Identifying and Evaluating Sources of Randomness in IoT Devices

Alex McNaughton

BSc (Hons) Ethical Hacking, 2024

School of Design and Informatics
Abertay University

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Abdul Razaq, for his continued motivation and support. Without his guidance, this project would not have come to fruition, and I am deeply thankful for his input.

I would also like to thank my friends and family for their support, and for their willingness to understand and provide feedback to the work I have done. For this, I express my deepest gratitude.

Abstract

Random number generation is a vital aspect of computing that has far reaching applications in various different disciplines. From cryptography to networking, the need to generate a sufficiently random number is a requirement for almost all types of computing platform. Many RNGs use physical phenomena such as user input, network timings, and thermal noise to seed their RNGs and provide a less deterministic output. However, on devices that do not have as many inputs, such as IoT devices, gathering entropy from physical phenomena can be more challenging, as the pool of physical sources is much smaller.

Most PRNG's and CSPRNG's are reliant on setting the initial state of their algorithm with a 'seed' which, if able to be guessed by an adversary, can lead to the output of the RNG being predicted, leading to issues in applications where security is needed. Seeding a PRNG with a physical noise source would make it much more difficult to predict a RNGs output.

Using a Wemos D1 R2 and a microphone, various different random number generation techniques were attempted in order to develop a reliable RNG. From directly sampling the microphones input to try and create a TRNG, to seeding various PRNGs with the microphones input to create a more unpredictable PRNG.

Using the results of statistical testing, it was possible to see how each method performed. Directly sampling the microphone proved to not be a suitable method for RNG, however seeding a PRNG with the microphones noise proved to be a more fruitful option for RNG on IoT hardware. Comparisons with well known random sources showed that some of the methods produced in this paper were able to match the statistical abilities of hardware random number generators.

Abbreviations, Symbols and Notation

CPRNG – Cryptographically secure pseudorandom number generator

CSMA – Carrier-sense multiple access

FIPS 140 – Federal Information Processing Standard Publication 140, *Security Requirements for Cryptographic Modules*

HRNG – Hardware random number generator

IoT – Internet of Things

LCG - linear congruential generator

MAC – Medium Access Control

NIST – National Institute of Standards and Technology

NOTMT – Non-overlapping Template Matching Test

PRNG – Pseudo random number generator

STS – Statistical test Suite

TRNG – True random number generator

XOR – Exclusive OR

Chapter 1 Introduction

This section will provide information regarding the background of the field of random number generation in security in relation to the project.

1.1 Background and Context

The use of random numbers is an absolutely vital part of modern computing. Secure communication over the internet provides billions of people with a reliable way to privately store files, messages, and other sensitive data, and none of it would be possible without the use of random number generators. Powerful encryption prevents a users data from falling into the wrong hands by scrambling normal data into an unreadable format using an algorithm, which can then only be made readable again using a special digital key , which is ideally generated in such an unpredictable way that it is impossible to predict the contents of that key. This makes it possible for an unfathomable amount of sensitive data to be communicated daily from many different types of computers. Everything from personal computers, mobile phones and IoT devices use random number generators in a multitude of ways to provide everything from secure communication, to accurate simulations, to even functioning network protocols.

The ability to abuse weak random number generators to obtain encrypted data is a real and immediate threat to anyone who relies on encryption to obfuscate their data. The use of hardcoded seed keys in specific software implementations of the ANSI Random Number Generator allowed researchers to develop a full passive decryption attack against FortiGate VPN gateway products which were affected by the insecure key generation (Shaanan N. Cohney, 2017). Examples like this highlight the importance of secure and reliable random number generation, as faults in generation algorithms can significantly affect the security of encryption processes.

Secure obfuscation of data in encryption relies on an unpredictable way to generate private keys. Because of this, encryption algorithms cant rely on wholly deterministic methods to be secure. In order to securely encrypt data, there needs to be a chaotic factor that cannot be predicted by attackers beforehand. To create unpredictability, encryption methods make use of random number generators, which use either physical phenomena or specific algorithms to produce a near unpredictable set of numbers.

Pseudo random number generators are generated entirely using software and provide a way to produce a repeatable and computationally efficient set of numbers, making them ideal for specific fields such as simulations and video games. However, many PRNG's are unable to securely perform in cryptographic environments due to their predictable nature. Hardware random number generators, unlike PRNG's, use physical phenomena to generate a random set of numbers, making their outputs less deterministic. This is beneficial in cryptographic functions as it prevents an outside party from being able to predict how an encryption algorithm will obfuscate data. In order to get truly random numbers, physical phenomena such as thermal noise, user input timings, and even the air turbulence in hard disk drives can be used as an unpredictable noise source.

Random number generators that make use of physical phenomena like the ones described are routinely labelled as 'true' random number generators, however the use of physical phenomena does not immediately make a number generator immune to predictability, and hardware random number generators with a low enough amount of entropy can be predicted given enough of a sample size (Dichtl, 2003). For this reason, Hardware random number generators require a high level of entropy from its noise sources in order to be unpredictable enough to be used in cryptographic applications.

As more and more devices join the internet of things, and because of the ever present need to securely provide a private way of communicating using secure encryption. It is important that we have a way to produce a suitably unpredictable set of random numbers with minimal hardware requirements in an efficient manner, so that secure encryption can be replicated on a large number of low spec devices such as the computers used in IoT devices.

For a considerable amount of devices, no on board components can provide enough entropy for physically based hardware random number generation. Some devices are capable of generating random numbers through HRNG's that are supplied with their device. However, these are in the minority, and the vast majority of devices have no hardware for RNG (Peter Kietzmann, 2021). Considering the importance of RNG in all aspects of computing, the need for a powerful and efficient way to generate random numbers with the hardware and software available on IoT devices should be explored and evaluated further.

1.2 Aims, Research Questions and Objectives

Aim:

To produce a method for creating hardware random numbers on IoT hardware, and confirm the validity of this method by performing various randomness tests.

Objectives:

- Investigate how HRNG's are designed
- Produce a HRNG for a typical IoT device
- produce a selection of methods for a wide range of devices
- Evaluate each method's effectiveness using statistical and standardized randomness tests, as well as testing for potential exploits.

Research Question:

How can a true random number generator be implemented on hardware typical to IoT devices?

Chapter 2 Literature Review

This section will highlight the retrospective work done on the generation of random numbers on computers, with the intent to critically evaluate the work done so far, and evaluate how this work can be compounded upon to achieve the aims of the project. This will be achieved by examining the current state of random number generation in computers, as well as the testing criteria required to achieve a sufficiently random number generator, and evaluating how these methods can be adapted for an IoT environment.

2.1 Current Generation Methods

Generating random numbers in modern computers has a variety of solutions based on the specific conditions of the situation it was designed for. The Linux Random Number Generator makes use of external user inputs such as the activity of the mouse and keyboard in addition to software events to create entropy high enough to be secure for file encryption and secure password storage (Gutterman, 2006).

For most applications, PRNG's are suitable enough at producing a stream of numbers that look unpredictable, for example in video games and some simulation software. The most common type of PRNG is the LCG, which has been used in many default runtime libraries such as the Microsoft Visual Studio C++ library as part of its rand() function (Microsoft, 2022). However, despite its commonality, the LCG is not capable of producing cryptographically secure random numbers due to its ability to be predicted (Chung-Chih Li, 2005).

However, cryptographic functionality should not be the only metric used to evaluate the strength of a RNG, as many PRNG's are capable of much faster, efficient, and computationally cheaper RNG than their cryptographic counterparts (Matthias Kaas-Mason, 2019). This allows for more efficient programming in applications that do not have the need for extreme unpredictability, and the applications for PRNG's in IoT devices are numerous. RNG is a vital component for specific networking protocols, such as CSMA, a MAC protocol used in IoT appliances to aid in transmitting traffic over a network that has many devices sharing the same transmission space, such as a cable or a wireless connection (L Kleinrock, 1975). When collisions happen between two devices in a shared medium, a collision resolution mechanism is required to prevent further collisions. In CSMA, a back-off algorithm is implemented to resolve such collisions, and part of the back-off algorithm requires a device to wait a random amount of time before sending new packets to avoid collisions. This is where a PRNG would be a better choice than a CSPRNG, as the wait time in the back off algorithm wait time approaches microseconds in current modern implementations of CSMA (Rafael Laufer, 2013). Such speeds require low security requirements, and fast production of random numbers, so a PRNG would be the best option for random number generation in this scenario. IoT devices require the use of PRNG’s almost as much as CSPRNG’s and so the implementation of a PRNG on IoT hardware should be explored further.

There are a wide range of PRNG solutions that have been devised over the years to varying degrees of success. One of the earliest solutions was RANDU, a type of LCG developed in the 1960’s that has been notorious for failing many of the statistical tests required for modern day RNG (Lewis, 1986). Since then, the statistical properties of PRNG’s have been put under much more careful consideration. Xorshift is a PRNG designed by George Marsaglia that is designed to be as efficient as possible, while also maintaining statistical strength through the refinement of the PRNG using non linear functions (Marsaglia, 2003). PRNG’s like this have a wide range of applications, and could be a suitable candidate for use on an IoT platform

However, PRNG’s that are capable of being predicted by an intelligent attacker should still never be implemented in security applications such as encryption key generation, as being able to predict the output of a random number generator can make it easier to predict encryption keys, and break encryption to leak potentially sensitive data.

For this reason, a more cryptographically secure approach is required. Many different CSPRNG’s have been developed for this exact reason, which attempt to minimise the predictability of their algorithms, such as the one included in Apple’s Secure Enclave, which uses a NIST standard algorithm called CTR_DRBG to generate secure random numbers (Apple, 2022). What is also notable about Apple’s RNG configuration is its use of multiple ring oscillators which provide a signal to seed its number generator. This method of random number generation provides higher cryptographic security, as the number generator is seeded on a physical phenomena that is near impossible to predict. Applying this method of using physical phenomena to an IoT device could help improve the cryptographic security of a RNG program that runs on an IoT device.

