Final Year Project Report

Full Unit – Final Report

A MINIMAL SET OF END USER FRIENDLY STATISTICAL TESTS

Verona Tu

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Darren Hurley-Smith

Department of Computer Science

Royal Holloway, University of London

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 10305

Student Name: Verona Tu

Date of Submission: 30/05/2022

Signature: VT

Table of Contents

Abstract

Random numbers are the cornerstone in achieving high level of security in the computing world. Many cryptographic applications and hardware place their trust in random number generators to be able to generate true random numbers. This leads to the question of how end users know they are getting true random numbers and not pseudo random numbers from RNGs. In my research I explored the different types of threats that RNGs face and how having accurate statistical testing can help reduce the risks of them. Also looked into the history of statistical test suites such as Diehard (Marsaglia, 1996) and its successor Dieharder (Brown, 2007) which took Garsaglia's Diehard Battery of Tests of Randomness and NIST SP800-22 to re-implement and expand upon to create what it claims to be a "Swiss army knife of random number test suites" [1] boasting a large library of 61 different statistical tests. The security issue that becomes apparent when looking into the uses and production line of RNGs is that they can be easily tampered with adversaries inserting malware such as trojan horse or backdoors into the RNGs [2]. This is where the importance of using statistical testing suites in order to verify the integrity of the RNGs once they've reached the end user. Within this report, I explore the integrity tests suites FIPS 140-2 and Ent. The research looks into how the statistical tests handle randomly generated data that has been tampered with, specifically if the tests are able to detect abnormalities and fail the test. The results showed that the statistical tests used in Ent (Entropy, arithmetic mean, Chi-square test, Monte-Carlo, and Serial Co-efficient correlation) were all able to successfully fail the adversarial generators with approximately 71% of the time under the Ent guidelines but under FIPS 140-2 guidelines it's discovered that most of the tampered data were able to pass all the tests undetected. This research became the foundation of the program I built in order to create a user-friendly program that users with little to nonstatistical knowledge would be able to comfortably use to validate their RNGs in order to be confident in their integrity.

Chapter 1: **Introduction**

This chapter will cover a brief overview of the research areas involved in this project and outline the scope. It will also address the concerns about RNG tests prompted my own research into statistical testing and the motivations for this project.

1.1 Overview

Random Number Generators are a core component of algorithms and protocols within cryptography. They are relied upon to produce sequences of numbers that are sufficiently random in a way where there are no patterns in the sequences that can exploited by adversary to use maliciously. Therefore, in order to authenticate the randomness of these generators, statistical tests of randomness were developed for and by researchers in the industry and academia. Many of the well-known test suites that I also look into: Diehard [3], TestU01 [4], ENT [5], CryptXS, and FIPS 140 all require a certain level of knowledge in statistical analysis in order to interpret the results they produce.

Random number generators gain credibility by being examined to have high entropy through entropy modelling. Issues about the integrity of RNGs arise once it reaches the end user since through the process of them receiving the products, they have no guarantee the RNGs haven't been tampered with or broken. One such case occurred to Intel processors [2] causing the generated keys to become predictable so adversaries could easily gain access. This is the purpose in which end-user need to have reliable methods to test that when they receive their RNGs it hasn't been tampered with and in turn the importance of test suites such as the ones mentioned before. Yet the number of resources available for business who are the end users of products that use random number generators are still few and far between. Many businesses will receive products that they rely on for the security of their company, but many will lack the time, knowledge, and money to either tests the products themselves or hire professionals to do so. They have to trust in the testing certification in which the manufactures have conducted as well as the fact that further up the production chain the products have not been tampered with.

One that is considered an industry standard is the FIPS that are used in NIST for industry hardware to get certificated have significant flaws. They are identified to be lightweight because of the they are often used in the industry to perform online tests on hardware devices and these sorts of quick statistical tests that in turn make them extremely lightweight [6]. Another of the more popular battery is Dieharder, the successor of Diehard. This battery is a very reputable and possess a large library compiling of 63 different statistical tests. In theory it would stand to reason that the more tests in a battery when applied would result in a higher chance of discovering and failing a tampered RNG. In reality, as Boris Ryabko mentions in his paper "the larger the battery, the less time it can spend on each test and, therefore, the shorter is the test sequence" [7]. This in turn causes minimises the ability of the tests to identify small deviations from randomness and can cause long process times.

Next in this report is Chapter 2 where I will go into depth and discuss through literature reviews on the threats RNGs face to understand the importance of statistical testing as well as understand the current batteries that exist for this purpose. Chapter 3 describes my analysis stage where I detail out my aims for the project, research method, look into professional issues that would concern my project, and the design outlines for the implementation of the program. Chapter 4 builds off from Chapter 3 and discusses the findings from my analysis and how I implemented those findings into my program. The final Chapter 5 wraps up the report with an evaluation and critical analysis of the end results.

Chapter 2: **Background**

In this chapter I will be displaying some literature reviews I conducted. They include the discussion on why there is a need for testing RNGs and the threats that they face. I will also have a brief look at what the main uses of RNGs are and how a defective RNG would impact them. Finally, I will introduce what current statistical test batteries exist for testing randomness, these include the following: Diehard [3], TestU01 [4], ENT [5], CryptXS, and FIPS 140 [8]. The purpose of looking at existing batteries is to evaluate what tests are currently in use and which ones I will consider analysing further to incorporate into my own set of statistical tests.

2.1 Reviewing Different Threats RNGs Face

In cybersecurity every part of a security system will have flaws and weaknesses that adversaries will exploit in order to perform actions with malicious intent. In the case of random number generators, one of their main purposes in a security system is essentially create the keys that keep the secret data secret. This means that it's a good target for attacks to focus on exploiting. RNGs that are of low quality will present entry points for adversaries to attack and in order to tell if what you're using is low quality, you would need statistical tests of randomness in order to tell you. This is what the importance of having an accurate statistical test suite is. In this paper I will look into show the importance of having accessible and accurate test suites by highlighting and explaining the threats and attacks that RNGs face when you deploy unfit RNGs that were given the pass by poorly performing test suites.

2.1.1 Different Types of Attacks

The following three attacks that I will be looking at, are attacks on software PRNGs. Some of the attacks are claimed to be more theoretical and academic but are said to be possible scenarios that can happen [9]. Most of the attacks I will look at are cryptanalytic attacks, however there are a large number of hardware attacks that can occur. Since I am looking at how statistical testing helps with minimising attacks, I will not be mentioning and going into depth about any hardware attacks but there are still a large threat to RNGs as well.

