True Randomness using the Randomness in Natural Phenomenon

A dissertation submitted for the Degree of Master of Information Security

H.K.R.K. Senadheera
University of Colombo School of Computing

Abstract

Patterns and ability to discover them, makes a system predictable. All of the modern day computers are finite state machines, which make them quite predictable in their nature. Yet, there are certain applications and cases which require a high level of unpredictability. This unpredictability, or sometimes referred to as non determinism, is characterised by *Randomness* of a system. Randomness attributes to the lack of knowledge on the causality behind a specific output, even if the system is known to fine details. Randomness is a key requirement in certain critical applications such as cryptography, simulations and so forth.

Randomness at the most abstract level is bi-fold. They are namely true randomness and pseudo randomness. True randomness is the ideal form, which is existing in the surrounding as various phenomenon, such as lightning, thermal noise, Brownian motion of particles and so forth. Even though these are available and truly random, most of the times they are far from practicality within the environment of a computing device, due to various reasons such as difficulty to measure and feed to the device, very low rates of change, generated bit strings being inadequate in size and so on. Often, capturing true randomness that is existing in the surroundings requires expensive hardware devices which are not feasible in the context of personal computing. Therefore, true random sources are mostly used to provide the initial seeds to a pseudo random generator.

Pseudo randomness on the other hand is the method of generating randomness by deterministically transforming an initial state called a seed). Often these systems generate randomness which has equal statistical qualities as true randomness, at much faster rates. Yet, output of the pseudo random generators almost all the times, repeat after a certain number of iterations. This is known as the period of a pseudo random generator and considered a weakness that is inherent in pseudo random generators.

This research study focuses on using the random variables within a typical system environment to generate randomness which is void of the inherent weaknesses of pseudo randomness and close to true randomness in terms of statistical quality. In order to achieve the said, feasibility of using Floating point, an existing number representation scheme was evaluated. Initially, a conceptual model was composed to address the different issues in generating randomness that is close to true randomness. Then each of the stage in the conceptual model was addressed with different possible strategies. For the generation of bits, a new generator is proposed which uses the concepts of floating point representation at its core. Then, the performance of the proposed model was tested using the Statistical Test Suite for Randomness provided by National Institute for Standards and Technology (NIST) by bench marking the results against some commonly used and recent pseudo random generators.

Acknowledgements

First and foremost, I would like to convey my sincere gratitude to Dr. Kasun de Zoysa, the coordinator of the Master of Information Security degree programme and Dr. Chamath Keppitiyagama, my supervisor assigned from the UCSC. Dr. Kasun deserves all the credit for allowing me to undertake this research project, among different regulations, and the guidelines given to us throughout out academic tenure of the masters. I fondly remind the kind guidance given by Dr. Chamath regarding the undertaking of this research project and how he put me on the right track where and whenever I have deviated from the right course. Most of all, Dr. Chamath was a great influence to awaken my eyes of wisdom towards having an own philosophy. Without the immense support of these two gentlemen, this research study would have been a far distant reality.

At the same time, I must remind and convey my sincere gratitude to my parents, who have been there for me every time no matter what. Their support and them tolerating my absence in important matters was immense. Who I am today, is primarily because of them. Also I would like to remind the moral support given by my brother.

Kasun Vimukthi Dissanayake, who happened to be my supervisor at my office front, has also been a great mentor to me. I fondly remind his support in various ways such as allowing me to take day offs where necessary, giving guidelines on how to complete this tedious task, putting me on the right track in certain cases and by encouraging me to complete the task. His immense support was a great influence for me to successfully complete the task.

Last, but not the least, each and every single person including but not limited to the Director and all the staff members of the UCSC, my batchmates of the MIS degree programme, my colleagues at hSenid Software International, my friends and my relatives who has been so much generous to offer me their helping hand in various aspects of successfully concluding this research project, are hereby reminded with my sincere gratitude. Thank you very much for being there for me. Every bit of their help was essential for this research study to be a fruitful one.

Kanchana Senadheera

Contents

List of Tables

List of Figures

Abbreviations

AES	Advanced Encryption Standard.
API	Application Programming Interface.
CBC	Cipher Block Chaining.
CBF	Cipher Feedback.
CPU	Central Processing Unit.
DRBG	Deterministic Random Bit Generator.
ECB	Electronic Code Book.
HTTP	Hypertext Transer Protocol.
IV	Initialisation Vector.
JSON	JavaScript Object Notation.
LCG	Linear Congruential Generator.
LFSR	Linear Feedback Shift Register.
MSE	Mean Squared Error.
MT	Mersenne Twister.
NIST	National Institute for Standards and Technology.
OBF	Output Feedback.
PC	Personal Computer/Computing.
PRNG	Pseudorandom Number Generator.
RAM	Random Access Memory.
SFMT	SIMD-oriented Fast Mersenne Twister.
SIMD	Single-Instruction-Multiple-Data.

TRNG	True Random Number Generator.
WELL	Well Equidistributed Long-period Linear.
WLAN	Wireless Local Area Network.

Chapter 1

Introduction

In this introductory chapter, it is focused on elaborating the overview and developing the rationale behind this study. This chapter is intended to introduce the reader to the concept of randomness, its need and the absolute purpose of this research study. Further, structure of this thesis is elaborated toward the latter part of this chapter.

1.1 Prologue

Human brain is known and identified to be an extraordinary organ among all of the known living beings to date. It has been identified to be capable of doing many different things and among them *pattern recognition ability* is known to be one of the most prominent, and also one of the prominent obsession. This ability and the obsession has led the human being to further expand the attempts to discover knowledge with the use of patterns and correlations.

Patterns and ability to discover them, makes a system predictable. This was absolutely necessary for most cases and applications. The availability and discover-ability of patterns in different systems, has paved the way to discover vast amounts of knowledge, causing many different advancements in each discipline. However on the contrary, this has led to some other systems to be less reliable. Certain areas in computing especially cryptography, relies on the attribute of the system being unpredictable. For the case of cryptographic systems, more the system is predictable, more vulnerable it would be. This, and some other applications requirements such as in simulations and so forth have craved for systems to be random in their behaviour.

1.2 Background

Throughout the early parts of the known history of mankind, chance and randomness were knitted together with fate. There are number of historic evidences that suggests that people threw dice to decide the fate which later has evolved into games of chance. Some of these games are spanned even to date. Also, evidences also suggest that various methods existed in most cultures which are attempts to circumvent randomness and fate.

Perhaps, the earliest people to formalise odds and chance are believed to be Chinese people, 3000 years ago. Historic evidences from Greece suggests that the Greek philosophers had dived deep in discussions on randomness, nonetheless only in subjective forms. It took time until the 16th century to initiate formalisation of the odds associated with various games of chance, by the Italian mathematicians. In the 19th century the concept of *Entropy* was introduced. With this and the invention of calculus was highly positive impact on the study of randomness. Then during the early part of the 20th century, formal analysis on the randomness has started to grow rapidly and steadily with proper mathematical foundations based on probability. In parallel, quantum mechanics changed the scientific perspective on determinacy and ideas of algorithmic randomness started to surface.

Due to the lack of scientific knowledge it has been long believed that randomness is comprehended to be an obstacle. Since deliberate introduction of randomness into computations was understood as an effective way for designing better algorithms, this has started to change. Ironically in certain cases, the best deterministic methods were outperformed by the randomisation algorithms.

There is a large number of applications such as cryptography, gaming, sampling, simulations and so forth, mainly in the domains of computing and statistics. Specially in cryptography, security of certain algorithms are entirely dependent on the seeds which are used as inputs and these seeds are generated using some form of randomness. At the same time, certain statistical applications which include but not limited to sampling and Monte-Carlo simulations needs some random data which are of heavy volumes. An parallel to these, it is also quite important that any form or generation be as fast as possible. However, it is generally accepted that generating complex and purely random numbers in large volumes at high speeds it the ideal combination which is far less feasible because these are conflicting requirements.

1.2.1 Randomness

Randomness in general sense could be comprehended as *absence of pattern or predictability in events*. In fact, a sequence of symbols or events which is deemed random does not stick to any tangible patterns. Popular examples of random events include flipping a fair coin and rolling a fair dice. The output of such events are believed to be truly random, hence provides the base for many cases and experiments. Apart from that, there are certain other phenomena which could be observed in our surroundings are also widely accepted to be random. Brownian motion, thermal noise and so forth could be considered as examples for such. Most other microscopic phenomena also are widely being used in different levels of abstractions, to generate certain forms of randomness.

Lack of patterns is believed to be true for most of the cases which are taken individually yet, in most of the cases as the process that generates the aforesaid randomness is repeated over a large number of iterations, the sequence starts to appear to be more and more predictable. When rolling two dice for an instance, the occurrence of any particular event is unpredictable, yet a sum of 7 will occur twice as often as 4. This is a prime example that highlights the fact that predictability of a system is dependent on the amount of information that is available regarding the system. With this perspective it could be concluded that randomness not mere *Haphazardness*, but a measure of outcomes. In fact, randomness is closely tied with concepts of *Probability* and *Information Entropy*.

The fields of mathematics, probability, and statistics use formal definitions of randomness. In statistics, a random variable is an assignment of a numerical value to each possible outcome of an event space. This association facilitates the identification and the calculation of probabilities of the events. Random variables can appear in random sequences. A random process is a sequence of random variables whose outcomes do not follow a deterministic pattern, but follow an evolution described by probability distributions. These and other constructs are extremely useful in probability theory and the various applications of randomness.

1.2.2 Existence of Randomness

The question *Does randomness truly exist?* is arguably one of the most popular question among scholars from many different disciplines. There seem to have different distinct philosophical strongholds which are considering different perspectives on randomness. However it is widely accepted that randomness is *an attribution* to the property of *lack of knowledge on the causality of a phenomenon*. Randomness is not a cause that results in different effects. Yet it is erroneously stated more often. In that sense, it could be safely concluded that as long as there exists some phenomena which the human being possess no knowledge of, there exists randomness. This could also be defined and portrayed using the relativity.

1.2.3 Forms of Randomness

Modern day, there are mainly two different forms of randomness, which are based on different sources. Those are namely True Random Number Generator (TRNG)s and Pseudorandom Number Generator (PRNG)s. Comparison between these two could be easily done by considering a simple flipping of a fair coin. It is widely accepted that flipping a fair coin demonstrates random behaviour. There are two approaches to use these flips as random. First, the coin could be flipped whenever there is a need of random binary result. Here, the source of randomness is provoked on-demand and the result is used. This is how the TRNGs theoretically function. That mimics getting the generator to flip the coin on-demand. However, it is physically infeasible to bring such macroscopic phenomenon to computers. Hence, additional hardware devices and sophisticated computer programs are utilised in capturing certain microscopic phenomenon and using those metrics to generate random numbers.

The other approach is to have the coin flipped a large number of times previously and their records are collected into a collection. Now, whenever there is a requirement of a random binary result, one could query the data set for the next random bit. This approach very closely mimic the behaviour of a PRNG. Instead of a large finite data set, almost all modern PRNGs have an algorithm which is switching between a very large number of states, which are determined by and highly sensitive to the initial condition/input (also known as *seed*).

