag χ^2 values that are too low, indicating suspiciously perfect uniformity. This exposes artificially balanced sequences that classical tests would accept.

$$\chi^2 = \sum_{i=0}^{k-1} \frac{(O_i - E_i)^2}{E_i} = \sum_i \frac{(v_i - \mathcal{N}/m)^2}{\mathcal{N}/m}$$

Where:

- k is the total number of categories. For n-bit patterns, $k = 2^n$.
- O_i is the observed count of the i-th n-gram in the sequence.

- E_i is the expected count of the i-th pattern, given by

$$E_i = \frac{N}{k},$$

  Where N is the total number of observed n-grams.

- v_i is the number of observations in bin i.
- $\mathcal{N}$ is the total number of observed n-grams.
- $m = 2^n$ is the number of possible n-bit patterns.
- Each pattern has an a priori probability of $1/2^n$ under the uniform-random assumption.

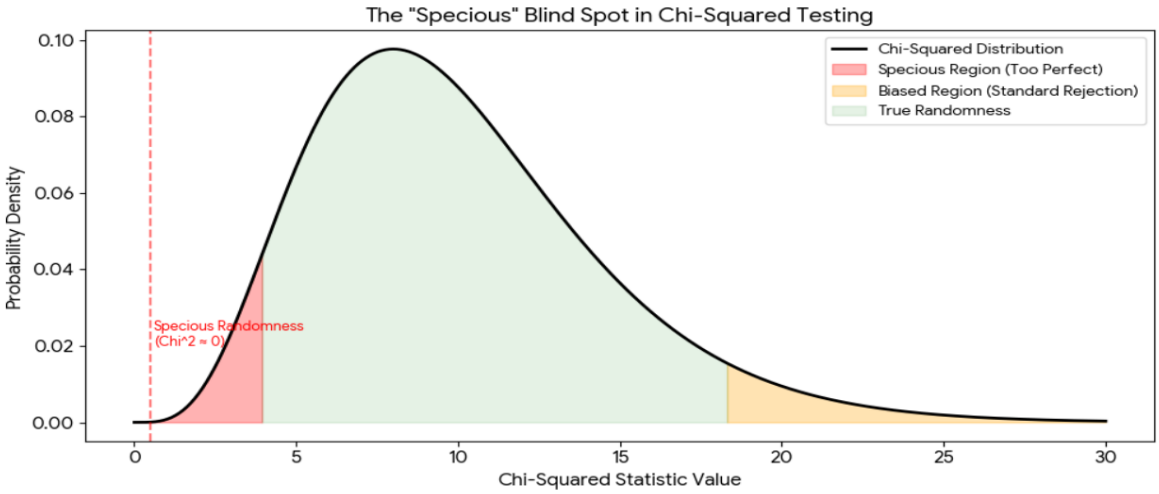

Figure 2: χ^2 Distribution graph with tail-end and front-end highlighted

3.2 Transformer Model as Random Number Classifier

Traditional validation methods, such as the NIST Statistical Test Suite, are computationally expensive, slow, and require sequential testing (one test at a time). This creates a bottleneck for high-speed applications. To improve upon Transformer Models, they were tested for their accuracy and speed. A lightweight Transformer architecture can act as a fast, unified validator for random-number sequences. Instead of running several statistical tests sequentially, the model performs multi-label classification, predicting whether a given binary sequence would pass or fail multiple NIST tests simultaneously. Binary sequences are tokenized, embedded, and fed into an encoder-only Transformer designed to detect subtle sequential patterns. Since non-random sequences often contain hidden structure or correlations, the self-attention mechanism becomes effective at identifying deviations that traditional tests flag.

The training dataset consists of truly random QRNG output mixed with artificially manipulated non-random sequences. Each sequence is labeled using the results of seven NIST STS tests. The model learns to match the output of the NIST test for whether the sequence would pass each test. The final architecture utilizes a single encoder layer, single-headed attention, and a 192-dimensional embedding, with an averaging layer to accommodate variable sequence lengths. This design was found to be the best trade-off between performance and efficiency.

Across input sizes of 512, 1024, and 2048 bits, the model achieves Macro F1-scores above 0.96, matching or slightly exceeding the performance of LSTM. The Transformer is significantly faster than both LSTMs and the NIST STS, processing large datasets in a fraction of the time due to parallelizable attention operations.

