CSCI474 - Course Project Proposal

Thomas Applegate
Colorado School of Mines
Golden, CO, USA
tapplegate@mines.edu

Kaelyn Boutin
Colorado School of Mines
Golden, CO, USA
kvboutin@mines.edu

Gabrielle Hadi
Colorado School of Mines
Golden, CO, USA
ghadi@mines.edu

Addison Hart
Colorado School of Mines
Golden, CO, USA
addisonhart@mines.edu

Et Griffin
Colorado School of Mines
Golden, CO, USA
egriffin@mines.edu

Isabelle Neckel
Colorado School of Mines
Golden, CO, USA
ineckel@mines.edu

Abstract—**The National Institute of Standards and Technology, NIST, defines fifteen separate tests for the randomness and unpredictability of random numbers [1]. True Random, Deskewed True Random and Pseudorandom are the three classes of random numbers that are going to be compared using the tests outlined by NIST. We plan to use a Python implementation that will provide us the P-values to compare and contrast to determine the random numbers that are closest to being truly random. The generation of the random numbers will also be considered when discussing the choice in relation to cryptography.**

Index Terms—**random numbers, pseudorandomness, NIST, python**

I. Introduction

Random number and bit generation is used heavily in cryptography and security applications, but it is difficult to get truly random numbers. Effective randomness is necessary to ensure security in cryptographic applications, as cryptanalytic attacks exist which can leverage patterns in encrypted data to decrypt without the necessary credentials. For example, a team led by Nadia Heninger found that poorly seeded pseudo random number generation, or PRNG, could be used at scale to derive RSA private keys [2].

The solution is not as simple as exclusive usage of true random numbers. Because the generation of true random numbers, or TRNG, is computationally expensive and requires a source of entropy, which is limited, TRNGs cannot be used for all the places in cryptographic algorithms where randomness is necessary, like the usage of nonces in OFB or initialization vectors in CFB, especially if large amounts of data is encrypted or the software is running on constrained devices. TRNGs may also be vulnerable to adversarial modification of the environment by the attacker, like changing the temperature of a device if the source of entropy is a temperature reading [3]. Additionally, an attacker may be able to leverage the loss in randomness as soon as the entropy pool empties, which can happen extremely fast if it is the only source of randomness used. Because of this the standard today is to utilize TRNGs only for the generation of seeds for PRNGs [4].

The purpose of our research is to evaluate the performance of three classes of randomness outlined by NIST using their statistical tests, named the SP 800-90. The SP 800-90 was created to provide guidelines for the generation of pseudo random numbers for cryptographic use. There are 3 parts to the SP 800-90 series: 90A talks about mechanisms for the generation of random bits using deterministic methods, 90B discusses how valid certain ideas of randomness and entropy are, and finally 90C specifically talks about construction and implementation of random bit generators. Random Bit Generation must be checked for randomness, so a total of fifteen statistical tests were developed and evaluated in order to determine the validity of a generator's statistical randomness. The fifteen tests are as follows

1) Frequency (monobits) tests: This test determines whether the number of ones and zeros generated in a sequence are approximately the same as should be generated for a true random sequence.
2) Test for Frequency Within a Block: This test tries to determine if the frequency of ones in a M-bit block is approximately half the size of the block.
3) Runs Test: This test determines whether switching between substrings is too fast or slow.
4) Test For Longest Run of Ones in a Block: This

test determines if the longest runs of ones in a block is similar to the longest run in a truly random generation.

5) Random Binary Matrix Rank Test: This test checks for linear dependence in a fixed length substring.
6) Discrete Fourier Transform (Spectral) Test: This test detects periodic features that could indicate the lack of randomness.
7) Non-Overlapping (Aperiodic) Template Matching Test: This test is used to reject sequence that show too many occurrences of a given non-periodic pattern (similar to the zero's run but allows for statistical independence amongst tests).
8) Overlapping (Periodic) Template Matching Test: This test is used to reject sequence that show changes from the expected number of runs of ones of a given length.
9) Maurer's Universal Statistical Test: This test detects whether or not a sequence can be compressed without loss of information, a overly compressible sequence is considered non-random.
10) Linear Complexity Test: This test determines if a sequence is complex enough to be considered random.
11) Serial Test: The test determines the number of occurrences of 2m m-bit overlapping patterns is what should be expected for a random sequence.
12) Approximate Entropy Test: This test compares the frequency of overlapping blocks of conservative lengths against what is expected for a random sequence.
13) Cumulative Sum (Cusum) Test: This test determines if the cumulative sum of partial sequence in the test sequences are too large or too small relative to random sequences.
14) Random Excursions Test: This test is used to determine if a number of visits to a state within a random walk except what is expected for a random sequence.
15) Random Excursions Variant Test: This test detects deviation from the expected number of occurrences of various states in a random walk.