2.1.1.1 Direct Cryptanalytic Attack

When an attacker is given a stream of random bits, they are able to spot and indicate the PRNG output from a true random output. Despite effecting many process that use PRNGs, there are a couple in which this attack would not apply too. One of which is when generating the keys for triple-DES keys since the keys themselves are not displayed or directly seen at any point of the process. Therefore, the attackers would not even have the opportunity to pull of this attack.

If we were to look at how a direct cryptanalytic attack would work on a real PRNG, one example that I will look at is the RSAREF PRNG. The main premise that you need to know about this PRNG is that it's comprised and built around the two operations: MD5 hashing and addition modulo 2^{128}. Out of these two operations, modulo 2^{128} is the one that has been shown to be the weaker function of the two and one of the attacks it can fall prey to is a partial precomputation attack. For this attack, the attacker will choose a number of expected successive outputs, *g*, and then calculates the hash of the *g*th possible counter value. An example given by Kelsey, Schneier, Wagner and Hall (2017), "*With* 2^{32} *outputs, an attacker would need to do a* 2^{96}*precomputation to mount the attack; with* 2^{48} *outputs, he would need to do a* 2^{80}*precomputation.*". [9]

Concluding from this, a direct cryptanalytic attack utilises repeated sequences that are produced within the stream of pseudo random numbers generated from low grade RNGs.

2.1.1.2 Input Based Attacks

This particular attack can alter the RNGs input in order to cryptanalyze the RNG. This method allows the attacker to expel the existing entropy from the system and place it into a known state. There are three types of input attacks that an attack can fall under:

> **Chosen input attacks** are used against tokens that are tamper-resistant such as smart cards through a cryptanalytic attack.
> **Replayed input attacks** are also typically used against similar targets as chosen input attacks, the only difference being the execution. Replayed input attacks demand less control needed from the attacker in comparison.
> **Known input attacks** works when the design of the RNGs inputs are made to be difficult to predict, however in turns makes it easier in some cases. One example mentioned in the paper by Kelsey, Schneier, Wagner and Hall (2017), "*is an application which uses hard-drive latency for some of its PRNG inputs but is being run using a network drive whose timings are observable to the attacker.*". [9]

2.1.1.3 State Compromise Extension Attacks

This attack is reliant on having insider knowledge about the RNG either through just previous knowledge or previous attack attempts. This information is used to attempt to foretell the future output. The goal of a state compromised extension attack is use the information learned about the internal state, *S*, and launch another successful attack to disclose unknown RNG outputs that were unknown prior to the compromission of *S*. How these leaks or insecure states happen can be due to an insufficient starting entropy. This will allow the attacker to obtain an initial guess. The other way I mentioned earlier that *S* could become compromised was from previous attacks.

- Backtracking Attacks is used to lean about previous RNG outputs when its state is compromised.

- Iterative Guessing Attack uses knowledge of when the RNG was in its compromised state in order to discover what the state will be at time $t + \epsilon$. This gathers a stream of guessable inputs for the attacker to use.

- Meet in the Middle Attacks is a combination of backtracking and iterative guessing attacks. Knowing S at the times *t* and $t + 2\epsilon$ will allow the attacker to backtrack and gain knowledge of *S* at time $t + \epsilon$.

- Permanent Compromise Attacks is something that can occur if when the attacker compromises *S* at time *t*, both future and past S values are susceptible to an attack.

This method of attack looks at exploiting the windows of opportunities that appear from the software of the RNGs themselves and there is very little to what statistical tests of randomness can help identify and minimise the risks for. Though checking that the RNGs are able to generate truly random numbers would deter the attackers from finding out the random numbers too quickly if they're less predictable.

2.1.1.4 Trojan Horse

Based on the Greek mythology, the concept behind the trojan horse attack is that attacker somewhere along the commercial line the attackers have tampered with the RNG devices to include a backdoor for the attackers to gain access the system whenever. So that when it reaches the end user, their devices have already been sabotaged. This can be software or hardware weakness with new research paper [2] that suggests it's possible to (Goodin, 2013) "*build into processors to surreptitiously bypass cryptographic protections provided by the computer running the chips*".

There was an attack on Intel processors that is mentioned in Goodin’s article that in turn damaged the RNGs instructions as well. This attack worked by diminishing the amount of entropy from 128 bits to 32 bits used by the RNGs. This meant that keys that were being generated by the tampered chips would be severely weakened to the point where attackers were able to guess them with little effort. The issues that arose from this attack was that these weakened RNGs were not being picked up for being deficient by the standard tests that were mandated at those times like NIST P800-90 and FIPS 140-2.

2.1.2 Evaluation

All things considered, there are many different types of threats that RNGs face and not all I can tackle with just a statistical test of randomness. The threats that do pose a large problem and statistical tests can help minimise are the trojan horse attacks and direct cryptanalytic attacks. The weaknesses that these attacks highlight is that generators that produce repeated sequences are low quality or have been tampered with and state renounced tests aren’t able to pick up on this. Therefore, for choosing which tests I would want to include in my test suite, I want to be looking for those that are able to identify tampered RNGs and pick up on repeats.

Having looked at the threats end-users face due to faulty product and the importance of having reliable statistical testing of randomness available for them, my next course of action was to take a look at what statistical tests and batteries exist currently. The main purpose for this research is to see what options are currently available for the public to use and the accuracy of these tests. Since I’m not aiming on creating my own statistical test but rather put together a suitable test suite for end users, this research also helped me to evaluate the possible tests that I would want to include in my own test suite.

2.2 Review on Existing Statistical Tests of Randomness

The goal for this review is to look at what the standard throughout the years was for testing the integrity of random number generators as well as exploring how each test is composed. I will be looking at how some of the first test suites such as diehard were later revised and why they worked well as well as the issues it possessed. Then going from there, we see the newer tests such as test user 01 and everything in between and see how throughout the years the improvement in randomness testing as well as where they fall short.

2.2.1 Timeline of Different Statistical Tests of Randomness

Diehard (1995)

The most notable early statistical test battery is the diehard tests. At the time in which the diehard tests were published which was in 1995, they were very highly approved by its academic peers. The tests were published by George Marsaglia who was already well known among his peers for developing algorithm RNGs. Marsaglia built diehard in order create a package that consists of well-established tests that can process large files produced by RNGs. At its core, the diehard tests are software tests that consist of 15 independent statistical tests that are applied over a stream of 32-bit integers and like those to come after it tests the randomness of pseudo-random number generators. Assessing the random of these generators refers to how breakable and usable the generators are to users.