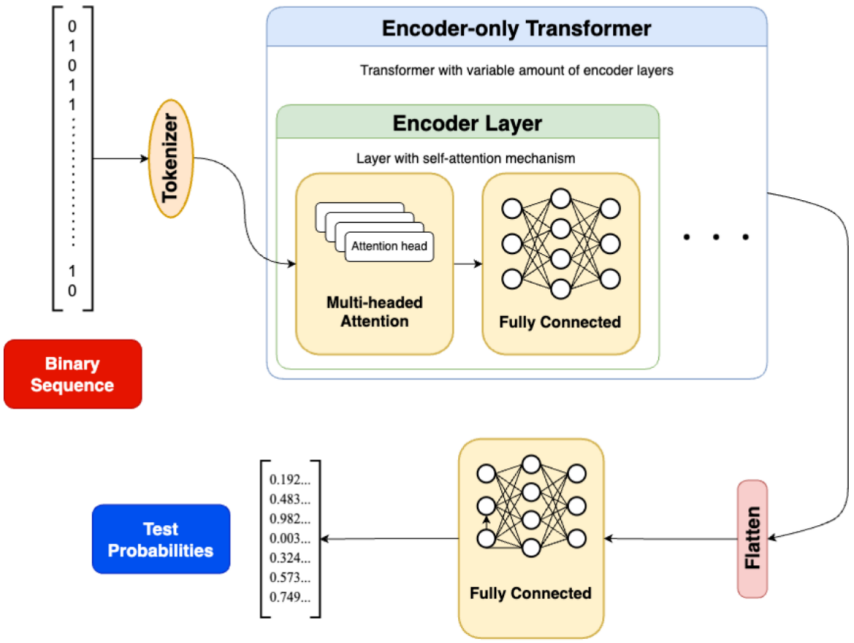

Figure 3: Transformer Model for Randomness Testing

The Transformer classifier provides two key advantages over classical statistical tests.

- First, it offers substantial efficiency gains. Because attention mechanisms operate in parallel, the model evaluates thousands of sequences far faster than the sequential structure of NIST STS or LSTM-based methods. This enables the rapid screening of large-scale random-number output, which is crucial for high-throughput QRNGs or real-time validation workflows.
- Secondly, the model introduces non-linear pattern detection capabilities that classical tests inherently lack. Statistical suites typically examine one feature at a time (e.g., runs of ones, block frequency, spectral peaks), relying on fixed formulas and assumptions such as independence and stationarity. A deep-learning model, by contrast, learns representations directly from data. This enables it to identify non-linear correlations, interactions between patterns, weak or high-order dependencies, and other similar anomalies.

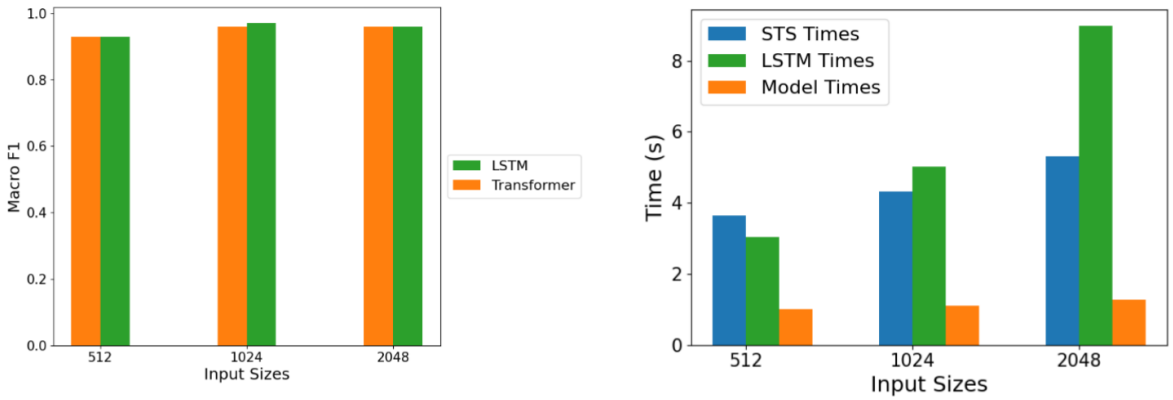

Figure 4: MacroF1 and Computation Time comparisons

In practice, this means the model can highlight sequences that appear fully compliant under traditional tests yet contain traces of dependency or pattern formation. Such capabilities make the Transformer a meaningful step toward identifying forms of non-randomness that result from complex hardware behavior, environmental drift, or deliberate manipulation.

3.3 TuRiNG Test

Modern randomness validation must account for hardware behaviors not captured by classical suites. A Physical Unclonable Function (PUF) is a hardware security technology that utilizes the inherent, random variations in a device's manufacturing process to create a unique, physical "fingerprint." These variations happen at the atomic scale, where quantum mechanics comes into play. Therefore, these devices constitute a closely related category of hardware-based random number generators. A PUF produces a fixed-length bitstring shaped by underlying physical microstructure, meaning tests must adapt to limited sample sizes, spatial dependencies, and cross-device correlations. The TuRiNG-style approach modifies classical tests to address these challenges.

Figure 5: PUF Array

The three main limitations that the Turing Test solves compared to statistical methods are,

- Reduce the high output requirement of some NIST tests by reparameterizing them. Examples include using 4×4 matrix rank tests instead of 32×32, and replacing CLT-based approximations with exact distribution computations for template matching. These reductions maintain statistical meaning while making tests feasible for short bitstrings.
- PUF outputs often come from a 2D physical array. Test results can vary drastically depending on how bits are concatenated. The framework tests all possible cases, Row ordering, Column ordering, and Block-based ordering, to ensure local spatial biases (e.g., defective regions, correlated adjacent cells) are not masked by a single canonical ordering. QRNGs with detector arrays or multi-pixel sensing architectures can benefit from identical procedures.