The goal of this research is to compare and contrast different random generators using these fifteen tests. Our research will compare True Random, Deskewed True Random, and Pseudorandom, to see how these classes stack up against one another and compare the time cost of each. Using a calculated P-value we can evaluate the results comparatively against each class. We also intend to show why some algorithms necessitate complexity, like hardware input or SHA hashing, and which algorithms are unnecessarily computationally intensive or complicated for the quality of their output. These results will provide insights into the strengths and weaknesses of each class, enabling cryptographers to make informed decisions on which class of RNG is suitable for their use case.

II. Methodology

To test the randomness of various algorithms, we will use the NIST randomness tests as implemented in Python [5]. The tests require a binary string of indeterminate length as the input. Each of the fifteen tests will output the P-value, as well as the result of whether the P-value means that the data can be considered truly random or not. For each of the algorithms that we test, we will use many samples of equal length to compare their randomness to each other.

III. Expected Results

We do not expect True Random numbers to perform the best, as despite their quality they lack a feature that Pseudorandom number generators are designed to have: follow an asymptotic distribution. True Random numbers are not generated to fit an asymptotic distribution so in practice they have significant skew, or bias, decreasing their entropy and making them perform worse in objective tests of quality. This is why we are testing Deskewed True Random numbers as well, as it reduces the most significant weakness of TRNGs. With this in mind we expect Deskewed True Random to perform the best, with True Random following and then Pseudorandom. We expect, however, that Pseudorandom will be found to be the fastest RNG, with Deskewed True Random and True Random tied afterwards.

IV. Experimental Investigation

There were a few stages to investigate the possible RNG classes against the NIST Randomness Tests: validating the code and collecting random data. These stages are detailed in the subsections to follow.

A. Validating Code

We started with python implementation of the NIST tests, NistRng, created by SAILab at the University of Sienna [6]. To validate the code, we conducted a code review, comparing the implementation to the NIST publication. We were able to find a number of errors in the implementation, all but one of which had fixes implemented by a GitHub user, zazuza7 [7], in a fork of the original library. A second code review was conducted on this implementation to verify that the noticed bugs were fixed. We could only find a single error, which we fixed in our own fork.

This review also revealed that our testing suite had a few requirement differences to the NIST specifications. Table I shows the inconsistencies found in the bit length requirements of the NistRng implementation and the NIST recommendation. We did not find an issue with these inconsistencies for our analysis, since we would be testing against data of larger than both requirements. It was also found that for the Non-Overlapping Pattern Matching Test that the NIST recommended parameters for pattern length were not quite met. NIST recommends (but notes that it does not require) using a pattern length of 9 bits, where NistRng only uses 8 bits to save on computation time. Including this extra bit added 10 minutes to the runtime of the tests, so we opted to leave it at 8 for our tests.

Test	NIST	NistRng
Matrix Rank	38912	9728
Mauer’s Universal Statistical	$\approx 4 * 10^6$	No Limit
Approximate Entropy	No Limit	512

Table I: Bit Requirements for NIST standard and NistRng Implementation

B. Collecting Data

We collected a set of large and small random numbers. Small length random numbers or length 256 bits were used for ease of use for the majority of tests that did not require large numbers, and large random numbers of length $4 * 10^6$ were used to assess against the whole testing suite.
True random numbers were collected using an online source of true random numbers. Originally we were using RANDOM.ORG, which uses astronomical noise to create a uniform set of random bytes [8]. However, this source was not able to produce enough true random bits for our large number requirement. For this, we used another source codebeautify.org [9]. This source claims to produce uniform true random numbers as well, however we could not verify the claim and this may have impacted the results.

Psudorandom numbers were collected using the python Random and numpy libraries. This mimics non cryptographic uses of PRNG. We used randint to generate 0 and 1 for the appropriate amount of bits for each experiment.

We were unable to collect a reliable source of skewed random numbers, as online resources all claimed to create uniform random numbers. For this, we simulated skewed random numbers using true random numbers to seed a normal distribution. From a set of normally distributed numbers, we used a cryptographic hash to deskew the data. This provided an approximation of deskewed random numbers for us to test against. We were unable to generate large deskeweed random numbers as the hashes available would not produce large enough results.