For the diehard tests to run appropriately, the files must be compromised of large sets of random positive integers. At the end it will produce a statistical p-value which can then be interpreted as evidence against non-randomness. The diehard tests consists of 15 different statistical tests: birthday space test, OPERM5 test, binary rank test for 32x32 and 6x8 matrices, bitstream test, OPSO test, count the 1s test, parking lot test, minimum distance test, random spheres test, squeeze test, overlapping sums test, runs test, and the craps test.

The diehard tests are highly regarded as one of the most difficult to pass, especially the Birthday Space Test. As mentioned in a study that included the Birthday Space Test by Marsaglia and Tsang [10], where RNGs such as gnu rand() which is a random number generator used in C/C++ programming language would fail. For this particular RNG, it showed there were far too many values generated that were less than the mean 4. This particular algorithm of Marsaglia's Birthday Space Test is claimed to be the "most-difficult-to-pass version" [10] and uses the following values for the n, m, λ; $n = 2^{32}$ and $m = 2^{12}$, $\lambda = 4$. The slight issue with this test is that since the $n=2^{32}$, it is implied to have a 32-bit limit.

A couple years later in 2004, Robert G. Brown created a test suite based off Diehard named Dieharder [1]. The aim of creating Dieharder was to create a way an easy way to test and time RNGs with the base of the tests being from Diehard with many more added. It has grown to amass a huge collection of tests and is able test files that are >2GB in size.

FIPS 140-2 (2001)

FIPS 140-2 cybersecurity certification program published by NIST which detail the security standards in which cryptographic modules created by private sector companies have to meet. FIPS 140-2 is the predecessor to FIPS 140-3, and it was industry standard that RNGs were required to be FIPS 140-2 compliant and is heavily relied on for the verification of randomness for RNGs. FIPS was designed to create a standard that evaluated both the hardware and software elements of RNGs. As part of this documentation, it would need to have some way of implementing statistical tests for the RNGs in the high-level security module [11]. For this NIST created a statistical test suite for the testing that is referred to as FIPS 140-2 STS. The documentation states that if tests statistical tests need to be run, they are to use a stream of 20,000 from the RNG in question on these tests:

- **The mono bit test**: Finds and counts the quantity of ones in the stream. Then assigns it to X and if it satisfies $9{,}725 < X < 10{,}275$ then it passes.
- **The poker test**: The bit stream is divided into 4 segments. Since the bit stream is 20,000 bits each segment would be 5,000. With *f(i)* representing the occurrences of each value, evaluate X as shown below

$$X = \left(\frac{16}{5{,}000}\right) \cdot \sum_{i=0}^{15} [f(i)]^2 - 5{,}000$$

- **The runs test**: This test counts the length of each uninterrupted sequence of 1's or 0's each one is considered as 'a run. Table 1 displays the passing conditions set by FIPS for this test. For each type of run there should only be that amount of runs per length as demonstrated by the table.

Length	FIPS 140-2
1	2,343 – 2,657
2	1,135 – 1,365
3	542 – 708
4	251 - 373
5	111 - 201
6+	111 - 201

Table 1: Runs Intervals

- **The long runs test**: FIPS 140-2 defines a long run as a run being larger than 25 bits. The probability of this is 0.000298 per block of 20,000 bits and if this occurs then it has failed the test
- **Continuous run test**: This checks if there are any consecutive identical 32-bit patterns. If any are detected, then it has failed the test.

For something that has been held to such a high standard within the industry, you would expect for it to deliver to the same standard. However, Hurley-Smith et al.'s research demonstrated how it failed to identify the adversarial bias [6]. When given data that has been tampered with, in this studies case it uses ε-hole, σ-counter and *t*-counter, they passed the tests used by FIPS. The tests in question are the monobit, Poker and Run-long tests that were run according to the FIPS 140-2 guidelines.

TestU01 (2007)

TestU01 was created by Pierre L'Ecuyer and Richard Simard. The battery itself is said to have everything the other big test suites have all available in the one software. The testU01 library is extensive and boosts to have a larger set of software tools compared to other batteries available such as diehard or Ent. TestU01 have four various groups for analysing RNGs depending on your need is. These include implementing pre-programmed RNGs, implementing specific tests, implementing batteries of statistical tests, and applying tests to entire families of RNGs. They also provide several batteries of tests of three different sizes. You have "small crush" which is 10 tests long and takes on average 4 minutes, "crush" which has 96 tests and would take around 1.4 hours, then the "big crush" which consists of 160 tests and takes on average 4 hours to complete.

Now taking all this into consideration, the testU01 has an immense library of tests which would give you accurate results. However, it is not very user friendly since, there are so many different tests a non-statistician would most likely not know how to understand and interpret the results. Other limitations that TestU01 faces is that it only takes in 32-bit inputs and can only interpret them as values within the range [0,1]. This causes it to become more sensitive to defects in the most-significant bits.

ENT (2008)

This Linux based testing tool was designed and implemented by John Walker. He then went on to release it with a free license, making this a very accessible battery for people to use. Ent applies a total of 5 different statistical tests on a stream of bytes and returns the results. Ent has two different operation modes: binary and byte. The difference between these two modes is simply in binary mode Ent will consider every bit in the sequence whereas in byte, the sequence is divvied out in bytes. Because of this difference in allocation, the two modes will in fact produce different results from the same file. When analysing this battery, I will only be looking at the byte version of Ent since it is the default mode. The five tests the encompass Ent are:

- Entropy is a scientific concept most notably associated with the state of randomness, disorder, and incertitude. It is also a measurable physical property. The maximum theoretical entropy of the sequence of bytes would be 8. So, the closer your result is to 8, the denser and more random it is.

- Chi-square tests are very common in testing for randomness but is very sensitive to errors in PRNGs and are prone to produce false negatives [12]. The test will compute the frequency of a symbol and compare it to the frequency of an expected uniform distribution. For the comparison, chi square statistic is computed.

- Arithmetic mean is what it sounds like, sum up all the bytes in the sequence and divide by the file length. A random sequence will have a mean of 127.5 and anything else means the values in the sequence is constantly high or low.

- Monte Carlo Value for Pi uses a sequence of 6 bytes to form 26-bit co-ordinates in a square. It then counts up the number of points fall within a circle that lays within the square. The number counted up is meant to estimate pi and a sequence that's truly random will be able to estimate pi to high degree of accuracy.

- Serial Correlation Coefficient checks the correlation of each consecutive value in the sequence. A truly random sequence would have an extremely low correlation.

When looking at the overall integrity of the Ent test suite, past research has shown it to be a well thought out and put together test battery. Even when increasing the number of statistical tests to 7, there was no drastic shift in results and the 5 tests were just as satisfactory and comparable.

