Lab 03: A Comparative Study of Hardware-Based TRNGs and PRNGs for Enhanced Security

Ganesh Veluru and Jwala Sri Hari Badam
University of South Florida
Tampa, FL 33620

I. Introduction

Randomness is a fundamental concept in various fields, from cryptography and data security to statistical analysis and simulation. At its core, randomness denotes the absence of a predictable pattern, making it essential for generating unpredictable sequences of numbers or events. Random number generators (RNGs) are indispensable tools in computing and technology for generating such sequences. There are several types of RNGs, including Pseudo-Random Number Generators (PRNGs) [1], which produce seemingly random numbers using deterministic algorithms and an initial seed, and Cryptographically Secure PRNGs (CSPRNGs), designed to withstand cryptographic attacks while maintaining efficiency. In contrast, True Random Number Generators (TRNGs) harness inherent physical phenomena, such as electrical noise or radioactive decay, to generate truly unpredictable and unbiased randomness. RNGs find applications in a wide range of domains, including secure communications, cryptographic key generation, statistical simulations, and games, where the quality of randomness is critical to their effectiveness and security. The schematic representation of TRNG is shown in Fig 1

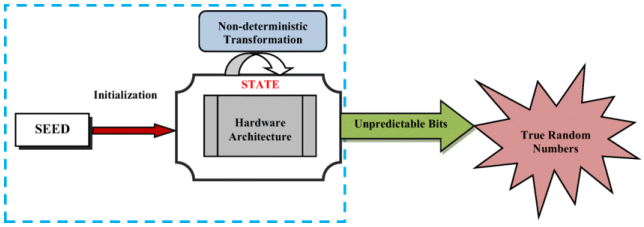


Fig. 1: Basic TRNG block diagram

In this project, we embark on the exciting journey of implementing, evaluating, and comparing both true and pseudo-randomness. To achieve this, we employ a Galois Ring Oscillator and a Linear Feedback Shift Register (LFSR) on an FPGA, leveraging physical processes and mathematical algorithms, respectively, to generate random bits. These bits will be subjected to rigorous testing using the NIST Statistical Test Suite, a comprehensive set of statistical tests designed to assess the quality of randomness. In parallel, we will utilize Python libraries to generate random bits through PRNG methods, enabling a direct comparison between pseudo-random and truly random sequences. By executing this comparative analysis, we aim to gain insights into the effectiveness and limitations of each approach, shedding light on the strengths and weaknesses of pseudo-randomness and true randomness in practical applications, particularly in the context of security and unpredictability.

II. Reading Check

Question 1: What kinds of applications are TRNGs used for?

Answer: TRNGs find applications in a wide range of domains, particularly in contexts where high-quality, truly unpredictable randomness is essential. They are used in secure communications and cryptographic systems to generate encryption keys, ensuring data confidentiality and protection against hacking. TRNGs are also employed in secure authentication processes, ensuring the integrity of digital identities and access control. In the field of scientific research, TRNGs are used for conducting experiments, simulations, and statistical analyses that demand truly random inputs. Additionally, they play a crucial role in gaming and gambling industries, enhancing the fairness and unpredictability of outcomes. In essence, TRNGs are integral to applications where robust security, reliability, and the absence of predictability are paramount.

Question 2: What are the 3 major components of a good TRNG design?

Answer: A well-designed True Random Number Generator (TRNG) typically consists of three major components [2]:

1. Entropy Source: The heart of a TRNG is its entropy source. This source gathers raw randomness from some physical process or phenomenon, such as electronic noise, radioactive decay, or unpredictable environmental factors like mouse movements. The quality of the TRNG heavily relies on the unpredictability and entropy provided by this source. It ensures that the generated numbers are truly random and unbiased.

2. Harvesting Mechanism: Once the entropy source collects raw randomness, a harvesting mechanism extracts and processes this data into a usable form. The harvesting process should be carefully designed to maximize the extraction of entropy without introducing any bias or patterns into the generated numbers. It's crucial that the harvesting mechanism doesn't disturb the natural randomness collected from the source.

3. Post-processing and Conditioning: The final component involves post-processing and conditioning of the

random bits generated by the harvesting mechanism. This step aims to remove any remaining bias or irregularities, ensuring that the output adheres to uniform randomness. Post-processing may involve techniques like whitening, error correction, or cryptographic hashing to enhance the quality and reliability of the generated random numbers.

