2024 Future Computing Summer Internship
at the
Laboratory for Physical Sciences (LPS)

An Analysis of Entropy and Randomness as it relates to True Random Number Generation

Nicole Baker
Joshua Hatfield
Kyrill Serdyuk
Boris Strots

Mentors:
Dr. Tyler Simon
Daniel Gauthier
Wesley Lui

August 9, 2024

ABSTRACT

Random numbers are a key component of many algorithms and cryptographic applications. However generating truly random numbers is not a trivial task. This paper demonstrates how physical phenomena can be observed in a variety of ways, like through sensors and hardware, as a method for generating entropy and random numbers. These devices are known as true random number generators. The true random number generators are evaluated based on speed in bits per second, their energy usage per bit, and how many NIST SP800-22 randomness tests they pass. The XOR Ring Oscillator preformed the best with a speed of 7.2 Mbps and pass rates from 98-100%. The XOR Ring Oscillator preformed favourably when compared to other hardware based true random number generators.

Keywords:

PRNG: Psuedo Random Number Generator
TRNG: True Random Number Generator
NIST STS: National Institute of Standards and Technology's Statistical Test Suite
FPGA: Field Programmable Gate Arrays

TABLE OF CONTENTS

1 INTRODUCTION

FCSI is so awesome! We learned about the relationships between Entropy[1] and some of the difficulties in computing Randomness[2]. We also compared our results with a True Random Number Generator (TRNG)[3], and our performance was better. We investigated many methods to generate random numbers through both deterministic means and harnessing entropy from physical phenomena, both generated different results. The key difference being that using physical sources to generate random numbers is a more cryptographically secure as the method of generation cannot be reproduced. With deterministic means, the seed value of the generator can be determined and the distribution of data can be calculated. This provides a creates unnecessary risk in RNG (random number generators) which can be holistically avoided by achieving true randomness. Throughout this paper, we discuss both methods we used and their statistical importance via their ratings in entropy, randomness, power consumption, and bit-stream generation rates.

2 BACKGROUND

Below we describe some of the Randomized algorithms that the team implemented. One of the goals of the Internship was to evaluate the performance of these with various hardware RNG's and entropy sources.

2.1 Monte-Carlo

The Monte Carlo method is a general technique that uses repeated random sampling as a way to obtain a numerical result of a problem or equation. Monte Carlo is commonly found where it is too difficult or impossible to use an analytical solution to derive the numerical result. A few examples are numerical integration and sampling Bayesian priors using Monte Carlo Markov Chains.

Since repeated sampling depends on a random number generator to create create new samples, the quality of the numeric result is tied to the quality of the random numbers. A common test is the approximation of π, which calculates the ratio of points under the curve inside a unit square. Better random numbers lead a more uniform distribution of points across the unit square which leads to less error in the numerical approximation.

Each team member implemented solutions to a three Monte Carlo problems to get a better understanding of the Monte Carlo method and for the problems to be used as rudimentary benchmarks for random numbers.

2.2 Frievald's Algorithm for Matrix-Multiplication

Frievald's Algorithm for Matrix-Multiplication is an example of a probabilistic algorithm. To determine $A * B = C$ for some matrices A, B, C, Frievald's runs in $O(n^2)$ with 2^{-k} probability of failure for k iterations while traditional deterministic algorithms only run in $O(n^{2.3739})$. A single iteration utilizes a random numbers to approximate whether $A * B = C$ and can erroneously accept the equality when its not actually true. By running the algorithm for multiple iterations, and only accepting if every iteration accepts, the probability of failure is minimized while still being faster than deterministic methods. But the probability of failure being 2^{-k} is only true for a uniform distribution of random numbers, as otherwise there is no guarantee that each iteration k checks a different set of random numbers.

2.3 Fisher-Yates Shuffle

We investigated the Fisher-Yates(FY) shuffle algorithm (also known as Knuth shuffle) because it is commonly used in applications where a random permutation of elements is required. The following algorithms all employ a FY shuffle:

- Sorting Algorithms with Randomized Partitioning:

  QuickSort with Randomized Partitioning: QuickSort can use the Fisher-Yates shuffle to randomize the pivot selection during partitioning. This helps avoid worst-case scenarios and improves average-case performance.

- Random Sampling:

  Algorithms that require random sampling of elements, such as selecting a random subset from a larger set, often use the Fisher-Yates shuffle to achieve unbiased randomness.

- Simulation and Monte Carlo Methods:

  In simulations and Monte Carlo methods, where random permutations of data or randomization of inputs are necessary, the Fisher-Yates shuffle is used to ensure randomness and statistical validity.

- Cryptographic Applications:

  In cryptographic applications or security-related algorithms where secure shuffling is required, adaptations of the Fisher-Yates shuffle may be used to ensure unpredictability and randomness.

- Machine Learning and Data Science:

  In machine learning and data science, especially during cross-validation or bootstrapping procedures, the Fisher-Yates shuffle is used to randomly shuffle data or partitions to ensure unbiased model evaluation.

- Game Development:

  In game development, especially in card games or board games, where shuffling of decks or tiles is required, the Fisher-Yates shuffle ensures fairness and unpredictability of the game outcomes. In summary, the Fisher-Yates shuffle is versatile and widely used in various algorithms and applications where random permutation of elements or secure randomization is needed. Its simplicity and efficiency make it a preferred choice in many contexts where randomization plays a crucial role.

2.4 Stochastic Arithmetic

- Stochastic Multiplication

  Suppose you have two random bitstreams, with the probabilities of a 1 appearing in bitstreams A and B being P_A and P_B respectively. If these bitstreams are fed into an AND operation to form bitstream C, then a 1 will be output whenever both A and B have a 1 in that position. The probability of both A and B having a 1 in a certain position is $P_C = P_A * P_B$. So, by measuring P_C, the product of P_A and P_B can be determined.

2.5 Entropy

For our studies we used Shannon entropy, as defined by Claude Shannon, which is $E = -\sum_i P(i) \cdot \log_2(P(i))$ where the sum is performed over every possible outcome in the sample space. Shannon entropy quantifies the amount of bits needed to represent information and is highly relevant in compression of data. Entropy is maximized when each outcome has equal probability. Therefore, ideal random data, which has a uniform probability distribution, has high entropy and is practically impossible to compress.

2.6 Von Neumann Extractor

We found some bias introduced in our experiments. This bias took the form of relatively long sequences of the same bit, 0's or 1's. In order to address this we introduced a filter or extractor. A Bernoulli trial is a random measurement that has 2 possible outcomes. A common example is a coin flip. For computers, this could be used to generate a random bitstream where 0 is equivalent to tails and 1 is equivalent to heads. A bitstream would have a maximum entropy of 1 bit per bit, which is achieved when the probability of 0 is equivalent to the probability of 1, which are both 0.5. However, what if the Bernoulli trial that's used as a source of randomness doesn't have the equal probability for the two outcomes? What if the coinflip is weighted? von Neumann[4] invented a way to remove this bias and normalize the probabilities, which works as follows: Generate 2 bits of data. If the 2 bits are 00 or 11, do nothing and drop those bits. If the 2 bits are 10, then write a 1. If the 2 bits are 01, then write a 0.

Suppose the probability of a 0 is p, while the probability of a 1 is (1-p). Then the probability of 00 is p^2. The probability of 11 is $(1-p)^2$ The probability of 10 and 01 are $(1-p)p$ and $p(1-p)$ respectively, which are equivalent. Thus, the probabilities of 0 and 1 in the processed bitstream will be equivalent. However, the von Neumann Extractor causes a non-constant time for each random bit to be generated. The time to generate 1 bit of random data grows as p moves farther away from 0.5.

2.7 XOR

The exclusive or operation is a binary operation with the following truth table.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

It's well known that the sum of independent, identically-distributed (iid) random variables tends toward a normal distribution. Similarly, taking the XOR of iid random variables will tend toward a uniform distribution [5]. This is a powerful way of using multiple entropy sources with imperfect randomness and combining them to create more entropic data.

2.8 Arduino

The Arduino is a popular open source micro controller that can interact with a variety of sensors and external modules. The Arduino has an ARM CPU running an extremely minimal operating system. The Arduino IDE compiles C++ using Arduino-specific libraries. Data communication and

transfer between the Arduino and a computer was done through a UART serial connection.

2.9 Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays are embedded systems that host a programmable logic block utilized for holding different circuits. This form of adaptive hardware makes it relatively simple for a plethora of different circuits to be synthesized and implemented using a HDL (hardware description language).

2.10 Statistical Test Suite and Dieharder

The Statistical Test Suite (STS) for Random and Pseudorandom Number Generators for Cryptographic Applications[6] is a comprehensive collection of statistical tests developed and curated by the National Institute of Standards & Technology. The tests check for both the validity and quality of the random numbers backed by hypothesis testing.

We used the STS as a more advanced and holistic metric of quality and true randomness than our Monte Carlo simulations. The percentage of tests passed, from a total of 188 test iterations, provided a simple and comparable metric for different collections of random numbers. We used a graphical application and STS backend developed by our mentors at LPS that required at least 1 MiB of random data to test.

The Diehard test suite is an older test suite originally created by Donald Knuth and later adapted by Robert G. Brown as Dieharder [7]. These add various new tests and require significantly more random data, more than 100 MiB to give accurate feedback about the entropy source. At a high-level, the Dieharder tests perform numerous Monte Carlo simulations using the random data and checks if the result falls close enough to the expected analytical values as a way of grading randomness.

3 MONTE-CARLO PROBLEMS

To begin the research venture, we were given 21 Monte Carlo to choose from [8] and we each chose 3 problems and wrote Monte Carlo simulations in C to find the approximate results.

3.1 A Parallel Parking Question (#3)

Suppose there are $n >= 2$ cars parallel parked in a long straight lot of length L. If you represent these cars as a point on a line of length L , then the position of the car can be written as the distance from the origin in the range of 0 to L. Each car will have only one nearest neighbor and two cars that have each other as their nearest neighbor will be considered mutual nearest neighbors. What is the probability that a randomly selected car is part of a mutual nearest neighbor pair?

Obviously, in the case of 2 cars, the probability of picking a mutual nearest neighbor is 1. And in the case of 3 cars, there is always 1 mutual neighbor pair meaning 2 out of 3 cars are mutual neighbors, so the probability is then $\frac{2}{3}$. The simulation to determine the probability for n > 3 cases involved generating n numbers from 0 to 1 then sorting the numbers from smallest to largest. Then, an array storing the index of each car's nearest neighbor was created. The number of cars that were mutual nearest neighbors were counted, then divided by the total number of cars to find the probability. In all cases $n > 2$, the probability of selecting a car that was a mutual nearest neighbor was $\frac{2}{3}$.

3.2 The Gamow-Stern Elevator Puzzle (#5)

Suppose a building with seven floors has an elevator that travels up and down the building continuously. When calling the elevator from the second floor to go up, what percentage of the time will the first elevator to arrive be going down? What about for more than one elevator?

Donald Knuth showed that the first elevator to arrive at the second floor is going down with probability $\frac{1}{2} + \frac{1}{2}(\frac{2}{3})^n$ [8]. For a single elevator, it will be going down 83.3% of the time. For two, it would be going down 72.2% of the time.

The simulation generalized the floors such that the bottom or first floor was 0 while the top or seventh floor was 1. From there, elevator(s) were generated with random position between 0 and 1 along with random direction (up or down). Then, the distance of each elevator is calculated and whether it is above or below the second floor. The elevator with the smallest distance will be the first to arrive. When simulated for 100,000 rounds, the simulation produced $n = 1$: 83.3%, $n = 2$: 72.2%, $n = 3$: 64.6%.

3.3 Steve's Elevator Problem (#6)

Steve works in a building with an elevator that makes 11 stops. Everyday Steve takes the elevator to the 9th stop and gets off there. He notices that the number of stops the elevator makes while he is in it increases when there are more people in the elevator. What is the average number of stops Steve will make while in the elevator when there are different numbers of people?

The simulation created an array the length of the number of people in the elevator other than Steve. It then assigned them an integer between 1 and 11 representing the floor the person will get off at. It then counts how many stops the elevator will make, starting from one since we know it will stop at least once when Steve gets off, then iterating from 1 through 8. If the integer is in the array then it increments the number of stops. It then repeats this process for the number of trials specified and divides the total sum by the number of trials to find the average number of stops.

3.4 The Pipe Smoker's Discovery (#7)

A pipe smoker has 2 boxes of matches filled with 40 matches (80 total). Each time they smoke, they take a match from either box randomly. The objective is to determine how many matches will be used before the smoker runs out of matches on either of the boxes. The Monte-Carlo implementation involves 100 iterations of running this scenario and averaging the results. At first glance, the problem appears quite trivial, but it does not have one single answer.