There has been a wide range of research undertaken in the field of generating cryptographically secure pseudorandom numbers on limited processing space, given the advent of mobile phones and other devices that need efficient programming. One of the algorithms that has been shown to produce secure pseudo random numbers on limited hardware quickly and efficiently is ChaCha20, which has shown to have a quicker encryption speed than more common encryption algorithms such as AES (Alen Salkanovic, 2021).

Seeding a CSPRNG like the one available in ChaCha20 with hardware based random numbers could produce true random numbers quickly and efficiently on limited hardware like IoT devices, therefore, it would be worth implementing this methodology as part of the work undertaken to achieve a true random number generator on an IoT device.

To reduce their predictability, many CSPRNG’s are tested using randomness tests, which apply statistical methods to measure the quality and predictability of the random number algorithm.

2.2 Testing Randomness

The prevalence of random number generation in security applications requires a thorough testing audit to ensure the apparent unpredictability of randomness algorithms. Due to the ever increasing reliance on computers to securely handle data, the problem of verifying randomness algorithms has been of upmost importance for security researchers and government bodies for decades.

In the context of random number testing for computers, assessment of random data is usually performed by analysing a stream of binary, through various statistical tests and simulations to highlight potential patterns in a bit stream. While methodology for these tests have existed as suggestions for computer programmers since the 1960’s (Knuth, 1968), a widespread software library containing a number of tests did not come to prevalence until the introduction of the Diehard tests in 1995. Created by George Marsaglia (Marsaglia, 1995), the original package contained 12 varied tests to run against random data to verify its perceived randomness.

The field of randomness verification is not confined solely to theoretical computer science, and has had extensive research performed by governing bodies due to their reliance on computers to securely store data. The National Institute for Standards and Technology, a subsidiary of the United States Department of Commerce, published its standards regarding random numbers as part of a series of papers called ‘FIPS 140 Security Requirements for Cryptographic Modules’. These papers, among other cryptographic recommendations, outline a variety of statistical tests that can be used to identify the quality of random numbers from a given RNG’s output.

Understanding the effectiveness of these testing methods is vital to the production of a program that can generate random numbers on an IoT device, as the functionality of any program that fails a significant number of these statistical tests will be extremely limited and unlikely to be used in any environment that requires appropriately unpredictable random numbers. For FIPS-140, its inclusion within a government standard which has the financial and intellectual backing of the United States government should give it some authority on the ability to determine a random number stream. However, aspects of the tests NIST provides have been met with criticism after analysis. Yongge Wang, a professor at University of North Carolina at Charlotte, proved that the statistical tests included as part of the NIST standards were not enough to validate the quality of a random stream (Wang, 2013). Knowing this, in order to test the efficacy of this projects program, a multitude of tests will be required to confirm the quality of the numbers generate, otherwise the project may be susceptible to exploitation.

Since the introduction of NIST’s FIPS 140 randomness tests, NIST has re-evaluated the testing suite and updated their offerings for statistical tests. In 2010, NIST published the NIST Statistical test suite, an extended implementation of the tests found in FIPS 140 that includes the original 4 statistical tests, as well as 11 more statistical tests designed to further evaluate the strength of a random source. While the initial FIPS 140 statistical tests are still useful and relevant to statistical testing, and can be used to provide a precursory evaluation of a random source, the NIST STS provides a much broader analysis, and this implementation should be used when further detail is needed.

2.3 Hardware Random Number Generation on IoT Devices

While some IoT capable devices contain HRNG capabilities, many lower level devices such as the Wemos D1 R2 development board are not supplied with one.

Finding sources of randomness can be much harder than expected on IoT devices, as the number of random sources it can use are much smaller than a typical personal computer.

Using external random inputs allows for a higher chance of unpredictability, and provides a more unpredictable method of number generation. While some devices, such as PC’s, are able to gather entropy from systems like user input, this kind of entropy extraction is less feasible on a device with limited user input. Most single board microcontrollers do not have user input devices such as a mouse or keyboard, and their input devices can change drastically depending on its function. For Arduino supported boards, which are commonly used in IoT devices, the standard reference manual suggests using its PRNG for random numbers, but also suggests that a pin configured as an input with no wires attached will provide ‘seemingly random changes in pin state’ (Arduino, 2023).This random change in pin state is attributed to electrical noise from the environment, which can provide a seemingly unpredictable signal, however further research has shown that this signal is in fact predictable, and therefore not safe for use within a cryptographic scenario (Kristinsson, 2012).

There are libraries available for the Arduino IDE that are capable of generating true random numbers using the natural jitter of the watchdog timer. However, hardware within IoT devices can vary greatly depending on its function, and its inputs could provide a source of randomness that its internal hardware may otherwise be unable to support natively. This is why using the pre-existing hardware of a IoT device to try and generate random numbers could potentially be a beneficial concept worth exploring, as the ability to use the pre-existing hardware to generate secure random numbers would be an extremely efficient option for devices that do not currently have these capabilities .

Chapter 3 Methodology

This chapter covers the development aspect of this project, that consists of effectively completing the project’s aim of a program that can generate a sufficiently random stream of bytes to pass the tests in the FIPS 140-2 testing criteria. By passing this testing suite, the program will be considered statistically random enough for use in random number generation applications, and will be further analysed using other testing suites to corroborate these findings. This chapter will initially describe the limitations of the hardware, the challenges that must be overcome to complete the aims, the reasoning for the development methodology, as well as the implementation of the testing standard to ascertain the effectiveness of each prototype.

3.1 Development Preface

There are some aspects of the development process that need to be considered before development can begin. Designing a randomness generator for IoT devices come with a variety of limitations that require such a program to be developed with these limitations in mind. The most pressing of these being the heavily restricted processing capabilities of most IoT devices. To understand these limitations and develop a solution that fits within them, using a suitable development environment that fits the profile of a general purpose IoT device was necessary. For this reason, development of the program was chosen to be performed on a Wemos D1 R2, an Arduino compatible development board with an ESP-8266EX microcontroller used for processing. This board matches the limited processing limitations of most IoT devices, and is capable of several standard IoT device functions, making it a very suitable candidate for a generic IoT device. What this board lacks however, is a suitably random hardware noise source that can provide enough entropy to a random number algorithm to generate sufficiently random numbers to pass FIPS 140-2. Due to this lack of hardware, a suitably generic external hardware device that is present in a wide range of IoT devices should be supplied with the testing environment to simulate the majority of devices.

For this hardware aspect, a microphone was decided as the hardware device to be used during development, as a wide range of IoT devices use or have microphones already, and using the background noise from a microphone was considered as a source of entropy. The microphones chosen for this project is a KY-037 microphone module, which was chosen due to its availability and its compatibility with Arduino supported microcontrollers like the Wemos D1 R2.

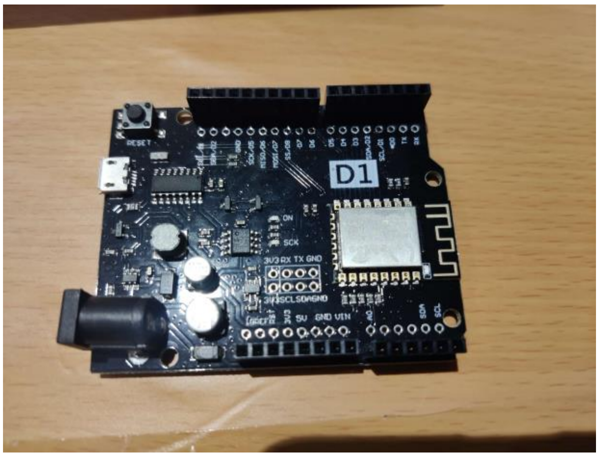


Figure 1 - Wemos D1 R2 Board

Choosing a suitable methodology for development of the program would improve the effectiveness of the development stage, and lead to an overall more suitable solution that fit with the project's aims. To decide on a methodology, the success of a specific program had to be defined, as deciding a finished state for the project could help decide the appropriate methodology. Since the aim of the program is to generate a sufficiently random stream of bits, passing the FIPS 140-2 standard was chosen as a reasonably obtainable requirement for the success of a randomness solution. The implementation of FIPS 140-2 provided by NIST is incredibly fast, capable of completing a test within milliseconds. This makes it possible to attempt multiple different approaches in quick succession until a passable condition is met. Because of the quick testing capabilities of the testing standard, a prototyping methodology was chosen as a suitable development strategy. In this strategy, various different methods of producing a RNG were developed until they produced a passable output, then this output would be tested on to confirm its randomness.

Once testing is complete, the results of the test would be analysed to determine how the next prototype would be developed.

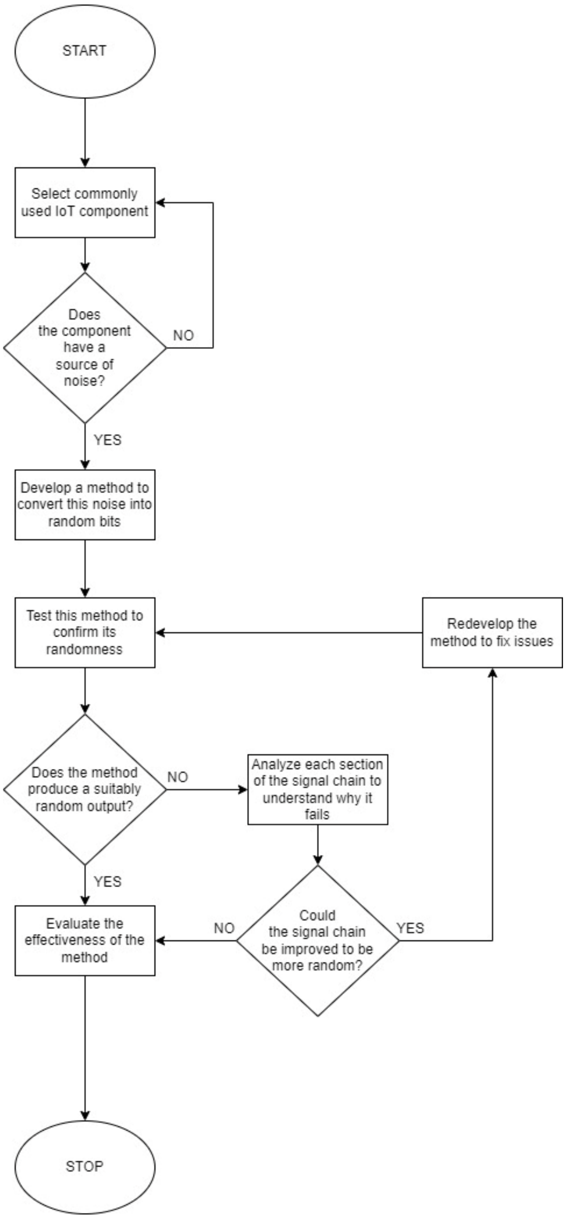

START
Select commonly used IoT component
Does the component have a source of noise?
NO
YES
Develop a method to convert this noise into random bits
Test this method to confirm its randomness
Redevelop the method to fix issues
Does the method produce a suitably random output?
NO
Analyze each section of the signal chain to understand why it fails
YES
Evaluate the effectiveness of the method
NO
Could the signal chain be improved to be more random?
YES
STOP

Figure 2 - Flow chart for development

Once the precursory testing of each prototype is completed, the use of the NIST STS and Dieharder suites will be used to provide a more extensive examination of the prototype. If the prototypes then pass these statistical tests, they will be considered fit for producing random numbers.