In essence, a robust TRNG design hinges on the effective coordination of these three components, guaranteeing that the output is truly random, unbiased, and suitable for various applications, including cryptography and secure communications.

Question 3: How can we evaluate whether or not a TRNG is truly random?

Answer: Testing whether binary sequences are random relies on well-established mathematical and statistical methods that have been around for a long time. Most current research efforts aim to improve these methods rather than invent new ones. In this context, the NIST Statistical Test Suite is a popular and widely used tool for checking the randomness of sequences [3]. The names of all tests is shown in Table I.

Certainly, here is a brief description of each test included in the NIST Statistical Test Suite [4]:

1. Frequency (Monobit) Test: This test counts the number of 0s and 1s in the binary sequence and checks if they are roughly equal. It's a basic test to ensure that the sequence has a balanced distribution of 0s and 1s.

2. Frequency within a Block: In this test, the sequence is divided into fixed-length blocks, and the frequency of 0s and 1s in each block is examined. It assesses whether the local frequencies within the sequence meet the expected statistical distribution.

3. Runs Test: The Runs Test checks for the presence of consecutive runs of 0s or 1s in the sequence. It helps identify patterns where the same value appears multiple times in a row, which would be unusual in a truly random sequence.

4. Longest Run of 1s in a Block: This test focuses on finding the longest consecutive run of 1s within specific blocks of the sequence. It helps detect long stretches of 1s that might indicate non-randomness.

5. Binary Matrix Rank Test: The Binary Matrix Rank Test involves analyzing submatrices within the sequence to determine their rank. Deviations from randomness can be identified by studying the properties of these submatrices.

6. Discrete Fourier Transform (Spectral) Test: This test applies the Discrete Fourier Transform to the sequence to examine its spectral properties. It can reveal patterns and deviations from randomness in the frequency domain.

7. Non-overlapping Template Matching: This test searches for predefined templates or patterns within the sequence. It helps identify specific sequences of bits that might indicate non-randomness.

8. Overlapping Template Matching: Similar to the non-overlapping version, this test searches for predefined templates, but it allows the templates to overlap, increasing the sensitivity to pattern detection.

TABLE I: List of NIST Statistical Tests

Number	Test Name
1	Frequency (Monobit) Test
2	Frequency Test within a Block
3	Runs Test
4	Longest Run of 1s in a Block
5	Binary Matrix Rank Test
6	Discrete Fourier Transform (Spectral) Test
7	Non-overlapping Template Matching
8	Overlapping Template Matching
9	Maurer's "Universal Statistical" Test
10	Linear Complexity Test
11	Serial Test
12	Approximate Entropy Test
13	Cumulative Sums Test
14	Random Excursions Test
15	Random Excursions Variant Test

9. Maurer's "Universal Statistical" Test: Maurer's test calculates the average number of bits needed to predict the next bit in the sequence, providing a measure of the sequence's complexity and unpredictability.

10. Linear Complexity Test: This test assesses the linearity of the sequence, which is an indicator of its predictability. Lower linearity suggests greater unpredictability. 11. Serial Test: The Serial Test checks for correlations between adjacent bits in the sequence. It helps detect patterns where the occurrence of one bit depends on the value of the previous bit.

12. Approximate Entropy Test: This test measures the complexity of the sequence by comparing the frequency of overlapping blocks of a fixed length. Lower entropy values indicate less complexity and potentially more predictability.

13. Cumulative Sums Test: The Cumulative Sums Test examines the cumulative sums of the bits in the sequence. It's used to detect deviations from randomness by looking at the cumulative behavior of the data.

14. Random Excursions Test: This test investigates the number of cycles in excursions away from zero within the sequence. It helps identify patterns that may indicate non-randomness.

15. Random Excursions Variant Test: A variant of the Random Excursions Test, this test includes additional conditions and is used to further evaluate the sequence's behavior during excursions away from zero.

These tests collectively provide a comprehensive assessment of binary sequences, ensuring their suitability for various applications where randomness is essential.

III. Methods

In this experiment, we are utilizing CMOD S7 board and We have a PC and PHS- virtual machine at our disposal, and

within this virtual environment, we are executing scripts to gather random bits.