For 100 simulations, it can be observed that depending on the dataset, the number of matches used can vary somewhere between 62 and 79. However, once the number of trials is increased, the average number of matches appears to lean towards the higher part of the interval. In fact, over millions of runs, it is common for the number of matches used to hover equal to or right below 79 matches. The solution to this Monte Carlo problem can be achieved by instantiating two 40 match box variables and generating a random number to determine which box is selected during each decision point. By keeping track of the number of matches used, you can then get an average using the total divided by the number of simulations.

3.5 The Forgetful Burglar's Problem (#9)

A burglar plunders homes on a linear path, but they struggle to remember which house they have already robbed. In this linear plane, the robber has two decisions to make at every interval: either to go 1 or 2 steps and whether to go left or right. The goal is to determine the likelihood for each k steps ($1 <= k <= 7$) for the robber to run into a house that they have already visited.

The first step is simple enough, the robber has just begun their spree, so there is a 0 percent chance that they would run into a house they have already been to. For the second step, there is a 25 percent chance that the burglar has already ran into their previously burglarised home. From here, the probability of the other steps can be achieved by keeping track of the previous steps and using that information to calculate the likelihood of visiting the previously visited houses.

3.6 The Case of the Missing Senators (#11)

The U.S. Senate is going to vote on a bill and in this scenario it is known that there are more senators for the bill than against the bill. So, we have A senators against the bill in which $A < 50$. The catch is that there are also M senators who will miss the vote on the bill for a random scenario. A senator missing from the vote does not have anything to do with whether or not they are for or against the bill. The randomness of this problem does not necessarily employ any random generation during the simulation; instead, the randomness would occur with the arguments provided for both A and M.

The solution can be achieved by finding $100 - A = Y$. Essentially, you would multiply the chance of this probability of a senator being for the bill. So $\frac{Y}{total} * \frac{Y-1}{total-1} * ... * \frac{Y-M+1}{total-M+1}$. This equation will reach the example solution where $A = 49$ and $M = 3$.

3.7 How Many Runners in a Marathon? (#12)

Suppose there are N runners running a marathon. Each runner is wearing a bib with a number on it numbered from 1 to N. As a spectator, you can read the number on the bib of each runner. If you write down the bib number of n runners and find that the highest number you recorded is E, then you can approximate the number of runners in the race using the formula $\frac{n+1}{n}(E-1)$. If you take the number of samples n as a percentage of the population, how does increasing the sample

size percentage affect the percent error between the predicted number of runners and the actual number of runners?

To estimate the number of runners in the marathon, we made a Monte-Carlo simulation for a race of 1000 runners. We created an array with numbers from 0 to 1000, then created a random permutation of those numbers. We then took n samples from the array and recorded the largest number in the sample and plugged it into our formula to predict the total number of runners. The percent error between the predicted and actual values decreased as sample size percentage increased.

3.8 A Police Patrol Problem (#13)

A long, straight stretch of road needs to be patrolled by a police car to be monitored for accidents. There are a few possible ways that the police could patrol the road, for example: the patrol car can sit at the midpoint of the road and wait for a call, it can continuously travel up and down the road waiting for a call, or multiple cars (2) can continuously travel up and down with the nearest responding to the accident.

It can be further complicated by considering two different scenarios, one where the road is divided by a grassy median that can be driven across, or when the road is divided by a concrete barrier that cannot be driven across. Given that accidents occur at random positions on either side of the road, which patrol strategy has the lowest response time across both scenarios?

Table 1: Average Time to Accident

Strategy	Median Grass	Median Concrete
Stationary	1/4	1
One Patrol Car	1/3	1
Two Patrol Car	1/5	2/3

3.9 Waiting for Buses (#17)

There are two independently operated bus lines in the city. One of the bus lines always arrives at the bus stop strictly on the hour 6 a.m., 7 a.m., 8 a.m., etc. The other line arrives at $(6+x)$ a.m., $(7+x)$ a.m., $(8+x)$ a.m., etc., where x is a positive constant. What is the average wait time a person who arrives to the bus stop at a random time should expect? What about for more than 2, like n, buses?

The average wait time follows the formula $Avg(Wait) = \frac{1}{n+1}$ where n is the number of busses. For 1 bus, the average wait time is 30 minutes. For two, the average wait time is 20 minutes.

3.10 Waiting for Stoplights (#18)

A pedestrian begins at $(m+1, m+1)$ and walks toward $(1,1)$ by moving either left or down. At every intersection, the pedestrian will encounter a green light in one direction and a red light in the other direction. At each intersection, the lights are random with probability 0.5 for the lights to be oriented in each direction. Once the pedestrian reaches $(1,n)$ or $(n,1)$ for some n, the pedestrian only walks in a single direction towards $(1,1)$. If the pedestrian encounters a red light going in that direction, the pedestrian must stop and wait until a green light. How many red lights, on average, will the pedestrian need to wait for on the journey from $(m+1, m+1)$ to $(1,1)$?

The problem at a given location can be broken into the problem at a closer location. At the

boundaries, $(1, y)$ and $(x, 1)$ for all x and y, the average number of red lights equals half of the remaining intersections remaining:

$$E(1, y) = \frac{1}{2}(y - 1)$$

$$E(x, 1) = \frac{1}{2}(x - 1)$$

Then, at each point, there is a 1/2 probability to go either way. So,

$$E(x, y) = \frac{1}{2}E(x - 1, y) + \frac{1}{2}E(x, y - 1)$$

Some values of m, for starting at $(m+1, m+1)$ are listed with their analytical solutions and Monte Carlo solution.

Table 2: Average stoplights waited for. 1000 iterations using PRNG for Monte Carlo.

m	Analytical Solution	Monte Carlo Solution
2	0.75	0.7450000000
5	1.23046875	1.2309999000
10	1.762	1.8000000000
20	2.5074	2.5569999000
50	3.97946	3.9790001000
100	5.6348479	5.4540000000

3.11 An Optimal Stopping Problem (#20)

Someone is dating and hoping to find their ideal partner from a pool of k potential partners. Each partner corresponds to a distinct numerical value that determines how good of a match that partner is, with 1 being the worst and higher numbers indicating a better match. The order of these partners is a random permutation. To find the best partner, the dater employ the following strategy: They try dating the first m people and eventually end the relationship each time. While doing this, they note what their best experience was. After the first m relationships, they continue dating and stop once they experience a relationship that was better than their best experience from the the first m relationships. What is the optimal sample size m such that the probability of the final partner being the best possible partner is maximized?

The probability that the best person is chosen is the probability that the jth person is the best person multiplied by the probability that the jth person is selected, summed over all possible selections. The probability of the best person being in the jth position is $1/k$. For sample size m greater than 1, the probability of the jth person being selected is the probability that the best person of the first $j - 1$ dates is within the the first m dates (otherwise, that person would be selected instead of the very best person of the population). So the probability of the best person being j and selecting that jth person is:

$$\frac{1}{k}\frac{m}{j-1}$$

Summing over all possible j selections results in

$$P(m) = \sum_{j=m+1}^{k} \frac{1}{k}\frac{m}{j-1}$$

Evaluating this expression for all possible sample sizes m for a population size k will determine the sample size with the best probability

For a population of 11 people, the theoretical probability of choosing the best person for various sample sizes is compared with the Monte Carlo probability.

Table 3: Probability of finding best person with population size of 11. 5000 iterations for Monte Carlo simulation, PRNG.

m	Analytical Solution	Monte Carlo Solution
0	0.0909	0.0883999990
1	0.2663	0.2696000000
2	0.3507	0.3458000100
3	0.3897	0.3826000100
4	0.3984	0.4050000000
5	0.3844	0.3890000000
6	0.3522	0.3468000000
7	0.3048	0.3066000000
8	0.2444	0.2424000100
9	0.1727	0.1750000000
10	0.0909	0.0890000020

3.12 Chain Reactions, Branching Processes, and Baby Boys (#21)

Assume that the number of male children a male produces, c, adheres to the following probability distribution:

$$P(0) = 0.4825$$

$$P(c) = 0.2126(0.5893)^c, c >= 1$$

What is the probability distribution for the number of sons born in each generation, beginning from a single male?

An analytical solution to this problem can be determined by using the method of generating functions. Associate the coefficient of x^c with the probability of c sons being born in a certain generation. Applying this to the probability distribution results in the following generating function for the first generation:

$$f_1(x) = 0.4825 + 0.2126 * 0.5893x + 0.2126 * (0.5893x)^2 + ...$$

After the first term, this is an infinite geometric series and the function is equivalent to the following:

$$0.4825 + \frac{0.2126(0.5893x)}{(1 - 0.5893x)} = \frac{0.4825–0.0717x}{(1–0.5893x)}$$

It's been shown that f(x) can be applied to itself to create the generating function for subsequent generations.

$$f_2(x) = f(f_1(x)) = \frac{.4825–0.0717(\frac{0.4825–0.0717x}{1–0.5893x})}{1–0.5893(\frac{0.4825–0.0717x}{1–0.5893x})}$$

$$= \frac{0.4479–0.2792x}{0.7157–0.5470x}$$

$$= \frac{0.4479}{0.7157} + \frac{0.0452}{0.7157^2}x + \frac{0.0452(0.5470)}{0.7157^3}x^2 + \frac{0.0452(0.5470)^2}{0.7157^4}x^3 + \cdots$$

where the final result can be determined by performing long-division. Performing the same procedure can be performed for future generations.

Table 4: Probability for number of sons at specific generation.

Generation Number	Number of Sons	Analytical Solution	Monte Carlo Solution
3	6	0.0205	0.0197500010

4 ARDUINO

Multiple sources of entropy/randomness were connected and tested on the Arduino, ranging from sensors, PRNG, and hardware quirks. Our goal was to capture the inherent randomness in the physical world to generate random data.

4.1 Watchdog Timer Clock Drift

The Arduino contains multiple hardware timers, including the watchdog timer. These crystal oscillators have naturally occurring drift, where they fall out of sync over time. This timer jitter, or inconsistent trigger times, can be used as a source of randomness.

Despite trying a variety of methods, like using total clock drift, using the jitter of every interrupt, 8-bit or 64-bit multiplication, etc., the results using the watchdog timer for randomness were never good. While the data seemed to have high entropy, hovering around 7.95 bits per byte, fully random data needed to be around 7.9999 bits per byte. When tested on monobit, the first STS NIST test, the majority of data samples had uneven amounts of 0 and 1 bits, failing monobit. Occasionally, about once every ten runs, the random data was able to pass monobit. The NIST test suite states that all tests after monobit assume that monobit passed. Since the data rarely passed monobit and doesn't achieve ideal entropy, clock drift is not a suitable source of randomness.

Table 5: WatchDog Timer Configuration Options

16ms	32ms	64ms	125ms	250ms	500ms	1000ms	2000ms	4000ms	8000ms

Table 6: Clock Drift Entropy

Name	Entropy	Pi Approx Error	NIST STS
Overflow Drift	7.9509	0.63	Too Small :(
No Drift Correction	7.9958	8.26	Too Small :(
FNV	7.9995	0.08	149/188

The least significant bit of the jitter appeared to be semi random, but when that bit is only generated every 16 milliseconds, its very slow to collect (62.5 bits per seconds). It was hypothesised that the least significant bit could be used to seed a pseudo random number generator as a way of getting good results.

On occasion the random numbers could appear to be random. When looking at a Monte Carlo simulation, specifically the Gamow-Stern Elevator Puzzle (#5) for 1 elevators, the numbers generated by the clock drift seemingly get the right answer (going up: 0.1729, going down: 0.8271) compared with rand() (going up: 0.1680, going down: 0.8320). But under further scrutiny, the random

numbers get the correct answer just on happenstance. When testing a different configuration of elevators the results were incorrect (going up: 0.1660, going down: 0.8340) versus (going up: 0.2783, going down: 0.7217).

Applying a simple hash like Fowler–Noll–Vo(FNV) greatly improved the distribution of random numbers, with the random numbers now being good enough to produce consistently correct results for the Gamow-Stern problem. The numbers also passed a significantly larger, around 80%, of the NIST test suite. On the 17th Monte-Carlo problem (Waiting for Buses) using the numbers generated by FNV had results that were consistently under the correct value. Remember that the waiting times followed the $\frac{1}{n+1}$ equation where n is the number of buses, so the FNV results were always slightly less than the ideal fraction (29.8, 20, 14.8, 11.8, 9.97, 8.51). Likely the numbers that FNV generated are not perfectly uniform and have some density bias, with consecutive numbers having less than the ideal mean distance between them.

4.2 Sound Sensors

The inventr.io sensor kit came equipped with a set of sound sensors: one small sound sensor and one big sound sensor. Both of these sensors are capable of outputting detected sound as either a digital or analog signal. Additionally, they have a set of adjustment knob which allows the user to alter the sensitivity of the sensor, making them useful for being able to detect different levels of sound. However, the difference between the two at detecting sound at the same level of sensitivity is negligible. Furthermore, difference between the two is negligible when it comes to generating randomness. The source being measured would have to vary in a way that would be unpredictable in order to achieve substantial randomness. A normal conversation (which many were sampled) did not provide a significant source of randomness.