Given the Wemos' support for Arduino software and libraries, the programming of the RNG would be performed in the Arduino IDE. However, reliance on specific Arduino libraries was kept to a minimum, in an attempt to make it easier for the finished product to be easily ported to other systems.

3.2 Prototype 1

The first prototype design involved designing a computationally low effort algorithm which took the signal generated from the microphone and converted it into a stream of bits that could be tested. However, before this could be done, ascertaining the microphones ability to produce a different result on every iteration had to be found. This was done by observing the signal produced by the microphone using the Arduino IDE's serial plotter.

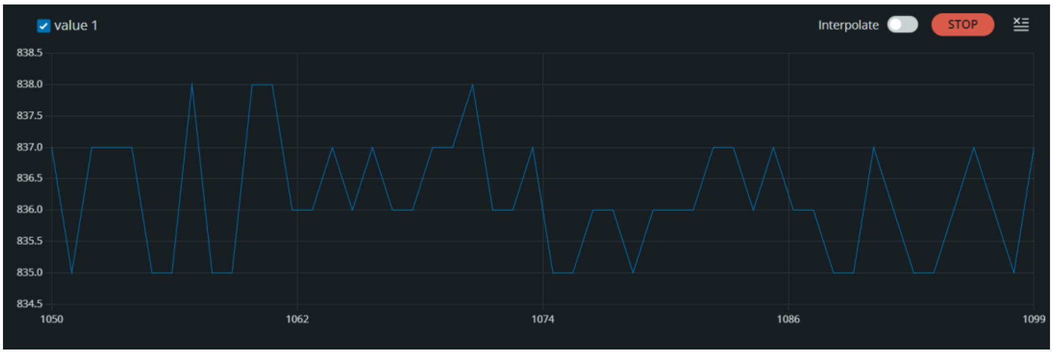


Figure 3 - Microphone signal stream

The signal produced by the microphone on standby produced a seemingly random set of numbers each time it was called. These values could not produce a secure bit stream on their own, so an algorithm had to be developed to divide these numbers into binary values in a bit stream. Since the range of values was always changing, the algorithm couldn't rely on producing bits when specific values were called. Eventually, dividing the stream into odd and even numbers, then classifying odds and evens into ones and zeros was decided as a computationally simple and potentially effective random number generation algorithm.

Once this algorithm was developed, the output of the algorithm was tested using the FIPS 140-2 recommendations implementation inside the Debian ‘rngtest’ package. Results of this testing proved that the output of this prototype was not sufficiently random enough to pass testing.

```
rngtest: FIPS 140-2 successes: 0
rngtest: FIPS 140-2 failures: 104
```

Figure 4 – edited output of prototype 1 FIPS test

Understanding why the algorithm failed is vital to the continued development of the program, so some statistical analysis of the results given was performed to further understand why the prototype failed. Performing a monobit test on the output from the prototype showed a large imbalance between the number of ones and zeroes in the output, signifying a large bias in the algorithm. In order to correct this behaviour, further post processing needs to be applied to the output of the algorithm to produce less biased results.

3.3 Prototype 2

Based on previous research and the results of analysis of the programs output. More post processing was considered for this iteration. Because of the bias in the previous output, a von Neumann Correction algorithm was implemented in this prototype. Von Neumann Correction algorithms are used within industrial applications to remove the bias from pseudorandom bit streams (Naccache, 2011) and so its implementation within the prototype was undertaken to see if it could remove the bias from the microphone’s output stream. Implementation of this algorithm was provided by an external library, which was written for Arduino devices to implement the von Neumann Corrector algorithm in an Arduino environment. Once implemented into the prototype, the output stream was analysed first to determine if the corrector algorithm had balanced the number of zeroes and ones in the output stream. Results of analysis concluded that the correction algorithm had not done enough to prevent bias from occurring within the output stream.

Table 1 - Comparison of output files from two prototypes

	No. of Zeroes	No. of Ones	Ratio (zeroes : ones)
Prototype 1	44439	55370	1:1.25
Prototype 2	782328	2178376	1:2.78

Testing the output of the prototype using FIPS 140-2 provided similar results to the first prototype, and both prototypes did not produce a suitable result. Since post processing had failed, redesigning the algorithm using previous research was considered as the next step in developing a working prototype.

3.4 Prototype 3

For this prototype, it was decided that the algorithm used to capture the signal was would not be capable of producing a truly random output, and the method for using the signal produced by the microphone would have to be re-designed. After further research, using the microphone signal to seed a PRNG was considered as a design method instead of directly using the microphone signal. To complete this solution , the Arduino Cryptography library was used due to its ability to import the microphone's signal as a noise source and feed it into the entropy pool of a ChaCha20 cipher stream (Weatherley, 2023). Implementing this method then testing it with the FIPS 140-2 suite produced a positive result, marking prototype 3 as the first prototype capable of producing a sufficiently random byte stream.

```
rngtest: FIPS 140-2 successes: 120
```

Figure 5 – Edited Prototype 3 FIPS result

While this prototype does match the aim of creating a program that can produce sufficiently random numbers on IoT hardware, it still has some issues that need to be addressed. Mainly, it relies on a standard Arduino library that may not be compatible with all microcontrollers, making it not as generally available to all IoT devices. For this reason, developing a version of the prototype that requires limited libraries, and includes all of the required code in one place, could improve the overall portability of the software.

3.5 Prototype 4

Since prototype 3 relied heavily on an Arduino standard library, Prototype 4 was developed with the intent to create a more portable solution, for devices that did not meet the requirements to use this library. Prototype 4 used a much simpler random number generator which used aspects of prototype 2's functionality to achieve a more portable option for generating random numbers, at the cost of security. Prototype 4 differs from previous attempts as it applies more post processing to the output signal, by obfuscating the signal further using a Xorshift algorithm (Marsaglia, 2003). The first version of the prototype was tested using the FIPS-140-2 library, and the results concluded that the initial version of prototyped 4 was able to pass the majority of tests, however it failed specific tests.

```
rngtest: FIPS 140-2 successes: 80
rngtest: FIPS 140-2 failures: 1
rngtest: FIPS 140-2(2001-10-10) Poker: 1
rngtest: FIPS 140-2(2001-10-10) Runs: 1
```

Figure 6 – Edited Prototype 4 Initial Version Test Result

Knowing this, attempts were made to improve the results of the tests. After redevelopment, applying a second run of the Xorshift algorithm on one of the entropy sources provided a better result for the algorithm.

```
rngtest: FIPS 140-2 successes: 81
```

Figure 7 – Edited Prototype 4 test result with further processing

Since the prototype received no failures, development of the prototype was ceased, as this result confirms its ability to produce hardware generated random numbers.

3.6 Testing Prototypes to Confirm Results

The results of the development phase so far have provided 2 different prototypes that are capable to producing sufficiently random numbers from a physical noise source, thus completing the projects aim of building a ‘true’ random number generator for an IoT device. However, confirming the abilities of these generation methods require more testing than the precursory tests done so far, due to the inadequacies in the testing criteria of the standard FIPS-140-2 test. Diversifying the tests applied by using multiple standard testing suites will prove the efficacy of the functional prototypes, and confirm the inadequacies of the prototypes that do not generate a random signal. Eventually, applying both the NIST statistical test suite and the Dieharder tests was considered as the primary way to confirm the results of the FIPS 140-2 Tests.

In order to test the outputs of each randomness method, an output file had to be generated. However, determining the proper size of the output file to provide enough data for accurate testing proved difficult, as documentation on output file size was scarce. FIPS 140-1, the predecessor of 140-2, recommended using an output stream of 20,000 bits to use for statistical tests (National Institute of Standards and Technology, 1994). However, at the time of writing, this standard has been superseded twice and is likely heavily out of date, and is intentionally crossed out in FIPS 140-2, suggesting that a stream of only 20,000 bits is not suitable for modern RNG’s, and a higher amount of output data would be required. Knowing this, an output file size of 200KB was chosen as it was substantially larger than the outdated requirement, and would require minimal time to generate for the IoT device.

To generate the output files, each prototype was set up to generate their output stream and send it to the serial console of the IoT device, where it would be captured by an external device running CoolTermWin64, a serial terminal viewer for Windows capable of capturing the output of a devices serial console. Once an output file has been successfully generated, the file would be moved to a Windows Subsystem for Linux Instance running Ubuntu, which contained the implementations of each testing suite used. Testing of the output files was then performed using each of the testing suites, and their outputs were recorded for further analysis later.

Chapter 4 Results

This chapter shows the results of the statistical testing performed on the random number generation techniques described in methodology. Results have been collated based on the testing suite used, to allow easier comparison of results between prototypes.

