3 Pseudo and True Randomness

Implementation and Evaluation

A Introduction

Random number generation is a critical component of many aspects of computing, and especially so within security domains. For instance random number generation is critical for random initialization of neural network parameters, creating unpredictable results in gambling/gaming systems, areas of science which require repeatable (non)-deterministic simulations and, of course, cryptographic applications. Random number generators (RNG) are often used to generate these random sequences of bits for such applications. That being said, there are important distinctions to make about the *components* of the RNG in order to properly assess its certification for different applications.

First, RNGs can be categorized into a few different classes; namely, pseudo-random number generators (PRNGs), cryptographically secure pseudo-random number generators (CSPRNGs) and true random number generators (TRNGs). PRNGs typically use an algorithm to generate a sequence of random numbers which are not *truly* random because they are determined by some initialization vector or *seed*. An example of a PRNG could be a linear feedback shift register (LFSR) which produces a (repeated) sequence of states that comprise the random bitstream. Consequently, they are not suitable for cryptographic applications, however enhancements may be made to make them suitable for cryptographic applications (i.e. CSPRNGs). Finally, TRNGs are dubbed *truly* random because they generate random numbers by harnessing the inherent randomness of certain physical phenomena, rather than purely algorithmically. For example, thermal noise, clock jitter, atmospheric readings and even quantum interactions [1, 2, 3, 4, 5]. In some scenarios, a TRNG may even be used for generating the random initialization vectors for a CSPRNG.

B Background

In general, a good TRNG consists of a few critical components [6]. First, the *entropy source* is the physical phenomena to be harnessed for generation and should be aperiodic and unpredictable. Second, the *harvesting mechanism* is the method which the TRNG uses to actually perform the extraction of entropy from the source. The harvester should be designed in such a way that it does not disturb the process itself while collecting as much information as possible. Finally, depending on the nature of the entropy source and harvester, a post-processing step may be necessary to mask imperfections or provide tolerance to faults and tampering. A diagram of a general TRNG structure is provided in Figure 1.

Naturally, this raises the question: *how can we determine whether or not a random number generator* ***really*** *is random?* Standards of testing binary sequences for randomness are heavily grounded in sound mathematics and statistics and have been for many years. Most ongoing research efforts generally consider optimizations to these methods rather than new ones. That being said, one of the most popular and widely used statistical testing frameworks used to assert randomness is the NIST Statistical Test Suite [7].

As seen in Table 1 NIST test suite consists of 15 statistical tests designed to test different features of binary sequences. In what follows we provide a brief description of each test: 1) *Frequency (monobit) test* focuses on the proportion of 0s to 1s in the entire sequence, 2) *Frequency test within a block* tests the proportion of ones within M-bit blocks, which should be approximately M/2, 3)

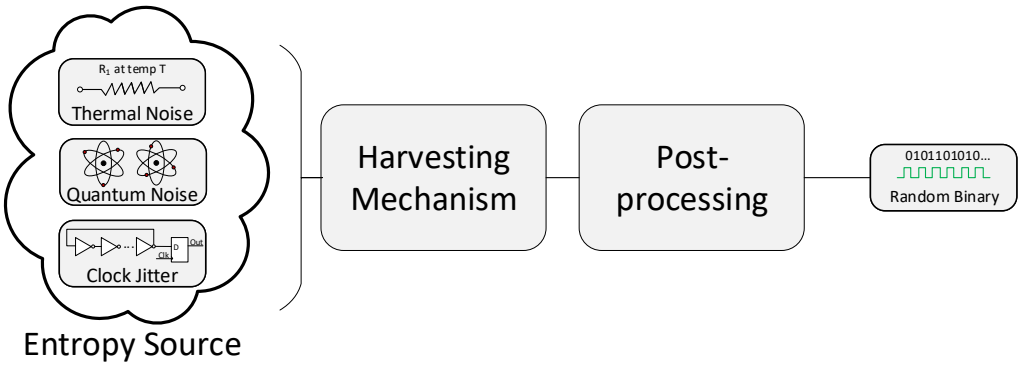


Figure 1: General TRNG structure consisting of entropy source (randomness), an extractor for that source and some post processing to finally output to binary.