- A major contribution is a statistical test for inter-device independence. Instead of considering only per-device randomness, the framework checks whether outputs from different PUF instances exhibit correlation patterns. This is achieved by calculating per-device p-values, enabling pass/fail behavior rather than relying on vague global statistics. QRNGs also face the analogous issue of multi-unit deployments, where identical component imperfections might induce correlated biases. A similar independence test can therefore strengthen QRNG fleet validation.

Table 1: Recommended orderings for each randomness test

Test	**Row?**	**Column?**	**Block?**
Monobit	✓	×	✓
Block Frequency	✓	✓	✓
Runs	✓	✓	×
Longest Runs	✓	✓	✓
Binary Matrix Rank	✓	✓	✓
Discrete Fourier Transform	✓	✓	×
4-bit Template Matching	✓	✓	✓
Serial	✓	✓	✓
Approximate Entropy	✓	✓	✓
Cumulative Sums	✓	✓	✓
Non-Overlapping Template Matching	✓	✓	✓
Overlapping Template Matching	×	✓	✓
Maurer's Universal Statistic	×	×	✓
Linear Complexity	×	×	✓
Random Excursions	×	×	✓
Autocorrelation Parameter 1	×	✓	×

```
There were 1000 devices in the dataset
The p-value for tests was 0.01
FHD Test: 985 devices out of 1000 passed

Randomness test pass counts (row order) at p-value 0.01:
Monobit Test: 988 of 1000, chi = 0.04903009680856367
Approximate Entropy: 989 of 1000, chi = 0.6993125708664081
Cumulative Sums: 989 of 1000, chi = 0.2368098152177318
Block Frequency: 986 of 1000, chi = 0.4807713143119623
Runs Test: 988 of 1000, chi = 0.8991707579982895
Longest Run of Ones: 990 of 1000, chi = 0.5422280239417125
DFT: 986 of 1000, chi = 0.394195382900628
Template Matching (fixed template): 983 of 1000, chi = 0.3907207096894158
Binary Matrix Rank Test: 989 of 1000, chi = 0.16998074885675502
Serial: 989 of 1000, chi = 0.9288565852283481
Serial Variant: 989 of 1000, chi = 0.784926882244637

Randomness test pass counts (column order) at p-value 0.01:
Approximate Entropy: 994 of 1000, chi = 0.7849268822446372
Cumulative Sums: 988 of 1000, chi = 0.15991006862766652
Block Frequency: 988 of 1000, chi = 0.25055820897829206
Runs Test: 989 of 1000, chi = 0.005932365924043212
Longest Run of Ones: 987 of 1000, chi = 0.6287904561747886
DFT: 986 of 1000, chi = 0.16904411493023286
Template Matching (fixed template): 988 of 1000, chi = 0.09657817385182599
Binary Matrix Rank Test: 995 of 1000, chi = 0.32520593104051193
Serial: 992 of 1000, chi = 0.4827068719844463
Serial Variant: 988 of 1000, chi = 0.973717809836578

Randomness test pass counts (block order) at p-value 0.01:
Block Frequency: 986 of 1000, chi = 0.4846458858263931
Cumulative Sums: 987 of 1000, chi = 0.6600123548951184
Binary Matrix Rank Test: 988 of 1000, chi = 0.032922821737753044
The p-values are written as arrays by Main for more information e.g. for running tests for uniformityThere were 1000 devices in the dataset
The p-value for tests was 0.01
FHD Test: 985 devices out of 1000 passed
```

Figure 6: TuRiNG Test Result on 1000 simulated PUF devices (1)

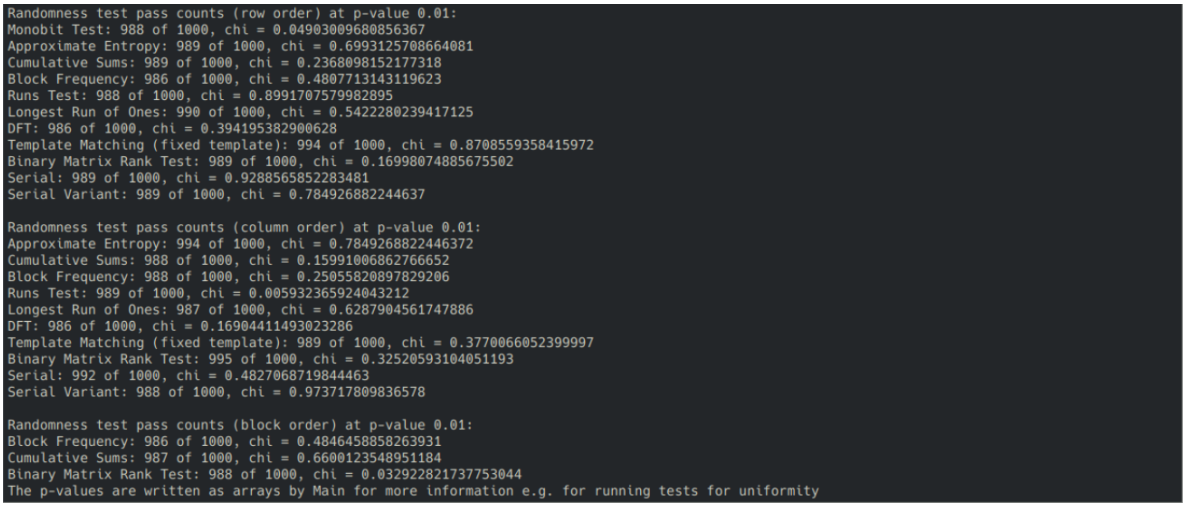

Figure 7: TuRiNG Test Result on 1000 simulated PUF devices (2)