CryptXS

CryptXS is a widely known and used battery in the commercial industry for statistical testing. The software for CryptXS is not publicly released so I have no access to use this software. However, I decided to mention it since many papers [9] [10] have referred to it as a popular battery that is currently used in the industry. What I do know about this commercial battery are some of the tests that is a part of the software. Tests include frequency, binary derivative, change point, runs, sequence complexity and linear complexity.

2.2.2 Conclusion

Having reviewed the current popularised test batteries that are available, I can conclude that the test batteries that are most beneficial for me to analyse further are Dieharder, FIPS 140-2 and ENT. If I were to describe the kind of program in which I intend to put together based on what test suites exist currently, it would be to incorporate the reliability and user friendliness of Dieharder but with the number of tests and speed of ENT. Hence why these are two of the test suites I plan on analysing their tests further and evaluate which statistical tests used in these two test suites I would want to use. FIPS 140-2 on the other hand is a testing suite that should hold the same prestige as Dieharder and is heavily used in the industry, however, there have been heavy criticisms towards this test suite and as mentioned above is unbearably lightweight. It would be useful to compare the results from FIPS which is consider an industry standard to ENT and Dieharder.

2.3 Summary

This chapter displays the background research I conducted in order to proceed with this project and to demonstrate my understanding of the importance of RNGs, the issues in which they are faced with and how statistical testing and analysis is important in dealing with these issues. The latter half of the literature reviews mainly focused on what statistical testing options are currently available and how can I incorporate and improve on what is already being used in this industry.

Chapter 3: **Research Method**

3.1 Significance of the project

Dieharder is the test suite that is most similar to what I aim to achieve with my project, in terms of the primary aim of making an easy to use and accessible method to test random number generators. The difference in direction is that my main aim is making a quick and reliable test suite which is comprised of a minimum amount of statistical test. Whereas with the intentions behind the design of Dieharder is to be able to handle any situation in which a user would use statistical testing, giving users a complete library of statistical tests and allowing them to pick and choose what tests they want to be using. The creator of Dieharder claims it to be a “Swiss-army knife of random number test suites” [1]. The issue despite it being very user-friendly and easy to understand how to use, to be able to understand the results from Dieharder tests the user would still need to have a basic understanding of how each statistical test works and which tests they should be using. I want to be able to create a program that has the UI user friendliness and result reliability of Dieharder but simplify the results further in a way that users who have no knowledge of statistical testing is still able to comprehend the results.

3.2 Aims of the project

The goal for the project is to tackle the issue of the lack of reliable and user-friendly statistical testing methods used to measure the randomness of random number generators. For the project, I have set out the following objectives that I aim to achieve by the end of the project. The completion of all the following objectives will constitute a successful end product:

1. Find a minimal number of statistical tests needed to still be reliable
    a. Ent is the smallest test battery I will be observing; therefore, the minimum should not be under the amount Ent has which is five.
    b. Tests should be able to detect the tampered random data presented as well as purposely non-random data.
2. Build a user-friendly GUI that is easy to navigate, features should include:
    a. Allowing users to upload their own files to be tested from their storage
    b. Easy to view results screen that will display the in a simple to read format the results from the statistical tests on whether or not user’s data passed
    c. Visible upload and progress bar
    d. Easy to navigate interface
3. Evaluate the results outputted by the different batteries of statistical tests through my own algorithm to determine how random the RNG data is based on test result patterns I have observed through experiments with each of the statistical tests
    a. This requires an analyse of a set of tests that will be conducted on the different statistical tests I plan to use. The test results I will be analysing will be coming from objective (1).

3.3 Research Approach

The first step in my project is to conduct a series of tests on the following existing test suites: ENT, FIPS 120 and Diehard. These tests will consist of giving the test suites 500 runs per condition in order to see how the different statistical tests evaluate the random data in each condition. The four different conditions include: a tampered file of random data produced by implementing a backdoor within the data; a small file filled with urandom data; a normal file around 100mb of urandom data;

and a large file of 1GB of urandom data. Urandom is a RNG that can provide a sufficiently random output created by IMB for the Unix operating system [13]. The reason these are the conditions chosen is because the areas of the statistical tests I want to look at are whether or not the statistical tests are able to detect malware within the data, in this case a backdoor, as well as looking at the speed and reliability to process small and large files respectively. Through this testing I will pick out the test in which I want to include in my own set of tests and will aim to include around 7 tests.

After completing this stage, I will move onto the actual development of my project. The basic design I will undertake for this project is have it run the users chosen RNG, catch the results that are outputted and pass them through each test battery. Then I will catch the results that each test battery releases and then run them through my own algorithm to review the test results and determine whether or not they have indeed passed the statistical tests and is random. I will be using an agile development method for the production of my project. The reason being I will implementing each test one at a time, so after each test is implemented, I will then test it to make sure it is working well by itself and with the other tests and components. The design of the program is also ever changing and agile methodology allows me to be more flexible with the requirements and design. Once the tests are all implemented, the next phase is to create an interface where the user can submit their file for testing and see the results on the screen. The results should also be easy to read and understand since part of my aim for the project is to produce something that's easy to use by end users.

3.4 Analysis

In section I will go over in depth the testing that I will subjugate to statistical testing. As mentioned before the three test batteries I will be looking into are: ENT, Diehard and FIPS STS based on FIPS 140-2 test standard. It is expected that ENT and Diehard tests to pass the testing without much error, the reason why ENT is chosen is because it's comprised of an already fairly solid set of statistical tests and is great tool that give a good initial understanding of the randomness of the data but less reliable on evaluating cryptographic RNGs. Similarly, Diehard is equally reputable battery with 10 more statistical tests than ENT. Ideally, I would analyse all the tests within Diehard, since having a larger sample to test would give me a greater insight into how the different statistical tests perform.

3.4.1 Generating Test Data

Any sequence of randomness must contain two important qualities which are uniformity and independence [14]. Uniformity is when every element is equally probable everywhere. Whereas independence is when each element has no relation to its prior element. Of course, both qualities are equally important to each other when testing for randomness, but most statistical tests fail to accurately measure both at once and favours one quality over the other. My final product is aimed towards users who want an easy-to-use program that can verify the integrity of their RNG, so the data in which I will be using is aimed to examine the test's ability to measure the independence of randomness. For this I will be taking a generator mentioned by Hurley-Smith et al [6] which they have named as the '*keyed ε-hole'*. This generator simulates leaking a key into a PRNG stream output which they have shown to pass FIPS140-2. For the control data I will be using pseudo-random numbers generated by urandom. Urandom is a reputable PRNG in the Unix toolbox that is well known to produce significantly random numbers. Therefore, I've created a small algorithm that will create an *x* number of files that are *x* mb large of urandom data depending on what size and how many I would need.