1.3 Motivation

It is experimentally proved that PRNGs are performing quite well. In fact in certain aspects, PRNGs appear to have outperformed most of the TRNGs. Yet there are certain inherent weaknesses are known to be associated with them. One such is that they are periodic i.e. after a certain number of iterations, the sequence that is generated is repeated. Even though these numbers are ranging from $2^{127}-1$ to $2^{19937}-1$ and even beyond that. Still, it is important to take into account the fact that computational power is also advancing at exponential rates. There is a growing argument that binary computers based on finite state machines are reaching their limits of performance envelopes. Still it is uncertain that it would occur in the near future. At the same time, there are new advancements in computing such as *Quantum Computing*. These at the moment are theoretically promising a new, powerful breed of computational devices. There is a chance that, PRNGs might fall short of the revised statistical requirements and testing criteria, based on the advancements of computing.

Being *Cryptographically Secure* is another important attribute of a better RNG. Even though there is a number of PRNGs available, most of them fall short in testing for cryptographic security. The reason why this is quite important is that random numbers are used in many cryptographic applications quite often. These include key generation, nonce values, salts in certain signature schemes and so forth. Also, the quality requirements that each application demands would be different from one application to another. Only a handful is being used and available in cryptographic applications. This is another major problem that PRNGs suffer from.

On the contrary, the counterpart TRNG is also suffering from known lapses and weaknesses. Even though the TRNGs provide statistically high-quality random bits, performance have been a major issue. Due to the high-end technological requirements and the portability issues introduced by such hardware devices, it is quite difficult to effectively employ the TRNGs specially in the context of personal computing, where *compactness* is deemed to be quite important. Apart from that, since TRNGs are based on physical phenomena, TRNGs in most of the cases are suffering from falling short of the volume requirements. It is quite time consuming to generate large volumes of random data, due to the fact that effective generation rate of the TRNGs are actually the rate of change in the physical phenomenon which is being monitored by the particular TRNG. Since these are mostly microscopic phenomena, the rates that which they are changing are quite low, giving a low frequency of change.

Therefore it is essential to look for sustainable options which offer better flexibility to suite the environment and the task being performed, while preserving the prime attributes and requirements of randomness. Also it is quite important to take the property of being cryptographically secure, into account during the discovery of alternatives. This research is intended to attempt to discover opportunities to fill that gap, by evaluating the system environments for possible alternatives, under the scope mentioned below.

1.4 Objectives and Scope

The primary objectives of this research study are

1. to explore and identify the possible sources of randomness within the system and its surroundings.
2. to identify and establish transformation strategies (hereafter referred to as *Distillation*) which are required to improve the complexity and volume.
3. to identify and establish the encryption strategies (hereafter referred to as *Hardening*) which are required to meet the cryptographic security requirements.
4. to identify evaluation strategies to assess the quality of the outputs.
5. to assess the quality of the outputs based on the complexity and volume requirements in order to determine the performance of the system.

It is important to take note on the following points regarding the scope of the study, as there are heaps of studies which are based on different perspectives, which have opened up a number of different paths to proceed.

- The domain of the study is restricted to *Computing*. There also, the primary focus is on the context of *Personal Computing*.

- The system being evaluated is as established in the latter part of the study. To briefly elaborate, the term *System* refers to the computational device being taken into consideration.

- The behaviour of the system is considered and activities such as network data flow, Random Access Memory (RAM) usage and so forth are monitored to observer their behaviour in terms of randomness. Along with that, natural effects of such behaviour is also monitored for the same.

- It is focused on possible sources of randomness that could be found in the near vicinity of a majority of computational devices. Also, a selected set of device related metrics also taken into consideration, as sources of randomness. Their suitability is also evaluated here

- Evaluation criteria for the randomness is fixed at the "***Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications***" published by National Institute for Standards and Technology (NIST) under SP 800-22 Rev. 1a. This was chosen due to its wider acceptance among the community.
- Implementations of any sort are done using the language *Python*. Python was chosen as the implementation language due to its expressive syntax and native support for certain large-date operations.
- Possible options to make the output cryptographically secure by hardening, are taken into account and evaluated. There the main focus will be on stream ciphers and *asymmetric encryption schemes*.

1.5 Structure of the Thesis

This thesis is comprised of the following chapters from this point forth. The content of each chapter is briefly summarised against each chapter name.

Chapter 2 - Literature Review

This chapter is dedicated to review the current literature and related work, in order to establish the foundation of the research. Related aspects such as formal definitions, terminology, existence of randomness, forms and sources of randomness, testing for randomness and so forth are addressed in fine detail in this chapter.

Chapter 3 - Methodology

Here it is focused on the methodology that the research is undertaken. Formal establishment of the scope, flow of work in the research, choices of different strategies for distillation and hardening and the justifications of those choices, choice of the testing strategy along with the justifications are discussed in this chapter.

Chapter 4 - Implementation

This chapter is including the details of the implementation of the above methodology. Different code snippets used along with their flow of activity and performance, other aspects related to implementation are addressed in this chapter.

Chapter 5 - Evaluation

All the details of the test execution results and the interpretation of those results are summarised in this chapter. It also includes a weighing of merits and demerits of the implementations which would be supportive to the conclusions that could be arrived at.

Chapter 6 - Conclusion

This chapter is a complete retrospect of the research which has taken three main perspectives into consideration. Initially, it is focused on the lessons learnt by the author during the research project. Then, the initial plan and the output artefacts are compared to determine and scale the success of the research project. Finally, it has enumerated various possible routes to proceed for future work related to this research.

Apart from the main body enumerated and elaborated above, the thesis also has the supplementary documents and written artefacts attached as appendices.

Chapter 2

Literature Review

This chapter is intended to explore the knowledge that is available in the related literature. The chapter begins with defining the concepts and the terminology. Then, the existing knowledge is taken into consideration, to explored the related facts and to develop the background of the study. Based on the knowledge gathered here, the conceptual framework for the research study is formulated and the methodology and the structure of the study is elaborated, towards the latter part of the chapter.

2.1 Terminology

Here, it is attempted to define and clarify the related terminology which there are a few yet, they are important. There are many different terms which are alternatively in use out there hence, it is crucial to clarify each terms, if they are synonymous, related or not.

2.1.1 Randomness

Randomness is defined in many different ways, taking many different aspects into consideration.

1. Cambridge English Dictionary defines randomness as
   (a) "*happening, done, or chosen by chance rather than according to a plan*"[15].
   (b) "*by chance, or without being chosen intentionally*"[15].
2. According to the Oxford English Dictionary, randomness is "*made, done, or happening without method or conscious decision*"[28].

As per the second definition provided by the Cambridge dictionary, the phrase *without being chosen intentionally* emphasises the fact that, there should not be some entity, that influences the choice made by the system. Further, the definition given by the Oxford dictionary also emphasises on the fact that *absolute lack of bias* should be a definite characteristic of randomness.

It is also worth noting that randomness is *not a cause, but an attribute*. There are common misconceptions such as "this was caused by random events" or "this is attributable to random variation" and so forth, which are ideally false. *Randomness* is a term which is coined to attribute when the cause(s) of a particular phenomenon is not known, at least yet.

2.1.2 Non-deterministic Behaviour

Behaviour of a system are said to be non-deterministic, even if everything that can be known about a system at a given time is known with all available details about the system, it is still not possible to predict the state at a future time. As per the Cambridge dictionary, "*deterministic*" is an adjective, which means "*Relating to the philosophical doctrine that all events, including human action, are ultimately determined by causes regarded as external to the will*"[27].

Some of the problems which demonstrate non-deterministic behaviours are modelled and examined in mathematics and computing related applications. According to Robert W. Floyd, a non-deterministic algorithm is "*a conceptual device to simplify the design of backtracking algorithms*"[18]. An arbitrary non deterministic algorithm having $f(n)$ levels may not be returning same outcome in different runs. Further, such algorithm might not complete its execution due to the size of the fixed height tree is potentially infinite. [18][5]. A prime example of a non-deterministic problem is *Prime Factorisation or Integers* [21] i.e. there is no algorithm that demonstrates deterministic behaviour, to derive the prime factors of a given integer [11][20]. Primality test is an extension of this problem. When both these problems are taken into consideration, even though we can easily predict the behaviour for small inputs, the system becomes unpredictable, as the inputs grow in their magnitude. There is no known deterministic algorithm that would determine if a given number is a prime or not, other than some repetitive and exhaustive methods.

2.1.3 Non-deterministic nature of Randomness

It is quite evident that the above two subsections 2.1.1 and 2.1.2 describes concepts which are going hand-in-hand. In fact, for a system to be random, one should not be able to precisely predict a future state of the system, even if everything is known about the system to the perfect detail. This leads to the fact that *randomness is non-deterministic* i.e. being non-deterministic is an attribute of randomness. More precisely, we could conclude that *randomness implies non determinism*[32]. However, the vice versa is not always true.

2.1.4 Existence of Randomness

"*Is randomness really there?*" has been a cause of fascination among scientists, philosophers and so forth. Previously we have established the fact that randomness should essentially have absolute lack of bias, which in other words means that there cannot exists variables that one could manipulate or influence, so that the output becomes predictable. This could also be interpreted as randomness is not something that would cause another.

Instead, randomness denotes that we are unaware of what causes the particular effect that we are taking into consideration. In other terms, either we are unaware of, or there is no access to variables that determines the effect of the cause in consideration, making those variables *hidden*. If the previous examples such as Brownian motion, chaotic behaviour of double rod pendulum and so forth, this fact applies on those cases as well. In that sense, we could conclude that randomness exists[32].

Another perspective one could look at this is that as per Feynman suggests, all interactions are actually taking place across all possible paths, without being ruled out by any other criteria. So there is no randomness involved, nevertheless the experimenter does not have access to absolutely all the information that might affect the set of possible routes to an outcome[10].

2.1.5 Attributes of Randomness - Summary

Based on the facts which have been discussed up to now regarding defining randomness, the attributes that true randomness should possess could be itemised as below.

- **Unpredictable** [15]
- **Absolute lack of bias** - This is bi-fold
  - System should not demonstrate any bias towards a particular state [15]
  - It should be infeasible for an outside observer to influence and manipulate the system to be bias to a particular state. This is feasible as long as an outside observer lacks in the knowledge of the system. [9]
- **Absence of patterns** [28]
- **Non deterministic**[32]

2.1.6 Forms of Randomness

Randomness exists in a variety of forms. It is generally accepted that, randomness could primarily be sourced by the origins described below.

Environmental Randomness

This form of randomness exists in the surroundings of a system. Certain phenomenon such as *Brownian Motion* and *Noise* in signal processing are prime examples for this form of randomness. Since these systems are either completely immune to external influences, or even under external influences the system is not considerably deviated from its non-deterministic state of behaviour, these systems are ideal in terms of the quality of the randomness. [8]

Randomness based on Initial Conditions

The behaviour of certain systems tend to demonstrate an extreme sensitivity to the initial conditions. For an instance we could consider *Double-rod Pendulum*. This is a rod which is rigidly hinged from one end, and has another rod hinged to the other end of the first rod. The farthest edge of the system which is free to move, will form a trajectory when the system is given with an external force. [8]

What is so fascinating about this system is that, a subtle variation of the initial conditions would cause the final trajectory to be drastically different from the previous. This behaviour is known as *Chaotic behaviour* in *Chaos Theory*. This system could be used as a source of randomness, as it demonstrates non-deterministic behaviour. Different initial conditions combined with temporal separation, will yield metrics which are non-deterministic.