3.4 PractRand and TestU01

PractRand and TestU01 represent two of the most potent modern statistical testing frameworks for randomness evaluation. Though initially designed for algorithmic PRNGs, both provide useful, though limited, coverage for QRNG validation when used appropriately.

3.4.1 PractRand

PractRand is a streaming-oriented test suite optimized for extremely long bitstreams.

- **Long-range correlation detection:** QRNGs may exhibit subtle drift from detector aging or environmental changes; PractRand reveals such effects, which are only visible over large data volumes.
- **Adaptive scaling:** Its incremental testing is useful for continuous QRNG monitoring, where deviations may emerge slowly over extended operation.
- **Cross-block structural checks:** Helpful for spotting extraction-layer artifacts or periodic interference created by optical or electronic components.

Limitations: Requires huge output volumes and cannot detect hardware-specific failure modes on its own.

3.4.2 TestU01

TestU01 (SmallCrush, Crush, BigCrush) remains one of the most statistically rigorous test batteries available.

- **Comprehensive coverage:** Advanced tests such as matrix-rank, linear complexity, and birthday spacings can reveal structural patterns caused by subtle physical bias or imperfect whitening.
- **High sensitivity:** BigCrush's analytical depth can detect anomalies that would not appear in smaller batteries, offering strong assurance for high-quality QRNG streams.
- **Effective against extractor induced structure:** If a QRNG extractor introduces periodic behavior or algorithmic artifacts, TestU01 can often detect them.

Limitations: Extremely slow, requires gigabit-to-terabit samples, and remains purely output-focused and unsuitable as the sole validator for physical entropy sources.

4 CHALLENGES AND FUTURE

4.1 Modern Limitations of Current Testing Suites

Even with advanced testing frameworks, such as specious-randomness detection, ML-based classifiers, PUF-inspired structural analyses, and advanced statistical testing suites, several fundamental limitations remain. These limitations arise from the hardware and extractor-specific attacks required for the creation of QRNGs. Modern suites significantly extend the detection surface, but they still cannot guarantee complete coverage against some of the failures outlined below.

- **Hardware Attacks That Preserve Output Statistics:** QRNGs remain vulnerable to physical manipulation that degrades true entropy without altering the uniformity of the output. An adversary can adjust optical power, alignment, or inject phase-correlated classical noise in such a way that the extractor continues to produce outputs that appear statistically uniform and perform well in testing suites but are flawed at a fundamental level. This occurs because modern testing suites primarily operate at the level of the final output and cannot detect changes at the hardware level.
- **Extractor-Side Vulnerabilities:** The post-processing extractor remains a critical weak point. Even sophisticated extractors may suppress or mask biases rather than expose faults when the underlying entropy source degrades. Specific extractor designs can absorb structured imperfections into their hashing process, producing outputs that appear uniformly random despite underlying failures.
- **Non-IID Drift Under Environmental Instability:** Real QRNGs rarely generate perfectly independent and identically distributed samples. Environmental variation, temperature drift, laser aging, and SPAD afterpulsing introduce correlations that can cause tests to pass or fail, creating significant vulnerabilities that may not be detected. Existing statistical and ML-based suites assume a more stable distribution than what valid hardware offers.
- **Invisible Attacks That Mimic Quantum Noise:** "Invisible attacks" are attacks that can stealthily bias the quantum source while remaining undetectable even after passing certified extraction and min-entropy estimation. These attacks rely on manipulating internal optical modes without affecting measured parameters, injecting classical noise that statistically mimics quantum shot noise, and controlling side channels (e.g., spectral components, timing jitter) that the extractor does not use. These "invisible attacks" survive traditional tests, advanced suites, min-entropy estimators, and even certified extraction pipelines.

4.2 Current Research Addressing Modern Limitations

Recent work in QRNG security is shifting toward methods that directly address the shortcomings of both classical and modern testing suites. These approaches move beyond output-only validation and introduce physical-layer monitoring, device-level verification, and real-time entropy assessment. Although none of these solutions completely closes all gaps, they represent the most promising directions for mitigating the four significant modern limitations.