4.3 Shock Sensor

The shock or vibration sensor is a digital sensor that detects vibrations in the environment. While the vibration sensor would be unlikely to provide a good source of entropy by itself, it was hypothesised that when combined with a source of external vibration, like phone or buzzer, it would produce random numbers.

Needless to say the sensor was either too sensitive or not sensitive enough to be effected by the buzzer. However when handling the sensor it was discovered that accidental bridging some connections on the sensor would result is random looking output. Despite the impracticality of the design (requiring someone to hold onto the sensor) it was still tested. The results were poor, with high π approximation errors that had similar performance with earlier clock drift attempts.

Table 7: Shock Sensor Entropy

Run	Entropy	Pi Approx Error
1	7.9983	0.16
2	7.9983	0.39
3	7.9983	0.43

4.4 Laser Sensor

We connected a laser and a laser sensor to the Arduino, with the laser pointed directly at the laser sensor. The Arduino constantly measured the laser sensor and used the variability in the mea-

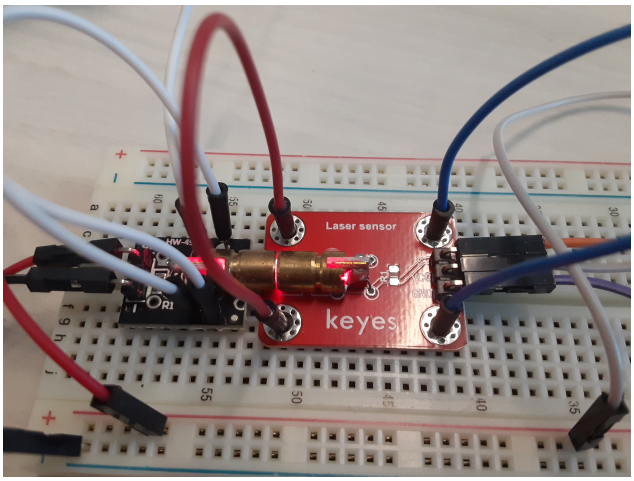


Figure 1: A laser pointing directly at a laser sensor. Some of the wires are used to hold the modules in place.

surement for randomness. The measured output of the laser was likely to vary, even when the time interval between measurements was short and the configuration of the laser and laser sensor was unchanged. The analog value of the laser was measured, and the least significant bit of this value was taken to output a random bit. Once 8 bits of supposedly random data were collected on the Arduino, the Arduino output 2 hexadecimal digits representing these random values. 2 MiB of data were collected in 2298 seconds, or 38.3 minutes. The data passed just 2 of the 188 NIST tests.

With this configuration, bits of 1 were more likely than 0, so a Von Neumann extractor was applied to attempt to have a uniform distribution of 1s and 0s.

The laser sensor took about 13851 seconds, about 3.8 hours, to collect 2 MiB of data when the von Neumann extractor was active. The resulting data passed 15 of the 188 NIST tests. It was successful for some of the random excursions tests and the linear complexity test.

The graph exhibits some self-similarity properties. Each 8-bit value X has the same probability of appearing as 255 - X = Y. Each probability peak has similar probability peaks on both sides. Because the left and right of the graph were symmetrical, at least 1 bit seems to be uniformly random. Therefore, data was collected again, but with only 1 bit output for each byte collected. The Arduino would take measurements of the sensor, with no prescribed delay, until the von Neumann extractor collected 8 bits. Then, the Arduino would output the least significant of these 8 bits. This made the data collection about 8 times slower, taking 81122 seconds, or 22.53 hours, to collect 2 MiB of data. However, this data was significantly more random than previous configurations.

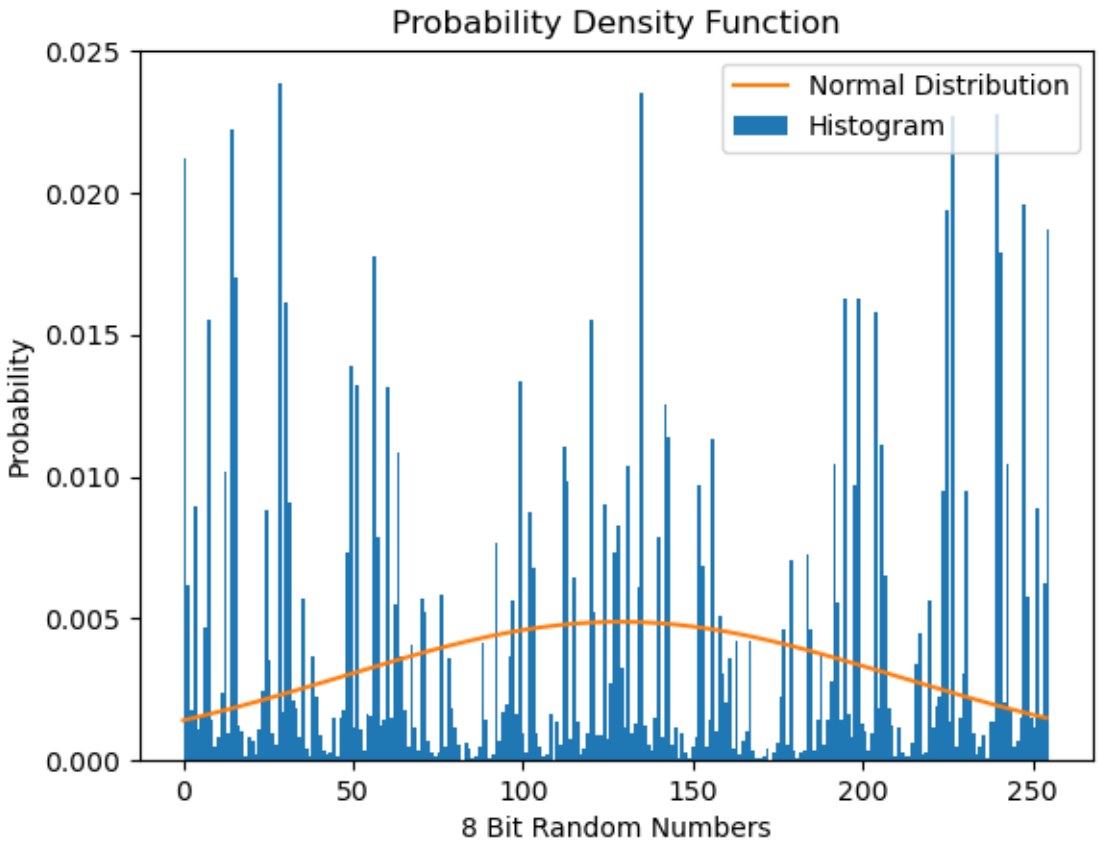
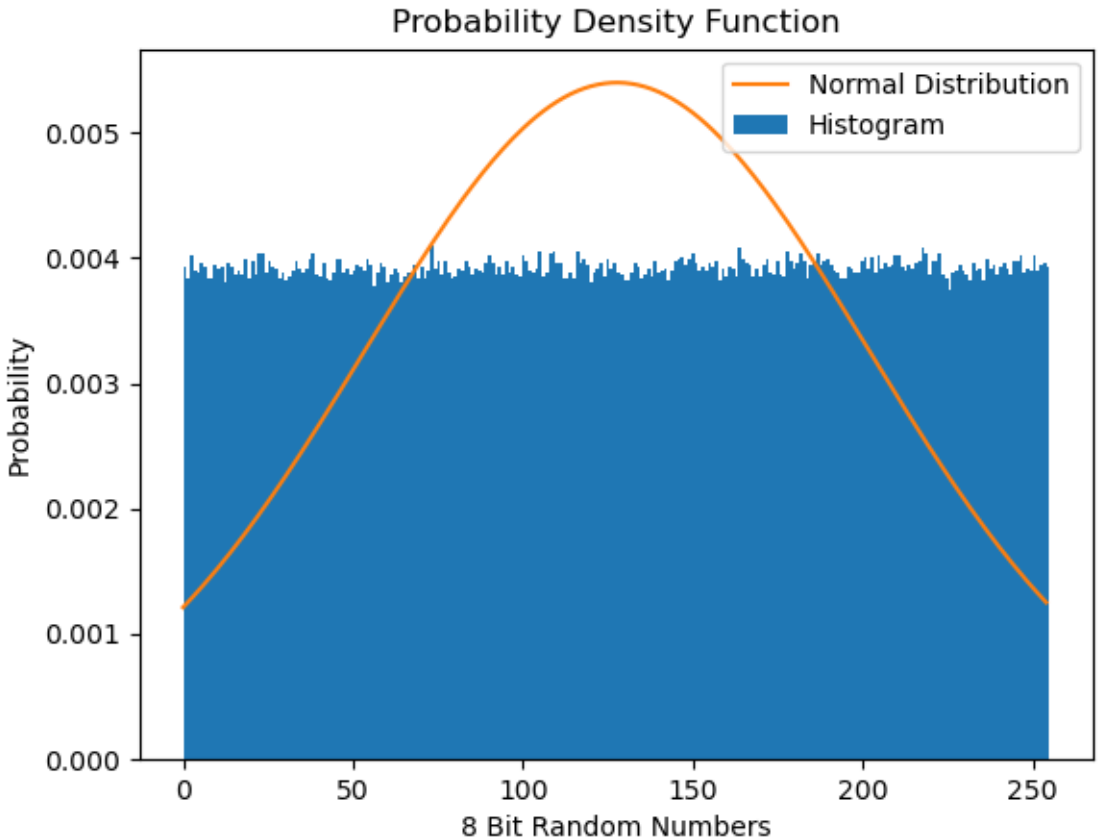


Figure 2: Probability Distribution of reading values from the analog laser sensor.



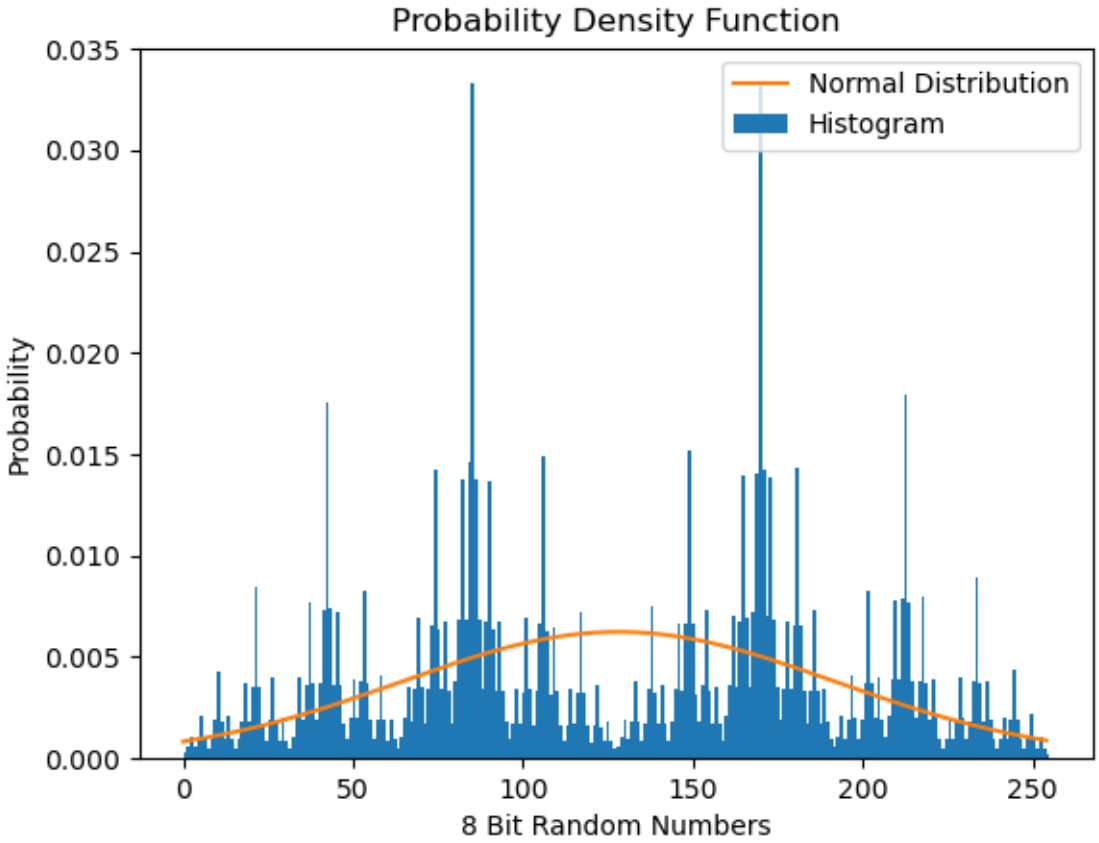


Figure 3: Probability Distribution of laser sensor with von Neumann extractor.