A. Hardware and Software Setup

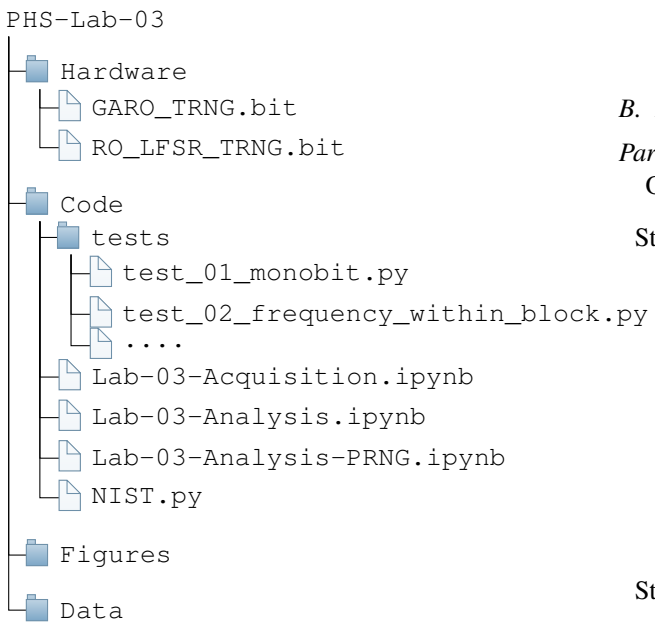
Our procedure for Hardware and Software setup will take the following steps:

Step 1: Connect FPGA Board-After starting the PHS-VM, connect the FPGA board to our PC via USB. Upon successful connection, we should see a green light blinking on the FPGA board, along with one red static light. To confirm that the device is properly connected, we can navigate to "Device" and "USB" in the VM. We should see "Digilent Adept USB Device" listed if the connection is established correctly. If not, we should double-check our hardware, software installations, and connections. The complete setup of our experiment is shown in the Fig 2.

Step 2: Access Jupyter Notebook-To start our investigation, we can open a web browser on our host machine and enter "http://localhost:8888/." This will allow us to access Jupyter Notebook locally. Inside Jupyter Notebook, we'll find various directories, and we should focus on the "PHS-Lab-03" directory.

Step 3: Unzip PHS-Lab-03 Content-Within the "PHS-Lab-03" folder, we can upload the provided "PHS-Lab-03.zip" file. Run the unzip code located in the "RUN-TO-ZIP.ipynb" file to extract the contents of "PHS-Lab-03.zip." Make sure that the name of the zipped file matches the current directory for successful extraction.

Step 4: Organizing the Extracted Content - After unzipping the provided archive, we will observe the folder structure illustrated in the accompanying figure.



a) Inside the "Hardware" folder, we find two essential bit files: RO_LFSR_TRNG.bit, GARO_TRNG.bit which are TRNG designs and will output random bits when implemented on FPGA. The first TRNG bit file RO_LFSR_TRNG.bit utilizes ten ring oscillators, which are enabled and combined using a Linear Feedback Shift Register (LFSR). The second bit file GARO_TRNG.bit" combines multiple Galois ring oscillators (GAROs), which are a hardware construct that incorporates elements of both ring oscillators and LFSRs. The process involves programming the FPGA with one of these bit files, collecting the required amount of random data from it, and then repeating the process with the other bit file.

b) Inside the "Code" folder, we discover two important files. The first is dedicated to "Data Acquisition." In this script, we will complete the necessary code to obtain random bits after implementing TRNG designs on the FPGA board. The second file pertains to "Data Analysis." Here, our objective is to analyze the acquired data using NIST.py. The "NIST.py" script found in the Code directory to assess the quality of your binary data by subjecting it to the NIST statistical test suite (STS). The results obtained from these tests will provide insights into the properties of your TRNG, allowing us to determine whether the generated bitstream exhibits favorable or unfavorable characteristics. Inside the "tests" folder we have code for all 15 tests which we used inside NIST.py to get results.

c) Figures generated during the analysis phase, such as plots and graphical representations, will be stored in the "Figures" directory.

d) The "Data" folder will be the destination for storing the random bits generated from the FPGA as well as using python random libraries. These responses are a crucial part of our experiment.