- **Real-Time Online Entropy Monitoring & Health Tests:** This method involves integrating continuous entropy assessment directly into the QRNG firmware. Real-time monitoring reduces the risk posed by non-IID drift and environmental instability. Instead of relying on large offline datasets, the QRNG actively checks its own health during operation.
- **Self-Testing and Public-Verification QRNG Architectures:** This research is about having internal consistency checks and structural redundancy. These systems aim to counter hardware-preserving attacks by providing a mechanism for external auditors to verify correctness without accessing the private output stream. This introduces transparency while maintaining security.
- **Advanced Entropy Estimation And Hardware Integration:** Deep convolutional or recurrent neural networks are being used to accelerate or augment entropy estimation, particularly in high-speed optical QRNGs. This approach helps address extractor-side vulnerabilities. By estimating entropy before extraction with higher sensitivity, the system is less likely to "launder" bias or hide issues introduced in the physical layer.

- **Device-Independent and Measurement-Device-Independent QRNG:** Research into DI and MDI QRNG removes trust assumptions from significant portions of the device. This direction directly targets hardware manipulation and invisible attacks by grounding randomness in device-independent quantum principles.

5 CONCLUSION

The evaluation of quantum random number generators requires methods that extend far beyond the capabilities of traditional statistical test suites. Classical frameworks such as NIST SP 800-22 and Dieharder were initially designed for algorithmic randomness and therefore assume stationary, independent, and uniformly distributed outputs. While they continue to serve as useful baselines, they cannot capture the wide range of structural, physical, and adversarial imperfections that arise in real QRNG systems. Their primary limitation is that they treat randomness purely as an output property rather than a manifestation of an underlying physical process.

Modern testing approaches offer a significant improvement by broadening the detection landscape. Techniques such as specious-randomness analysis, two-sided χ^2 evaluation, compressibility checks, Transformer-based classifiers, and PUF-inspired structural testing introduce sensitivity to both engineered uniformity and more complex dependency patterns. Long-range statistical suites, such as PractRand and TestU01, further extend evaluation into the high-volume regime, enabling the detection of slow or accumulated deviations that classical tests overlook. Together, these methods provide deeper insight into the statistical and structural integrity of QRNG outputs.

However, modern frameworks still face inherent limitations. Hardware-preserving attacks, extractor-level masking, non-IID drift, and adversarial noise engineering remain difficult to detect through output-only inspection. These unresolved vulnerabilities underscore the fundamental challenge of QRNG validation: without explicit access to or trust in the physical entropy source, even the most advanced statistical techniques cannot guarantee that the observed bitstream accurately reflects genuine quantum randomness.

Contemporary research is actively addressing these challenges. Real-time entropy monitoring, self-testing architectures, deep learning–based entropy estimation, and device-independent QRNG designs represent promising directions. These approaches shift the focus from pure output analysis to more integrated, physically grounded validation, thereby reducing reliance on assumptions that adversaries or hardware faults may exploit.

In summary, the path toward fully reliable QRNG testing lies in layered validation: classical tests for baseline detection, modern statistical and ML-based tools for structural and non-linear anomalies, and physical-layer or device-independent methods for resisting sophisticated attacks. No single tool suffices, but together, these approaches form a robust framework for assessing and safeguarding the integrity of quantum-derived randomness in practical deployments.

6 REPRODUCTION WORK

To validate and better understand the behavior of modern randomness-testing techniques, a practical reproduction of specious randomness is performed. This reproduction involves:

1. Specious Random Number Generator
2. Specious Random Number Tester

The reproduction work focuses on generating perfectly balanced n-gram distributions and measuring their deviation under χ^2 analysis. This reproduction provides a hands-on demonstration of how specious randomness can be constructed and how enhanced statistical tests can detect it. The corresponding code is included below for completeness and reproducibility.

6.1 Specious Random Number Generator

First part of the code is a random number generator that creates a specious randomness. To generate a speciously-random bitstring first we generate all N bit strings.

Example, if N=3, then generate ["000", "001", ..., "111"]

Afterwards these strings are randomly shuffled and concatenated. This step is repeated until desired length is achieved. This data is finally copied to a txt file such that every line contains only 8 bits at a time to NIST test compliant.

```python
import random

def generate_specious_stream(N=16, target_length=None, shuffle=True,
    outfile="specious_bits.txt"):

    # Step 1: Generate all N-bit patterns.
    # -----------------------------------
    # For every integer i from 0 to (2^N - 1), format it as a binary string
    # padded with leading zeros such that each pattern has exactly N bits.
    # This ensures we cover the ENTIRE space of all possible N-bit sequences
    # exactly once, making the resulting concatenated stream "specious".
    patterns = [format(i, f"0{N}b") for i in range(2**N)]

    # Step 2: Shuffle of the entire pattern list.
    # -------------------------------------------
    # Shuffling prevents the patterns from appearing in sorted (lexicographic) order.
    # It obscures the underlying sequential structure of the enumerated bit patterns.
    if shuffle:
        random.shuffle(patterns)

     # Step 3: Concatenate all N-bit patterns into one big bitstream.
    # ---------------------------------------------------------------
    # Joining the list of binary strings creates a single long string of bits.
    stream = "".join(patterns)

     # Step 4: Adjust stream to match user-specified target_length.
    # ------------------------------------------------------------
    # If target_length is given:
    # Case A: stream is already long enough → truncate it to target_length.
    # Case B: stream is too short → repeat the entire concatenated stream
    #         enough times to reach or exceed target_length, then truncate.
    # This is useful for producing fixed-length test datasets for statistical
    # randomness testing frameworks.
    if target_length is not None:
        if len(stream) >= target_length:
            # Simply take the first target_length bits.
            stream = stream[:target_length]
        else:
            # If too short, compute how many times we must repeat the full stream.
            multiplier = (target_length // len(stream)) + 1
            # Repeat and then trim to exact target_length.
            stream = (stream * multiplier)[:target_length]

    # Step 5: Write the final bitstream to a file in 8-bit chunks.
    # ------------------------------------------------------------
    # The output file will contain 8 bits per line. This format is required for
    # NIST testing suite. If the total length of the stream is not divisible by 8,
    # the last line will contain fewer than 8 bits.
    with open(outfile, "w") as f:
        for i in range(0, len(stream), 8):
            f.write(stream[i:i+8] + "\n")

    # Final console message summarizing the total produced bit count and output file.
    print(f"[OK] Generated specious stream of {len(stream)} bits → saved as {outfile}")
    return stream