FIPS 140-2

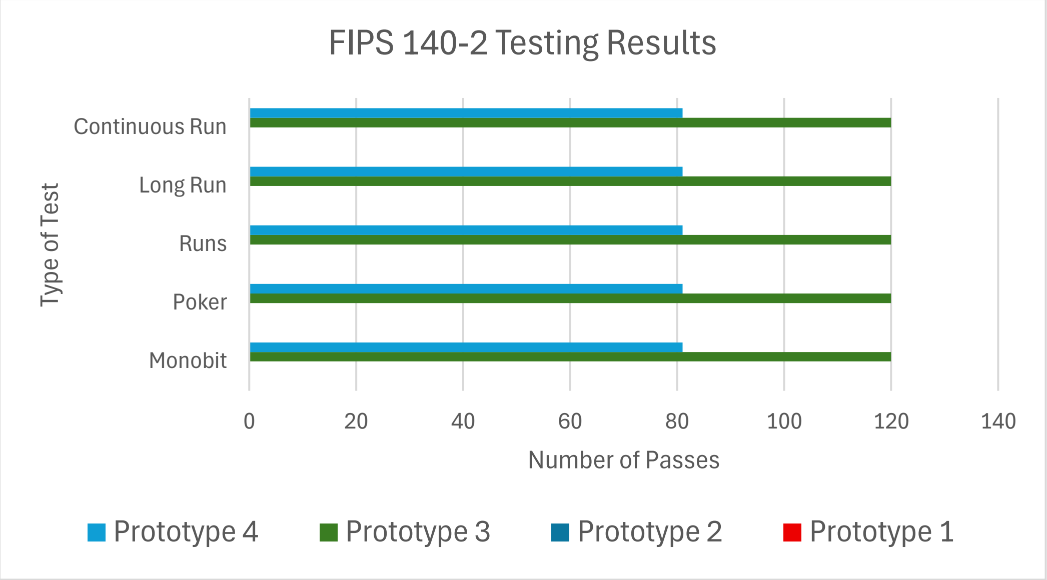


Figure 8 - FIPS-140-2 Testing Results

FIPS 140-2 delineates the success of a random number generator based on whether its output passes a number of statistical tests. This test suite divides the output file into 20,000 bit blocks, and runs each of its five tests on each block to determine if a block is sufficiently random enough. If a block passes, it is added to the number of passed tests. For this testing suite, prototypes 1 and 2 passed none of the tests.

NIST STS

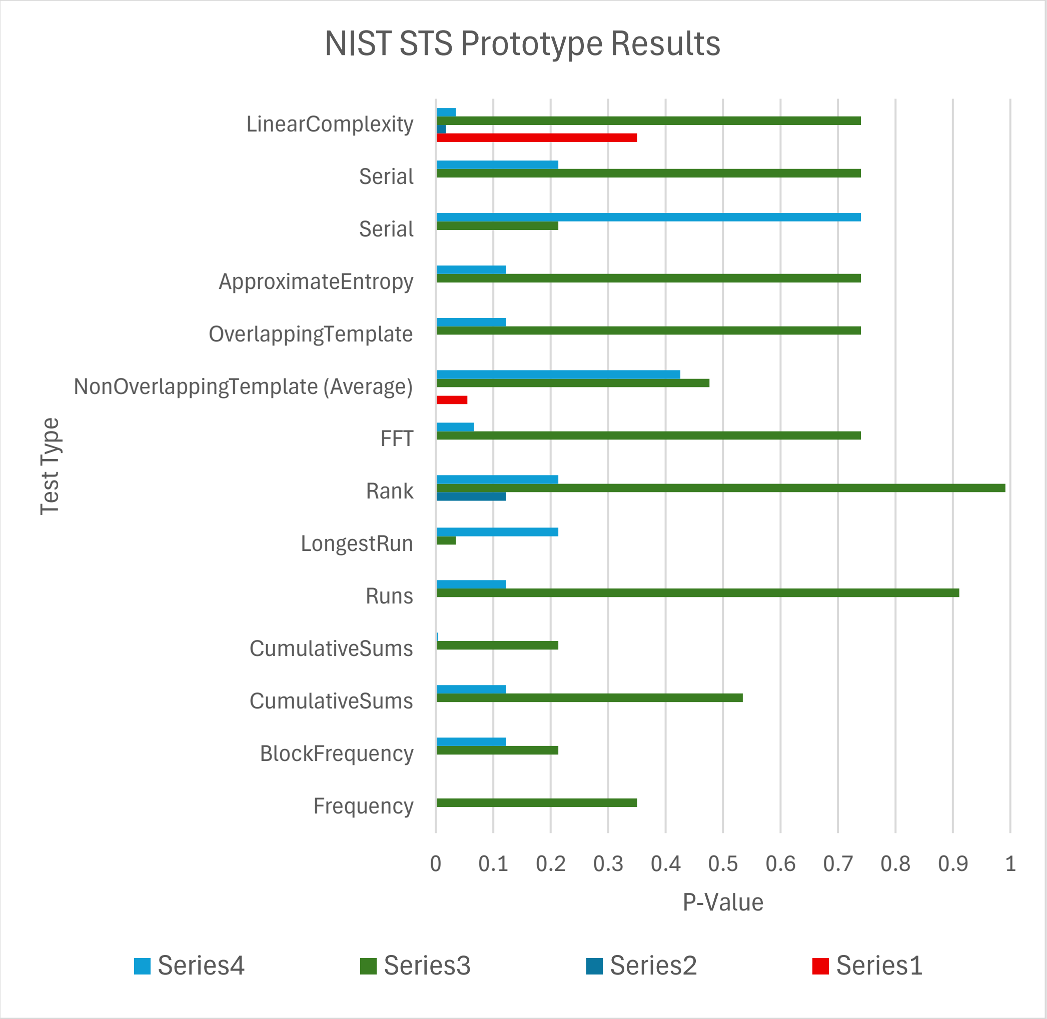


Figure 9 - NIST STS Prototype Results

The NIST STS works in a similar way to the FIPS 140-2 suite by applying statistical tests to a prototypes output stream, except that the suite performs its tests on the entire output instead of in chunks. Each test for each prototype was performed 10 times to ensure consistency in the results of each test. Unlike the pass/fail results found in the FIPS 140-2 testing suite, the strength of the tested random data is determined by its P-Value, with the higher P-value delineating a more random signal. This is described in more detail in section 5.1.2.

Dieharder

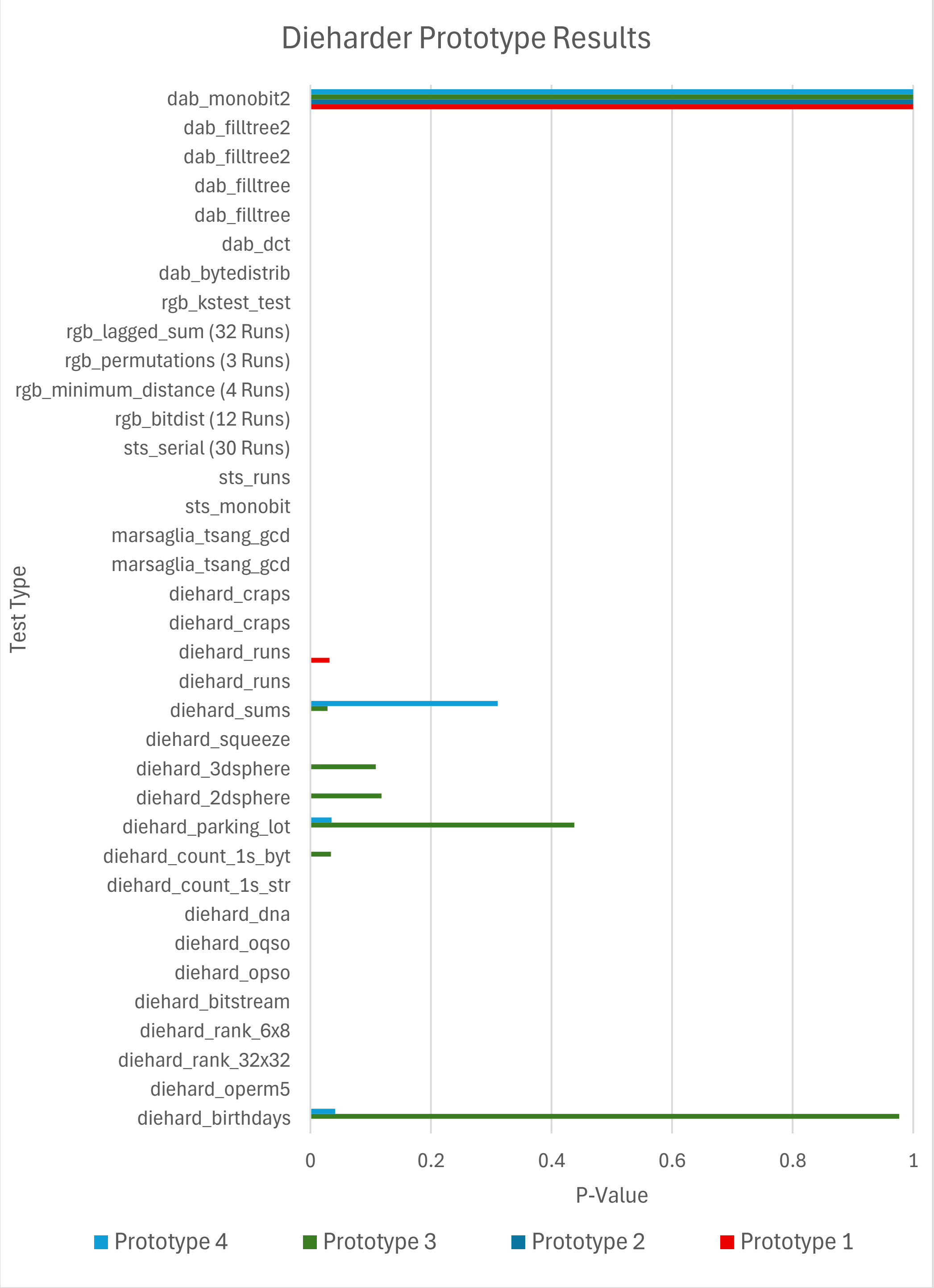


Figure 10 - Dieharder Prototype Results

For Diehard’s testing results, P-Value is defined in the same way as the NIST suite, except for some differences: any p-value that reaches 1 is considered a failure due to errors in the suites design, and not actually a truly random signal. Further analysis can be found in section 5.5.1.