B. Experimental Procedure

Part A: Data Acquisition

Our procedure for Part A will take following steps [5]:

Step 1: To send and receive data from the TRNG design, we connect the Cmod S7 board to our computer using a USB connection. This board has a special chip that helps translate USB messages into a format that a Universal Asynchronous Receiver/Transmitter (UART) circuit can understand. The provided PHSVM already has the necessary drivers installed, creating a Virtual Port. This allows us to send messages using software on our computer or through a programming language especially python with a serial library.

Step 2: To begin, we first import the necessary Python libraries and ensure that our FPGA board is properly connected. We use the command "! djtgcfg enum" to enumerate and list all connected Digilent JTAG devices, including FPGAs.

Step 3: Before we can send or receive data, a crucial step is programming the FPGA board with the provided bitstream files using a Bash script. This step is essential for configuring the FPGA.

Step 4: To establish communication, we need to determine which serial port the FPGA board is connected to. To accomplish this, we utilize the ”serial.tools.list_ports.comports()” function in Python. Once we have identified the correct port, we initialize a serial connection using ”serial.Serial(port, baudrate)” with the specified port and baudrate settings.

Step 5: Next, our task is to get random data through the ‘collect_serial ()‘ function. This involves utilizing the serial connection object, providing 3 bytes input that is expected number of bits and using ‘ser.write()‘ for transmission. Subsequently, as we receive the random bits in bytes format and we will save the data in a file.

Step 6: Here we are generating 1 million and 10 million bits for each TRNG design. The code script to write input and read ouptut is shown in the Fig 2.

```
""" complete the code """
def collect_serial(ser: serial.Serial, size: int):
    # convert int size to bytearray, 3 bytes, big endian
    byte_array =  size.to_bytes(3,byteorder="big")

    print(bytearray)
    # send bytearray to FPGA
    ser.write(byte_array)

    # read back from FPGA
    r_bytes = ser.read(size)

    # return data collected
    return r_bytes
```

Fig. 2: Python Script to get random bytes from FPGA

Part B: Data Analysis

For the Analysis Part we are duplicating the Analysis Jupyter notebook for the python random modules implementation.

After importing the necessary libraries and loading data from files, Instantiating a TRNGtester object to load the random data. Then, we have to call the run_nist_tests() method from this object to generate the list of list of p-values. Each sub-list correspond to each test. Some test may have multiple p-values . therefore we are considering the minimum values from the list. The code snippet is shown in Fig 3.

We also generating random data (1M and 10M) using python PRNG modules like numpy random, os.urandom and generic random and saving as raw bytes in a file. The code snippet for this is showing in Fig 4. The random data generated using python modules also sending to TRNGtester run_nist_tests() to test the randomness using NIST statistical suite.

```
# storing filepaths into variables.
ro_1M = "../Data/RO_LFSR_TRNG_1M.bin"
ro_10M = "../Data/RO_LFSR_TRNG_10M.bin"
ga_1M = "../Data/GARO_TRNG_1M.bin"
ga_10M = "../Data/GARO_TRNG_10M.bin"
```

```
# storing varaibles in dictionary
dict_values={
    'RO_LFSR_TRNG_1M':ro_1M,
    'RO_LFSR_TRNG_10M':ro_10M,
    'GARO_TRNG_1M':ga_1M,
    'GARO_TRNG_10M':ga_10M
}
```

```
# method for returning the min values from the list passed
def get_min_lst(lst):
    res = [min(i) for i in lst]
    return res
```

```
#script to run all tests for the data collected.
res_values ={}

for i,j in dict_values.items():
    # creating an object
    rand = TRNGtester(j)

    # Running all test
    rand_final = rand.run_nist_tests()
    print(rand_final)
    values = get_min_lst(rand_final)
    res_values[i]=values

    print("================================================================
```