Runs test focuses on the total number of runs (sequence of identical bits) there are in the sequence, 4) *Longest run of 1s in a block* tests for the longest run of 1s within M-bit blocks, 5) *Binary Matrix rank test* focuses on the rank of disjoint (32x32) sub-matrices of the entire sequence, 6) *Discrete Fourier Transform (Spectral) test* focuses on the peak heights in the DFT of the sequence and detects periodic features which indicate uniformity (non-randomness), 7) *Non-overlapping template matching test* computes the number of non-overlapping occurrences of a supplied bit template pattern, 8) *Overlapping template matching test* is similar to the non-overlapping template test, but the matches are allowed to overlap through the sequence, 9) *Maurer's "Universal Statistical" test* focuses on the number of bits between matching patterns in the sequence - essentially detecting whether or not the sequence can be compressed without information loss, 10) *Linear Complexity test* which divides the sequence into N M-bit blocks and computes the linear complexity of each block, which corresponds to the length of the shortest LFSR that generates all the bits in that block, 11) *Serial test* focuses on the frequency of all overlapping m-bit patterns in the sequence, 12) *Approximate Entropy test* is similar to the Serial test, however it compares the frequency of overlapping blocks for block sizes of m and m+1, 13) *Cumulative Sums test* computes the cumulative sum of the adjusted sequence mapped from $0, 1$ to $-1, 1$ and examines the maximal excursion from 0 of the random walk of the cumulative sums (when a line graph of the sequence oscillates over/under 0), 14) *Random excursions test* computes the same cumulative sum metric as the Approximate Entropy test, but focuses on the number of cycles having exactly K visits in the random walk between a range of (-4, +4), and 15) *Random Excursions variant test* which is very similar to the Random Excursions test except within a range of (-9, +9).

C Hardware and Software Tools

This experiment requires the Cmod S7 board, Digilent Adept 2 v2.26.1 software, and a Jupyter Notbeook environment with Numpy. You will be provided the bitstreams for a TRNG, with which you will program onto the Cmod S7 board. You will then use the pySerial to interface with the FPGA.

You are provided a Python script **NIST.py** which calls separate scripts (located under "Code/tests" directory) for all of the 15 statistical tests as specified in [7]. This script provides a class **TRNGTester** which implements all 15 of the NIST tests on your input. You may instantiate an object of this class and use it to load and test multiple different binary inputs.

No.	Test Name	No.	Test Name
1	Frequency (monobit)	9	Maurer's "Universal Statistical" Test
2	Frequency within a block	10	Linear Complexity
3	Runs Test	11	Serial test
4	Longest run of 1s in a block	12	Approximate Entropy
5	Binary Matrix rank	13	Cumulative Sums
6	Discrete Fourier Transform (Spectral)	14	Random Excursions
7	Non-overlapping template matching	15	Random Excursions variant
8	Overlapping template matching	-	

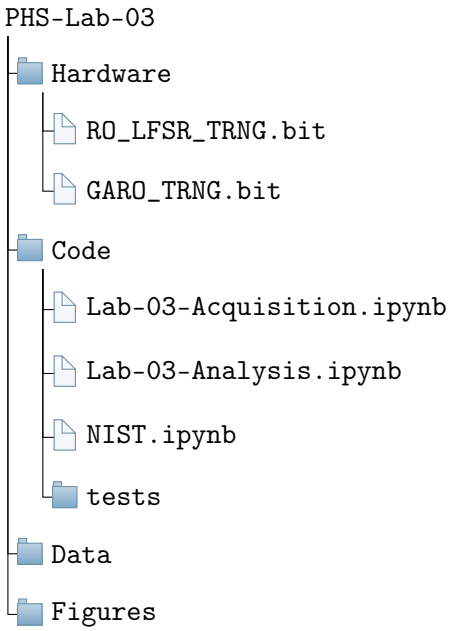
Table 1: All statistical tests recommended by the NIST Statistical Test Suite [7].