Chapter 5 Discussion

5.1 Interpretation and Analysis of Testing Results

The outcomes of the statistical tests applied to each prototype can be used to confirm their ability to complete the aims of the project. While the FIPS-140-2 results and the NIST STS provided generally accurate results for each prototype, Dieharder provided an overwhelmingly negative result for every single prototype. However, using the test results from the other testing suites and data from the dieharder output, its possible to prove that dieharder's test results may be misconstrued.

5.1.1 Dieharder Test Anomalies

Dieharder's extreme test results require further analysis to understand why they are so different to the other testing suites' results. One way this can be done is by analysing the results of similar tests that all 3 suites run. One of these tests is the monobit test, which is common among statistical testing suites as a starting point to measure the randomness of a set of data. In prototype 3's testing results, the monobit test passes every single time it is run through the FIPS 140-2 suite, as well as every time it passes through the NIST STS. However, on dieharder, it fails. This pattern also occurs with other similar tests like the runs test. Such a deviation from the results of the other two suites while running the same tests imply that the dieharder suite is providing a faulty test result. If this is true, understanding why the suite does not provide accurate test results is crucial to improving the reliability of the tests.

Analysis of the raw test output could provide insight into diehards failure to provide accurate results. For many of the tests run, the result of the test is provided with a warning that the file has been rewound a certain number of times:

```
# The file file_input_raw was rewound 63985 times
sts_monobit| 1|   100000|    100|0.00000000|  FAILED
```

Figure 11 - Section of raw prototype 3 dieharder test result

Rewinding in this context means that the RNG output files data has been duplicated 63,984 times to fit dieharder's minimum size requirement for the test. This likely creates patterns in the test input, and therefore makes the resulting input data less random, leading to a test result that is inconclusive due to the software's need to rewind the input data.
Using the size of the input data and the number of rewinds applied to this test, we can estimate that dieharder needs approximately 13 gigabytes of random input data for this test alone, much larger than the 200 kilobytes of input data, and significantly greater than the 20,000 bits recommended for the FIPS 140-2 tests. This amount of required data isn't even close to the reported 228GB of data required for running every single randomness test in the dieharder suite (Darren Hurley-Smith, 2021). This explains the egregious results given by dieharder, as its tests have been giving a fraction of the amount of data required to complete its tests. While regenerating an output of the required size would likely make dieharder's results align more closely with the other testing suites, the time taken to generate such a large amount of input data for each prototype on IoT hardware is well beyond the scope of the project and likely well beyond the capabilities of the projects hardware. Dieharder's results may have provided inaccurate results, and may have added very little to the understanding of the generation techniques of each prototypes, but the test suites large input file requirements highlight an issue with the current testing standards in relation to IoT hardware.

For many devices with HRNG capabilities, the throughput of their RNG is not high enough to meet the demands of Dieharder's input file size requirements within a reasonable time frame. For a relatively small device such as the MIFARE® DESFire® EV1, extraction of 64MB of test output can take up to 12 days (Darren Hurley-Smith, 2021). This amount of data isn't even close to the requirements of dieharder, and to reach these requirements at the speed of the DESFire's RNG would take approximately 668 days, far beyond the scope of most research projects.

While Dieharder's tests might be adequate for personal computers, which can generate random numbers at incredibly high speeds, its testing requirements make it harder to verify RNGs on less sophisticated hardware such as IoT devices.

5.1.2 Analysis of FIPS 140-2 and NIST STS Results

Knowing that the Dieharder test results are likely not indicative of the actual capabilities of each prototype, analysis of each prototypes abilities will be performed using FIP 140-2 and the NISTS STS' results. Out of the 4 prototypes, prototypes 3 and 4 are the only ones that receive a positive result from both test suites. Repeated success from both working test suites indicate that these generation methods are suitably random enough to be considered random number generators. Its possible to compare the quality of the random numbers generated by both working prototypes by analysing the P-Values calculated during NIST's STS tests.

P-Values in NIST's STS are used to indicate the strength of a random signal, by comparing a signal to a hypothetical purely random noise source. P-Value's of 0 indicate an entirely non-random source, while P-Values of 1 are indicative of a purely random sequence (Andrew Rukhin, 2010). For the purposes of computer generated random numbers, any P-Value that is above 0 is considered random. We can use this to identify how random the NIST STS considers each prototype with incredibly low P-Values for prototypes 1 and 2, and much high ones for 3 and 4. One point of note however is that the NonOverlappingTemplate test almost always returns at least one positive P-value, even when other tests fail. This is likely due to the design of the test, which is expounded upon in NIST's SP 800-22. As described in the report, the Non-overlapping Template Matching Test searches for specific bit phrases throughout the input data, and scores higher if the phrase is found. This means that specific sequences of bits are being found throughout the input data, however the fact that only specific sequences are being found as opposed to all of the sequences in the NOTMT indicates that the test data is not random, and the phrases being found are merely coincidental.

This combined with the lack of results from other tests indicate that prototypes 1 and 2 are not fit for producing random numbers.

Prototypes 3 and 4 provide generally positive results, as they both pass the majority of tests that the FIPS 140-2 and NIST STS can provide. However, analysis of the NIST STS test results can provide a more granular insight into the capabilities of each prototype. Knowing that the P-Value denotes a similarity to perfect random noise, its possible to compare the two prototype test results to better understand how these two prototypes perform:

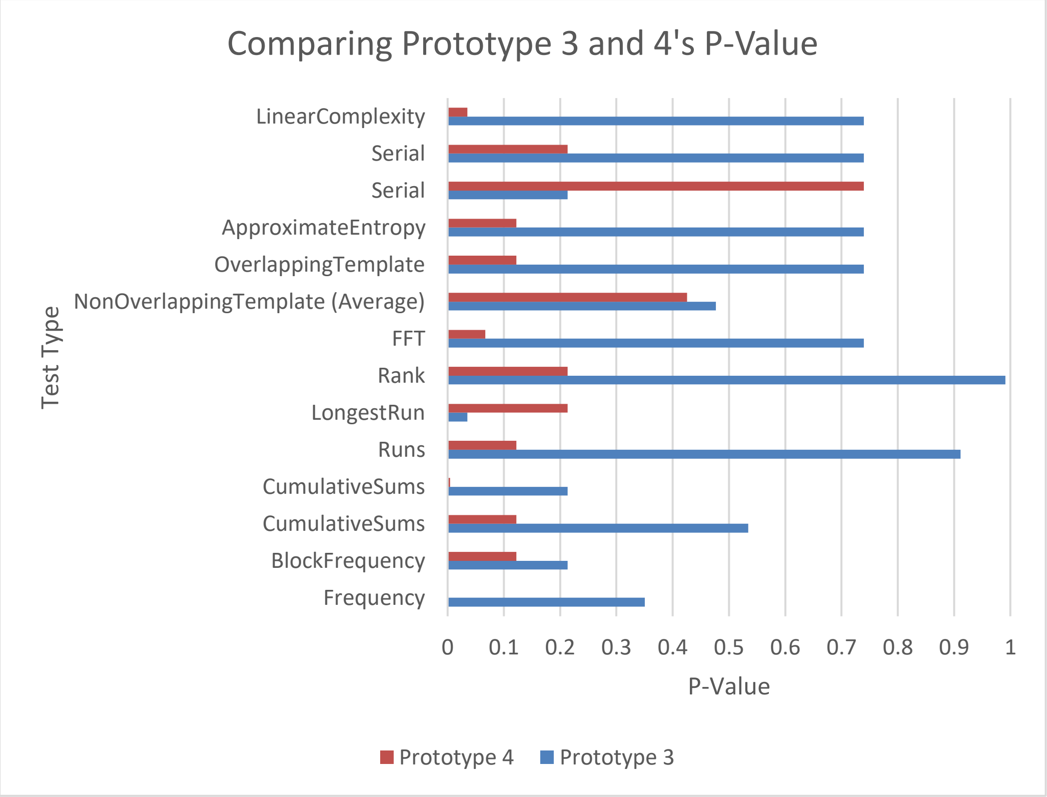


Figure 12 - Comparing Prototype 3 and 4's P-Value

Comparing the test data of both prototypes reveals that prototype 4 tends to score significantly lower than prototype 3 in multiple different tests. Since P-Value is a metric that delineates the closeness of a sequence to true random noise, its safe to assume that the output of prototype 4 is considerably less random than the output of prototype 3. This has significant implications for the security capabilities of prototype 4 and will require further analysis of the generation methods of each prototype to understand why each prototype gave such significantly different results.

FIPS 140-2’s results are significant less detailed than the NIST STS, but they can still be used to corroborate the findings from other test suites. Knowing that the output files from all prototypes are roughly the same, and knowing that the FIPS 140-2 suite works in 20,000 bit blocks, it is reasonable to assume that similarly performing prototypes would pass a similar amount of times for roughly the same amount of data. As seen in Figure 8 - FIPS-140-2 Testing Results, the results of prototype 3 are much higher than the results of prototype 4, indicating that prototype 4’s block divided output is not passing as much as prototype 3, despite being similar sizes. This implies that the output of prototype 4 is measurably worse that prototype 3, as sections of the prototype 4 output are failing statistical tests.