Fig. 3: Python script to get p-values using NIST suite

```
# using random module to generate 1M and 10M bits
random_array_rand_1m = []
for i in range(125000):
    random_array_rand_1m.append(random.randint(0,255))

random_array_rand_10m = []
for i in range(1250000):
    random_array_rand_10m.append(random.randint(0,255))
```

```
# OS Random Module to generate 1M and 10M bits
osrandom_array_rand_1m = []
for i in range(125000):
    random_byte = ord(os.urandom(1))
    random_byte %= 256
    osrandom_array_rand_1m.append(random_byte)

osrandom_array_rand_10m = []
for i in range(1250000):
    random_byte = ord(os.urandom(1))
    random_byte %= 256
    osrandom_array_rand_10m.append(random_byte)
```

```
# Numpy Random to generate 1M and 10M bits
random_array_numpy_1m = np.random.randint(0, 256, 125000)
random_array_numpy_10m = np.random.randint(0, 256, 1250000)
```

Fig. 4: Generating random data using python modules

IV. Results

In the NIST Statistical Test Suite, each test is assessed as either a pass or fail by evaluating its p-value(s). If the p-value is less than 0.01, the test is considered a failure, whereas a p-value greater than or equal to 0.01 indicates a passing result. This method is fundamental to the evaluation of tests within the NIST suite, a collection of statistical tests used for various

TABLE II: Statistical Test Results for TRNG and PRNG

Dataset	Count Passed
RO_LFSR_TRNG_1M	8
RO_LFSR_TRNG_10M	7
GARO_TRNG_1M	15
GARO_TRNG_10M	14
GENERIC_RAND_1M	14
GENERIC_RAND_10M	15
NUMPY_RAND_1M	3
NUMPY_RAND_10M	3
OS_RAND_1M	15
OS_RAND_10M	15

purposes, including assessing the quality of random number generators.

The statistical results regarding the number of tests passed using p-values for both TRNG designs are calculated using the python loop shown in Fig 5. This is repeated for the random data generated using python modules also.

The combined statistical test results for TRNG and PRNG are shown in table II.

```
# Counting the passed tests for each TRNG(1M and 10M)
print("Dataset","    Count Passed")
print("==========================")
for i,j in res_values.items():
    count_passed = 0
    for k in j:
        if k > 0.01:
            count_passed+=1
    print(i,"--->",count_passed)
    print('--------------------------')
```

```
Dataset     Count Passed
==========================
RO_LFSR_TRNG_1M ---> 8
--------------------------
RO_LFSR_TRNG_10M ---> 7
--------------------------
GARO_TRNG_1M ---> 15
--------------------------
GARO_TRNG_10M ---> 14
--------------------------
```

Fig. 5: Python script to get Statistical Results

The plots are generated using p-values on y-axis and tests on x-axis and the corresponding result for each test is shown on the bar graph. The plots are generated for 1 M and 10 M bits and for both PRNG and TRNG designs.

The plots for GARO_TRNG are shown in Fig 6

The plots for RO_LFSR_TRNG are shown in Fig 7

The plots for Generic Random module are shown in Fig 8

The plots for Numpy Random Module are shown in Fig 9

The plots for OS Random Module are shown in Fig 10

V. Discussion

From the statistical analysis of RO_LFSR and GARO TRNGs, 15 NIST statistical tests are passed for GARO TRNG for 1 million data whereas only 8 are passed for RO_LFSR. For 10 million data 14 tests are passed for GARO TRNG and only 7 for RO_LFSR.

The results of the NIST statistical tests reveal a significant disparity in the performance of the GARO TRNG and the RO_LFSR. While the GARO TRNG successfully passes all 14 NIST tests, indicating a high degree of randomness and statistical independence in its output, the RO_LFSR only passes 7 out of the 15 tests, suggesting weaknesses in generating truly random and statistically independent sequences. These findings imply that the GARO TRNG is better suited for applications requiring strong randomness, such as cryptographic systems, while the RO_LFSR may be more appropriate for non-cryptographic applications where weaker randomness is acceptable. However, the choice of generator should consider the specific requirements of the intended application and may require additional analyses beyond the NIST tests to make an informed decision.

Coming to comparison with PRNG random modules in python the data generated by numpy random module only passed 3 tests for both 1M and 10M data. Whereas data generated by generic random and os random module passed 15 out of 15 tests even they are PRNG modules.