```

```python
if __name__ == "__main__":
    N=16
    generate_specious_stream(
        N,
        target_length=(N)*(2**(N)),
        shuffle=True,
        outfile="specious_bits.txt"
    )
```

```
[OK] Generated specious stream of 1048576 bits → saved as specious_bits.txt
```

For comparison purposes, the output of the code for generating specious random number string was tested by NIST testing suite. As expected the NIST tests failed to identify the recurring patterns and instead gave passing results.

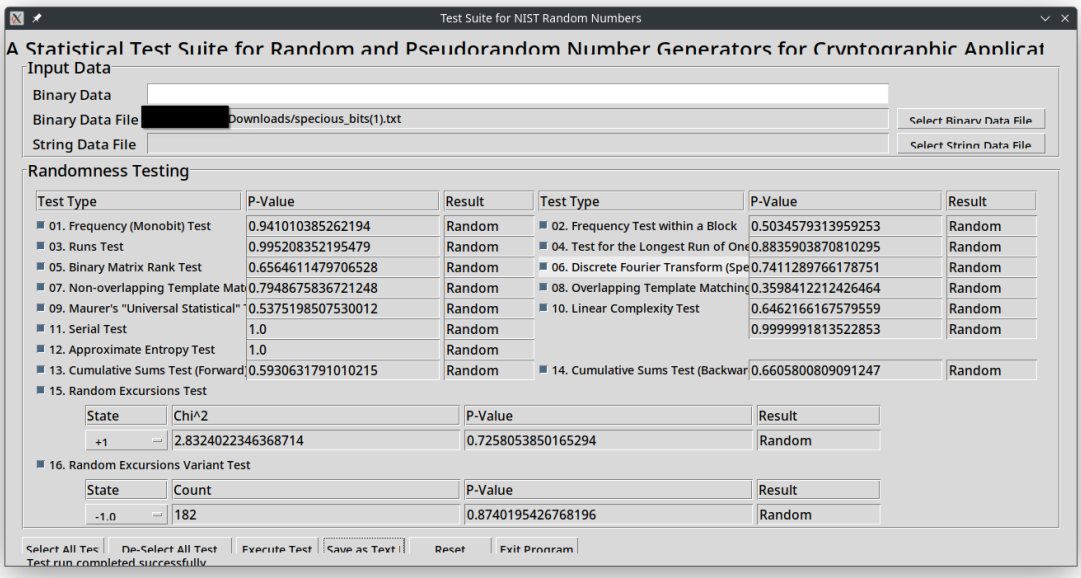

Figure 8: NIST test passing the "specious random" number generated

6.2 Specious Randomness Number Tester

The test is designed to detect artificially uniform n-gram frequencies, something normally not detected by the NIST randomness test suite. A truly random sequence should occasionally deviate from the expected frequency distribution. A specious sequence is constructed so that all n-grams occur with suspiciously equal frequency, causing a lower-tail χ^2 failure.

This notebook evaluates the binary file by computing:
- n-gram frequency distributions
- χ^2 statistic
- lower-tail cumulative probability
- double-sided rejection

```python
import math
from collections import Counter
from pathlib import Path
from typing import List, Dict
from scipy.stats import chi2
```

Step 1: χ^2 Distribution: PDF and CDF

The χ^2 test evaluates how far the observed n-gram frequencies deviate from the expected uniform distribution. The paper defines the chi-square probability density function (PDF) as:

$f(k, x) = \frac{1}{2^{k/2}\Gamma(k/2)} x^{k/2-1} e^{-x/2}$

Instead of integrating this PDF manually—as done in the paper—we use:

- scipy.stats.chi2.cdf (exact)
- a Gaussian approximation (for very large degrees of freedom)