Randomness generated Intrinsically

Randomness could also be generated intrinsically. This refers to the behaviour of certain algorithms, routines and so forth, which appears to demonstrate non-deterministic behaviour, given that the initial state of the system is concealed. *Pseudorandom Number Generator (PRNG)s* which are in use in most of the modern day applications, are falling under this classification. This form of randomness is much closely reviewed in later chapters, to determine their performance in terms of complexity, volume and quality. [8]

2.1.7 Random Generators

Random generators are machines or systems that would generate a specific type of an output that which it is random. A prominent example of a random generator is a *Lottery Ball Selector*[4] which shuffles the fair light weight balls using a high-speed air flow streamed into an enclosed chamber. It will output a type **Ball** which is said to be random. In most of the cases, the ball being taken out at random is replaced and the process is repeated. This is considered to have the absolute lack of bias, given that balls are fair.

In applications related to computing, this is slightly more complex. According to National Institute for Standards and Technology (NIST) USA, one could comprehend generating a random bit sequence, as flipping an unbiased coin successively with each side labelled as 0 and 1 respectively. Due to the coin is unbiased, every flip has a probability of 0.5 of producing either 0 or 1[34]. Further it is worth noting that these flips are independent of each other, i.e. previous flip has no influence whatsoever on the next flip. This is outlined in NIST-SP-800-22 as a perfect random bit generator due to the facts that,

1. Values are selected independently at each flip. [34]

2. Probability on each outcome of the sample space is uniformly distributed, i.e. there is **absolute lack of bias**. [34]

It is obvious that this model would not be practical in a cryptographic application. Yet, the hypothetical output of this idealised generator serves the purpose of a benchmark to evaluate the quality of Random Number Generators in general.

2.1.8 Quality Attributes of Randomness

Previously we have arrived at a conclusion that randomness is an attribute of a system. We are leveraging that attribute to be used in certain other systems. For this goal to be attained, we mush essentially look for the following quality attributes of randomness.

1. ***Complexity***: Complexity of a RNG is a bi folded attribute. On an attacker's perspective, it should be computationally infeasible to determine the next bit. In that sense, it should be above a predetermined threshold, and higher the complexity, better the system would be. On an owner's perspective, the system should be less complex to implement and maintain. With the least possible effort, the owners should be able to implement and utilise the system.
2. ***Volume***: Here it focuses on the amount of random data that could be generated within a given unit time duration. Higher the volume, better the generator would be.
3. ***Performance***: Under this attribute, there is a multitude of aspects that need to be taken into consideration. These aspects would include computational complexity in terms of time and space, resource usage, security and so forth. This is also closely related to the volume requirements.

2.2 True-Random Generators

One type of sequence generator is a True Random Number Generator (TRNG). A TRNG is primarily composed of two main modules. Primarily there is an entropy source at its core. This source is usually some sort of a microscopic phenomenon that generate "noise" signals which are of low-level and having statistical random properties. Examples of such are thermal noise of an electric circuit, the photoelectric effect, involving a beam splitter, timing of user activities or interactions with input devices (e.g. key strokes or mouse movements) of a computer, quantum effects in a semiconductor and so forth. Even combination of such could be effectively used. Other component is there to eliminate or mitigate the weaknesses and to improve the quality of the random bits. This process is called "*distillation process*". This distillation process is essential to eliminate possible flaws of the entropy source which could result in producing non-random numbers (e.g. very long strings of zeros or ones). [8]

The outputs of a TRNG could be used out of the box as a random value. Also, could be fed into a PRNG as a seed. To be directly usable without any further processing or transformations, the output of any TRNG should comply with *Strict Randomness Criteria* as evaluated by statistical assessments to determine if the actual sources of the TRNG inputs demonstrate adequate randomness. For an instance, an entropy source (e.g. electronic noise) might sometimes contain certain regular and repetitive structures, (e.g. waves, other periodic phenomena) which might have the appearance of randomness, yet would fall short during statistical tests. [8]

2.2.1 Limitations of TRNGs

Even though TRGNs are promising, in terms of quality and randomness, there are several different limitations that would yield them far from usability in applied scenarios. Some of these limitations could be identified as follows.

To be used in cryptographic applications, unpredictability of the outputs is a must. However, certain commonly available sources (e.g. date time vectors) are predictable. Problems as such could be avoided by combining a variety of outputs of different forms of random sources to be used as the source of a TRNG. However, the resultant might still be low quality when assessed by statistical tests. Apart from that, generation of random numbers which are of high-quality might be excessively laborious, leaving such production unsuitable in cases where a the quantity of required random numbers is quite large. Hence, PRNGs may be preferred over TRNGs for generation of large quantities of randomness. [8][7]

On the other hand, most of the phenomenon that has been taken into consideration above, are microscopic. Hence, it is computationally expensive in most of the cases to achieve the performance requirements and most of the times fall short in front of volume requirements. Therefore the required quality attributes are not achieved and they would not be suitable for certain applications such as *Monte-Carlo Simulations* and so forth, which would require large volumes of data. [8]

Another major drawback in TRNG is the requirement of additional hardware which might introduce new dimensions of problems including costs, compatibility and compactness. Almost all of the personal computing is rapidly floating towards leaner and more compact form factors. For an instance we could consider a mobile phone. Plugging in a Hardware RNG to such a device could be far from practicality due to various reasons including size, power supply, processing power required, portability and so forth. So, if there is an RNG which targets at Personal Computer/Computing (PC) devices, these factors should essentially be taken into consideration.

Following is an itemisation of the companies that manufacture the commonly found TRNGs in modern applications. [3]

- Araneus Alea
- ComScire

- Entropy Key
- Fox-IT Fox RandomCard
- ID Quantique
- OneRNG
- BitBabbler
- ProtegoST
- ubld.it TrueRN
- Real Random EaaS

Detailed and in-depth examination on these generators are left out from this point forth so as to confine the scope of this study.

2.3 Pseudo-Random Generators

The other type of generators is Pseudorandom Number Generator (PRNG)(also referred to as Deterministic Random Bit Generator (DRBG))[12]. This at the core is an algorithm that outputs sequences of numbers whose properties approximate the attributes of random sequences. A PRNG generates multiple "pseudorandom" numbers with the use of one or more inputs which are also referred to as seeds. The seed should essentially be unpredictable for this to be utilised in contexts that demands unpredictability. Hence, by default, a TRNG should be coupled with PRNGs. [8].

The outputs of a PRNG are a deterministic function of the seed being used. This causes the ideal randomness of the generator to be confined to the seed generator being used. The term *pseudorandom* is derived from the deterministic behaviour of the process. Since each element of a pseudorandom sequence could be regenerated using the initial seed, saving the seed is adequate and necessary if it is required to validate or regenerate a previous random sequence [8].

A key weakness of PRNGs is repetition and patterns. Almost all the PRNGs would repeat their sequence after some n number of outputs. Lower the value for n, more the system is vulnerable and predictable. For the PRNG to be considered *Cryptographically Secure*, the value of n should be higher. This property is inherent to all the PRNGs. This is why a common form of attack of keeping track of the numbers of the PRNG is possible. However, when the value of n becomes extremely large (e.g. a power of two), it would become temporally infeasible to keep track of each number. Hence it would generate outputs with the required robustness.

In practice, the output from most of the commonly used PRNGs demonstrate certain errors causing them to fail *statistical pattern-detection tests*. These include [8]:

- Shorter periods than expected for certain initial states (also known as *weak* seeds)
- Inconsistent degree of distribution, when larger quantities are generated
- Existence of correlation among successive values
- Outputs with poorly distributed dimensions
- The distribution of the distances between where certain values occur vastly differ from those in a distribution of a random sequence.

However, pseudorandom numbers in most often cases appear to demonstrate better randomness than random numbers obtained from physical random sources. If a pseudorandom generator is constructed with adequate care, each value in the sequence will contribute to add more randomness to the subsequent value. Such transformations in series are capable of eliminating the statistical auto-correlation shared among input and output. Thus, the outputs of a PRNG may tend to demonstrate better statistical qualities while being faster in generation, than a TRNG. [8]

There are many different PRNG algorithms and systems which are in use in the modern day computing applications. Some of these could be enumerated as follows [3].

- Middle-square method (1946)
- Lehmer generator (1951)
- Linear Congruential Generator (LCG) (1958)
- Lagged Fibonacci Generator (LFG) (1958)
- Linear Feedback Shift Register(LFSR) (1965)
- Wichmann–Hill generator (1982)
- Rule 30 (1983)
- Inversive Congruential Generator (ICG) (1986)
- Park-Miller generator (1988)
- MIXMAX generator (1991)
- Add-with-carry (AWC) (1991)
- Subtract-With-Borrow (SWC) (1991)
- Maximally periodic reciprocals (1992)
- KISS (1993)

- Multiply-with-carry(MWC) (1994)
- Complementary-Multiply-With-Carry(CMWC) (1997)
- Mersenne Twister(MT) (1998)
- Xorshift (2003)
- Well Equidistributed Long-period Linear(WELL) (2006)
- A small noncryptographic PRNG (JSF) (2009)
- Advanced Randomization System (ARS) (2011)
- Threefry (2011)
- Philox (2011)
- SplitMix (2014)
- Permuted Congruential Generator (PCG) (2014)
- Random Cycle Bit Generator (RCB) (2016)
- Xoroshiro128+ (2018)

Out of these, Linear Feedback Shift Register (LFSR) (1965), Mersenne Twister (1998) and its selected varieties and Xoroshiro128+ (2018) are considered in the following subsections for detailed analysis. These choices are justified under the corresponding subsection.

2.3.1 Linear Feedback Shift Register

In the simplest form, a Linear Feedback Shift Register (LFSR) (also known as *Tausworthe generator*) is a *Shift Register*, that will derive its next state based on the immediate previous state. Even though this is susceptible to many vulnerabilities and deviations from the standards of true randomness with cryptographic security, this concept has paved way to many different implementations and innovations in PRNGs such as Linear Congruential Generator (LCG)s [16].

Basic Implementation

The absolute fundamental implementation is based on the most common bit-wise operator XOR. A popular and simple variety of LFSR is Fibonacci LFSR. Here, certain bit positions are predetermined to be affecting the next state. These predetermined positions are called *Taps*. Current bit at each tap is sequentially XORed together, would be the next bit, which will prepended after shifting the whole register by one bit position[16]. This could be graphically depicted as follows (figure 2.1).

The same implemented using Python is as follows.