The comparison of NIST statistical test results between the PRNG modules in Python reveals that the data generated by ‘numpy.random‘ exhibits significantly poorer randomness, passing only 3 out of 15 tests for both 1 million and 10 million data points. In contrast, the data generated by the generic ‘random‘ module and ‘os.random‘ module, also PRNGs, successfully passes all 15 tests. This discrepancy underscores the variance in randomness quality among PRNG algorithms, with ‘numpy.random‘ performing less satisfactorily in this regard. When comparing these PRNG results with the earlier TRNG findings, it reaffirms that TRNGs, relying on inherently unpredictable physical processes, generally outperform PRNGs in providing high-quality randomness, especially in security-critical applications. The choice between TRNGs and PRNGs should be contingent on the specific demands of the application, where strong randomness is imperative or where weaker randomness may suffice.

VI. Conclusion

In our project, we delved into understanding the differences between Pseudo-Random Number Generators (PRNGs) and True Random Number Generators (TRNGs). We took a hands-on approach by setting up a TRNG using FPGA hardware, focusing on comparing how they work and where they find use, especially in terms of security. An essential part of our study involved using the NIST Statistical Test Suite, a tool that helped us assess the quality and randomness of the numbers we generated. Through this project, we gained a deeper understanding of how important randomness is in various fields, especially in making cryptographic systems secure. We also learned that TRNGs, which rely on hardware, are more unpredictable and safer for security-related tasks compared to PRNGs, which use mathematical algorithms.