Perhaps the earlier configurations were sampling data too frequently, so that some pattern emerged. To test this, more data was collected with a specified 1 ms delay between output bits. This data also faced a issue as the laser data without any bits thrown away, namely having a symmetrical probability distribution, indicating that at least 1 of the bits seems completely random. Although this graph was more uniform and the entropy was higher than other tests without bits thrown away, the random data did not pass more NIST tests. So, its possible that the source of stochasticity in the data is related to the UART output process of the arduino.

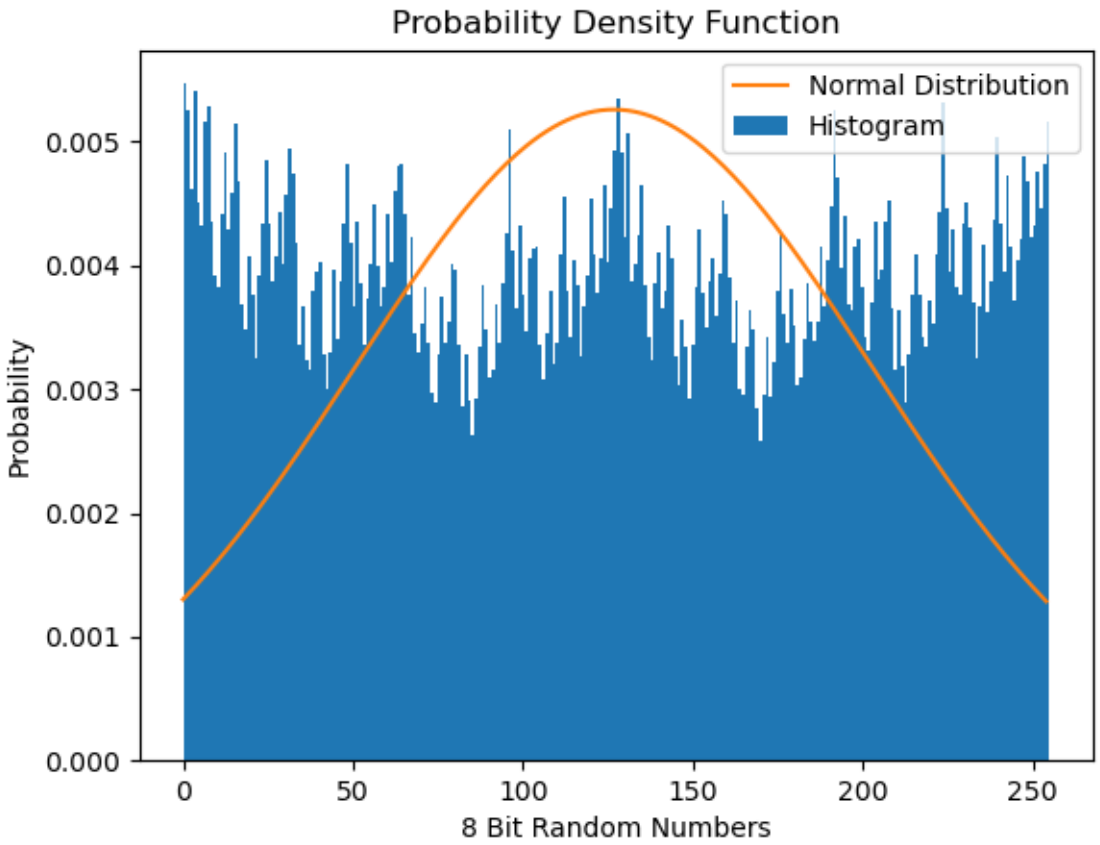


Figure 5: Probability Distribution of laser sensor with a specified 1 ms delay between random bits.

Table 8: Laser Sensor Benchmarks

(2 MiB)

Configuration	Time (seconds)	Bits per second	Entropy	Pi Approx Error	NIST STS
Analog Only	2298	912.6	6.977447	6.28%	2
von Neumann	13851	151.4	7.414846	6.77%	15
1 of 8 bits, von Neumann	81122	25.9	7.999817	0.04%	164
von Neumann, 1 ms delay	31426	66.7	7.983913	0.92%	2

4.5 Gas Sensor

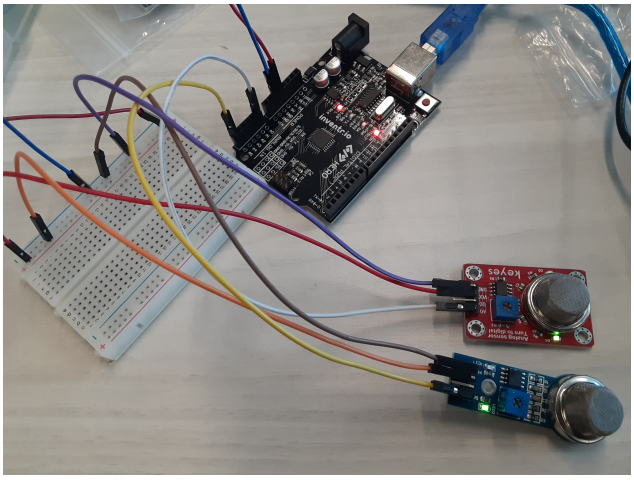


Figure 6: Two gas sensors

The gas sensor is able to detect LPG, I-butane, propane, methane, alcohol, hydrogen, and smoke. Based on statistical mechanics the microstate, or the configuration of air molecules in this system, is completely random at different moments in time. With enough precision, the gas sensor may be able to measure these random variations. From a different perspective, the gas sensor will also naturally experience fluctuation in its analog output due to noise and measurement error. The outputs of 2 gas sensors were connected to the Arduino, where they were XORed. Then, the least significant bit of the XOR result was fed into a von Neumann extractor to output random bits.

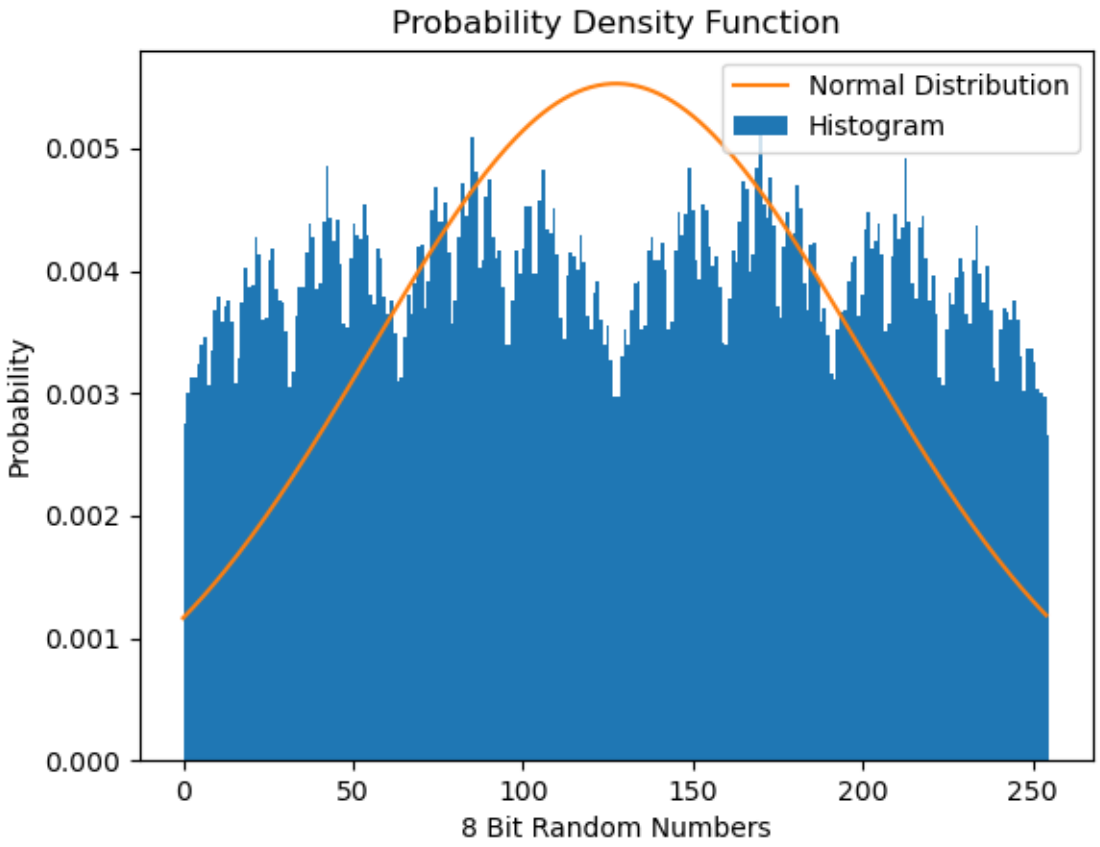


Figure 7: Probability Distribution of 2 gas sensors.

Like the laser sensor, this probability graph is symmetric and appears self-similar, but is more uniform than the single source, von Neumann laser data. More data was generated using just a single gas sensor to see if the XOR operation was bringing the output data closer to a uniform distribution.

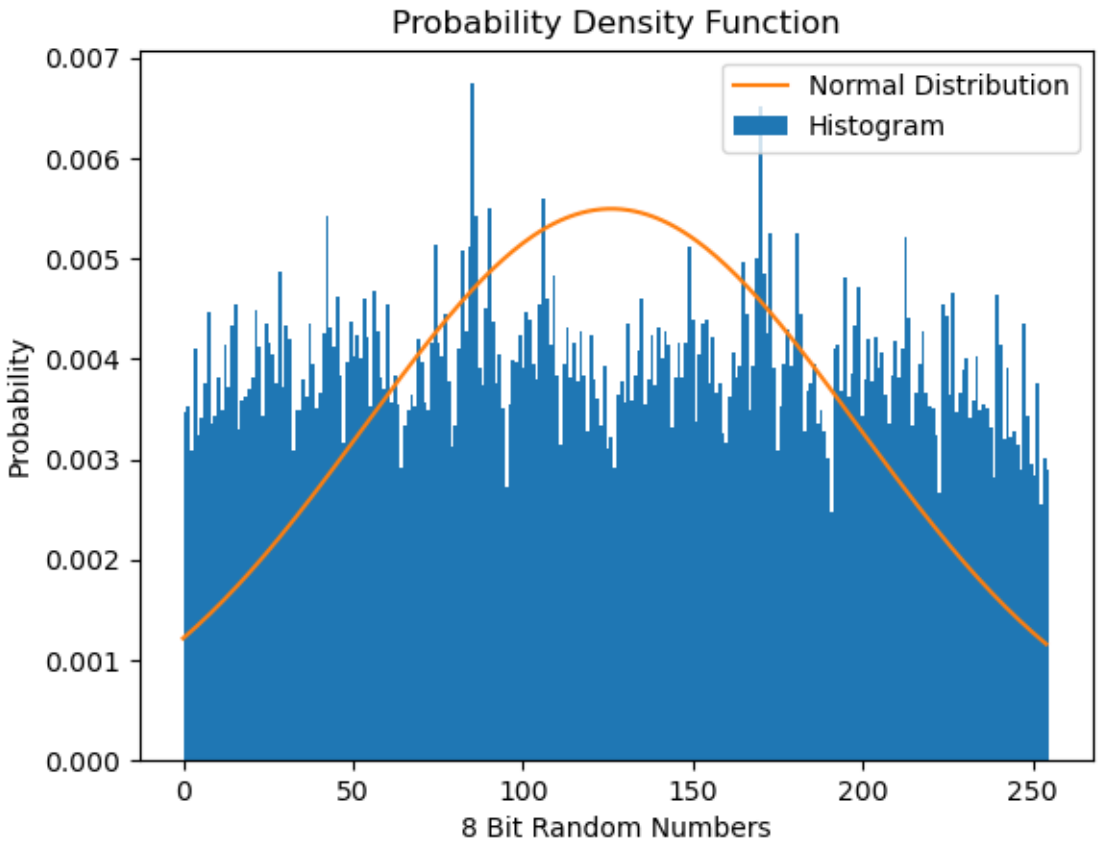


Figure 8: Probability Distribution of 1 gas sensor.

This data with 1 gas sensor has greater variation for the highest and lowest probabilities than the data with 2 gas sensors, and was evaluated to be less random. Furthermore, this data took substantially longer to collect, because 1 gas sensor will encounter less measurement noise than 2 gas sensors will, and the von Neumann extractor depends on variation to generate random data.

Table 9: Gas Sensor Benchmarks

(2 MiB)

Configuration	Time (seconds)	Bits per second	Entropy	Pi Approx Error	NIST STS
2 Gas Sensors XOR	30556	68.6	7.989307	1.07%	4
1 Gas Sensor	52315	40.1	7.983378	2.14%	2

4.6 Floating Analog Pin

On the Arduino, it's possible to read analog values from pins that are floating, i.e. not connected to anything, and use it as a source of entropy. The voltage that was read from the pin of the Arduino naturally fluctuated over time along and also responded to external stimulus, like a circuit changing the value of a pin.