Reading Check

1. What kinds of applications are TRNGs used for?
2. What are the 3 major components of a good TRNG design?
3. How can we evaluate whether or not a TRNG is *truly* random?

D Getting Started

Download and open the Lab-03 directory. The directory is structured as follows:



There are two parts to this lab: data acquisition and data analysis. For data acquisition, you will program your Cmod S7 board with two different TRNG designs and sample some random bytes from each. The first TRNG bit file is located in "Hardware/RO_LFSR_TRNG.bit", and is implemented using 10 ring oscillators which are enabled and XOR'd using an LFSR. The second bit file is located in "Hardware/GARO_TRNG.bit", and XORs multiple Galois ring oscillators

(GAROs) – which is a hardware construct similar to a combined RO and LFSR. You will program the FPGA with one bitfile, sample the required amount of data from it, then do the same with the other bitfile.

To sample random bits, you will need to once again use the pySerial library to communicate with the board. *The provided controller expects as input a **3 bytes** which specifies how many bytes to expect from the FPGA.* In Python, convert this number to a `bytearray` in *big endian* format, and transmit this to the board using a serial `write()` function. The circuit will generate (pseudo)random data for you to read. You can use the serial `in_waiting` attribute to check that the FPGA is sending data in a loop, then accumulate all of the bytes across all the reads from the serial interface. Once the total number of bytes read exceeds the number requested, exit the loop, and save the data to file in the **Data** directory for further processing.

For data analysis, you will use the "NIST.py" script (located in Code directory) to pass your binary data through the NIST statistical test suite (STS). The results will indicate good/bad properties of the TRNG (evident by the bitstream). To use the NIST STS, import the "TRNGtester" class from "NIST.py" and create an instance of it. Its constructor takes as arguments: the path to your binary file (from the **Data** directory), the number of bits to read/test (-1 is default and means *all bits*). The parser expects your binary data to be in raw bytes format. You can write raw bytes to a file by using the "wb" mode descriptor when opening the file to write to. Once you have a "TRNGtester" object with your data loaded in it, you can call the "run_nist_tests()" function to call all 15 of the tests. The function will return a *list of list* containing the p-values for all the tests (sorted as in Table 1). Some of the tests return multiple p-values. The NIST test documentation is inconsistent on how to dictate pass/fail with multiple p-values – so just extract the *minimum* p-value from each test.

To reiterate, here are some high-level takeaways to consider while coding for this lab:

1) The TRNG design – while running in the FPGA – expects **exactly 3 bytes** as input to tell it how many random bytes to generate. So, if you want to generate 1000 bits, you would have to send it the value 125 (125 bytes = 1000 bits) formatted as 3 bytes in **big endian** format – in this case, $b'\backslash x00\backslash x00\}'$.
2) The TRNGs are designed to run at 10 MHz, which means it should only take around 10 to 11 seconds to generate 1M bits.
3) The TRNGtester expects your random data to be in *raw bytes format.* You can just write the bytes directly to a ".bin" file.
4) Some of the tests return multiple p-values. Just report the *minimum* p-value from each test.
5) Every single test in the NIST STS is specified as pass/fail based on the test p-value(s). A p-value of <0.01 means the test fails, and $>=0.01$ is a pass.

E Lab Assignment

In this assignment, you are given two TRNG designs and are tasked with implementing them on your FPGA board (any), gathering random binary data from them, and running the data through a Python implementation of the NIST framework. Ultimately, your results will lead you to conclude whether or not the TRNGs can be considered good enough for cryptographic applications. In practice, a TRNG must pass rigorous testing in order to be qualified for use in the field. We do *not* guarantee that the provided TRNG would be good for a practical cryptographic application. We

provide the same "assurances" of the cryptographic strength of the given TRNG that are provided in the NIST specification [7].