3.4.2 Method

I will be conducting an experiment on ENT, FIPS STS based and FIPS 140-2 using the same data files under the same conditions and comparing the results to each other and what I expected. There are three types of PRNG data that will be tested: 4kb keyed PRNG data and 50mb urandom data. The first is the is see whether the tests would claim that an infected stream is still considered random and passes it when it should be failing it. The second is the control since urandom is significantly random enough to be considered random and should pass all tests with no issues. Each type of PRNG data is run 100 times and what I will look at is the average of the 100 runs.

3.5 Design

In the design phase, I will be considering the results I found in the analysis stage about how the different statistical tests perform and implement them into a working program that allows users to test their PRNG generators. The basic flow of the program is shown below in Fig 1. After the PRNG has run through the statistical tests, I will implement an additional function that will catch the results and analyse them based on results I have found in the analyse stage. The output will also be in a more qualitative method instead of just releasing the pure statistical results as they can be hard to interpret without statistical knowledge.

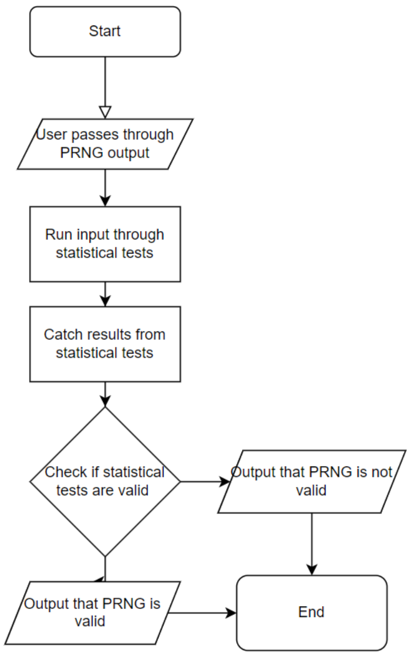

Start
User passes through PRNG output
Run input through statistical tests
Catch results from statistical tests
Check if statistical tests are valid
Output that PRNG is not valid
Output that PRNG is valid
End

Figure 1 Basic flowchart of program

3.6 Project Tools

For the production of this project, there are certain code or programs that I will implement into my own program. These include the use of some existing PRNG generators and the algorithms for the different statistical tests. All these are all open-source and have been credited accordingly within the code or in a text file within the zip file of my project, which ever option is most appropriate for each source used. I will be employing the use of a range of software development tools that are all either free software licence or open-source licence as listed below in Table 2.

Tool Name	*Description*	*License*
Linux/ Ubuntu	Operating System used for the production of this project	Open-source, GNU General Public License
Python	High level, object-orientated scripting language	Python license in a BSD style, compatible with GNU

PyCharm	Dedicated python integrated development environment	Education License, Apache 2 licensed, open-sourced

Table 2: Project Tools

3.7 Limitations

The limitation of this project is:

- It will not have as an extensively researched library as initially intended. I initially intended to look at all the statistical tests included within the Dieharder test suite library and TestU01 and see how they perform against my testing to possibly include in my program. However, I could not find an appropriate python version of either to use and my limited time made it difficult to do as well. Therefore, I believe I could have chosen a range of tests that were much more reliable.

- There is only so much testing of different PRNG outputs I could do. So, I cannot consider every type of problem and backdoor method that could occur and simulate it with the different statistical tests.

- The time and resources for this project is very limited due to unforeseen circumstances therefore UI and in depth analyse of statistical test results may not be achieved as stated in the scope of the project.

3.8 Professional Issue: Licensing

Inventions, creations, and designs all fall under the category of intellectual property since they originate from a person's original idea. Along with this you have the idea of ownership, you would want everyone else to know that this design was conceptualised by you so that other people could not steal and also claim they thought of it first. The way this situation is handled is through a legally binding document of copyright stating the ownership of said product, which gives the owner the sole rights to claim the work as their own. This notion carries through to software development as well, developers are able to copyright their programs, algorithms, and code. Although this is the case, many developers want to be able to make their programs accessible for public use while maintaining the rights to the code, this is where software licensing comes in. A software license agreement is a legal contract between the software developer/vendor and the user of the software which contains a set of regulations and rules on what the user is able to do and not do with the software. The license used is called the End User License Agreement (EULA) [15]. They are only allowed to make use and access the software if they agree and abide by the rules set out by the agreement. This protects the owners from theft and misuse of their software. In the UK there is a legislation named "The Copyright (Computer Programs) Regulations 1992" which was created as an amendment to the current "Copyright, Designs and Patents Act 1988" after cases [16] revealed complications with the CDPA in determining the ownership of emerging technology.

One of the most common violations of the software licensing is the theft and distribution of said software, otherwise known as software piracy. One of the main ways in which criminals commit software piracy is to download and create a copy of the licensed data and then resell them at a fraction of the original price for a profit. Other methods that are also a violation of the licensing agreement is modification to the software in any way in which isn't permitted by the written agreement, this can include: allowing users to access features that they would have needed to pay for first; and adding customisation changes that are specified to be not permitted to be affected in the agreement. This is a growing problem in the world with around $9.1b worth of unlicensed software installations occurring in the US alone in 2015 with other countries like China, India, and the UK with $8.7b,

$2.7b and $1.9b worth respectively being installed [17]. There are several levels of punishment with being found guilty of software piracy depending on what the specific offense is. With the example mentioned prior, the illegal sharing and distributing files you have not paid can run you a maximum fine of £5,000 and imprisonment for a maximum of 5 years [18]. Ancillary orders may be incurred in addition to a fine and imprisonment when being sentenced for digital piracy. Within the realms of possibility, you may be informed to compensate the owner between £20 and £170 as a victim's surcharge as well as: restraint, reparation, financial reporting, and confiscation orders.

Understanding and correctly following the guidelines of software licensing is important to me since it is an issue that directly impacts my project. This is because I will be implementing and incorporating existing statistical RNG tests as mentioned earlier, namely ENT into the code of my program. To use these algorithms, I must understand the concept of open-source licensing. Open-source licensing is a form of software licensing in which it's a legal binding contact between the author of the software and the user that specifies that the components are allowed for individual/commercial use under specific conditions. There are two main categories of open-source licensing, copyleft and permissive. The one that will be applied to my scenario is copyleft. A copyleft license for when the creator copyrights their work but also releases it a with a statement which allows people the right to use, modify, share, and deploy their work either as a whole or substantiation portions of it. In the documentation of the copyright of ENT by R.F. Smith [6], it mentions in the license that in any copies or substantial portions of it must include the copyright notice and all permissions mentioned within the included copyright notice is to be applied. This particular software is licensed under the MIT license which also means that the author cannot be liable for any claim damages or other liabilities arising from out of or in connection with the software. These protections the author being prosecuted for any criminal activities that include the use of his software. All this means that under the licensing agreements, I am fully permitted to implementing parts of R.F. Smith's ENT software into my own project but must bear in mind the license stating all this must be included in documentation of the project. In order to fulfil this, I will be making a folder within the project file that includes the documentation of all the necessary licensing agreements.