When initially measured the values were poorly distributed and not very random. By switching the mode the Arduino used to read voltage to use reference voltage, the distribution of values of

read values became less biased, but the values still fell into bands of voltages that the Arduino was running through. Thus, the resulting numbers had a distinct pattern to them.

Table 10: Floating Pin Entropy

Run	Entropy	Pi Approx Error	NIST STS
1	1.6637	27.32	1/188
2	2.3483	27.32	1/188
3	3.5876	27.32	2/188
4	5.6868	13.30	2/188
5 (XOR)	7.997813	0.36	118/188

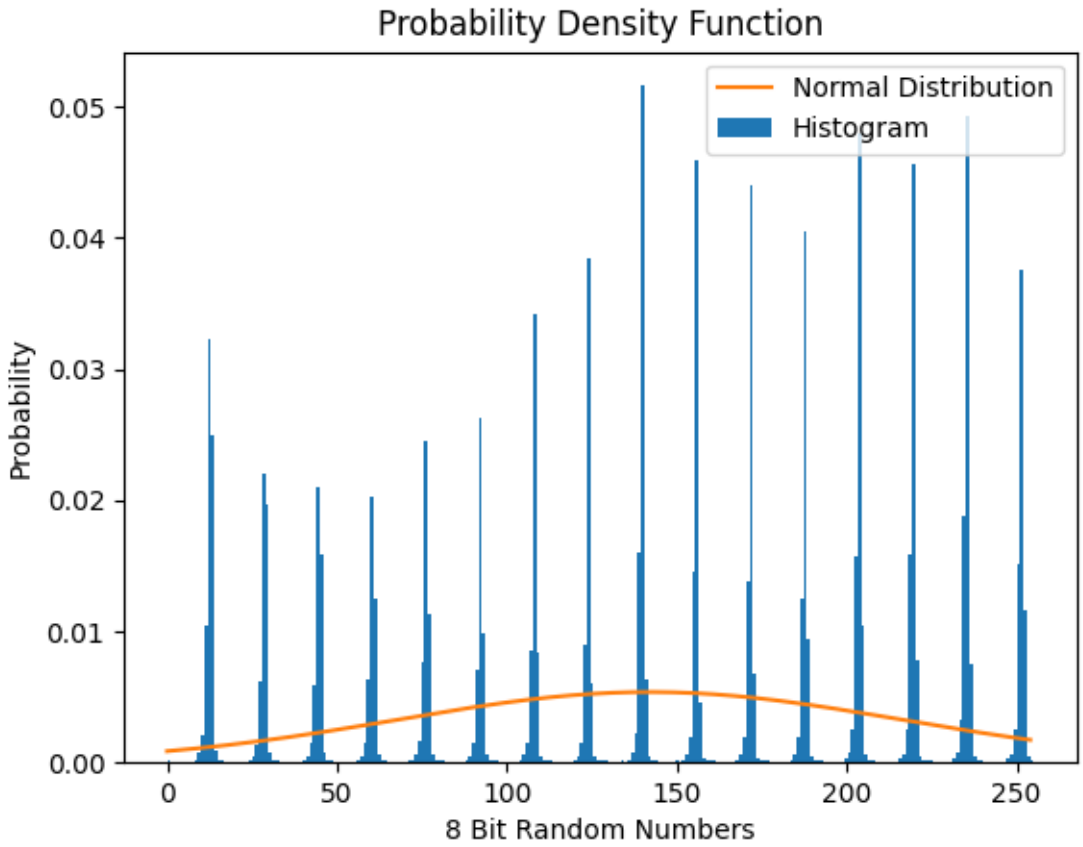


Figure 9: Probability Density Function of numbers generated by a floating analog signal.

Because XORing independent random variables can only improve entropy, more data was collected and XORed together by reading all the analog pins to generate a more uniform distribution. In addition, only the least significant bit was used. The analog pins were connected to male-male wires whose other end wasn't connected to anything else. These resulted in a significant improvement, but was extremely slow, taking 56716 seconds.

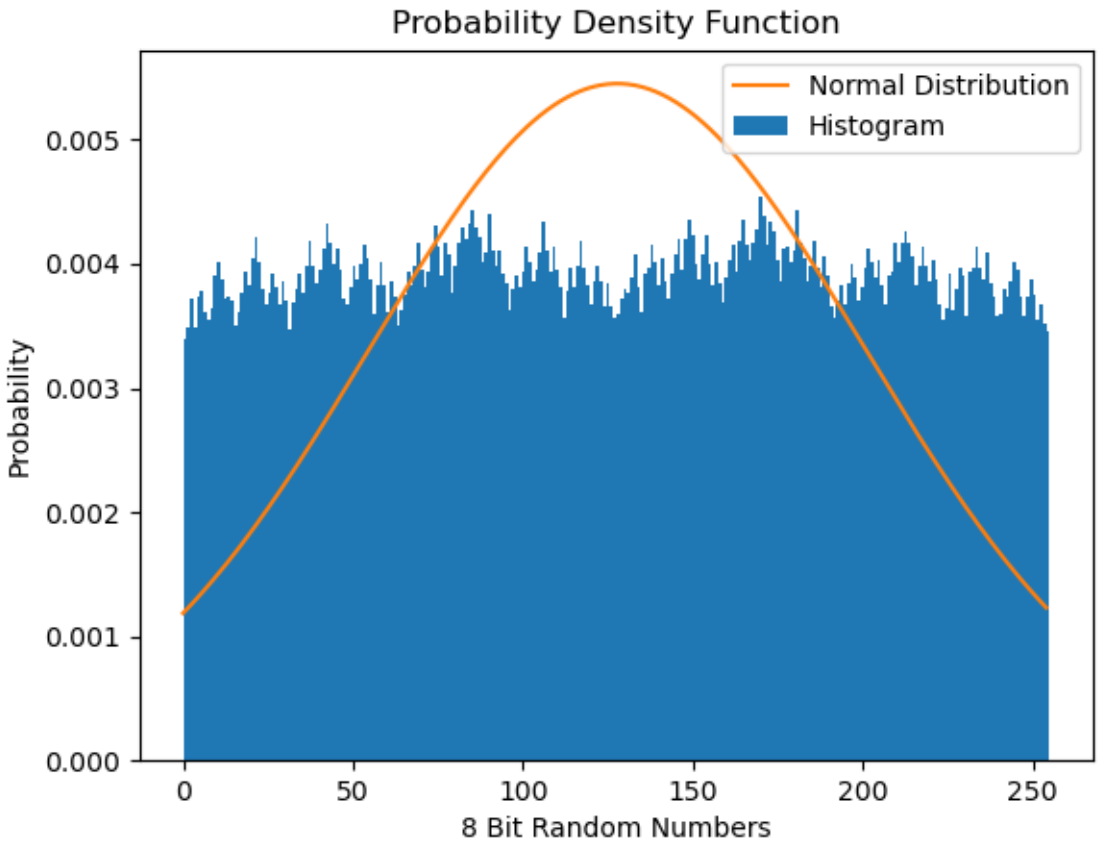


Figure 10: Probability distribution of 6 analog pins, least significant bit XORed. The distribution is approaching a uniform distribution.

4.7 Radio Antenna

Another sensor that was used to generate random bits was a radio antenna, although this was interfaced directly with a PC computer rather than an Arduino. The antenna was plugged in with USB and the

Listing 1: Radio Antenna command

```
'rtl_fm -f <freq>e6 -s 200000 -r 48000'
```

Then, the least significant bit of each sample was taken. For each 16 bit sample collected from the antenna, 1 random bit was outputted. This method consistently achieved a high NIST passrate. 2 mebibytes of random data were taken from frequencies 10 MHz to 200 MHz, with steps of 10 MHz. The resulting NIST passrates and entropy are graphed below. Each data collection took 350 seconds, amounting to 5991.9 bits per second for the least significant bits and 95869.8 bits per second for the raw data. There are multiple trends evident in this graph. Consumer electronics are common users of the lower MHz frequencies. There's an obvious drop around the 90 to 110 MHz frequencies, which correlates with the fact that many radio stations broadcast meaningful data near these frequencies, making it less random. As the frequency becomes higher, a combination of the constant radio antenna sampling rate reading quicker data and more sparse radio usage

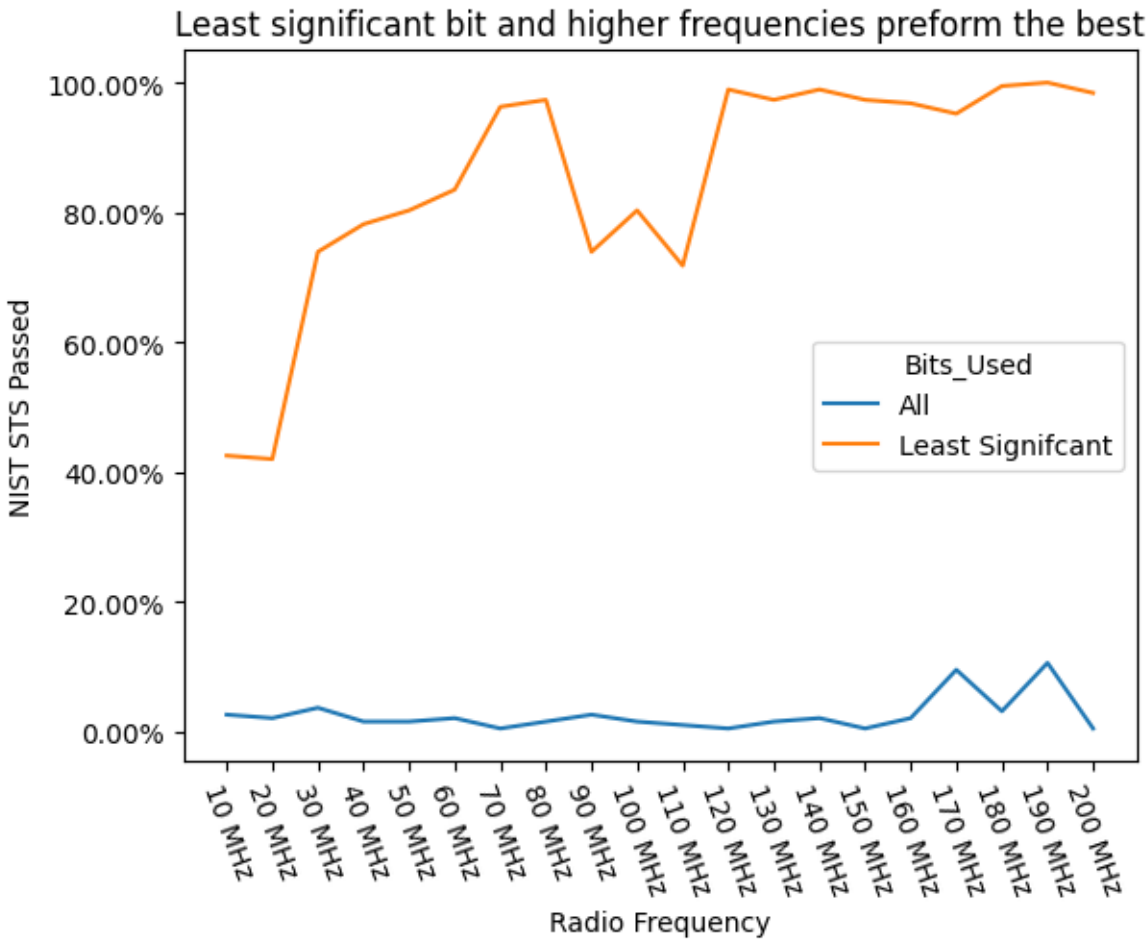


Figure 11: Radio NIST tests passed.

spectrum make the data more random.

4.8 Pseudo Random Number Generators

Many of the Arduino entropy sources suffer from not being able to output a lot of random bits quickly, but a few can manage decently random bits very slowly. Since it takes too much time to gather more than a few bytes, its possible to use those few random bytes as an initial seed for a pseudo random number generator.

The advantage of a good pseudo random number generator (PRNG) is that they will mathematically always guarantee a uniform distribution of random values. Even a simple 128-bit XOR shift-based PRNG was able to pass almost all of the NIST tests. If the initial seed generated from an entropy source is truly random, then a cryptographically secure PRNG could be used to generate random data. The issue with this technique is determining whether the seed is random and secure, as the Arduino sensors are dubious in this regard.