V. Analysis

Each set of random numbers was run against all eligible tests and the average score, time and tests passed were recorded. See table II. The tests that could not be run due to insufficient bit length were binary matrix rank, overlapping template matching, maurers universal and linear complexity. For a bit length of 256, the true random numbers achieved the highest score but also passed the least amount of tests. The pseudorandom numbers had the lowest score but passed the most tests. The deskewed numbers had a high score and passed a good amount tests but did take a little longer to run. Overall, the deskewed random numbers preformed the best and are easier to generate than true random numbers.

	2^8			$4 * 10^6$	
	TR	PR	D	TR	PR
Score	.4913	.3765	.4836	.1785	.2337
Time(ms)	1.45	1.27	3.33	21768.73	28864.13
Passes	8/11	10/11	9/11	4/15	6/15

Table II: Results of Tests

When comparing the long random numbers, they both preformed worse overall than the smaller random numbers. Pseudorandom numbers had a higher score and passed two more tests than the true random numbers and both took extremely long to run.

VI. Limitations and Future Research

Some notable limitations with these tests, outlined by NIST, are:

1) The neccesity of large enough samples. In the case of the Maurer’s Universal Statistical test, the large sample size was neccesary to find the asymptotic distribution of the data. We had difficulty generating a sufficient amount of bytes of TRNG for the $4 * 10^6$ byte tests.

2) The inflexibility of using explicit values for P-value significance. There is a lack of research into what significance level is neccesary for security, so NIST chose values for illustrative purposes. They state this when they say "For the examples in this document, [the significance level] has been chosen to be 0.01. Note that, in many cases, the parameters in the examples do not conform to the recommended values; the examples are for illustrative purposes only" [1].
3) The inflexibility of a fixed list of limited tests. NIST has stated their intention to keep this list up to date when new flaws in methodology become known, but it remains that this test suite cannot say for certain whether a source of randomness is sufficient, the tests in combination are hoped to be sufficient.
4) The quantity of parameters that may be tweaked during testing. The test suite requires choosing a sequence length, sample size and block size, among other parameters which are best effort. They provide recommendations, like the recommendation for sample size being the inverse of the significance level, but an implementer of the test suite may choose to modify parameters until the results look the best, otherwise known as P hacking. Without a standard on what these parameters must be the test results cannot be compared between different implementers and the potential for implementation error increases.

There are also limitations with our analysis. The generation of our random numbers likely affected the results. It is difficult to generate true random numbers and know that they have a sufficient amount of entropy. Furthermore, the different lengths of true random numbers in our experiment cannot truly be compared since they were generated from different sources.

For future work, the experiment would be repeated with larger sets of random numbers with additional lengths being tested. The random numbers would be generated with a Linux operating system and openssl would be used to hash larger bit lengths. These changes would allow us to compare sources of random numbers with less confounding variables due to data collection.

VII. Conclusion

NIST provides 15 tests to compare and contrast random number generators. These tests promote different qualities of ideal random numbers. We have used python libraries for the NIST tests, and various pseudorandom and deskewed true random number generating algorithms to compare them to true random numbers. These tests are important for the verification of randomness used in cryptographic systems, and we some of the implementations we looked at did not provide correct results. We recommend that authors of software implementing these tests should strive to better validate their solutions given the importance of this test suite in cryptography.

We will summarize the results of these comparisons, and then advise on the generating algorithms suitable for cryptographic use.

References

[1] A. Rukhin, J. Soto, J. Nechvatal, *et al.*, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications* (NIST special publication ; 800-22), eng. U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, Apr. 2010.

[2] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your ps and qs: Detection of widespread weak keys in network devices," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 205–220.

[3] B. Barak, R. Shaltiel, and E. Tromer, "True random number generators secure in a changing environment," in *Cryptographic Hardware and Embedded Systems-CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings 5*, Springer, 2003, pp. 166–180.

[4] K. Lee, S.-Y. Lee, C. Seo, and K. Yim, "Trng (true random number generator) method using visible spectrum for secure communication on 5g network," *IEEE Access*, vol. 6, pp. 12 838–12 847, 2018. DOI: 10.1109/ACCESS.2018.2799682.

[5] S. Ang. "Randomness testsuite." (), [Online]. Available: https://github.com/stevenang/randomness_testsuite (visited on 03/14/2024).

[6] L. Pasqualini. "Sailab nistrng." (), [Online]. Available: https://github.com/InsaneMonster/NistRng/blob/master/README.rst (visited on 03/23/2024).

[7] zazuza7. "Nistrng." (), [Online]. Available: https://github.com/zazuza7/NistRng (visited on 03/24/2024).

[8] "Random.org - true random numer service." (), [Online]. Available: https://www.random.org/bytes/.

[9] "Code beautify." (), [Online]. Available: https://codebeautify.org/.