3.9 Summary

This chapter identified the significance of the project and the aims I wish to fulfil by the end of the project. I then discussed the analysis process of the project and how I was to go about conducting the experiments and designing the program. Expected limitations of the upcoming development stage is acknowledged. Tools that will be used to aid this project are recognised and a short literature review on the importance of licensing is addressed in the ways it applies to my own project.

Chapter 4: **Theory and Development**

Within this chapter, I analyse the results from the experiments on the statistical tests as mentioned in the analysis section 3.4.2 and determine what the results of this means for how I implement the evaluation of the statistical tests in my own test suite program. Then I will explain how I actually implemented these findings and the process I did in order to create the program. The process I took with building the program is to have the skeleton of how I wanted the layout of the program to look like and then I adapted the functionality to be used to test folders of data files and to write the results to a CSV file in order for me to be able to process the large amount of data into a manageable format to view and process into graphs. Then I am to use the findings from the experiments to determine the pass and fail conditions for the statistical tests so users can qualitatively see the results of their PRNG.

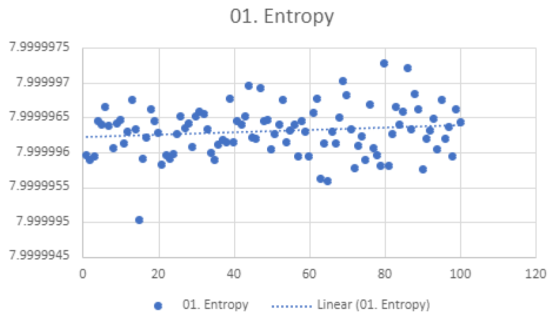
4.1 Statistical Test Experiment

The purpose of running these tests are to determine the pass and fail conditions for the statistical tests that I will implement into my test suite program. Having created files of test data for the two conditions (urandom data and tampered data) through methods mentioned in section 3.4.1, I ran them through a modified version of my test suite program in order to run 100 times per condition and store the results of each condition into two separate CSV files in order to analyse the results and be able to display the results graphically to gain a visual representation of my findings then further. Within this section I will discuss the results for each of the 10 statistical tests I am looking at and use the findings and results to conclude how I will set the passing conditions for my test suite program.

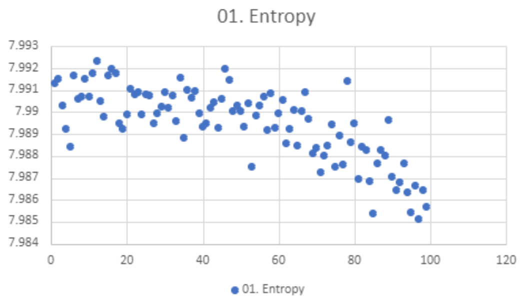
4.1.1 Findings

As mentioned before the tests that I will use are the tests that are used in the popular Ent and FIPS 140 STS. The tests that these batteries include are Chi-square test, arithmetic mean, Monte Carlo value for Pi, serial correlation coefficient, Monobits, poker test, runs test, long run test, continuous run test as well as an entropy test. For each of the two conditions, 100 iterations were run with each file being 20kb for the *e-hole* and 50mb for the urandom data. All results can be found in more detail in the CSV files attached. Observations for each test are as follows.

Entropy: An approximate entropy describes the density of the information given, so the denser the data the more random it is. The determining factor is the closer the value is to 8 the more random it is considered. Fig.2 demonstrates the results of the approximate entropy test for urandom (a), and e-hole(b). Looking at the urandom results, they are all within at least $5 \cdot 10^{-5}$ of 8, whereas with the e-hole it shows none of the data has an entropy of higher than 7.993 which is $6.996 \cdot 10^{-3}$ difference from the average of urandom and has an average of approximately 7.989. So, in order to pass the approximated entropy test, it must have a result of >7.99999.



(a) urandom



(b) e-hole

Figure 2. Entropy results

Chi-squared test: The way the chi-square test is implemented in Ent is by calculating the chi-square distribution for the stream of bytes in the file and then a percentage of how frequently the sequence would exceed the calculated value is determined. The value we look at is called the percent distance of probability of the X^2 from centre. The parameters set in Ent was that the percentage was to be less than 40% to be considered random, *40<p<45* was close to random but not perfect and anything *>45* is not random. So, from looking at the results, 78% of the urandom data would have passed the ent chi-squared test while, 55% of the *e-hole* data passes. By also looking at the *p-values* of the two conditions, the *e-hole* files that failed had a p-value of *p<0.01.*

Arithmetic mean: This is just a normal mean calculation of the sum of all the bytes divided by the file length. Using the mean calculated we can see how close to random it is. If the value is around 127.5 then it is close to random. Rounded to 4 decimals, the average of the arithmetic mean for urandom data is 127.4998 with the range of values being 127.47<m<127.53 giving it a 0.03 error margin either way. To compare this to the range of *e-hole* data being 125.9<m<128.7 with 87% of them falling outside of the range set by urandom. We see that it is able to accurately fail the 87% of the data if I were to go by the range set by the urandom data.

Monte Carlo: Monte Carlo value is used in order to calculate the value of Pi. The closer the calculated value is to Pi (3.143560572) the more random it is to be considered. For urandom the majority passed with a 0.001 error margin and only a 0.003 error margin needs to be considered for a 100% pass rate. If the 0.003 error margin is considered then only 7% of the e-hole data would have been considered random, in comparison to the ent standard of an 0.06 error margin which would have passed 67% of the urandom data.

Serial correlation co-efficient: This test measures the independence of the sequence of bytes by measuring the extent in which each byte depends on the previous byte. This value can be either positive or negative but the closer it is to 0 the less correlation will be detected, hence a more random sequence. Under the ent guidelines, urandom and *e-hole* has 100% pass rate for this test which shows how easy it is for tampered data be tested undetected. However, when looking at the results, urandom shows a range of -0.0005<p<0.0004 with the top 80% deviating 0.00015 up and down from 0. In comparison, the *e-hole* results boasting a much wider range from -0.025<p<0.0153 showing a significant percentage of them could be failed while still keeping a 100% pass rate with urandom data.