Table 11: PRNG Entropy

Run	Entropy	Pi Approx Error	NIST STS	Size (MiB)
1	7.9998	0.06	178/188	1.1
2	7.9998	0.18	174/188	1.1
3	7.9998	0.01	173/188	1.2
4	7.9999	0.01	184/188	3.3
5	7.9999	0.04	185/188	11.0
6	7.9998	0.14	173/188	1.1

5 FPGA

FPGAs (Field Programmable Gate Array) are embedded systems that host a programmable block utilized for holding different circuits. This form of adaptive hardware makes it simple for a plethora of different circuits to be synthesized and implemented using a HDL (hardware description language). In this case, we used Verilog HDL in order to integrate circuits such as a 1-bit arithmetic logic unit (ALU), half-adder, shift registers, and ring oscillators. The FPGA boards also host traditional von Neumann processors, known as the processing system (PS), which are able to interact with the configurable logic fabric through Advanced eXtensible Interface(AXI) connections.

5.1 Boards

- Zedboard
  The Zedboard is based on the Zynq-7000 System on a Chip (SoC) which includes a dual-core ARM Cortex A9 Processing System. The board has 8 LEDS, 8 switches, and 5 buttons connected to the Programmable Logic. It also includes an LED and 2 buttons can be controlled by the PS via Multiplexed I/O (MIO). There are micro-USB connections for onboard JTAG programming, UART output, and USB interfacing.

- Zybo Z7 - 10
  Similar to the Zedboard, the Zybo board has Xilinx Zynq-7000 family using an ARM prcocessor. This board included 4 switches and 4 toggling single value LEDs. As a result, implementing something such as the linear feedback shift register (LFSR) was limited in display due to the board's minimal amount of LEDs.

- Basys 3
  The Basys 3 board utilizes an Artix-7 FPGA, and has 16 switches, 16 LEDs, 5 buttons, numerous PMOD ports, and UART output that was useful for uploading the random bits generated by the FPGA.

5.2 Tools Used

- Verilog
  Verilog is a hardware description language that was used to create circuits for the FPGA board using the environment in Vivado. Unlike traditional programming languages, Verilog is structural and describes how an electronic circuit is connected, instead of giving a series of instructions to perform a certain function.

- Vivado

Vivado is a software suite used with hardware design languages such as verilog and VHDL. It has tools for design, simulation, synthesis and implementation. Vivado allows you to select the board that you are using when creating a project, which gives the software information about your board. You can then easily write Verilog code and synthesize and implement that code to be run on the board.

- Vitis
  Vitis is a software development environment that allows for the hardware created from Vivado on the FPGA to be read and manipulated. For example, the workflow of these mechanisms would usually entail creating a circuit in Vivado (written in Verilog) which was then exported with a bitstream to an XSA file. This XSA file would then be used to create a platform in Vitis that is specific to the target FPGA board. From here, C/C++ code can be written in Vitis to be able to read the output from the circuit to a printf() function which could be read out serially.

- Chisel
  Chisel is one of multiple novel alternative HDLs. It's a Scala library, where the engineer writes code to generate digital hardware when the program is run. Chisel has the ability to generate Verilog, which can then be used to synthesize a design for the FPGA. Chisel places emphasis on portability of the generated digital hardware designs, so specific board IO and certain digital hardware cannot be directly implemented with Chisel. Therefore, it requires Verilog implementations for these features, which Chisel is able to interface with. Boris used Chisel for his FPGA designs.

- F4PGA
  F4PGA is a free, open-source FPGA toolchain supported by the CHIPS Alliance. Compared to Vivado, F4PGA is less bloated with features and provides better insight about the process of synthesizing and implementing hardware designs. But, F4PGA is more difficult to begin using, less mature, and doesn't have the same community knowledge base as Vivado. Some issues arose when using F4PGA to create designs that didn't have any sequential elements, needed to interface with the ARM processor, or contained combinational loops. When a design didn't use any register elements, the following error occurred within the F4PGA synthesization process, where <module> corresponds to the name of the instantiated module that incorporates a clock without using any register elements:

Listing 2: F4PGA error

```
Message: Clock name or pattern '<module>.clock' does not correspond to any
nets. To create a virtual clock, use the '–name' option.
```

This could be fixed by either including some register elements in the design, or manually editing the generated .sdc file to add '-name' before the <module>.clock. To communicate with the ARM processor, the programmable logic design needed to write to extended MIO registers. This capability was provided by the PS7 Verilog module. While connecting wires to the module to read output from the ARM processor immediately worked, connecting wires to the PS7 processor input was not working upon initial attempts. Later, the input wires seemed to work, and it wasn't clear what had that enabled them to work. Finally, the F4PGA flow doesn't seem to synthesize combinational loops. Connecting output LEDs to ring oscillator outputs did not result in the LEDs oscillating. Due to these problems, the Verilog hardware designs were also synthesized with the Vivado command line interface (CLI). When using the Vivado CLI, these problems were not encountered.

- OpenFPGALoader
  OpenFPGALoader is a simple command line tool that is able to program the FPGA programmable logic (PL) by uploading an FPGA bitstream.

5.3 FPGA IO

For our basic designs that were meant to simply demonstrate that we could successfully program the FPGA, we simply used the onboard switches and LEDs for input and output. For our later designs, we needed comparatively high-speed output to read random data. Our team members used different approaches due to the different boards and tools they used.

GPIO: The Zynq processing system has GPIO registers that allow communication between the PS and PL. These registers are accessible from the PL using the PS7 Verilog module. From the PS side, these are accessible through memory mapped IO. The registers begin at base address 0xE000A000. Setting up the GPIO includes writing to the direction registers and output enable registers. There are 4 GPIO banks, with 0 and 1 corresponding to Multiplexed IO, which is connected to various FPGA device pins, while 2 and 3 correspond to Extended Multiplexed IO, which is connected to the PL.

AXI: Hardware designs could use AXI connections to simulate and connect to GPIO pins.

UART: UART, or universal asynchronous receiver-transmitter, is a relatively simple protocol for devices to communicate over a connection. The Zedboard has a Cypress CY7C64225 that functions as a USB-UART bridge, which is accessible from the PS and was used as our main source of output to a PC through a USB connection. Then, the output data can be read by a serial program listening to the USB device. When collecting our random data on the FPGA, the UART was the main bottleneck for moving generated bits off the FPGA to our computers.

5.4 FPGA Designs

The FPGAs feature programmable electronic components which allows for a wide variety of circuits to be created. At first, simple circuits were created in order to help adjust to the new programming environment. From there, more complex circuits were designed.

- Half-Adder
  The half-adder is a simple circuit composed of two inputs, A and B. In turn, there are two outputs, sum and carry. The two operations for sum and carry are XOR and the binary AND logic operations respectively. Due to its simplicity, it runs very quickly and helped in getting our foot in the door for circuit design.

A	B	SUM	$CARRY$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- 1-bit ALU
  A small upgrade from the half-adder is the 1-bit ALU ,which utilizes the logic from the half-adder in addition to other logic gates. For our design, we used a multiplexer in order to choose between the different logic gates. Although both the half-adder and the 1-bit ALU don't generate random numbers, they are useful in understanding the design of circuits, which will be used later for the LFSR and the ring oscillator.

A	B	C_{in}	C_{out}	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

- LFSR
  An LFSR is a circuit made up of a shift register where the output is XORed with several bits to provide the input to the shift register. Given a non-zero seed and properly chosen taps, it is possible to create a LFSR that outputs $2^n - 1$ pseudo random bits. In order to create a maximum length LFSR, the taps must be chosen to represent a primitive polynomial. The output bit of the LFSR appears random but is actually deterministic and can easily be predicted if the internal LFSR state is known.
- Ring Oscillator

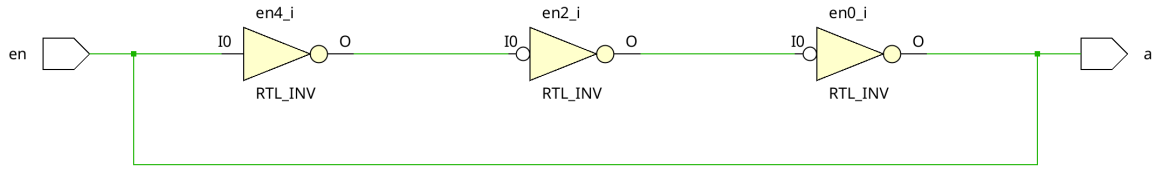


Figure 12: Example Ring Oscillator circuit generated in Vivado

A ring oscillator consists of an odd number of NOT gates, with the output of the final NOT gate fed as an input into the initial NOT gate. With this setup, the logic will never reach a stable point and will keep oscillating. Due to imperfect manufacturing and thermal noise, the circuit will oscillate at inconsistent periods. This jitter can be observed to help create a form of TRNG. One method of retrieving randomness from the Ring Oscillator was by allowing the circuit to run multiple times and combine that elapsed time in a variable. The method used that achieved 100 percent pass rate from the NIST test suite followed this procedure, logging the start time, end time, and elapsed time.

$$((Start * end * elapse) + elapsedTotal) mod 256$$

- XOR Ring Oscillator The XOR ring oscillator combines multiple ring oscillators, using the output of the final gates as a random bit and applying the XOR operation to increase the entropy of the data. Figure 5 shows the probability distribution for a ring oscillator RNG. Each oscillator consists of 5 NOT gates. 100 oscillators, with no intermediate flip-flop, are XORed together to generate the output bits. This configuration was created 32 times for 32-bit output from the

FPGA. This probability function has fractal properties. The leftmost peak gives way to more

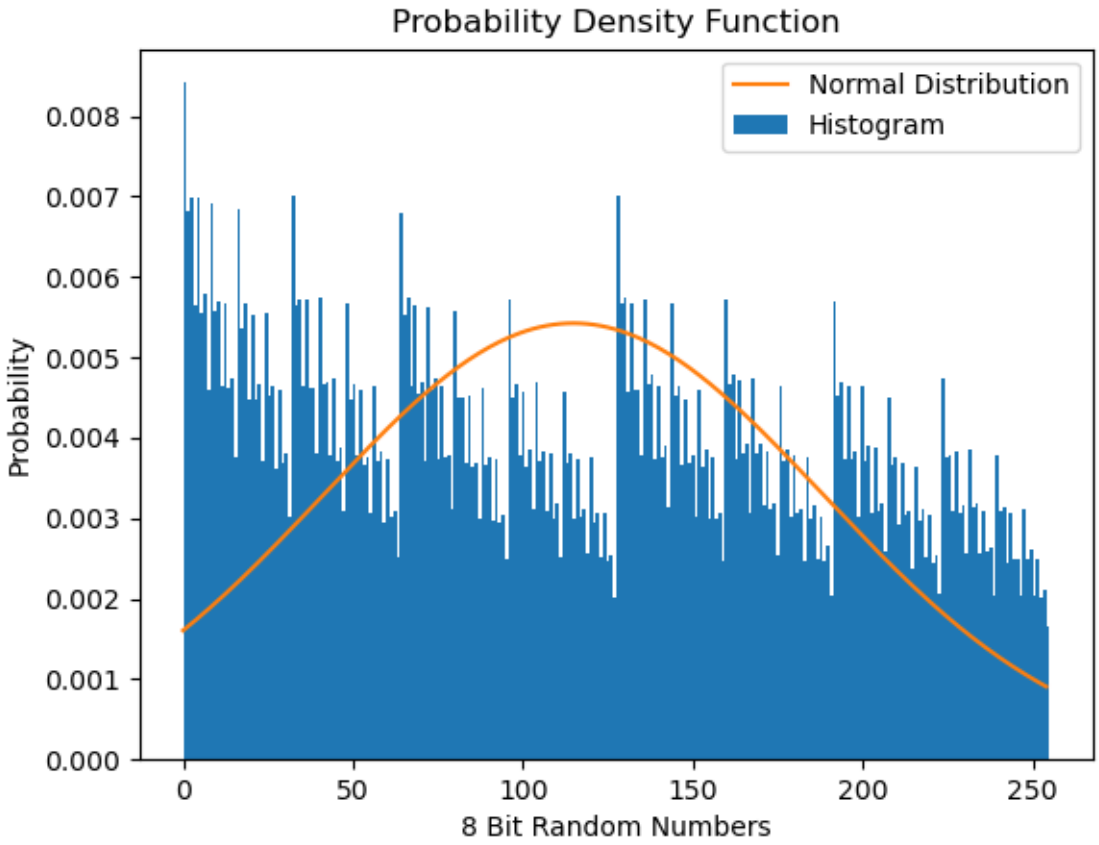


Figure 13: Probability of some numbers.

peaks on the right which become increasingly spaced from each other. Each of theses peaks then repeats the process with ever-decreasing probabilities. The graph is strongly related to the hamming weight., which counts the number of 1 bits in the binary representation of a number. While a binary binomial distribution would count the total 1s in a binary number, relating to the number of combinations of a certain amount of 0s and 1s, this function treats each permutation separately. Each 1 or 0 is generated by an independent, identically-distributed Bernoulli trial.

The following function gives the value of the nth digit of a binary representaton of x.

$$b(x,n) = \text{mod}\left(\text{floor}\left(\frac{x}{2^n}\right), 2\right)$$

which can be summed to determine the hamming weight for the first m digits

$$H(x,m) = \sum_{i=0}^{m} b\left(x,i\right)$$

The probability distribution then becomes

$$P(x,m) = (1-p)^{m-H(x,m)}p^{H(x,m)}$$

where p is the probability of a 1.