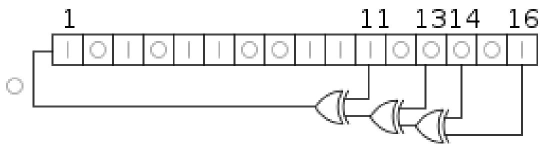


Figure 2.1: Linear Feedback Shift Register - Tap Positions [16]

```python
import numpy as np

class LSFR:
  def __init__(self, seed, l = 16):
    seedBin = format(seed, '0{0}b'.format(l))
    regTmp = [int(e.encode('ascii')) for e in seedBin]

    self.reg = regTmp[(len(regTmp) - l):]

  def next(self):
    l = len(self.reg)
    nb = self.reg[l - 6] ^ self.reg[l - 4] ^ self.reg[l - 3] ^ self.reg[l - 1]
    self.reg = ([nb] + self.reg)[:l]
    return nb
```

Listing 1: Python implementation of Fibonacci LFSR

Performance and Caveats

Due to the inherently fast nature of the core operations being used, LFSRs are known to be highly performant in terms of the speed of generation, which makes them useful in applications that requires fast generators. Moreover, the resultant bit sequence demonstrates a considerable statistical quality. Yet, it suffers from the major weaknesses enumerated below.

1. LFSR generates a deterministic output stream. Given that the present state of the LFSR along with the positions of taps, one could derive the next states easily.
2. Output streams are reversible; a LFSR with mirrored taps enable regenerating the output bit sequence in the reverse order of the original string.

2.3.2 Mersenne Twister

Mersenne Twister (MT) is a 623-dimensionally equidistributed uniform pseudorandom number generator. The most commonly used version of the Mersenne Twister algorithm is based on the Mersenne prime $2^{19937} - 1$ (hence the name). This is precisely the period of the particular implementation. The algorithm is based on a definition of a k-distribution as *a very reasonable definition of randomness*[23]. This was chosen mainly due to that fact that this is one of the most commonly adopted random generator in many applications and libraries including but not limited to Microsoft Excel[25], MATLAB[22], PHP[29], Python[31][30], C++ (V.11 and forth)[14] and CUDA[26]. As for the pros and cons, following could be enumerated [23].

Pros

- Licensed permissively and royalty-free for all of its variants except CryptMT.
- A substantially large interval of $2^{19937} - 1$ (in MT19937 implementation)
- Included in many programming languages and libraries.
- Passes a variety of statistical tests, including but not limited to the Diehard tests and a majority of TestU01 tests.
- k-distributed upto 32bit for each k in $1 \leq k \leq 623$
- Generally faster than other methods. A study indicates that the Mersenne Twister is capable of generating 64-bit floating point random numbers about 20 times faster than the RdRand (a hardware-implemented instruction set which is based on the processor).[33]

Cons

- Relatively large state buffer(2.5 KiB), (TinyMT variant is to overcome this problem).
- Mediocre throughput as per the modern standards (SFMT variant is to overcome this).
- Multiple instances that differ only in seed value (but not other parameters), are not considered to be suitable for Monte-Carlo simulations, which require different and independent random generators. However, there exists some techniques of choosing multiple sets of parameter values which are void of such problems.
- Could take a considerably long time before it actually starts generating output that passes randomness tests, if the initial state is lacks statistical qualities of true randomness. This is particularly if the initial state is composed a large number of 0's. This is also known as the *Zero-excess Initial State*. Consequentially, with closely similar initial states, will output sequences which are almost same for a large number of iterations, before diverge from the similarity eventually. Update for MT published in 2002 has improved the initialisation, so that such initialisation states were yielded unlikely.
- All implementations except CryptMT variety, are considered to be not cryptographically secure. This is due to the fact that observing an adequate number of iterations (624 in the case of MT19937; size of the state vector from which subsequent iterations are generated) will enable an attacker to predict all the future iterations. This matter is further discussed below with a sample implementation.

In the definition 1.1 of the article, Matsumoto and Nishimura suggest and establish that

"A pseudorandom sequence x_i of w-bit integers of period P satisfying the following condition is said to be k-distributed to v-bit accuracy: let trunc$_v(x)$ denote the number formed by the leading v bit of x, and consider P of the kv-bit vectors

$$(\mathrm{trunc}_v(x_i), \mathrm{trunc}_v(x_{i+1}), \ldots, \mathrm{trunc}_v(x_{i+k-1}))(0 \leq i < P)$$

Then, each of the 2^{kv} possible combinations of bits occurs the same number of times in a period, except for the all-zero combination that occurs once less often. For each $v = 1, 2, \ldots, w$, let $k(v)$ denote the maximum number such that the sequence is $k(v)$-distributed to v-bit accuracy[23].

Mersenne Twister is based on the linear recurrence

$$x_{k+n} := x_{k+m} \oplus (x_k^u \mid x_{k+1}^l)A$$

where $k \in \mathbb{Z}_0^+$. According to Matsumoto and Nishimura, "*We have several constants: an integer n, which is the degree of the recurrence, an integer r (hidden in the definition of x_k^u), $0 \leq r \leq w-1$, an integer m, $1 \leq m \leq n$, and a constant $w \times w$ matrix A with entries in $\mathbb{F}_2$*" [23]. Here w is the word length, hence $x_i \in \mathbb{F}_2^w$ would be word vectors of length w. Here A is,

$$A = \begin{bmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots, a_0) \end{bmatrix}$$

The term $x_k^u \mid x_{k+1}^l$ denotes the concatenation of x_k^u and x_{k+1}^l. x_k^u is the *upper $w-r$ bits* of x_k and x_{k+1}^l is the *upper r bits* of x_{k+1}. For MT19937, the parameters are set to values $w = 32$, $n = 624$, $m = 397$, $r = 31$ and a is chosen such that $a = \texttt{0x9908B0DF}$. [24]

Next, for the initialisation, with a parameter f which initialises the internal state as follows. Value of f is 1812433253, which is chosen without a particular reason[24].

$$x_i = f \cdot (x_{i-1} \oplus (x_{i-1} \gg (w-2))) + i$$

There exists a *tempering* process (T), which is intended at *compensating for the reduced dimensionality of equidistribution.* The tempering routine is defined for the Mersenne Twister as

$$\begin{aligned} y &= x \oplus ((x \gg u) \ \& \ d) \\ y &= y \oplus ((y \ll s) \ \& \ b) \\ y &= y \oplus ((y \ll t) \ \& \ c) \\ z &= y \oplus (y \gg l) \end{aligned}$$

Parameters for the above equations take values as $u = 11$, $s = 7$, $b =$ 0x9D2C5680, $t = 15$, $c =$ 0xEFC60000 and $l = 18$[23].

However, this version of MT is not considered to be cryptographically secure due to the fact that tampering process T is bijective. Since then there exists T^{-1} it is possible to create an *untempering function*. If one could record n consecutive outputs of the Mersenne Twister and untemper them using the function, it would reveal the internal state of the generator and will enable predicting all future values. The untemper routine implemented in Python is as follows (Listing 2)[24].

```
def untemper(y):
    y ^= y >> MT19937.l
    y ^= y << MT19937.t & MT19937.c
    for i in range(7):
        y ^= y << MT19937.s & MT19937.b
    for i in range(3):
        y ^= y >> MT19937.u
    return y
```

Listing 2: Untemper routine of MT, implemented in Python

This is precisely the same steps of temper function, performed in the reverse order. This routine will make the inner state of the MT instance visible to an attacker, leaving all the future values of the MT instance predictable [24].

There is a number of varieties of the Mersenne Twister, which are improvements to the original MT, to make it usable in different specific applications. These varieties include,

- **CryptMT** - This variety functions as a stream cipher which is also a cryptographically secure PRNG. This is patented unlike the other varieties. CryptMT has a Mersenne Twister at its core. The cryptographic security is provided by stream encryption[24].
- **SIMD-oriented Fast Mersenne Twister (SFMT)** - This variety is also based on the Mersenne Twister which is optimised for Single-Instruction-Multiple-Data (SIMD) operations specially of 128-bit, which are most common in modern day computers. This is also the base for the CryptMT. The implementation demonstrates following characteristics[24].
    - Approximately two times the speed of the generic MT.
    - Demonstrates better equidistribution than the MT. However it is not as good as Well Equidistributed Long-period Linear (WELL).
    - Demonstrates faster recovery from Zero-excess Initial State than the MT.
- **TinyMT** - This is a variety that was proposed to overcome the problem caused by the large state buffer. However due to the significant reduce in the state buffer to 127-bits, this suffers having a low period of $2^{127} - 1$. Therefore this is recommended only for applications that is limited in memory[24].

2.3.3 Xoroshiro128+

This is one of the most recent PRNGs which is developed by Sebastiano Vigna in collaboration with David Blackman. It is identified as an algorithm which has demonstrated drastic improvements in speed (e.g. generation times well lesser than a nanosecond per 64bit integer) along with significant improvements in statistical quality [37].

Algorithmic Details

A key function in the algorithm is the rotation. This could be graphically depicted as follows 2.2.

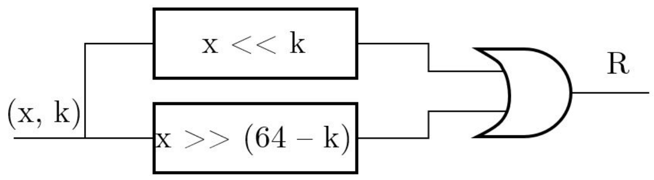


Figure 2.2: Rotation in Xoroshiro

The algorithm is named after its routine of transformations being used. The core of the algorithm performs *XOR*, *RO*tate, *SHI*ft and *RO*tate, which is the reason for the name. This routine is performed over two Initialisation Vector (IV)s which will be replaced after transformation, to the next iteration. Random bit string would be the arithmetic sum of the IVs, before transformation [37].

Generator initialised with IVs s_1 and s_2. Each iteration begins by adding the current values of the IVs to generate the next bit string (ε_i). Then, s_1 is replaced by $s_1 \oplus s_2$, followed XOR with Rotation and Shifting is performed over s_1 and Rotation is performed over s_2. At the end of the iteration, the value ε_i is returned [37].

Statistical Quality and Performance

According to the statements made by the authors,

> "*It passes all tests we are aware of except for the four lower bits, which might fail linearity tests (and just those), so if low linear complexity is not considered an issue (as it is usually the case) it can be used to generate 64-bit outputs, too; moreover, this generator has a very mild Hamming-weight dependency making our test (http://prng.di.unimi.it/hwd.php) fail after 8 TB of output; we believe this slight bias cannot affect any application* [36].

The algorithm has a 2^{128} period which is comparatively much less than that of the MT. Yet, it is capable of generating pseudorandom numbers at a rate which is as high as 1.2 nanoseconds per 64-bit number as claimed by Matt Gallagher, in the study he has conducted on the random generators used in Swift [19]. The authors of the algorithm also have taken measures to verify the results by testing the algorithm using *PactRand* and *TestU01*.