Note: you do not need to use both FPGAs. But you can use both FPGAs to spread the work.

For this assignment, you will write 2 Python scripts which accomplish the following:

1) The first script (data acquisition) reads random data from the FPGA using pySerial and saves the data to a file (e.g. RO_1M.bin). You are required to gather both 1 million and 10 million bits from each TRNG.
2) The second script (data analysis) utilizes the **TRNGTester** to read your binary data and runs the NIST STS on it.
3) Run the tests on the 1M and 10M bits from both the RO/LFSR TRNG and the GARO TRNG.
4) Plot the p-values in a vertical bar graph (`matplotlib.pyplot.bar`) and compare the results (4 total vertical bar charts). Annotate (`matplotlib.pyplot.annotate`) which test pass/fail on the respective bar.

   For the graduate students, we also require that you:

5) **Graduate Only:** On a new data acquisition script, generate random data (1M and 10M as well) from *two* other available PRNGs in Python. For example, using Numpy's random library and Python's built-in random number generator (generic random, os.urandom, etc.). *Remember to save them as raw bytes.*
6) **Graduate Only:** On a new data analysis script, repeat analysis steps 2-3 on this data, compare the results to what was measured on the FPGA for the TRNG implementations. You should have 4 additional bar charts and accompanying text in your report.

   Note: name the new scripts the same, but add "-PRNG" at the end; i.e., "Lab-03-Acquisition-PRNG.ipynb" and "Lab-03-Analysis-PRNG.ipynb", respectively.

F Badges and Achievements

This experiment involves **FPGAs**, **Data Processing**, and **Data Visualization**. By successfully completing the lab you can unlock the following achievements:

- FPGA2: Digilent Adept Tool
- FILE3: File Manipulation
- STAT3: Statistical Analysis
- PLOT3: Data Visualization

Depending on the demonstrated skill, each achievement will be further divided into bronze, silver, and gold levels.

G Deliverables

Set up your experiments along the guidelines described above, and organize your findings into a report according to the guidelines in the Appendix A. The report should also contain the following:

- Answers to the reading check questions.
- Bar charts with annotations of your results from the NIST tests.
- Discuss your findings/conclusions about the given TRNGs (and PRNGs for grad students). Is one better than the other? why or why not?

Submit your report along with all code. All analysis code should be runnable and should produce the output submitted with your report.

References Cited

[1] N. Bochard, F. Bernard, and V. Fischer, "Observing the randomness in ro-based trng," in *2009 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2009, pp. 237–242.

[2] V. Fischer, F. Bernard, N. Bochard, and M. Varchola, "Enhancing security of ring oscillator-based trng implemented in fpga," in *2008 International Conference on Field Programmable Logic and Applications*. IEEE, 2008, pp. 245–250.

[3] J.-L. Danger, S. Guilley, and P. Hoogvorst, "High speed true random number generator based on open loop structures in fpgas," *Microelectronics journal*, vol. 40, no. 11, pp. 1650–1656, 2009.

[4] S. Srinivasan, S. Mathew, R. Ramanarayanan, F. Sheikh, M. Anders, H. Kaul, V. Erraguntla, R. Krishnamurthy, and G. Taylor, "2.4 ghz 7mw all-digital pvt-variation tolerant true random number generator in 45nm cmos," in *2010 Symposium on VLSI Circuits*. IEEE, 2010, pp. 203–204.

[5] K. Yang, D. Blaauw, and D. Sylvester, "An all-digital edge racing true random number generator robust against pvt variations," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1022–1031, 2016.

[6] B. Sunar, W. J. Martin, and D. R. Stinson, "A provably secure true random number generator with built-in tolerance to active attacks," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 109–119, 2007.

[7] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.