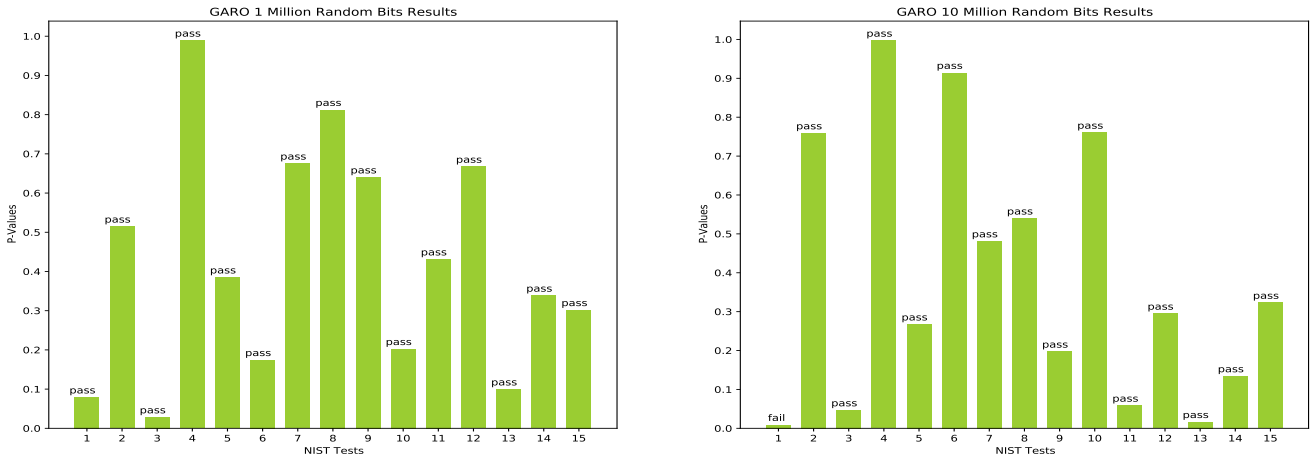


Fig. 6: Bar Chart showing p-values and test result for GARO TRNG

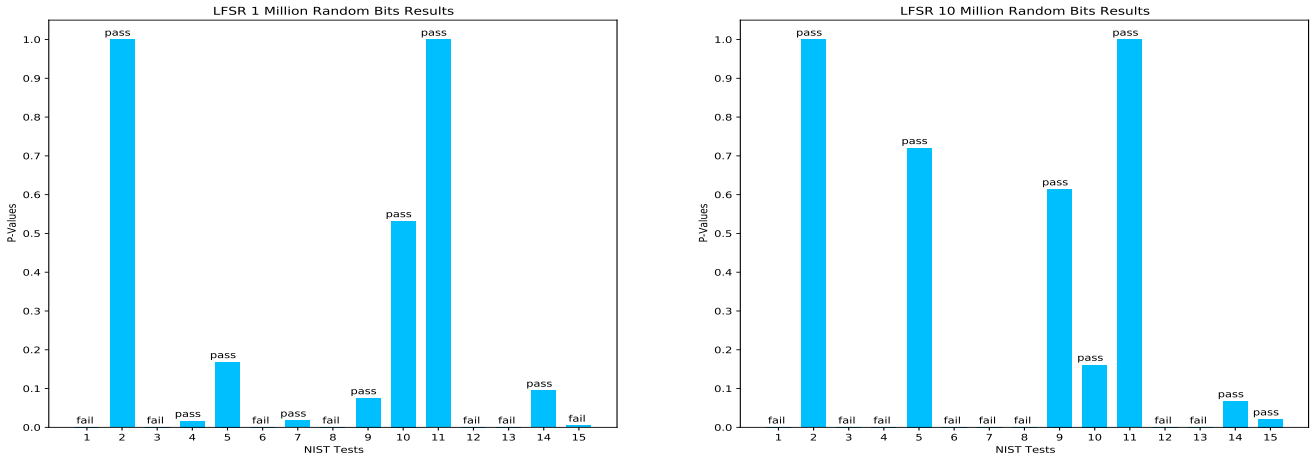


Fig. 7: Bar Chart showing p-values and test result for RO LFSR TRNG

Looking ahead, there are exciting areas we can explore further. One interesting path is investigating natural phenomena like radioactive decay or quantum effects as sources of true randomness. Understanding how these natural processes create randomness and finding practical uses for them, especially in the field of cryptography, could be fascinating for future research. Additionally, we can continue working on making TRNGs more efficient and secure, which will help improve overall security measures. In summary, this project has shown us the crucial role random numbers play and has opened up exciting opportunities in the world of information security and beyond.

REFERENCES

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Lecture Note 3: Pseudo- and True RNGs/PRNGs," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[2] ——, "Lecture Note 3: Pseudo- and True RNGs/Good TRNG Design," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[3] ——, "Lecture Note 3: Pseudo- and True RNGs/NIST Statistical Test Suite," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[4] ——, "Lecture Note 3: Pseudo- and True RNGs/NIST Tests," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[5] ——, "Experiment 3: Pseudo- and True Randomness Implementation and Evaluation," in *Practical Hardware Security Course Manual*. University of South Florida, Sep 2023.