2.4 Environmental Randomness

As Linus Trovalds outlines in the documenting comments of the source code of the random generators `/dev/random` and `/dev/urandom`

> "*Computers are very predictable devices. Hence it is extremely hard to produce truly random numbers on a computer as opposed to pseudo-random numbers, which can easily generated by using an algorithm. Unfortunately, it is very easy for attackers to guess the sequence of pseudo-random number generators, and for some applications this is not acceptable. So instead, we must try to gather "environmental noise" from the computer's environment, which must be hard for outside attackers to observe, and use that to generate random numbers. In a Unix environment, this is best done from inside the kernel.*[6]

Here, the term *Environmental Noise* is an interesting topic. By the same, it is referred to the internal environment of a computer, which is composed of the hardware platform and the OS platform. However for the case of this study, this concept of environment has further extended up to the surrounding environment. According to Trovalds, the sources used in the above random generators were chosen carefully to demonstrate the following attributes.

- Non-deterministic [6]
- Difficult for an outside observer to access [6]

The first attribute above traces back to the characteristics of true randomness outlined previously in subsection 2.1.5. Furthermore, the idea of difficulty for an outsider to observe, is a relative matter which depends on how much information and probes are available to the said observer. In that sense, to the eye of an outside observer who does not possess sufficient information and probes to access a particular system, the system might appear truly random, given that the system demonstrates other characteristics of true randomness. This leads to the fact that true randomness in the current context, is relative.

2.5 Evaluation of Randomness

When choosing a source, model or a system that generates random data, it is primarily important to determine its quality of randomness. This would require different metrics of quality. To compare between many different sources of randomness for their quality, it is important that these metrics be comparable. This essentially emphasises the fact that these metrics should be quantitative and objective.

It is possible to find a handful of practical measures and tests of randomness of a binary string. Almost all of these tests include measures which are based on statistical tests, different sorts of transforms and complexity or a combination of these types. For instances one could consider the use of Hadamard transform that measures randomness. This suite was proposed by S. Kak and further developed by Phillips, Yuen, Hopkins, Beth and Dai, Mund, and Marsaglia and Zaman[32].

Among the available tests, the publication with the title "*A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*" published by the NIST under the publication number NIST Special Publication 800-22, appears to have gained wider acceptance throughout the community. This document provides a detailed elaboration of a collection of statistical tests which could be performed over a given potential input data. These tests, according to NIST could be used in different combinations to evaluate the quality of randomness of the input.

2.5.1 BSI Evaluation Criteria

The Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik, BSI) Germany, has established a four-fold criteria to asses and rank the quality of deterministic random number generators[35]. Those could be summarised as follows.

- **K1** – The probability of generated sequences of random numbers being different from each other should be high.
- **K2** – A sequence of numbers which is indistinguishable from 'true random' numbers according to specified statistical tests. These tests should include

    1. Monobit test (equal numbers of ones and zeros in the sequence)
    2. Poker test (a special instance of the chi-squared test)
    3. Runs test (counts the frequency of runs of various lengths)
    4. Longruns test (checks whether there exists any run of length 34 or greater in 20 000 bits of the sequence)—both from BSI and NIST
    5. Auto correlation test

    In essence, these requirements are a test of how well a bit sequence: has zeros and ones equally often; after a sequence of n zeros (or ones), the next bit a one (or zero) with probability one-half; and any selected subsequence contains no information about the next element(s) in the sequence.

- **K3** – It should be impossible for any attacker (for all practical purposes) to calculate, or otherwise guess, from any given subsequence, any previous or future values in the sequence, nor any inner state of the generator.

- **K4** – It should be impossible, for all practical purposes, for an attacker to calculate, or guess from an inner state of the generator, any previous numbers in the sequence or any previous inner generator states.

For cryptographic applications, generators which are meeting the K3 or K4 standard only would be accepted.

2.5.2 Unpredictability

NIST emphasizes that *Random and pseudorandom numbers generated for cryptographic applications should be unpredictable*[34]. Further, it spans the discussion towards PRNGs and highlights the fact that given that the seed of a PRNG is kept a secret, next output number of the sequence should be unpredictable, in spite of any knowledge of the previous random numbers. This is identified as *forward unpredictability*. Along with that, it is also important that, given the entire or a part of the sequence of random numbers generated is known, it should be infeasible to determine the seed used for the given sequence. This attribute is known as *backward unpredictability*. There should not be any obvious or computable correlation between the seed and any of the random numbers in the sequence [34].

2.5.3 Statistical Tests Suite of NIST

NIST, in their special publication 800-22 highlights a number of statistical tests which could be applied on a random sequence to assess and compare the sequence to determine the quality of its randomness. Randomness is widely accepted to be a probabilistic attribute; i.e., the attributes of any random string can be characterised in terms of probability. The probable outcome of any statistical evaluation, when applied on a truly random sequence, is known *a priori*. This then can be explained in probabilistic terms. A number of statistical tests are out that could be utilised that which each is evaluating if particular patterns are present or absent which, if identified, would denote that the string being considered is not random. Since there is an abundance of tests for evaluating the randomness and its degree of a given sequence, it is not possible to deem that *this finite set of tests is deemed "complete"* [34].

Apart from that, SP 800-22 also highlights that the results of statistical testing on a random sequence requires being interpteted with due care in order to mitigate possible erroneous conclusions about the generator being tested [34]. According to NIST, how the test results should be interpreted is extremely sensitive to the sequence and its utilisation. This matter is discussed in detail later under this section.

A statistical test is specifically composed to evaluate a predetermined *Null Hypothesis* ($H0$). In the test suite given by NIST, the null hypothesis being tested is that *the sequence being tested is random*. The *Alternative Hypothesis* (Ha) associated with this null hypothesis, which in the test suite, is that *the sequence is not random*. Each and every test in the NIST suite, has a decision or conclusion derived for *whether to accept* *or reject the null hypothesis*, i.e., whether the sequence being produced by the generator is random or not[34].

For each of tests taken, an applicable randomness statistic should be chosen and used to conclude if $H0$ is accepted. Such a statistic would be composed of a distribution of possible outcomes, under an assumption of randomness. A hypothetical reference distribution for the selected statistic under $H0$ would be determined by various mathematical models. Then, based on the reference distribution derived, a *critical value* is derived. Each test is designed in a manner that generates a test statistic value with statistical computations upon the sequence under test and, the resultant is compared against the previously determined critical value. For the test statistical values that exceeds the critical value, the $H0$ for randomness will be rejected[34].

Statistical hypothesis testing is for generating conclusions. The possible outcomes of a statistical hypothesis test are two folded. Those are namely,

1. Accept $H0$ (i.e. the input being considered is random)
2. Accept Ha (i.e. the input being considered is non-random)

How each *TRUE situation* is related to each possible outcome could be tabulated as follows (Table 2.1).

TRUE SITUATION	CONCLUSION	
	Accept $H0$	Accept Ha (reject $H0$)
Data is random ($H0$ is true)	No error	Type I error
Data is not random (Ha is true)	Type II error	No error

Table 2.1: Conclusions on Each Hypothesis[34]

There might be a rare case that chooses to reject $H0$ (i.e. the data stream is not random). If it is arrived at such conclusion, it is called a *Type I error*. Similarly, if it is chosen to accept $H0$ where the sequence being tested is not actually random (i.e., to accept that the data is actually random), this case is called a *Type II error*. The cases to that accepts $H0$ when the sequence is really random, and rejects $H0$ when the sequence is non-random, are deemed accurate[34].

The probability of a Type I error is being occurred is known as the *level of significance*, a property of the test. This probability (denoted as α) could be computed before a statistical test is taken. For any of the tests, α denotes the probability that a sequence would appear to have non-random properties even when a “good” generator produced the sequence. Most commonly, values of α (as applicable to cryptographic applications) are about 0.01[34].

The probability of a Type II error (denoted by β) is for any of the tests, the probability that a "bad" generator would produce a sequence which appears to demonstrate attributes of quality randomness. Unlike α, β could be any of many different values as there are many ways that a particular stream shows no randomness, and β is based on the data stream. Computation of β is quite laborious thank α because of the many different possibilities of types of non-randomness[34].

These tests primarily aim to minimise the probability of a Type II error. The computed variables α and β are related mutually and also to the size n of the sequence being tested in a way so that when two of those values are specified, the other could automatically be determined. In general a sample size n and a value for α (level of significance) are chosen upfront. Then a critical value for a given statistic is chosen in such a way so that β is smallest. In other words, aa appropriate sample size along with an acceptable probability of the generator being bad when the sequence demonstrates statistical random qualities, are chosen. Then the threshold for acceptability would be chosen so that the probability of erroneously accepting a sequence is random is the smallest[34].

Each test at its core computes a statistic value, using a function of the data being tested. Given the test statistic value S and the critical value t, value for α is computed by

$$P(S > t \mid\mid H0 \text{ is true}) = P(\text{reject } H0 \mid H0 \text{ is true})$$

and the β by

$$P(S \leq t \mid\mid H0 \text{ is false}) = P(\text{accept } H0 \mid H0 \text{ is false})$$

Using the test statistic S, a P-value is calculated to quantify the evidence which are against $H0$. For these tests, each P-value is the likelihood that an ideal random generator would have created a succession, which is less random from the sequence under test, given the sort of non-randomness that the test is aiming to assess. For P-values equal to 1, the sequence appears to demonstrate statistically perfect randomness as opposed to P-values equal to 0 indicates that the sequence appears to be completely non-random. P-values less than or equal to α would make $H0$ accepted. Otherwise (P-value is greater than α), $H0$ is rejected. α is chosen in the range [0.001, 0.01] in general[34].

2.6 Testing Strategy of NIST Statistical Test Suite

The 15 tests described in the NIST statistical test suite can be briefly enumerated as follows.

1. Frequency (Monobits) Test
2. Frequency Test within a Block
3. Runs Test
4. Test for the Longest Run of Ones in a Block
5. Binary Matrix Rank Test
6. Discrete Fourier Transform (Spectral) Test

7. Non Overlapping Template Matching Test
8. Overlapping Template Matching Test
9. Maurer's "Universal Statistical" Test
10. Linear Complexity Test
11. Serial Test
12. Approximate Entropy Test
13. Cumulative Sum Test
14. Random Excursions Test
15. Random Excursion Variant Test

The NIST publication 800-22 elaborates in detail, how these tests should be undertaken and the facts and points that needs to be taken into account when the results of tests are interpreted[34].

2.7 Common Sources of Bit Strings

There are a variety of mathematical components such as constants, functions and so forth which are yielding infinite strings of numbers. It all begins at the principles of counting and number theory. There are several different classes of numbers such as integers ($\mathbb{Z}$), real numbers ($\mathbb{R}$), rational ($\mathbb{Q}$) and irrational ($\mathbb{Q}'$) and so forth. All these categorisations are based on how they behave when they are graphically depicted on the *Line of Numbers*. Out of these, rational numbers which are not integers (i.e. $(\mathbb{Q}-\mathbb{Z})$ and irrational numbers ($\mathbb{Q}'$) are quite interesting in terms of *Number Representation in Digital Circuits*. This is because of the properties of binary numbers and their behaviour in different representations. All of the above numbers ($(\mathbb{Q}-\mathbb{Z})\cup\mathbb{Q}'$) are having fractions of whole numbers (e.g. $1/2$ is also represented as 0.5 which is a part/fraction of a number). Based on how they behave in decimal number system, they are categorised as follows.