5.1.3 Comparison of prototype 3 and 4 generation methods

Understanding how both prototypes scored significantly different results require analysis of the generation methods implemented for each prototype. Looking at prototype 3, the generation method used by the Arduino Standard Library is based on the ChaCha20 algorithm, an encryption algorithm that is designed to generate random numbers unpredictable enough for cryptographic applications (Weatherley, 2023). Comparing this to prototype 4, prototype 4’s RNG algorithm is a Xorshift algorithm, a type of LCG that is designed to be quick and efficient in generating random numbers, but has been proven to have issues with statistical tests in the past (François Panneton, 2005). From this research, it is reasonable to conclude that the reason that prototype 3 scores higher during statistical testing is due to its algorithm being designed with a higher requirement to provide unpredictable numbers, and a greater need for security in mind, and therefore produces a higher level of random output to achieve this. Prototype 4’s algorithm by comparison produces a weaker random output, and the reasons for this should be studied to understand its strengths and weaknesses.

5.1.4 Prototype 4 and Xorshift

Prototype 4’s Xorshift algorithm has some advantages over prototype 3’s ChaCha based algorithm, but at the cost of serious cryptographic security flaws. The Xorshift algorithm requires a fairly small amount of computational effort to function, as its design requires a small amount of instructions to be performed. In comparison to the ChaCha Based algorithm of the Arduino Cryptographic Library, which uses a much more complex algorithm, the Xorshift algorithm is computationally more efficient and much faster than the ChaCha algorithm. However, the Xorshift algorithm’s speed comes from the fact that the algorithm is a linear operation (François Panneton, 2005), this means that the output of the algorithm is capable of being reversed, meaning that the input stream can be exposed. Since statistical testing of the raw noise source (prototype 1) concluded that that the source is not random, it may be deterministic. Since Xorshift has design issues that make it predictable, it is not fit for use within a cryptographic environment. However, this does not mean that the algorithm is completely without merit. While the focus of this project has been to generate a HRNG that could primarily be used for cryptographic applications, RNGs have a variety of uses outside of cryptography that could benefit from using such a generation method as designed in prototype 4.

5.1.5 Comparing prototype outputs and testing suites to a control output

Understanding the functionality of the prototypes developed over the course of the project can be hard to measure. While statistical testing can prove a given methods randomness, it is hard to understand how well it performs unless compared to an already proven method of random generation. Random.org is a website that provides random numbers using atmospheric noise that is skew filtered to provide a source of fairly reliable random numbers from a physical noise source (Haahr, 1999). Using the sites “Raw Bytes” generator, it is possible to generate a output file similar to the ones generated for each prototype.

Using this tool, a 128KB control output file was generated, and this file was run through the same statistical tests as the other prototypes.

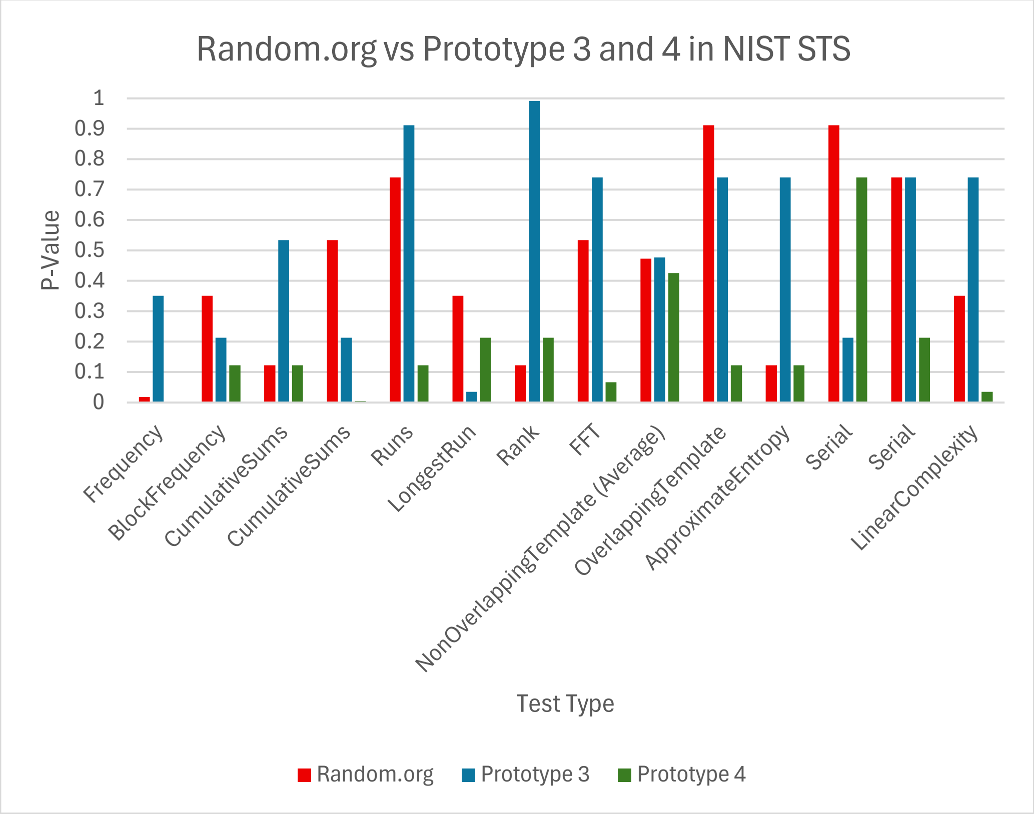


Figure 13 - Random.org vs Prototype 3 and 4 in NIST STS

While prototype 4 performed much worse than the output of random.org, prototype 3 was able to out-perform random.org's output on multiple tests, while still losing to it in others. While the test results are mixed, this shows that prototype 3 is able to generate random numbers at a similar level to the atmospheric noise in random.org's output.

Chapter 6 Conclusion and Future Work

6.1 Conclusion

Considering the outcomes of the testing results from all prototypes, it is difficult to determine the success of the project in terms of creating a suitable ‘True’ random number generator. While prototype 2’s design is similar to other RNGs that mainly use physical sources, results of statistical testing show that prototype 2’s output is substantially worse than other solutions, and is considered essentially non-random by the NIST STS. This is likely due to the noise output of the microphone being more deterministic than previously thought, and shows that a number generator that uses just a microphone’s input is not a suitable solution for generating sufficiently random numbers. Where the project’s value lies is in its ability to provide a statistically better random number generator solution, by using the raw noise signal to seed a variety of PRNGs. While the attempts at generating a purely HRNG proved fruitless, the combination of hardware noise sources and a PRNG provided a better solution.

Evaluating the abilities of each prototype was fairly successful, despite the issues with dieharder’s testing criteria. While Dieharder failed to provide an accurate test result, the results from the NIST STS allowed for a reasonable statistical analysis of each of the working prototypes, and allowed for comparison with known random sources.

Overall, while the design methodology of the HRNG attempts were flawed, a better solution to the issue of generating random numbers on IoT hardware was found. While issues were found during testing, the NISTS STS and the FIPS 140-2 statistical tests provided enough data to evaluate the abilities of each prototype. However, there is still a wide range of research to be done into the field of IoT random number generation, as well as improvements that could have been made to the testing process applied in this project.

6.2 Future Work

While the completion of the aims of the project have proven that it is possible to create and use a sufficient random number generator on an IoT device, there are several aspects of the project that could benefit from continued research. This section will discuss some of the major elements of the project that could be researched further to better understand the effectiveness of the solution found during artefact production.

6.2.1 Diversification of testing equipment

Over the course of the development stage of the project, the testing hardware used remained essentially the same throughout: a Wemos D1 R2 with a Arduino compatible microphone used as a noise source. While this setup was able to achieve the projects aim of producing a HRNG, and considering the likelihood that a majority of IoT devices would contain a microphone, its likely that the produced code would work on a majority of IoT devices. However, testing the code on another device would have provided greater evidence that the solutions provided in this paper were possible on a majority of devices. Work towards developing RNGs for similar devices has been undertaken in the past (Gangurde, 2017) that confirms the possibility of a HRNG working on a similar device, the possibility of the software solution devised during this project working on another device is unconfirmed. Testing the prototypes developed during this project on other devices would have made the conclusions drawn much more concrete, and would have added value to the research performed.

Prototype 3’s use of Arduino libraries would make it a likely candidate for supporting a wide range of IoT devices which are Arduino compatible. Considering the cross compatibility that Arduino libraries are designed to have with hardware, using any Arduino device with an A0 pin should be possible. Combined with any sensor that provides a suitable amount of entropy, prototype 3 should run seamlessly on such theoretical hardware. For example, using a Arduino Nano 33 IoT and a accurate enough temperature sensor like a BME280 would theoretically provide a similarly random result to the hardware used in this project. Continuing testing the prototypes developed during this project on different hardware would confirm the projects functionality in various hardware environments.

6.1.2 Diversification of Testing Suites and Testing Procedure

To confirm the abilities of the prototypes developed. A sample of the data each prototype produced was ran through Robert Browns ‘dieharder’ tests, the FIPS 140-2 statistical tests as well as the NIST Statistical Test Suite. While using both of these tests in conjunction confirmed that the outputs of these prototypes were random enough, diversifying the testing suites used could have further proved that the prototypes were performing adequately enough. Outside of these three tests, the selection for testing suites is small, however using a testing suite like TestU01 (Pierre L'Ecuyer, 2007) could help further confirm the findings of the two other testing suites.

Generating larger output files for each prototype could have revealed certain quirks with the generation of random numbers. During research into the types of pseudorandom number generators, it was found that a very common type of PRNG, the Mersenne twister, fails after producing a large enough output (James Hanlon, 2022). While the possibility of this is low for the prototypes produced, generating a larger output file may have improved the accuracy of the statistical tests. Generating larger output files wasn’t performed during this project due to the slow speeds of the serial port on the Wemos D1 R2, and because of the inefficient outputting code present on most of the prototypes. Despite the kilobytes of bits produced for each prototype being enough to accurately confirm if a RNG is sufficiently random enough, producing a larger file in the range of megabytes to gigabytes could provide a more accurate picture of the generation of random number to the statistical test suites used.

The results of the Dieharder tests were definitely affected by the small size of the bit streams generated for testing. As a result of the smaller output files, dieharder's tests were forced to implement a process called rewinding, where the input file is read through multiple times as a result of not having enough data to process. This process definitely skewed the results of these tests and would have benefitted from having a much larger input file.