This avoids numerical overflow for large n, such as n=24 (df $\sim$ 16 million).

```python
# ------------------------------------------------------------
# Safe chi-square CDF and inverse CDF
# ------------------------------------------------------------
def chi2_cdf_safe(k, x):
    if SCIPY_AVAILABLE:
        return float(chi2.cdf(x, k))
    # Gaussian approximation for large k
    # mean = k, variance = 2k  → Z = (x - k)/sqrt(2k)
    z = (x - k) / math.sqrt(2 * k)
    # standard normal CDF
    return 0.5 * (1 + math.erf(z / math.sqrt(2)))


def chi2_inv_cdf_safe(k, p):
    if SCIPY_AVAILABLE:
        return float(chi2.ppf(p, k))
    # Gaussian inverse using erf^-1:
    # x  k + sqrt(2k) * Z
    # Z = ^{-1}(p)
    from math import sqrt
    from math import erfcinv

    # ^{-1}(p) = -sqrt(2) * erfcinv(2p)
    Z = -math.sqrt(2) * erfcinv(2 * p)
    return k + math.sqrt(2 * k) * Z
```

Step 2: Counting n-Grams

A binary string of length L contains:

- overlapping n-grams: $L - n + 1$ samples
- non-overlapping n-grams: $\lfloor L/n \rfloor$ samples

In a specious sequence, all n-grams occur almost equally often. This is the behavior that the new randomness test aims to detect.

```python
# ------------------------------------------------------------
# n-gram counting
# ------------------------------------------------------------
def count_ngrams(bits: str, n: int, overlapping=True) -> Dict[str, int]:
    L = len(bits)
    out = Counter()
    if overlapping:
        for i in range(L - n + 1):
            out[bits[i:i+n]] += 1
    else:
        for i in range(0, L - n + 1, n):
            out[bits[i:i+n]] += 1
    return out
```

Step 3: Computing χ^2 from n-Gram Frequencies

The paper defines the χ^2 statistic for discrete uniform distribution as:

$\chi^2 = \sum_{i=1}^{m} \frac{(\nu_i - N/m)^2}{N/m}$

Where:

- ν_i = observed count of the i-th n-gram
- N = total number of observed n-grams
- m = 2^n possible patterns
- expected frequency = N / m

A low $\chi^2 \rightarrow$ observations are too close to expected uniform distribution
A high $\chi^2 \rightarrow$ observations deviate too much
Both extremes indicate non-randomness.

```
#-------------------------------------------------------------
# ² statistic (Eq. 18)
# -------------------------------------------------------------
def chi_square_stat(counts: Dict[str, int], n: int):
    m = 2 ** n
    N = sum(counts.values())
    expected = N / m
    chi = 0.0

    for i in range(m):
        key = format(i, f'0{n}b')
        observed = counts.get(key, 0)
        chi += (observed - expected) ** 2 / expected
    return chi, N, m
```

Step 4: Lower and Upper Critical Limits

The novelty of the paper's test is:

- Upper χ^2 cutoff (standard NIST-like test)
- Lower χ^2 cutoff (detects "too uniform" specious randomness)

If:

$\chi^2 < \chi^2_{p/2} \rightarrow$ FAIL (too uniform)

or

$\chi^2 > \chi^2_{1-p/2} \rightarrow$ FAIL (too uneven)

Otherwise $\rightarrow$ PASS

```
# -------------------------------------------------------------
# Full n-gram randomness test (double-sided)
# -------------------------------------------------------------
def test_n(bits: str, n: int, p=0.01, overlapping=True):
    counts = count_ngrams(bits, n, overlapping)
    chi, N, m = chi_square_stat(counts, n)

    k = m - 1  # degrees of freedom

    # Lower tail
    lower_p = chi2_cdf_safe(k, chi)

    # Critical values
    lower_crit = chi2_inv_cdf_safe(k, p/2)
    upper_crit = chi2_inv_cdf_safe(k, 1 - p/2)

    result = "PASS" if (lower_crit <= chi <= upper_crit) else "FAIL"

    return {
```

```
        "n": n,
        "df": k,
        "m": m,
        "chi2": round(chi, 2),
        "l_crit": round(lower_crit, 2),
        "u_crit": round(upper_crit, 2),
        "result": result,
    }
```

Step 5: Running Across All n

We run the test for:
n=1,2,...,N
If the input file is specious randomness, the test will fail for many n. If the file is truly random, the χ^2 values will stay within acceptable limits.

```
# ---------------------------------------------------------------
# Main runner
# ---------------------------------------------------------------
def run_full_test(filepath: str, n_values: List[int], p=0.01, overlapping=True):
    bits = Path(filepath).read_text().replace("\n", "")
    bits = "".join(ch for ch in bits if ch in "01")

    results = []
    for n in n_values:
        print(f"Testing n={n} ...")
        results.append(test_n(bits, n, p, overlapping))
    return results
```

Step 6: Execute the Test

This cell processes the file generated by your specious randomness generator (N = 16). As we can see from the result this test failed for many of the cases but as we saw previously this same bitstream passes NIST test.

```
# Example runner:
if __name__ == "__main__":
    N=16
    filepath = "specious_bits.txt"
    n_range = list(range(1, N+1))
    results = run_full_test(filepath, n_range, p=0.01, overlapping=True)

    for r in results:
        print(r)
```