1. Terminating Decimals - Numbers that has a finite number of places beyond the decimal point (e.g. $\frac{1}{2}$)
2. Recurring Decimals - Numbers that which its fraction has a whole or part of a finite numbers, which are recurring as a pattern. There could be two different types.
   (a) The decimal part, called the period, is repeated endlessly (e.g. $3.222... = 3.\dot{2}$ and $3.217217... = 3.\dot{2}1\dot{7}$)
   (b) The period has an irregular part followed by a regular part repeated endlessly (e.g. $0.00522222... = 0.005\dot{2}$ and $4.55127127... = 4.55\dot{1}2\dot{7}$)

3. Non terminating/Infinite Decimals - Fractional part is extended infinitely. (e.g. $\pi = 3.141592653\ldots$)

Certain researches highlights the statistically random qualities of numbers such as the value of Pi (π) which is an irrational number [1]. Furthermore, when these numbers are represented in a digital computer, there were several different challenges that which some of them still exist in certain cases. Today, most common representation of decimal numbers, used in modern binary computers is *Floating Point Representations*. However, it is a widely accepted fact that floating point is not an exact representation of a fractional value. In reality, floating point representation is an approximation of the actual fractional value. This result however, yields some interesting results which are identified and meticulously discussed in section 3.4.3.

In addition to these, enormous amounts of information are being shared in the modern world. Due to the amount of different types of devices, different connectivity technologies available for communication over Hypertext Transer Protocol (HTTP) and also the extensive usage of HTTP for business application development has paved the way for an environment which is rich of electronic data. Depending on the different protocols technologies being used, large amounts of metadata which are required to establish, secure, maintain and terminate connectivity between different interfaces are also included in this electronic data rich environment. So, it is fair to conclude that in such an environment there could be certain possible sources which their results/output is random. The existence of such and if available, how useful they are, needs to be taken into account. During this, it is important to focus on the following aspects.

- Availability - Whether such data sources are available
- Accessibility - Whether accessing such data sources is feasible
- Mechanisms for extracting data from such sources
- Mechanisms for distillation of such data
- Mechanisms for hardening of such data so that they are useful in *security critical applications* such as cryptography

2.8 Encryption

Encryption is the process of concealing the true meaning of a message by means of substitution and transposition or some other mathematical means such as *trapdoor functions*. There is a heap of research and literature that could be considered in this aspect. However, the focus on encryption for the scope of this study, is to use it in the process of hardening the generated output to meet the cryptographic security requirements.

2.8.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) (also known as *Rijndael*) is a standard for encrypting data communications accepted by the NIST in 2001. AES infact is a refinement of the Rijndael block cipher Vincent Rijmen and Joan Daemen, two Belgian cryptographers. Rijndael encompasses a family of encryption schemes that supports a multitude of block and key sizes. For AES, three members of the Rijndael family were chosen by the NIST, with block lengths of 128, 192 and 256 bits.

AES encryption is four staged process which could be outlined as follows.

1. `KeyExpansion` — round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.
2. Initial round key addition: `AddRoundKey` — each byte of the state is combined with a block of the round key using bit wise xor.
3. 9, 11 or 13 rounds for each key length respectively:
   (a) `SubBytes` — a ***non-linear*** substitution step that replaces each byte based on a lookup table.
   (b) `ShiftRows` — a transposition step where the last three rows of the state shifted to right in a cyclic manner.
   (c) `MixColumns` — a linear mixing operation which operates on the columns of the state, The four bytes of each column are combined here.
   (d) `AddRoundKey`
4. Final round (giving a total of 10, 12 or 14 rounds for each key size):
   (a) `SubBytes`
   (b) `ShiftRows`
   (c) `AddRoundKey`

2.8.2 Block Cipher Modes of Operation

Block ciphers have various modes of operation, which could be enumerated as follows.

1. Electronic Code Book (ECB) - This is the simplest form that a block cipher could be employed in. There, each input block is encrypted in isolation from the previous blocks. Due to this isolation of blocks in operation, certain properties of the plain text such as regions, might be preserved in the cipher text. Hence this is considered to be weak. Yet, this is useful in certain communication channels which are inherently non-reliable, due to the fact that communication errors are not propagated. The process could be graphically depicted as follows (Figure 2.3).

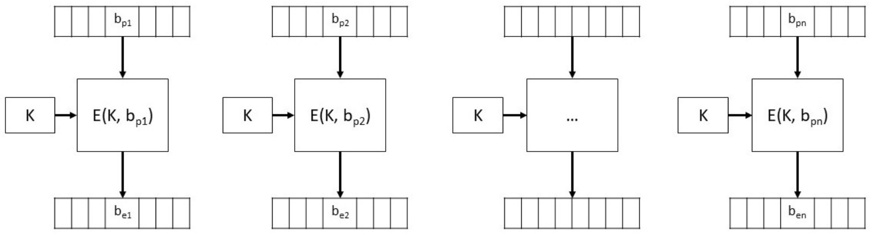


Figure 2.3: Electronic Code Book (ECB) Encryption

2. Cipher Block Chaining (CBC) - This model introduces an additional step that includes an XOR operation on the plain text block and an IV, that which its output is encrypted with the key. For each successive blocks, the previous block's cipher text becomes the IV. The process could be graphically depicted as follows (Figure 2.4).

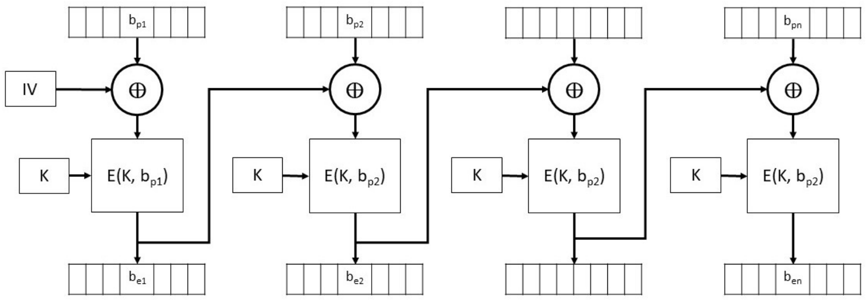


Figure 2.4: Cipher Block Chaining (CBC) Encryption

Even though this effectively ties each block together, in order to to allow decryption of the message by the recipient, the IV should be shared with the recipient. If an intruder manages to predict the IV, then the encryption would not be resistant to *chosen plain text attacks*[1]. If the IV is chosen to be some input such as a user password, then it requires to be encrypted using a separate key. IVs should be changed after some time, so as to not to make the system vulnerable to chosen plain text attacks. Also, an isolated error of a single bit during the transmission of the cipher text would be propagated across the rest of the cipher text, yielding the whole cipher text useless. in decryption.

3. Cipher Feedback (CBF) - This mode of operation is quite similar to the CBC, except for that the IV for each iteration/block is encrypted with the key. Then, the resultant is XORed with the plain text block, which will provide the IV for the next iteration. The process could be graphically depicted as follows (Figure 2.5).

   This is also vulnerable to error propagation during encryption, due to the tying of blocks using the cipher of each block.

[1]a cryptanalyst can choose arbitrary plain text data to be encrypted and then he receives the corresponding cipher text

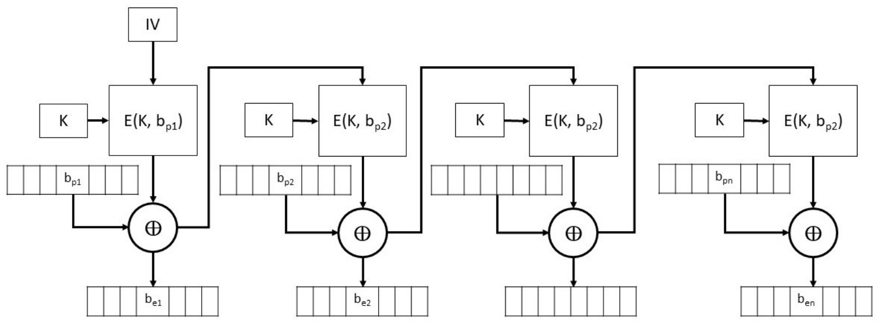


Figure 2.5: Cipher Feedback (CBF) Encryption

4. Output Feedback (OBF) - The IV is repeatedly encrypted to provide the IVs for each successive blocks in this mode. This is also a slight variation of the CBF mode discussed previously. The process could be graphically depicted as follows (Figure 2.6).

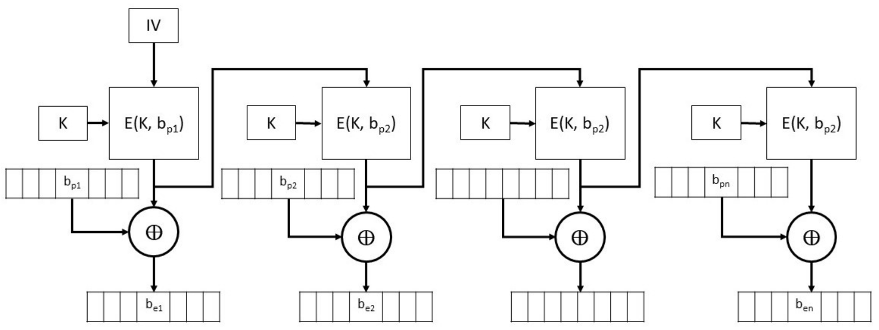


Figure 2.6: Output Feedback (OBF) Encryption

If an error occurred in a single bit of plain text or cipher text (for an instance due to a transmission error), only one corresponding cipher text or respectively plain text bit is damaged as well. This could however be recovered using various correction algorithms to restore the previous value of damaged parts of the message. The most critical drawback of OFB is that the state of the system might be repeated due to the repeated encryption of the same plain text. Even though the probability of occurring such situation is quite low, in such a case the plain text blocks will be encrypted with the same corresponding state as previous.

2.9 Conceptual Framework

Based on the knowledge established in the previous sections, the following conceptual framework is formulated to be used in establishment of the scope and further development of the research project (Figure 2.7).

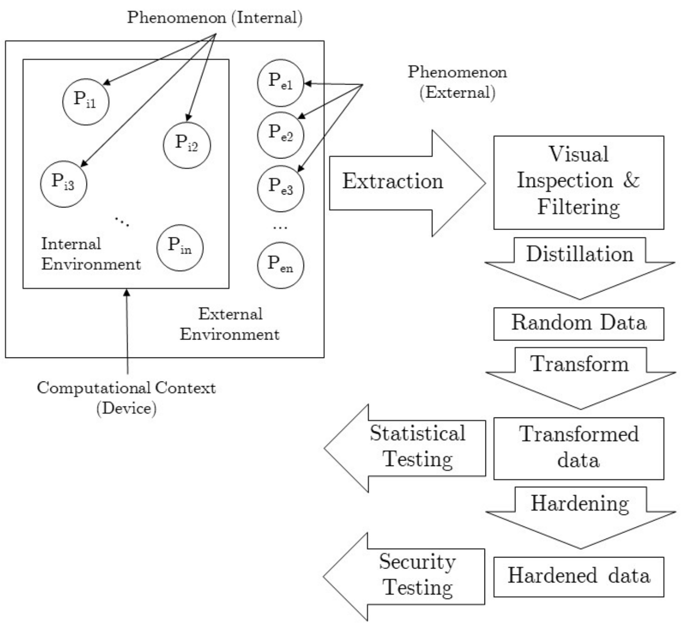


Figure 2.7: Conceptual Framework

2.9.1 Elaboration of the Concept

The research is based on the internal and external phenomena that are observable internally. These sources are monitored and recorded during the *Extraction*. Caution must be exercised when choosing these sources to ensure that an external manipulator has no knowledge or access (ideally) or the least practical knowledge and access. Based on these criteria, the observations recorded are filtered during the *Visual Inspection and Filtering* stage.