As p approaches 0.5, the probability distribution approaches the uniform distribution. In this instance, m is 8 because this is the distribution of 8-bit random numbers. According to the graph,

$$P(0) = (1-p)^8 = 0.0084142$$

and

$$P(255) = p^8 = 0.0016438$$

Therefore

$$p \approx 0.449$$

This probability distribution seems very simple and fundamental, like the binomial distribution, but it doesn't seem to be well-studied.

For positive integers, the hamming weight is the sum of ones in the integer's binary representation. Each bit alternates between 0 and 1, with increasing period for higher value bits, so each bit can be represented as a square wave. Each square wave can be represented as a Fourier series

$$f(x,m) = \frac{1}{2} + \sum_{n=1}^{\infty} \frac{(1-\cos(n\pi))}{n\pi} \sin\left(\frac{2}{m}n\pi x\right)$$

where m corresponds to a binary digit and 2^m is period of the square wave. Each square wave must be offset by 2^m so that the first $0..2^{m-1}$ are 0.

$$f(x,m) = \frac{1}{2} + \sum_{n=1}^{\infty} \frac{(1-\cos(n\pi))}{n\pi} \sin\left(\frac{n}{2^m}\pi(x-2^m)\right)$$

Then, these are summed over all bits to form the hamming weight:

$$H(x) = \sum_{n=0}^{\infty} f(x,n)$$

The differing probability between 0 and 1 could be caused by the threshold voltage needed for a low or high output. For example, for a low and high voltages of 0 V and 5 V respectively, the threshold voltage might be 3 V, rather than the 2.5 V that's in the direct middle of the range. If the ring wire spends a uniform amount of time at the voltage levels, then a 0 is more probable than a 1. Connecting these "floating" wires to the XOR gate will result in the XOR gate also "floating" and suffering from the same issue. However, adding flip-flops after each ring oscillator will push the output wire from "floating" to either a low or high. Then, XORing multiple oscillators moves the distribution to a uniform distribution. After adding latches for each ring oscillator output and XORing those, the data formed a random uniform distribution. The following heatmap shows the NIST passrates and entropy for output random data for varying amounts of NOT gates per oscillator and oscillators XORed per bit.

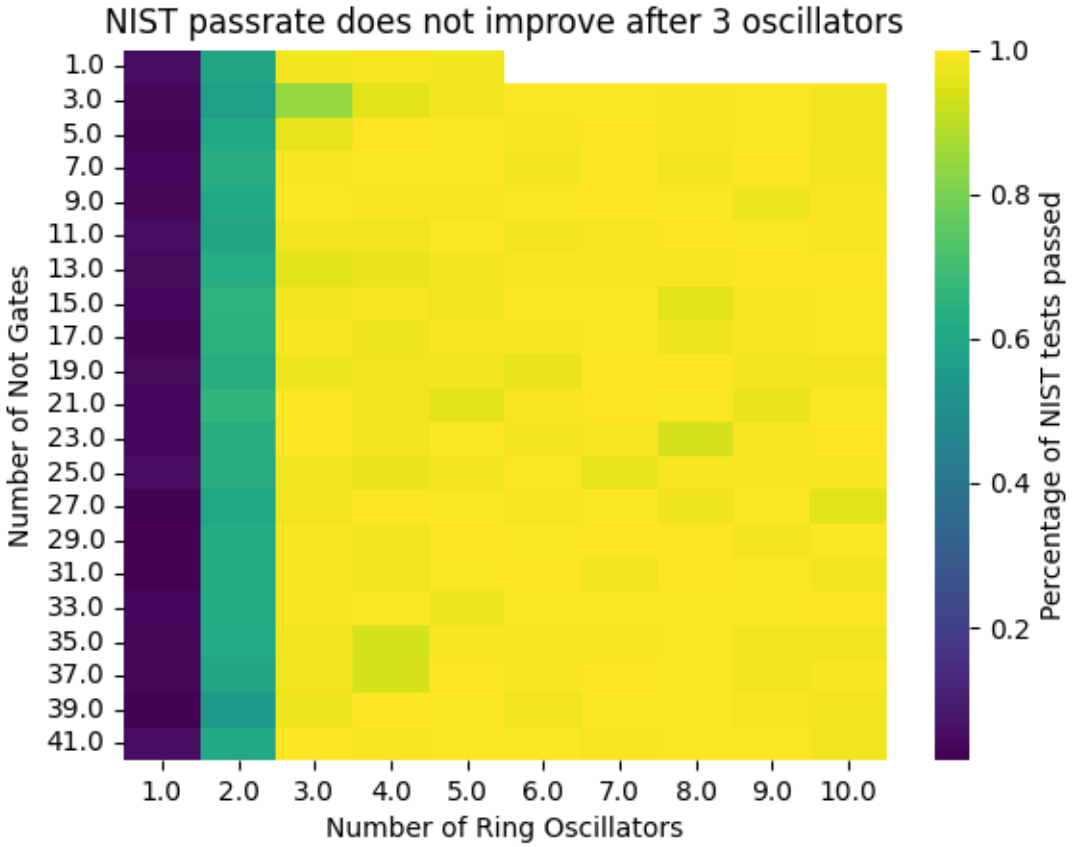


Figure 14: NIST tests passed for amount of NOT gates and amount of oscillators.

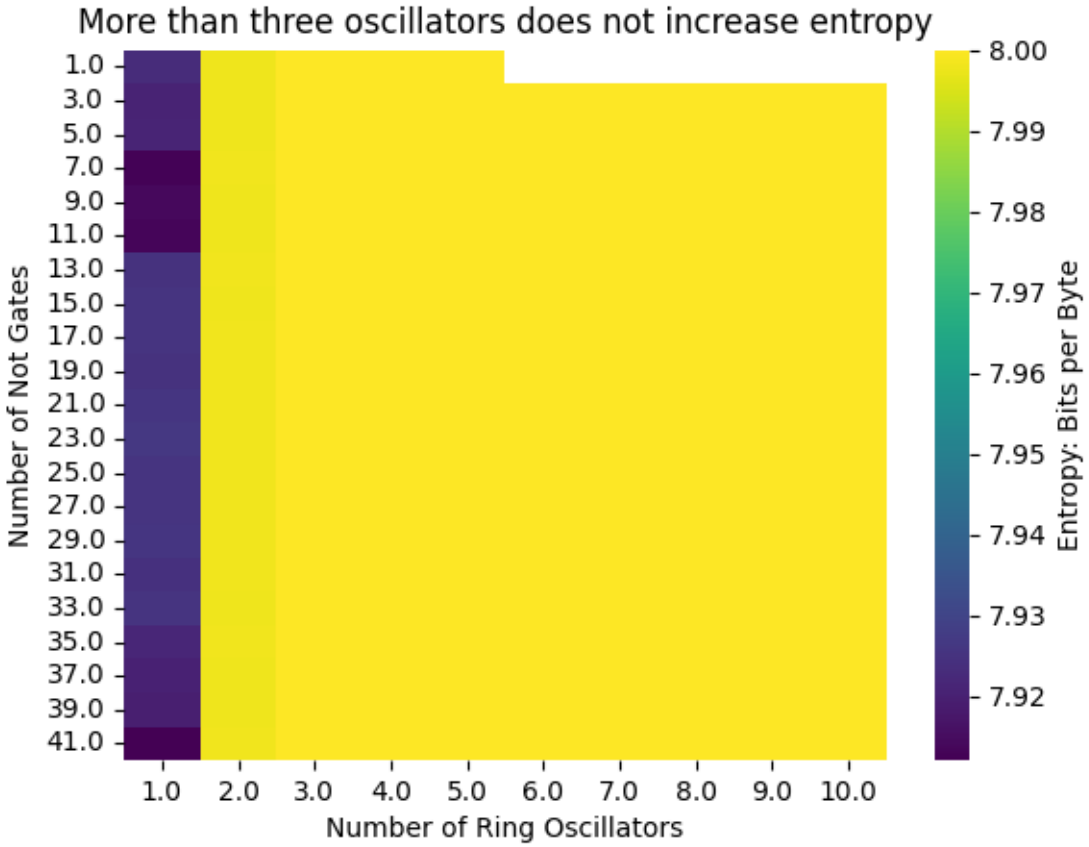


Figure 15: Data entropy for amount of NOT gates and amount of oscillators.

6 PERFORMANCE ANALYSIS

6.1 Arduino Results

- Serial UART

  Quite a few of the sensors seem to cap out at 5,849 bytes per second for generating random numbers. This limit is not from generating entropy but from the serial UART connection that was used to read the numbers of the Arduino. The Arduino's baud rate was set to 115,200 Bd and values were encoded as readable hexadecimal before being sent over. This limited the maximum number of bytes that could be transferred in a second and decreased the perceived speed of the Arduino sensors as sources of entropy.

- Data Size

  Across all sensors, it was noticed that the size of the data passed into the NIST test suite was positively associated with the number of tests it would pass. This was observed with the pseudo random number generator, where runs 4 and 5 passed around 10 extra tests compared to other runs. Tripling the size of the data, from 1.1 MiB to 3.3 MiB, increased the pass rate from 92.6% to 97.8%. Tripling the data again, from 3.3 MiB to 11.0 MiB, only lead to a small increase in

pass rate. With weaker random number generators, the NIST pass rate experiences logarithmic growth with number of bytes.

- Sensors

  The majority of sensors either depend on external stimulus or are not sensitive enough to capture small fluctuations in their surrounding environment. In a stagnant environment the values produced by the majority of sensors (sound, reed switch, temperature probe, etc.) have low variance and are consistent. When considering what these sensors are made for, hobbyist electronics, it makes sense that the output is consistent and not overtly sensitive. But what makes them good sensors for hobbyists makes them less then ideal as sources of entropy.

- WatchDog Timer

  On the Inventr.io Hero, or micro-controllers in general, there are semi-consistent timers called watchdog timers which are meant to help to mediate crashes in software by resetting the system's CPU. However, the watchdog itself is not 100% consistent in this. As a result, this can help to generate random numbers. Whenever the watchdog timer is interrupted, we can store this time in a variable and then compare that to the next interrupt time in order to get a measurement of the elapsed time between the interrupts. If the watchdog timer was configured to be 16 milliseconds, the variation between the timer interrupts was less than 1 millisecond. This lack of change can be seen across all 10 configurations for the watchdog timer and thus does not make the watchdog timer a good source of randomness.

Table 12: Watchdog performance for different configurations

Configuration	Average MS	Average Drift Change
16ms	16.55667	0.61685
32ms	33.06419	0.76229
64ms	66.11122	0.97896
125ms	132.21342	0.46693
250ms	264.38375	0.69940
500ms	528.85168	0.97094
1000ms	1056.58618	0.91483
2000ms	2114.03002	1.89178
4000ms	4227.77050	4.13126
8000ms	8454.22070	5.54910

- WatchDog + Sensors

  In combining the inconsistency of the watchdog timer and the entropy of the sensors on the micro-controllers, we are able to generate random numbers by blending both of these methods together.

6.2 FPGA Results

- LFSR

  The bits generated by the a 32 bit LFSR had a 99% pass rate from the NIST test suite, when over a MB of data was collected.

- Ring Oscillator With the Ring Oscillator, sufficient randomness was achieved only at around 1200 bytes per second at 1.519 watts or joules per second, which was derived from an estimate

from Vivado. This was exported as 2 characters of hexadecimal digits and then transferred to binary using a Python script. From there, multiple tests were performed to capture the randomness of the output data. 95-100% was passed using the STS client NIST tests, and 56-84 percent for the Dieharder tests suite.