Monobits: Counts the number of ones in the stream of bytes. In accordance with FIPS 140-2 requirement for this test, X must satisfy 9275<X<10725 to pass. Under these parameters, *e-hole* files all easily pass this test and showing similar results to the urandom data. This shows how easy it would be to tamper with a stream of data and have it still pass the standardised tests set by FIPS. One thing to note about the average for urandom and *e-hole* are 10011.99 and 9980.394 respectively, so the tampered data will gather a lower X value on average compared to random data.

Poker test: The stream is broken up into 4 chunks and passed through an equation to get the X value. In the FIPS 140-2 guidelines they require that 2.16<X<46.17 in order to pass. Under these guidelines, all data passes in both urandom and *e-hole*. They both show similar results and have an average of 14.884 and 15.277 respectively.

Run test: This counts the length of each uninterrupted sequence of 1's or 0's and there should be a certain amount of each length in order to be considered random. Under the FIPS 140-2 guidelines *e-hole* failed 9% of the time. The two conditions even have very similar averages for each of the different 'runs. By decreasing the extrema of the conditions for each of the types of 'runs', the pass rate for *e-hole* can be decreased making it more slightly more reliable in detecting tampered data.

4.1.2 Conclusion

Through all this analysis, the statistical tests used by ent are very reliable in detecting flaws in the flawed generators in comparison to the tests used by FIPS 140-2 which struggled in finding the flaws in the adversarial generators. The guidelines set out by FIPS 140-2 are still decent at determining the randomness of a generator so for the average consumer who would just want to check whether their PRNG is generating random enough variables then it can be just as helpful. In order to increase the statistical tests integrity more when it comes to detecting adversarial generators, I will be changing the passing requirements to what is specified above for each statistical test.

4.2 Implementing Test Tools

One of the first things I did was to implement the experiments on the statistical tests as discussed in my section 4.1. Within this stage of the project, I wanted to run some urandom and tampered data through the statistical tests to see how well they performed and identify which tests are flawed. Instead of just building a separate program that would run these experiments, I built up the skeleton of my test suite program and modified it from what the end goal functionalities are to better fit the current task. The GUI was the first thing I built as seen in Fig.3 using the python tool tkinter. The aim of this stage was just to gather a basic layout of the program that had all the important elements such as an input bar for the user to select a file, all the necessary labels indicating each of the statistical tests that I have included in the test suite.

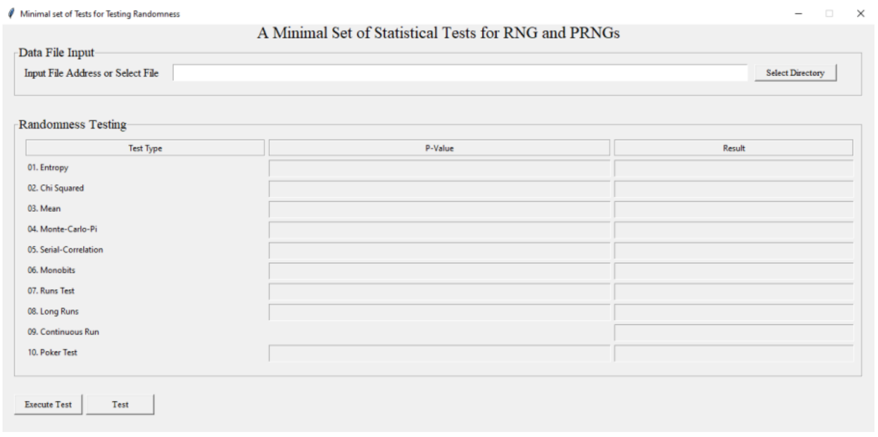


Figure 3. GUI of program

Next thing I did was set up the statistical tests. I am using a pre-existing Python implementation of the ent and FIPS statistical tests. I partial implemented these batteries into my program to use the statistical algorithms since if I were to create each statistical test from scratch, it would have taken too much time and is counterproductive. In order to better organise and manage the different tests since they are located in two separate files, I created the batteries.py file which calls all the relevant functions from each of the ent and fips files to collect the results from them. At this stage, since I needed to run hundreds of files, I had the select file function set to select directory and have it iterate through the generated data files and writing the results of each file to a CSV file which I later used for analysing as seen in section 4.1. Listing 1 shows the python script showing how I handled the hundreds of data files and writing the results to a CSV file.

```
def test(self):
    files = [f for f in listdir(self.__file_name.get())]
    numFiles = len(files)

    header = ['01. Entropy', '02. Chi Squared', '03. Mean', '04. Monte-Carlo-Pi', '05.
    Serial-Correlation', '06. Monobits', '07. Runs Test', '08. Long Runs', '09.
    Continuous Run', '10. Poker Test']
    filename = 'frngResults.csv'     with open(filename,'w', newline="") as file:
        csvwriter = csv.writer(file, delimiter=',', quotechar='"',
                    quoting=csv.QUOTE_MINIMAL)
        csvwriter.writerow(header)
        for i in range(numFiles):
            resultEnt =
            batteries.ent("C:/Users/Verona/PycharmProjects/FYP/data/frng/"+files[i])
            entropy, chi, sc, mean, mc = resultEnt
            r, resultFips =
            batteries.fips("C:/Users/Verona/PycharmProjects/FYP/data/frng/"+files[i])
            mono, poker, run, longrun = resultFips             contrun = r[4]
            csvwriter.writerow([entropy, chi, mean, mc, sc, mono, run, longrun,
                               contrun, poker])
            print("File tested"+str(i))
        print ("CSV file created")
        file.close()
```
Listing 1: Python function of the process of running a directory of files and writing the results to a CSV

The last stage of the development process was to implement the findings from my analysis of the statistical tests and make the program functioning. I added in the pass conditions of the statistical tests into the program and so when the user presses the execute button the program will run and display the p-values of the statistical tests and the result of whether or not the RNG passes the statistical tests and therefore deemed random.

4.3 Summary

This chapter displayed and discussed the findings of my experiments with the different statistical tests in order to find out the best optimum range of success conditions that could distinguish between pure RNGs and adversarial generators. The method of development and the process behind how I created the program is discussed as well as the project plan that was proposed.

Chapter 5: **Conclusion**

This chapter will conclude the report in which I will evaluate the work done this far on the program and the project as a whole. Then a critical analysis is to be conducted on the achievements made within this project as well as how it could have gone better, followed by a short discussion on future work the can be made on this project.