The data collected from the filtered sources would undergo a distillation process to extract the exact values and filter them based on the criteria demanded by the transformation work flow, which would play the role of *seeds* which would undergo the *Transformation*, which would magnify the values that has been distilled. These transformed data would be taken into evaluation as inputs to the statistical tests suite. To meet the cryptographic security requirements, the data could optionally be hardened by the hardening strategies identified and assessed in the corresponding section 3.5.

Chapter 3

Methodology

This chapter is dedicated to elaborate the design and the methodology of execution of the research project. The chapter commences with a thorough elaboration of the research methodology, followed by the detailed description of the sources of data which have been taken into consideration. Afterwards, the details of data collection strategy, data sanitisation and transformation strategy is described. Towards the conclusion of the chapter, the resultant data is summarised into appropriate visual forms of presentation.

3.1 Computational Device as an Environment

According to the *Systems Thinking*, it is possible to portray the *Computer* as a system. Typically a system,

- has predefined objectives.
- has components interacting with each other.
- components cooperate and coordinate with each other to achieve the objectives.
- enclosed by a boundary.
- might or might not interact with the *surrounding environment*.

An abstract schematic that graphically depicts this outline could be sketched as in figure 3.1. It is quite straightforward that this approach enables identifying two distinct partitions of the space that the system exists. Those are namely,

- *Internal Environment* - The space that which is within the system's boundary
- *External Environment* - The space that which is out of the system's boundary

This research identifies both of these partitions as *the Environment*. The research focuses on the phenomenon that takes place in this environment. The observations made by the author along with the assumptions and interpretations are discussed in detail in the following chapter.

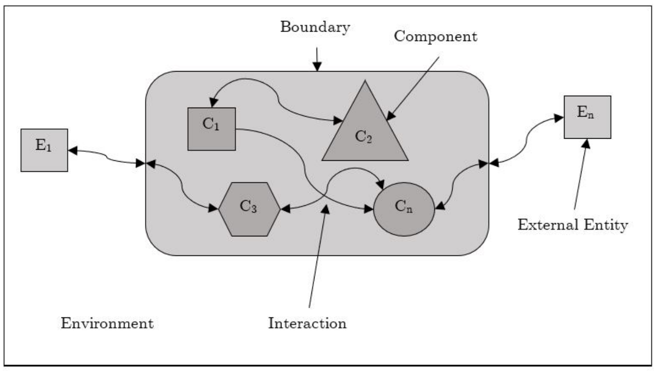


Figure 3.1: Abstract Schematic of a System

3.2 Extraction and Visual Inspection

Identifying different phenomenon from this environment of the computational device and collecting the metrics related to those are the activities which are due on the *Extraction* phase. There, primarily the following sources were considered as data sources for extraction of randomness.

1. Device metrics such as Random Access Memory (RAM) usage, Central Processing Unit (CPU) usage, temperature measures and so forth.
2. Network packet flows. Here, it was mainly focused on the wireless network adapters that the device is having built in. Specifically the Wireless Local Area Network (WLAN) adapters that the device is having and operating in the ***monitor mode*** (also known as ***promiscuous mode***) was extensively used.

The reason behind choosing these sources was primarily due to the readily availability. One of the main motives behind this study is to discover the ability for an ordinary personal computational device to generate randomness while being portable, cost effective and performant. Costly peripherals attached to the devices via cables or any other source, would not be positively supporting to uphold that motive. Moreover, large volumes of systematic, structured and intentional data flows are quite effective in concealing subtle, less important yet useful effects which the causes are uncertain and unpredictable.

The data extracted from the sources were visually inspected in the second stage so as to filter out any obvious and frequently repeating data points and items, as an added measure to further ensure the quality and the suitability of these measures in security critical applications. Even though the network traffic data was collected, they were not taken into consideration during the next steps, due to the added complexity and overhead of reliably filtering those data to filter out the less secure measures, as those data are originated well away from the system boundary. However, these data and the network data as a source of entropy, could be considered and accounted in expanding this study further.

3.3 Distillation

Distillation is the phase where the data points which sequentially together demonstrate absence of patterns, are extracted from the filtered sources. During the extraction of data, initially the collected data pool was visually inspected using small *sliding windows*[1] of time. The variation of each observed attribute across the sliding window was considered and the attributes which does not demonstrate rapid changes were dropped out. For the ease of observation, these data were tabulated and visually inspected over approximately 1000 samples. End result of the distillation would be microscopic data points/values, or macroscopic values which could be used to generate microscopic results with the use of some operation or a combination of operations, which will provide the seeds for transformation in the next step.

3.4 Transformation

During the transformation, the seeds from the previous stage are transformed into different representations in order to meet the volume and complexity requirements of the randomness. Since the requirement is a large bit string which is sufficiently complex, the primary alternative considered was *Floating Point Number System.*

3.4.1 Floating Point Representation System

Floating Point Representation is a binary representation of a fractional number, which is which is widely used in modern day computers. A Floating Point representation attempts to capture and represent the following details about the particular number.

1. Sign (represented by s)
2. Exponent (represented by E) - A measure of the position of the decimal point of the number being represented.
3. Significand (also known as *Mantissa*, represented by m) - The actual value of the number.

Usually, to represent the sign, a single bit would be allocated. For the other two parameters there are standard value combinations as well as arbitrary settings. When a number is represented using floating point

1. The number is represented in the binary number system.
2. Binary number is normalised (discussed in subsection 3.4.2 below).

[1]A predetermined time frame of fixed length, that will be sled across a sequence from the beginning to the end. At any point in time, what belongs to within the time frame is taken into consideration

3. The index of two (2) is obtained and transformed using the bias (discussed in subsection 3.4.2 below) and the biased exponent is converted to binary. This will be assigned to E.
4. Fractional bits of the normalised number is assigned to m.
5. CONCATENATE(s, E, m) will be the final representation of the number.

3.4.2 Normalisation of the Binary Number and Exponent

If the number $+13.25_{10}$ is taken into consideration for an instance, it's binary representation would be $+1101.01_2$. In applied mathematics, a number is said to be normalised when it is written in scientific notation with one non-zero decimal digit before the decimal point [17]. So, the above number in decimal could be written as $+1.325 \times 10^1$. The same process on the binary representation would result in $+1.10101 \times 2^3$. It is also important to note that, if the first two steps are interchanged, the result would be drastically different.

As per the steps above, next is to convert the index of two to binary. Now if an imaginary result of a binary conversion of a fraction is considered such as 0.000101, normalising such would result in 1.01×2^{-4}, which the index of two is a negative number. This causes another problem of representing signed magnitudes. This could in a way be overcome by representing this using a number representation such as *2's complement*. However, since 2's complement would make the forward process and the reversal process too much complicated, and since this is not directly used in any arithmetic operations, it is possible to reliably use a *bias value*, which would map an interval $\{-n..n+1\}$ to an interval $\{0..2n-1\}$. The bias value is based on the number of bits allocated as the exponent and for i which is the number of bits in exponent, bias value is given by $2^i - 1$.

3.4.3 Analysis on Attributes of Floating Point

Previously it has been established that floating point is an approximate representation of a fractional number. If the fractional number 3.2 is considered for an instance, representing this as a binary number would result in $11.001100110011\ldots$ Upon closer examination, it is evident that the fractional part of the binary representation above, has a repetitive pattern of $\dot{0}01\dot{1}$. Apart from those special cases, many floating point representations were appearing to have an uneven spread of binary digits across the representation. So, it is decided to closely review the behaviour of floating point representations to identify its usability as a *model of transformation* of certain metrics. This review was conducted according to the following routine.

1. Implement a routine for floating point conversion of arbitrary structures.
2. Starting from one digit at the mantissa in the decimal number, convert the given mantissa to binary. Here, the length of the mantissa m was chosen to be around

10^6. It could be extended beyond that however, such cases will be sensitive to the performance of the device being used.

3. Once the bit string is generated, plot the bits into an $x \times y$ (where $x \cdot y = m$) bitmap for visualisation.

4. At the end of each iteration, append a randomly chosen digit (here the digit is chosen by author with no conscious intervention) to the input on the previous stage, and repeat from step 2.

A separate routine that generates the bitmaps were implemented using Python with the use of `matplotlib`. `matplotlib` is a powerful module that could be plugged into Python, for a vast variety of image processing tasks. Once the above process is repeated for over 1000 times for different choices of digits, it was decided to visually and mathematically evaluate the bitmaps. Based on those, it is possible to discover the patterns and assess their closeness to randomness. For the benchmark of this activity, a bitmap which is captured from a television which was out of synchronisation with a visual signal, was used (Figure 3.2) and each bitmap was compared visually, mathematically and then the data and the observations were tabulated.

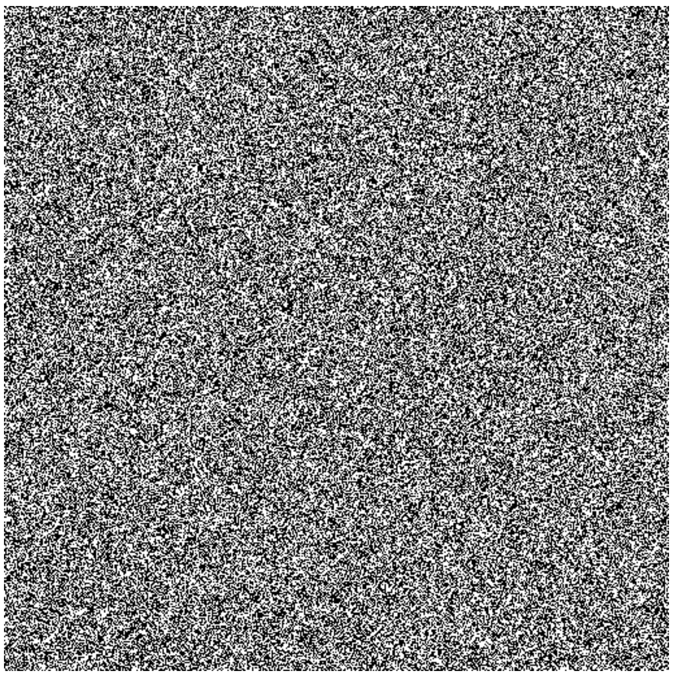


Figure 3.2: Television Noise generated, during absence of a Signal

Each bitmap generated above, were structurally compared using Mean Squared Error (MSE). MSE is a measure of average of the squared errors for each pixel of the image. For each sample, the MSE is calculated and plotted in a graph, in order to evaluate the variation as the number of digits increased. A visual inspection on each image also was carried out in order to note the perceivable differences and variations. This test is hereafter identified by the name *Initial Random String Test*.