The project's focus on statistical testing may have prevented exploration of other elements of the prototypes output. The NIST STS specifies in SP 800-22 that "no set of statistical tests can absolutely certify a generator as appropriate for usage in a particular application, i.e., statistical testing cannot serve as a substitute for cryptanalysis" (Andrew Rukhin, 2010). This means that no amount of statistical testing can fully prove a RNG's usefulness in cryptography, and that further detailed analysis of a RNG's design is required to properly certify a RNG for use in cryptographic applications. While statistical analysis has proven the efficacy of prototypes 3 and 4, their cryptographic readiness cannot be confirmed by the testing performed in this project. Considering the previous research done on the RNG methods in prototypes 3 and 4, the ChaCha20 algorithm in prototype 3 is more likely to be cryptographically secure, while prototype 4's Xorshift algorithm is definitely not cryptographically secure, and could not be used in any cryptographic applications securely. However, without a proper cryptographic analysis, it is difficult to ascertain their cryptographic abilities in a more detailed context.

6.1.3 Further Research and Development for RNG Algorithms

While the algorithms used throughout the course of the project were capable of producing a sufficiently random number, much of the code used for this process was developed by a third party. While the addition of third party code greatly improved prototype turnaround, their inclusion prevented any of the algorithms from being custom fit to the aims of the project.

If research into the subject matter of the project were continued, integrating a custom built RNG that was able to follow a more cryptographically secure design that also appealed to the hardware based entropy requirements of the project would significantly benefit the comprehensiveness of the project artefacts.

Bibliography

Alen Salkanovic, S. L. S. L., 2021. Analysis of Cryptography Algorithms Implemented in Android Mobile Application. *Information Technology and Control,* L(4), pp. 786-807.

Andrew Rukhin, J. S. N. S. B. L. L. V. B. H. D. V., 2010 . *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,* s.l.: s.n.

Apple, 2022. *Apple Platform Security,* s.l.: s.n.

Arduino, 2023. *Arduino Reference.* [Online]
Available at: https://www.arduino.cc/reference/en/
[Accessed October 2023].

Chung-Chih Li, H. S. R. K. S., 2005. Security evaluation of email encryption using random noise generated by LCG. *Journal of Computing Sciences in Colleges,* Volume 20(Issue 4), p. 294–301.

Darren Hurley-Smith, J. H.-C., 2021. Challenges in Certifying Small-Scale (IoT) Hardware Random Number Generators. In: *Security of Ubiquitous Computing Systems.* s.l.:Springer, Cham, p. 165–18.

Dichtl, M., 2003. *How to Predict the Output of a Hardware Random Number Generator.* Berlin, Springer.

François Panneton, P. L., 2005. On the Xorshift random number generators. *ACM Trans. Model. Comput. Simul.,* Volume 15, pp. 346-361.

Gangurde, S., 2017. *Accelerometer As Random Number Generator.* [Online]
Available at: https://www.ee.iitb.ac.in/~stallur/wp-content/uploads/2017/02/Saurabh.pdf
[Accessed April 2024].

Gutterman, Z. P. B. a. R. T., 2006. Analysis of the linux random number generator. *IEEE Symposium on Security and Privacy,* pp. 15-pp.

Haahr, M., 1999. random.org: Introduction to Randomness and Random Numbers. *Statistics,* Volume June, pp. 1-4.

James Hanlon, S. F., 2022. *A Fast Hardware Pseudorandom Number Generator Based on xoroshiro128.* [Online]
[Accessed April 2024].

Knuth, D., 1968. *The Art of Computer Programming.* Massachusetts: Addison-Wesley.

Kristinsson, B., 2012. *The Arduino as a hardware random-number generator.* [Online]

Available at: https://arxiv.org/abs/1212.3777

L Kleinrock, F. T., 1975. Packet Switching in Radio Channels: Part I - Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics. *IEEE Transactions on Communications,* 23(12), pp. 1400-1416.

Lewis, P. A., 1986. *Graphical analysis of some pseudo-random number generators.* 1st ed. Monterey, California: Naval Postgraduate School.

Marsaglia, G., 1995. *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness.* [Online]

Available at: https://ani.stat.fsu.edu/diehard/

Marsaglia, G., 2003. *Xorshift RNGs.* [Online]

Available at: https://www.jstatsoft.org/article/view/v008i14

Matthias Kaas-Mason, G. P. S. S., 2019. Comparison of Pseudo, Chaotic and Quantum Random Number Generators and their use in Cyber Security. *Group,* 4(1).

Microsoft, 2022. *rand.* [Online]

Available at: https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/rand?view=msvc-170

Naccache, D., 2011. *Encyclopedia of Cryptography and Security.* Boston: Springer, Boston, MA.

National Institute of Standards and Technology, 1994. *FIPS 140-1 Security Requirements for Cryptographic Modules.* [Online]

Available at:
https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=917970

Peter Kietzmann, T. C. S. M. W., 2021. A guideline on pseudorandom number generation (PRNG) in the IoT. *ACM Computing Surveys (CSUR),* 54(6), pp. 1-38.

Pierre L'Ecuyer, R. S., 2007. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software,* 33(4), p. 1–40.

Rafael Laufer, L. K., 2013. On the capacity of wireless CSMA/CA multihop networks. In: *2013 Proceedings IEEE INFOCOM.* s.l.:IEEE, pp. 1312-1320.

Shaanan N. Cohney, M. D. G. H., 2017. *Practical state recovery attacks against legacy RNG implementations.* [Online]
Available at: https://duhkattack.com/paper.pdf

Wang, Y., 2013. [Online]
Available at: https://webpages.charlotte.edu/yonwang/papers/liltest.pdf

Weatherley, R., 2023. *Arduino Cryptography Library, Generating random numbers.* [Online]
Available at: https://rweather.github.io/arduinolibs/crypto_rng.html

Appendices

Appendix A –Testing Output Tables

FIPS 140-2

Prototype 1

Table 2 - Prototype 1 FIPS 140-2 Results

Test Type	No. of Passes	No. of Fails
Monobit	0	104
Poker	0	104
Runs	0	104
Long Run	0	1
Continuous Run	0	0

Prototype 2

Table 3 - Prototype 2 FIPS 140-2 Results

Test Type	No. of Passes	No. of Fails
Monobit	0	75
Poker	0	80
Runs	0	80
Long Run	0	0
Continuous Run	0	0

Prototype 3

Table 4 - Prototype 3 FIPS 140-2 Results

Test Type	No. of Passes	No. of Fails
Monobit	120	0
Poker	120	0
Runs	120	0
Long Run	120	0
Continuous Run	120	0

Prototype 4

Table 5 - Prototype 4 FIPS 140-2 Results

Test Type	No. of Passes	No. of Fails
Monobit	81	0
Poker	81	0
Runs	81	0
Long Run	81	0
Continuous Run	81	0

NIST STS

Prototype 1

Table 6 - Prototype 1 NIST STS Results

Test Type	P-Value	Passes/No. of Tests
Frequency	0	0 of 10
BlockFrequency	0	0 of 10
CumulativeSums	0	0 of 10
CumulativeSums	0	0 of 10
Runs	0	0 of 10
LongestRun	0	0 of 10
Rank	0	0 of 10
FFT	0	0 of 10
NonOverlappingTer	0.05509016	N/A
OverlappingTempla	0	0 of 10
ApproximateEntrop	0	0 of 10
Serial	0	0 of 10
Serial	0	0 of 10
LinearComplexity	0.350485	10 of 10

Prototype 2

Table 7 - Prototype 2 NIST STS Results

Test Type	P-Value	Passes/No. of Tests
Frequency	0	0 of 10
BlockFrequency	0.000003	4 of 10
CumulativeSums	0	0 of 10
CumulativeSums	0	0 of 10
Runs	0	0 of 10
LongestRun	0	0 of 10
Rank	0.122325	0 of 10
FFT	0	0 of 10
NonOverlappingTer	0.00030966	N/A
OverlappingTempla	0	0 of 10
ApproximateEntrop	0	0 of 10
Serial	0	0 of 10
Serial	0	0 of 10
LinearComplexity	0.017912	10 of 10

Prototype 3

Table 8 - Prototype 3 NIST STS Results

Test Type	P-Value	Passes/No. of Tests
Frequency	0.350485	10 of 10
BlockFrequency	0.213309	10 of 10
CumulativeSums	0.534146	10 of 10
CumulativeSums	0.213309	10 of 10
Runs	0.911413	10 of 10
LongestRun	0.035174	10 of 10
Rank	0.991468	10 of 10
FFT	0.739918	10 of 10
NonOverlappingTer	0.47666725	N/A
OverlappingTempla	0.739918	10 of 10
ApproximateEntrop	0.739918	10 of 10
Serial	0.213309	10 of 10
Serial	0.739918	10 of 10
LinearComplexity	0.739918	10 of 10

Prototype 4

Table 9 - Prototype 4 NIST STS Results

Test Type	P-Value	Passes/No. of Tests
Frequency	0.000199	10 of 10
BlockFrequency	0.122325	10 of 10
CumulativeSums	0.122325	10 of 10
CumulativeSums	0.004301	10 of 10
Runs	0.122325	10 of 10
LongestRun	0.213309	10 of 10
Rank	0.213309	10 of 10
FFT	0.066882	10 of 10
NonOverlappingTer	0.42579993	N/A
OverlappingTempla	0.122325	10 of 10
ApproximateEntrop	0.122325	10 of 10
Serial	0.739918	10 of 10
Serial	0.213309	9 of 10
LinearComplexity	0.035174	10 of 10