5.1 Evaluation

The main goals in which I initially established for the end result of this project were (1) find a minimal set of tests that can be used to reliably identify adversarial generators (2) create a user-friendly GUI in which users who don't know anything about statistical analysis are able to use the program and (3) through my own research determine how the statistical tests are to be evaluated in terms of what is random or not random. For the first aim, I was not able to fulfil this goal as much as I strived to. I only chose a select few statistical tests I believed to be a good amount without much testing in order to find the optimum amount and tests to assemble. The second goal I accomplished to a substantial extent with a working GUI that is simple and easy to navigate as well as functional. The third aim I also have good evidence of having accomplished this aim. I compared how the statistical tests performed with tampered PRNG data and urandom data to establish the ideal parameters for the pass conditions.

5.2 Critical Analysis and Discussion

This section is a discussion of what I have achieved by the end of this project and how certain aspects could have gone differently or how I would have gone through with certain actions differently.

5.2.1 Achievements

Having completed the goal of having a user-friendly functioning application that users are able to use to test their RNGs for any tampering that may have happened during the production line with the ease of a click of a button is one of the main goals that I have achieved in this project. The biggest personal achievement was the understanding and analysing the statistical tests. Since all this content was new to me, being able to understand how to read and interpret the statistical data in order to implement it into my own work is the biggest achievement I've accomplished during the project process.

5.2.2 Reflections

The main weakness of the program is that it only runs either a file at a time and displays those results or runs through a folder of files and stores the results in CSV file which is not very useful to the average user who may not know how to interpret the pure results. This is one feature I wish I could have implemented, as a combination of the two functions mentioned above. This would increase the usefulness of the program a lot more since running just one file is not enough to justify the RNG is not broken or tampered with. Having many tests and then finding the average of how many of those tests failed or succeed and then outputting the results would have been a much better functionality to add to the program.

Also, one aspect that the project could have benefitted from is having fulfilled the first aim from the list of goals I established at the beginning of the project. The group of tests that I have incorporated into my program are not the optimal tests in which I could have chosen to implement. If there was more testing on other statistical tests that could have been done, the group of tests that end up being implemented could have been more reliable and created a more rounded test suite.

5.3 Future Work

From where the current status of the project stands, it would be worth changing the input method from taking in a singular data file to processing the random number generator instead and directly taking the data used from a user selecting the path of their random number generator. This could then prompt research into looking at what the optimum percentage of successes after an *x* number of iterations would be enough to determine how random the RNG is. This remove the need of having to produce hundreds of files beforehand to test and having a larger sample size to evaluate from, increases the validity of the results given back to the users on whether or not their random number generators are indeed random and haven't been tampered with down the production line.

Bibliography

[1] Brown, R., n.d. *Robert G. Brown's General Tools Page*. [online] Webhome.phy.duke.edu. Available at: <https://webhome.phy.duke.edu/~rgb/General/dieharder.php> [Accessed 29 May 2022].

[2] Gasior, G., Tingle, S. and Tingle, S., 2013. *Researchers demonstrate 'hardware trojan' attack on Ivy Bridge*. [online] The Tech Report. Available at: <https://techreport.com/news/25386/researchers-demonstrate-hardware-trojan-attack-on-ivy-bridge/> [Accessed 29 May 2022].

[3] "The Marsaglia Random Number CDROM including the Diehard Battery of Tests," Jan. 25, 2016. https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/ (accessed Dec. 10, 2021).

[4] "TestU01: A C library for empirical testing of random number generators: ACM Transactions on Mathematical Software: Vol 33, No 4." https://dl.acm.org/doi/10.1145/1268776.1268777 (accessed Dec. 10, 2021).

[5] J. Walker, "Pseudorandom Number Sequence Test Program", *Fourmilab.ch*, 2008. [Online]. Available: https://www.fourmilab.ch/random/. [Accessed: 11- Dec- 2021].

[6] D. Hurley-Smith, C. Patsakis, and J. Hernandez-Castro, "On the unbearable lightness of FIPS 140-2 randomness tests," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2020, doi: 10.1109/TIFS.2020.2988505.

[7] Ryabko, B., 2020. Time-Adaptive Statistical Test for Random Number Generators. *Entropy*, [online] 22(6), p.630. Available at: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7517164/>.

[8] CSRC, 2001. *Security Requirement for Cryptographic Modules*. USA: NIST.

[9] Kelsey, B. Schneier, D. Wagner and C. Hall, "Cryptanalytic Attacks on Pseudorandom Number Generators", *Academia.edu*. [Online]. Available: https://www.academia.edu/9890532/Cryptanalytic_Attacks_on_Pseudorandom_Number_Generators. [Accessed: 11- Dec- 2021].

[10] Marsaglia, G. and Tsang, W., n.d. *Some Difficult-to-Pass Tests of Randomness*. [online] Sematics scholar. Available at: <https://pdfs.semanticscholar.org/95f0/143b4ba87a144977bacc19a4eb2950ff9199.pdf> [Accessed 29 May 2022].

[11] Hasegawa, A., Kim, S. and Umeno, K., n.d. *IP Core of Statistical Test Suite of FIPS 140-2*. [online] Design And Reuse. Available at: <https://www.design-reuse.com/articles/7946/ip-core-of-statistical-test-suite-of-fips-140-2.html> [Accessed 29 May 2022].

[12] M. L. McHugh, "The chi-square test of independence," *Biochem Med (Zagreb)*, vol. 23, no. 2, pp. 143–149, 2013, doi: 10.11613/bm.2013.018.

[13] Ibm.com. 2022. *random and urandom Devices*. [online] Available at: <https://www.ibm.com/docs/en/aix/7.2?topic=files-random-urandom-devices> [Accessed 29 May 2022].

[14] Haahr, M., 2022. *RANDOM.ORG - Introduction to Randomness and Random Numbers*. [online] Random.org. Available at: <https://www.random.org/randomness/> [Accessed 29 May 2022].

[15] En.wikipedia.org. 2022. *End-user license agreement - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/End-user_license_agreement> [Accessed 29 May 2022].

[16] GOV.UK. 2011. *Copyright Act*. [online] Available at: <https://www.gov.uk/government/publications/copyright-acts-and-related-laws> [Accessed 29 May 2022].

[17] Richter, F., 2016. *Infographic: The Cost of Software Piracy*. [online] Statista Infographics. Available at: <https://www.statista.com/chart/5164/use-of-unlicensed-software/> [Accessed 29 May 2022].

[18] Stuart Miller Solicitors. n.d. *Want to know the sentence for a digital piracy copyright offence in 2022.* [online] Available at: <https://www.stuartmillersolicitors.co.uk/sentences/sentence-for-digital-piracy-copyright-offence/> [Accessed 29 May 2022].