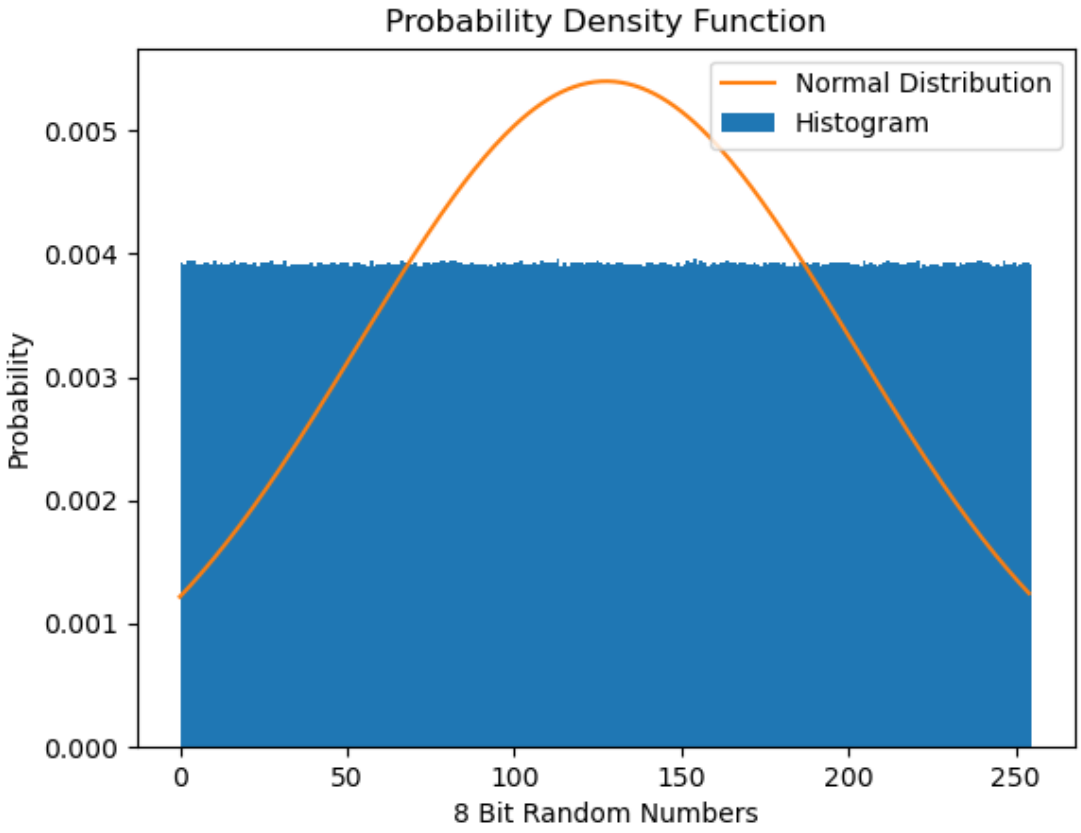


Figure 16: Output generated using jitter in ring oscillator circuit

Fgure 16 is a representation of a 25 MB file that was generated using the jitter/shot noise in the circuit. For this example, the data passed 100% of the NIST tests and was a concatenation of prior files which ranged from 1 MB to 5 MB.

- XOR Ring Oscillator

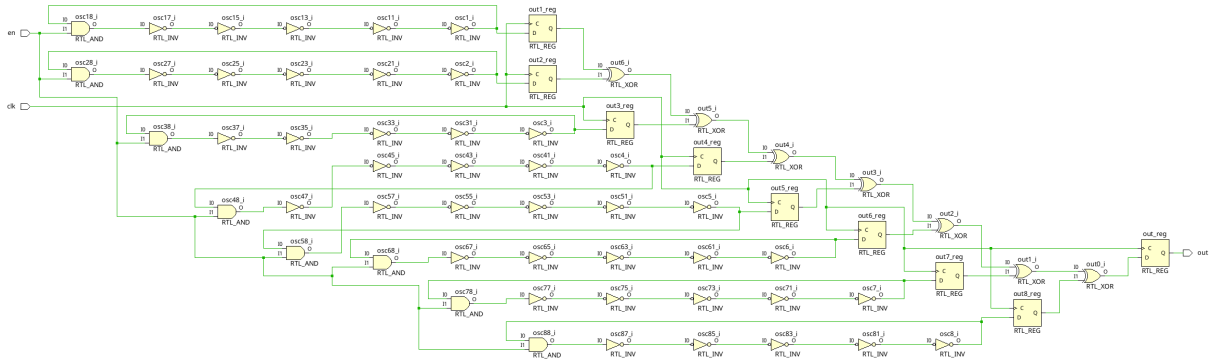


Figure 17: Example XOR Ring Oscillator circuit generated in Vivado

The XOR Ring Oscillator is a modified version of the normal ring oscillator with a significant difference in its ability to achieve TRNG. Instead of measuring the time as seen above in the ring oscillator, the XOR Ring Oscillator is able to generate random bits by utilizing multiple oscillators per one bit [9]. By creating multiple oscillators that exhibit shot noise, they can be XOR'd at the end of the circuit to create a random number generator.

When implemented on the ZedBoard, the reported power usage by Vivado of the logic circuit and IO was 0.005 watts. With an average mega bits per second of 7.293, the Ring Oscillator averaged 685.589 pico joules per bit. However, these results are not indicative to the true performance of the FPGA and XOR Ring Oscillator. The timing information and random bits were recorded using the onboard Zynq-7000 ARM Cortex SOC. The Zynq-7000 consumes at least 1 watt of power according to Vivado, including the reading the values provided by the ring oscillator. Given that the Zynq-7000, and the given C code running on it through Vitis, are not synchronized with the ring oscillator module, there is not a perfect transfer of bits. The bit read speed may be limited by the IO connection, and it is not guaranteed to read every bit that the oscillator produces. Without being able to synchronize the ring oscillator output and bit reading the actual performance is unknown, but above is a good estimate.

Table 13: XOR Ring Oscillator Preformance

Run	Data Size	MegaBits Per Second	NIST Pass Rate
Run 1	20 MiB	7.285	99%
Run 2	5 MiB	7.274	99%
Run 3	10 MiB	7.286	99%
Run 4	15 MiB	7.345	99%
Run 5	5 MiB	7.273	99%
Avg		7.293	99%

6.3 Penn State True RNG, "Dark Crystal"

The materials scientists at Pennsylvania State University have used a certain material to create a transistor[3]. Two transistors are then used to create a transistor-transistor logic (TTL) NOT gate. Then, 3 NOT gates are connected to make a three-stage inverter. Penn states graph shows that the by

chaining more NOT gates, the output voltage more sharply moves toward either 0 or 1. The chip has 2 TSIs that are XORed to generate output of 1-bit width. While the researchers tested various power supply voltages, the standard configuration was a $V_{DD} = 2V$ and $V_{in} = 0.25V$. The input voltage is held constant, but the output bit supposedly randomly fluctuates. This occurs because the input voltage is in the hysteresis window of the TSI. Due to properties of the material used for the transistor, the random charge trapping and detrapping will affect the output voltage of the TSI. While various true random number generators already exist, the main benefit of this new generator is its extremely low energy usage, purporting to use 30 pJ/bit. The researchers measured the average current of the circuit and size of the transistors, then used the following formula to estimate the energy used.

The dark crystal could generated a bit-stream at 1 kbps, as sampling the chip at a faster rate would deteriorate the randomness of the bits. Under further scrutiny the dark crystal only scores around 92% on the NIST test suite, which is lower than the XOR ring oscillator. This may be because the dark crystal doesn't use d flip-flops, which are critical in the XOR ring oscillator. Also, it may be that like the XOR ring oscillator two TSI are not enough to achieve good enough randomness, as three is the minimum oscillators for the XOR ring oscillator. Increasing the number of TSI may result in better random number generations and higher NIST pass rates.

6.4 Eve scheme (randomness vs math)

Another test of our RNG's was in the evaluation of a simple password authentication scheme adapted from Klein[10]. A user inputs a username and password. The server has a list of users and passwords. The server's task is to check if the inputted password is the same as the password in its database. The server proceeds in the following: The server contains a mapping of each password character to a binary vector. In our implementation, only the lower case letters were allowed, with a corresponding to 0 = 0b00000 and z corresponding to 25 = 0b11001. Internally, the server has a list of binary vectors. For each character in the input password, the hamming distance between that character and each binary vector is calculated. The server performs this by XORing the character and the binary vector, then summing the bits in the result. The server does the same for the correct password stored in its database. If all the hamming distances match, then the server accepts the password. Otherwise, the server rejects.

Example:

Correct password: a

Internal binary vectors:

$$\begin{matrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{matrix}$$

Hamming distances for correct password:

$$H(00001, 00000) = 1$$

$$H(00010, 00000) = 1$$

$$H(00100, 00000) = 1$$

$$H(01000, 00000) = 1$$

$$H(10000, 00000) = 1$$

Inputted password: a = 0 0 0 0 0
Hamming distances:

$$H(00001, 00000) = 1$$

$$H(00010, 00000) = 1$$

$$H(00100, 00000) = 1$$

$$H(01000, 00000) = 1$$

$$H(10000, 00000) = 1$$

Hamming distances the same, so server accepts.
Inputted password: b = 0 0 0 0 1
Hamming distances:

$$H(00001, 00000) = 0$$

$$H(00010, 00000) = 2$$

$$H(00100, 00000) = 2$$

$$H(01000, 00000) = 2$$

$$H(10000, 00000) = 2$$

Hamming distances differ so server rejects.
Different configuration:
Correct password: t = 19 = 1 0 0 1 1
Changing Internal binary vectors:

$$\begin{matrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{matrix}$$

Hamming distances for correct password:

$$H(00001, 10011) = 2$$

$$H(01000, 10011) = 4$$

$$H(10000, 10011) = 2$$

Inputted password: v = 1 0 1 0 1

$$H(00001, 10101) = 2$$

$$H(01000, 10101) = 4$$

$$H(10000, 10101) = 2$$

Hamming distances match, so the server accepts.

7 CONCLUSION

We rocked and rolled our way to a better understanding of our entropic universe. Firstly, when it comes to evaluating and testing a random source, most sources NIST pass rate seems to follow

a normal distribution. Each bit stream experiences some deviation from the average pass rate. For example, the ring oscillator circuits were able to achieve 100% pass rate on the NIST test suite on occasion with sometimes falling below that threshold around 95% pass rate. Ideally the TRNG would be consistent in passing the NIST test suite.

The performance of a random number generator can be categorized regarding the space/energy efficiency, how random the output was, and how many bits could be generated per second. Most sources of entropy we tested as random number generators could only fulfill two of categories. If a generator was able to generate bits quickly and efficiently with regards to energy, the results had poor randomness and failed the majority of the NIST tests. Sources that had good randomness, passing 95%+ of NIST tests, often were less than ideal in speed. This is partly because increasing the randomness of sources through tools like XOR and the von Nuemann extractor increase the amount of data required. To compensate for this loss of speed, the random number generators could be put into parallel, but that would increase the energy and space required by the generator.

For future works, we hope that someone is able achieve all 3 of course, but even our counterparts at Penn State are currently having a similar issue with their endeavors in the dark crystal. In their case, they have both efficiency and effectiveness, but their speed is lacking. In future works, they plan to continue chipping away at this problem and well maintain TRNG using their dark crystal. True random is the friends we made along the way.

REFERENCES

[1] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948. [Online]. Available: http://plan9.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.

[2] R. T. Kneusel, *Random Numbers and Computers*. Springer, 2018, pp. 1–259, ISBN: 978-3-319-77696-5.

[3] H. Ravichandran, D. Sen, A. Wali, *et al.*, "A peripheral-free true random number generator based on integrated circuits enabled by atomically thin two-dimensional materials," *ACS Nano*, vol. 17, no. 17, pp. 16 817–16 826, 2023, PMID: 37616285. DOI: 10.1021/acsnano.3c03581. eprint: https://doi.org/10.1021/acsnano.3c03581. [Online]. Available: https://doi.org/10.1021/acsnano.3c03581.

[4] V. Neumann, "Various techniques used in connection with random digits," *Notes by GE Forsythe*, pp. 36–38, 1951.

[5] T. C. Ekeh, *Entropy post XOR — tekeh.github.io*, https://tekeh.github.io/2020-07-19-entropy-post-XOR/, [Accessed 08-08-2024].

[6] L. Bassham, A. Rukhin, J. Soto, *et al.*, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, en, 2010-09-16 2010. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=906762.

[7] R. Brown, D. Eddelbuettel, D. Bauer, and R. Urban, *Dieharder — rurban.github.io*, https://rurban.github.io/dieharder/AUTHORS.html, [Accessed 07-08-2024], 2021.

[8] P. Nahin, *Digital Dice: Computational Solutions to Practical Probability Problems* (Paradoxes, Perplexities & Mathematical Conundrum for the Serious Head Scratcher). Princeton University Press, 2008, ISBN: 9780691126982. [Online]. Available: https://books.google.com/books?id=bmhuaGP3FOEC.

[9] S. Choi, Y. Shin, and H. Yoo, "Analysis of ring-oscillator-based true random number generator on fpgas," in *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, 2021, pp. 1–3. DOI: 10.1109/ICEIC51217.2021.9369714.

[10] P. Klein, *Coding the Matrix: Linear Algebra Through Applications to Computer Science*. Newtonian Press, 2013, ISBN: 9780615880990. [Online]. Available: https://books.google.com/books?id=3AA4nwEACAAJ.

[11] H. Ravichandran, D. Sen, A. Wali, *et al.*, "A peripheral-free true random number generator based on integrated circuits enabled by atomically thin two-dimensional materials," *ACS Nano*, vol. 17, no. 17, pp. 16 817–16 826, 2023, PMID: 37616285. DOI: 10.1021/acsnano.3c03581. eprint: https://doi.org/10.1021/acsnano.3c03581. [Online]. Available: https://doi.org/10.1021/acsnano.3c03581.