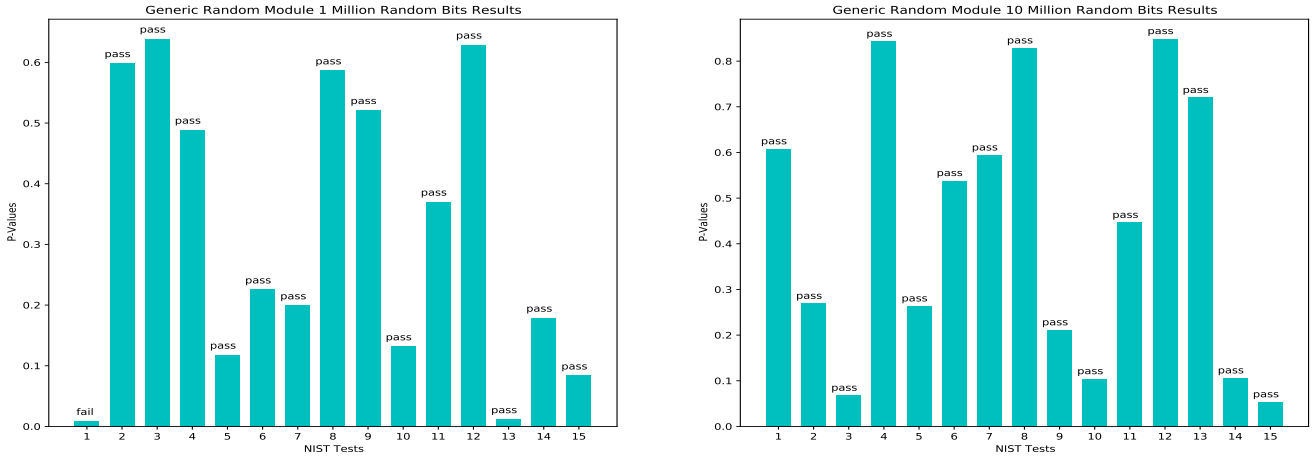


Fig. 8: Bar Chart showing p-values and test result for Generic Random Module

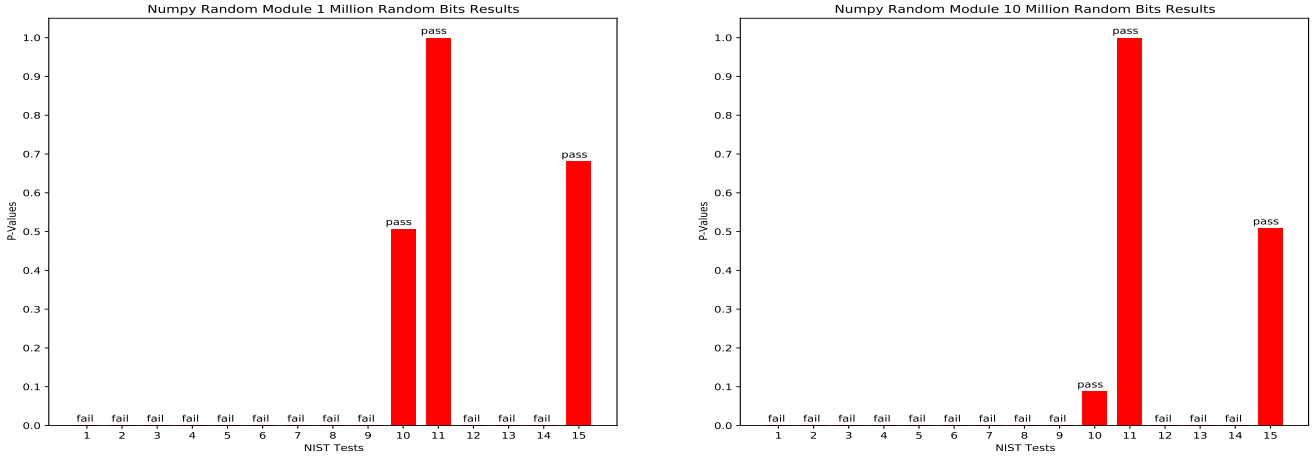


Fig. 9: Bar Chart showing p-values and test result for Numpy Random Module

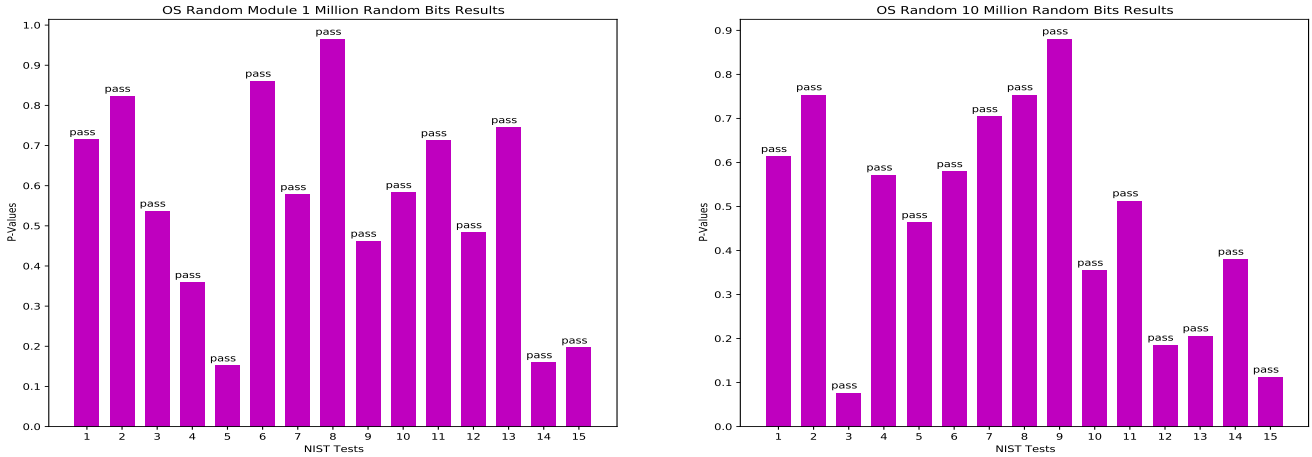


Fig. 10: Bar Chart showing p-values and test result for OS Random Module