Dieharder

Prototype 1

Table 10 - Prototype 1 Dieharder Results

Test Type	P-Value	Assessment
diehard_birthdays	0	FAILED
diehard_operm5	0	FAILED
diehard_rank_32x32	0	FAILED
diehard_rank_6x8	0	FAILED
diehard_bitstream	0	FAILED
diehard_opso	0	FAILED
diehard_oqso	0	FAILED
diehard_dna	0	FAILED
diehard_count_1s_str	0	FAILED
diehard_count_1s_byt	0	FAILED
diehard_parking_lot	0	FAILED
diehard_2dsphere	0	FAILED
diehard_3dsphere	0	FAILED
diehard_squeeze	0	FAILED
diehard_sums	0	FAILED
diehard_runs	0.00015719	WEAK
diehard_runs	0.03149772	PASSED
diehard_craps	0	FAILED
diehard_craps	0	FAILED
marsaglia_tsang_gcd	0	FAILED
marsaglia_tsang_gcd	0	FAILED
sts_monobit	0	FAILED
sts_runs	0	FAILED
sts_serial (30 Runs)	0	FAILED
rgb_bitdist (12 Runs)	0	FAILED
rgb_minimum_distance (4 Runs)	0	FAILED
rgb_permutations (3 Runs)	0	FAILED
rgb_lagged_sum (32 Runs)	0	FAILED
rgb_kstest_test	0	FAILED
dab_bytedistrib	0	FAILED
dab_dct	0	FAILED
dab_filltree	0	FAILED
dab_filltree	0	FAILED
dab_filltree2	0	FAILED
dab_filltree2	0	FAILED
dab_monobit2	1	FAILED

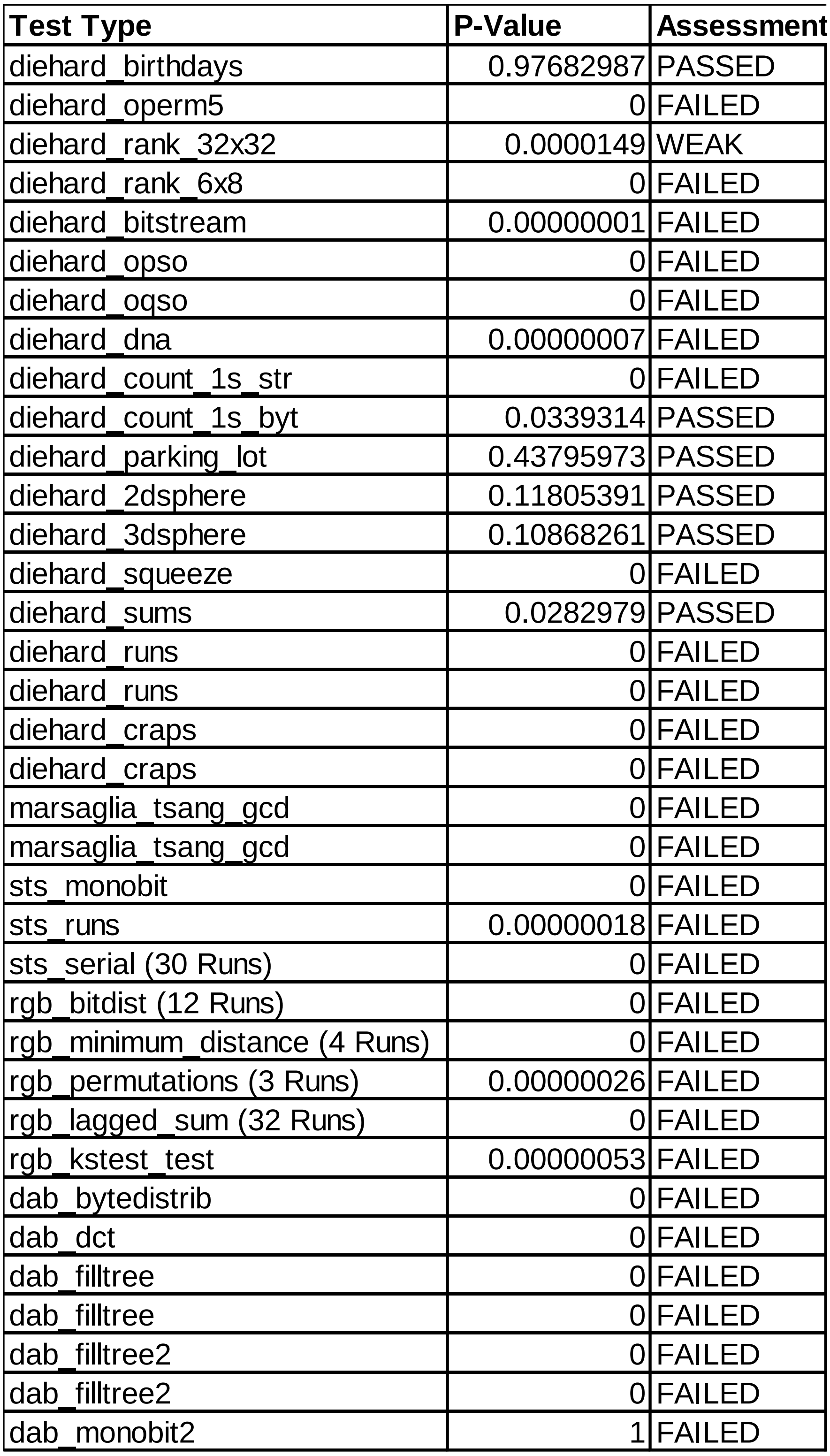
Prototype 2

Table 11 - Prototype 2 Dieharder Results

Test Type	P-Value	Assessment
diehard_birthdays	0	FAILED
diehard_operm5	0	FAILED
diehard_rank_32x32	0.00025672	WEAK
diehard_rank_6x8	0	FAILED
diehard_bitstream	0	FAILED
diehard_opso	0	FAILED
diehard_oqso	0	FAILED
diehard_dna	0	FAILED
diehard_count_1s_str	0	FAILED
diehard_count_1s_byt	0	FAILED
diehard_parking_lot	0	FAILED
diehard_2dsphere	0	FAILED
diehard_3dsphere	0	FAILED
diehard_squeeze	0	FAILED
diehard_sums	0	FAILED
diehard_runs	0	FAILED
diehard_runs	0	FAILED
diehard_craps	0	FAILED
diehard_craps	0	FAILED
marsaglia_tsang_gcd	0	FAILED
marsaglia_tsang_gcd	0	FAILED
sts_monobit	0	FAILED
sts_runs	0	FAILED
sts_serial (30 Runs)	0	FAILED
rgb_bitdist (12 Runs)	0	FAILED
rgb_minimum_distance (4 Runs)	0	FAILED
rgb_permutations (3 Runs)	0	FAILED
rgb_lagged_sum (32 Runs)	0	FAILED
rgb_kstest_test	0	FAILED
dab_bytedistrib	0	FAILED
dab_dct	0	FAILED
dab_filltree	0	FAILED
dab_filltree	0	FAILED
dab_filltree2	0	FAILED
dab_filltree2	0	FAILED
dab_monobit2	1	FAILED

Prototype 3

Table 12 - Prototype 3 Dieharder Results



Test Type	P-Value	Assessment
diehard_birthdays	0.97682987	PASSED
diehard_operm5	0	FAILED
diehard_rank_32x32	0.0000149	WEAK
diehard_rank_6x8	0	FAILED
diehard_bitstream	0.00000001	FAILED
diehard_opso	0	FAILED
diehard_oqso	0	FAILED
diehard_dna	0.00000007	FAILED
diehard_count_1s_str	0	FAILED
diehard_count_1s_byt	0.0339314	PASSED
diehard_parking_lot	0.43795973	PASSED
diehard_2dsphere	0.11805391	PASSED
diehard_3dsphere	0.10868261	PASSED
diehard_squeeze	0	FAILED
diehard_sums	0.0282979	PASSED
diehard_runs	0	FAILED
diehard_runs	0	FAILED
diehard_craps	0	FAILED
diehard_craps	0	FAILED
marsaglia_tsang_gcd	0	FAILED
marsaglia_tsang_gcd	0	FAILED
sts_monobit	0	FAILED
sts_runs	0.00000018	FAILED
sts_serial (30 Runs)	0	FAILED
rgb_bitdist (12 Runs)	0	FAILED
rgb_minimum_distance (4 Runs)	0	FAILED
rgb_permutations (3 Runs)	0.00000026	FAILED
rgb_lagged_sum (32 Runs)	0	FAILED
rgb_kstest_test	0.00000053	FAILED
dab_bytedistrib	0	FAILED
dab_dct	0	FAILED
dab_filltree	0	FAILED
dab_filltree	0	FAILED
dab_filltree2	0	FAILED
dab_filltree2	0	FAILED
dab_monobit2	1	FAILED

Prototype 4

Table 13 - Prototype 4 Dieharder Results

Test Type	P-Value	Assessment
diehard_birthdays	0.04106123	PASSED
diehard_operm5	0	FAILED
diehard_rank_32x32	0	FAILED
diehard_rank_6x8	0	FAILED
diehard_bitstream	0	FAILED
diehard_opso	0	FAILED
diehard_oqso	0	FAILED
diehard_dna	0	FAILED
diehard_count_1s_str	0	FAILED
diehard_count_1s_byt	0.00000673	WEAK
diehard_parking_lot	0.03499926	PASSED
diehard_2dsphere	0.00000294	WEAK
diehard_3dsphere	0	FAILED
diehard_squeeze	0	FAILED
diehard_sums	0.31083204	PASSED
diehard_runs	0	FAILED
diehard_runs	0	FAILED
diehard_craps	0	FAILED
diehard_craps	0	FAILED
marsaglia_tsang_gcd	0	FAILED
marsaglia_tsang_gcd	0	FAILED
sts_monobit	0	FAILED
sts_runs	0	FAILED
sts_serial (30 Runs)	0	FAILED
rgb_bitdist (12 Runs)	0	FAILED
rgb_minimum_distance (4 Runs)	0	FAILED
rgb_permutations (3 Runs)	0	FAILED
rgb_lagged_sum (32 Runs)	0	FAILED
rgb_kstest_test	0	FAILED
dab_bytedistrib	0	FAILED
dab_dct	0	FAILED
dab_filltree	0	FAILED
dab_filltree	0	FAILED
dab_filltree2	0	FAILED
dab_filltree2	0	FAILED
dab_monobit2	1	FAILED