The exact same file was used in this modified χ^2 test and again the results shows great success as it manages to correctly identify the specious randomness and failed every single iteration. Following is the result that was obtained:

```
Testing n=1 ...
Testing n=2 ...
Testing n=3 ...
Testing n=4 ...
Testing n=5 ...
Testing n=6 ...
Testing n=7 ...
Testing n=8 ...
Testing n=9 ...
Testing n=10 ...
```

```
Testing n=11 ...
Testing n=12 ...
Testing n=13 ...
Testing n=14 ...
Testing n=15 ...
Testing n=16 ...
{'n': 1, 'df': 1, 'm': 2, 'chi2': 0.0, 'l_crit': 0.0, 'u_crit': 7.88, 'result': 'FAIL'}
{'n': 2, 'df': 3, 'm': 4, 'chi2': 0.03, 'l_crit': 0.07, 'u_crit': 12.84, 'result': 'FAIL'}
{'n': 3, 'df': 7, 'm': 8, 'chi2': 0.13, 'l_crit': 0.99, 'u_crit': 20.28, 'result': 'FAIL'}
{'n': 4, 'df': 15, 'm': 16, 'chi2': 0.55, 'l_crit': 4.6, 'u_crit': 32.8, 'result': 'FAIL'}
{'n': 5, 'df': 31, 'm': 32, 'chi2': 2.4, 'l_crit': 14.46, 'u_crit': 55.0, 'result': 'FAIL'}
{'n': 6, 'df': 63, 'm': 64, 'chi2': 9.04, 'l_crit': 37.84, 'u_crit': 95.65, 'result': 'FAIL'}
{'n': 7, 'df': 127, 'm': 128, 'chi2': 34.98, 'l_crit': 89.7, 'u_crit': 171.8, 'result': 'FAIL'}
{'n': 8, 'df': 255, 'm': 256, 'chi2': 92.48, 'l_crit': 200.59, 'u_crit': 316.92, 'result': 'FAIL'}
{'n': 9, 'df': 511, 'm': 512, 'chi2': 204.24, 'l_crit': 432.41, 'u_crit': 597.1, 'result': 'FAIL'}
{'n': 10, 'df': 1023, 'm': 1024, 'chi2': 484.86, 'l_crit': 910.25, 'u_crit': 1143.27, 'result': 'FAIL'}
{'n': 11, 'df': 2047, 'm': 2048, 'chi2': 1072.01, 'l_crit': 1885.95, 'u_crit': 2215.57, 'result': 'FAIL'}
{'n': 12, 'df': 4095, 'm': 4096, 'chi2': 2353.56, 'l_crit': 3865.65, 'u_crit': 4331.86, 'result': 'FAIL'}
{'n': 13, 'df': 8191, 'm': 8192, 'chi2': 5190.68, 'l_crit': 7865.07, 'u_crit': 8524.44, 'result': 'FAIL'}
{'n': 14, 'df': 16383, 'm': 16384, 'chi2': 11370.69, 'l_crit': 15920.5, 'u_crit': 16853.02, 'result': 'FAIL'}
{'n': 15, 'df': 32767, 'm': 32768, 'chi2': 24749.64, 'l_crit': 32111.35, 'u_crit': 33430.16, 'result': 'FAIL'}
{'n': 16, 'df': 65535, 'm': 65536, 'chi2': 53425.7, 'l_crit': 64606.21, 'u_crit': 66471.3, 'result': 'FAIL'}
```

The entire code is uploaded on to Github for reference.

References

[1] J. Almlöf, G. Vall Llosera, E. Arvidsson, *et al.*, "Creating and detecting specious randomness," *EPJ Quantum Technology*, vol. 10, no. 1, 2023. doi: 10.1140/epjqt/s40507-022-00158-7.

[2] R. Goel, Y. Xiao, R. Ramezani, "Transformer models as an efficient replacement for statistical test suites to evaluate the quality of random numbers," arXiv:2405.03904, 2024. Available at: arxiv.org/abs/2405.03904.

[3] Crypto Quantique, *TuRiNG: A PUF randomness test suite* (Version 2023) [Computer software]. GitHub repository. Available at: github.com/cryptoquantique/TuRiNG-A-PUF-randomness-test-suite.

[4] Y.-F. Chen, D. Wang, Y.-B. Zhao, L. Cheng, Y. Zhang, Y. Zhang, "Statistical Invisibility of a Physical Attack on QRNGs After Randomness Extraction," arXiv:2508.21498, 2025. Available at: arxiv.org/abs/2508.21498.

[5] Y. Liu, Q. Zhao, M.-H. Li, *et al.*, "Device-independent quantum random-number generation," *Nature*, vol. 562, pp. 548–551, 2018. doi: 10.1038/s41586-018-0559-3.

[6] Almlöf J, Vall Llosera G, Arvidsson E, Björk G. "Specious randomness data sequences for various number systems," 2021. Available at: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-298306.