3.5 Hardening

Hardening is an optional phase, which would only be required in the context of security critical applications such as cryptography. Intention of this phase is to make sure that the data being generated becomes more tamper proof. This stage is primarily essential to inculcate the attributes of a *Cryptographically Secure Random Number Generator*, which in brief are

1. Given that the first k bits of a random sequence is known, there should not exist a polynomial time algorithm that would predict the $k+1^{\text{th}}$ bit, any better than 50% of success.
2. In case if a part of the sequence is successfully asserted, it should not be possible to predict the part of the sequence before the revelation.
3. If the generator uses some sort of an entropy input, it is expected to be infeasible to utilise the knowledge on the state of the input for predicting future conditions of the generator.

Specially, the 3^{rd} point on the above list is crucial in a security critical application. In order to achieve this, following alternatives were taken into consideration.

3.5.1 Symmetric vs. Asymmetric Key Encryption

Symmetric Key Encryption (also known as *Private Key Encryption*) is one of the two fundamental flavours of data encryption. The absolute purpose of encryption is to hide the meaning of some message from everyone else except the intended recipients. This is primarily some function $E(m,k)$ where m is the message and k is the encryption key. The reason for this to be known as symmetric is that, to decrypt and reveal the original plain text, the same key that was for encryption is used, which is where the symmetry of the process is considered at.

Asymmetric Key Encryption (also known as *Public Key Encryption*) is the counterpart flavour of the previous. Popular and widely used encryption schemes such as RSA, ECC and so forth. Here also exists some function $F(m,K)$ where m is the message and K is the encryption/decryption key. Here, the asymmetry comes from the fact that the K in fact are two interrelated keys k_p and k_u in such a way that, what is encrypted with k_p could only be decrypted by k_u and vice versa. This addresses certain inherent issues in symmetric key encryption such as key distribution problem. Since the two keys are entirely different, the term *asymmetric* is coined. However, schemes of this model are having certain other drawbacks such as implementation complexity and specially, most of the encryption schemes of this family requires a random Initialisation Vector (IV), which causes a *cyclic dependency* between random number generation and asymmetric encryption, for most of the algorithms.

Due to the obvious reasons it has been decided to lean towards symmetric encryption. This leaves the following concerns to be addressed when an encryption scheme is chosen.

1. Resistant to attacks
2. Less implementation complexity for being compatible with the personal computing devices
3. Less consumption of computational power for being compatible with the personal computing devices

As discussed previously in section 2.8.1, AES has some powerful features which are quite useful along subtle weaknesses that could be overcome with caution. When AES works in CBC mode, it requires an IV, that which could be provided as a password for the system, which enables enforcing standards over the password. Since the encryption process does not require to have a decryption process, the administrator(s) will not have the requirement to remember the password, which will enable enforcing strict standards. This will provide the additional security required to meet the cryptographic security standards.

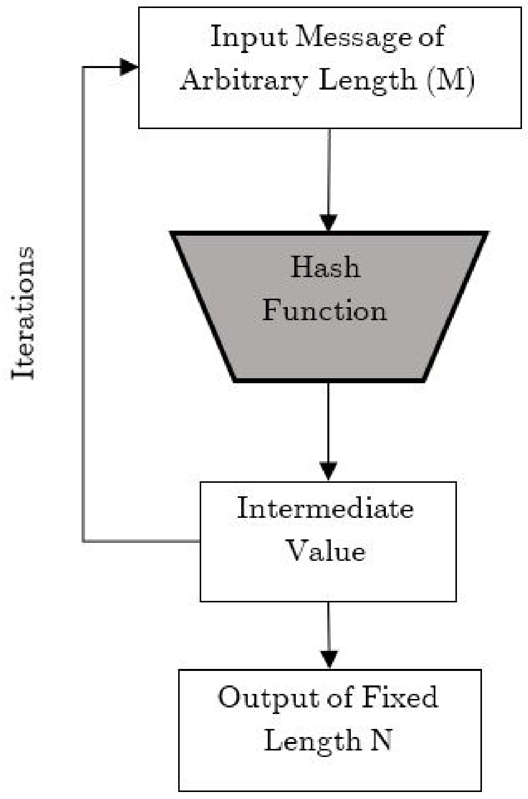


Figure 3.3: Work flow of a Hash Function

3.5.2 Message Digest and Hashing

Hashing, which is also known widely as *Message Digest* is a process of mathematically transforming an input of an arbitrary size, to an output which is most of the times of fixed size. Precisely, hash functions generate an output which is called the *Message Digest*. Typical work flow of a hashing algorithm is as depicted in Figure 3.3.

The most important attribute of hashing is that, given only the output it is a process which cannot be reversed to obtain the initial input or the state of the algorithm. This makes hashing a powerful function, which has attributes that could be leveraged in generating random numbers. There already are a wide variety of RNGs which are based on hashing algorithms. Such algorithms include but not limited to *Hash_DBRG, HMAC_DBRG, CTR_DBRG* and so forth [13].

3.5.3 Use of Hashing within the Context

Hashing was considered as an option for hardening in the context of this research. Since hashing is an irreversible process, a block which appears to be arbitrary could be easily hidden by using hashing, as the reverse mapping is not present and computationally infeasible to generate that. However, all the available hashing algorithms are generating lengthy hashes (about 128 bits and above), which would bring in the requirement of clipping their outputs. This would add more overhead and dependencies on additional functions such as sponge functions[2]. Hence hashing was left out from further consideration, as a possible further work.

[2]Finite state algorithms that take a bit stream of any length to produce a bit stream of a given length

Chapter 4

Implementation

This chapter is dedicated to the implementation related details of the models discussed in the previous chapters. In order to capture and thereafter execute the conceptual model discussed previously, a variety of mechanisms and software implementations were required. Details related to those implementations and the justifications for the corresponding decisions and choices are included and elaborated in this chapter.

4.1 Capturing of Data

In order to assess the behaviour of a system, collection of relevant and accurate data is a must. For the purposes of this study, a number of readings were collected over a period of more than 30 days. Data capturing routines were implemented using Python programming language. Python was chosen as the language for implementation, due to its simple and expressive syntax, support for extended arithmetic operations such as power and floor division and the comparatively higher availability of various modules and extension Application Programming Interface (API)s.

During the capturing of data, reading and transformation concerns such as fast and easy access, was taken into consideration. Hence, the data being captured were stored in `JSON` initially. JavaScript Object Notation (JSON) was chosen as the storage structure due to the simplistic nature and flexible support for that format awarded by python. Further, the data that was captured was processed later as a batch and written into a database in MongoDB. MongoDB was chosen again due to the freedom interfacing with python and freedom and performance in querying.

As discussed previously, this study focuses on both the internal and the external environments, under the constraints of

- Technological Feasibility
- Portability
- Economical Feasibility (i.e. Cost effectiveness)

There are certain phenomenon such as software and hardware interrupts, context switches and so forth, which are deemed to be sufficiently random by certain applications, such as `/dev/random` and `/dev/urandom` in Unix Systems [**?**].

Python has a module named `psutil` which is designed to capture the CPU and RAM metrics and so forth. A python routine was implemented which consumes these functions from the `psutil` which is attached as appendix B. When collecting data, the following functions in `psutil` were invoked.

1. `psutil.cpu_stats()` - Collects CPU statistics including hardware and software interrupt counts, number of system calls and number of context switches
2. `psutil.sensors_temperatures(1)` - Core temperature metrics of the CPU
3. `psutil.virtual_memory()` - Various counts of memory of the computer including but not limited to available memory, used memory, cached memory and so forth
4. `psutil.disk_usage("/")` - Disk usage of a given mount point (in linux)

A sample of such a result, represented as a MongoDB document is indicated in the code listing below (Listing 3).

```
{
    "_id" : 1555041606,
    "sdiskusage" : {
        "total" : NumberLong(51471126528),
        "used" : NumberLong(42555625472),
        "percent" : 87.1,
        "free" : NumberLong(6277283840)
    },
    "scpustats" : {
        "interrupts" : 2092921,
        "soft_interrupts" : 1537679,
        "syscalls" : 0,
        "ctx_switches" : 4502862,
        "temperature" : {
            "nouveau" : [
                [
                    "",
                    120.2,
                    203.0,
                    221.0
                ]
            ],
            "acpitz" : [
                [
                    "",
                    123.8,
                    null,
                    null
                ]
            ],
            "coretemp" : [
                [
                    "Core 0",
                    111.2,
                    221.0,
                    221.0
                ],
                [
                    "Core 1",
                    111.2,
                    221.0,
                    221.0
                ]
            ]
        }
    },
    "svmem" : {
        "available" : 1991725056,
        "used" : 1804324864,
        "cached" : 1569783808,
        "percent" : 51.9,
        "free" : 391614464,
        "inactive" : 837197824,
        "active" : NumberLong(2480824320),
        "shared" : 48029696,
        "total" : NumberLong(4140666880),
        "slab" : 303345664,
        "buffers" : 374943744
    }
}
```

Listing 3: Example of CPU data collected into a MongoDB document

Two systems were monitored with this routine where one system was running at idle and the other was being used by a software developer. A total of 1838879 data documents from idle system and 1999681 data documents from the working system, which maps to each second of a mix of continuous and noncontinuous time frames were collected and, each document was assigned with the corresponding epoch timestamp[1] as the ID.

4.2 Distillation and Transformation

The metrics obtained in the previous step were observed initially and the numeric values were extracted. During the extraction the values that were potential seeds, the primary concern was those values to demonstrate variations over time and the *rate of change* to be as high as possible. In order to do a systematic comparison, two systems were monitored simultaneously, that which both of them were having the same version of the same operating system running, one running at idle, and the other one being used by a software developer which engages in various software development work. Then, the metrics were evaluated for their quality as seeds for the transformation process.

The distillation was mainly based on statistical analysis and visualisation of data points using graphs. for the primary filtering of data items, values for each metric for ten seconds were visually inspected for their appearances of suitable variations. This was repeated over a 100 different samples and then the observations were tallied. A sample of such observations is graphically depicted below (Table 4.1).

[1]The Unix epoch (or Unix time or POSIX time or Unix timestamp) is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT), not counting leap seconds (in ISO 8601: 1970-01-01T00:00:00Z).[2]

sdiskusage.percent	83.5	83.5	83.5	83.5	83.5	83.5	83.5	83.5	83.5	83.5
sdiskusage.free	8080375808	8080375808	8080375808	8080375808	8080375808	8080375808	8080375808	8080375808	8080375808	8080375808
scpustats.interrupts	62734861	62735347	62735834	62736290	62736681	62737063	62737443	62737847	62738260	62738686
scpustats.soft_interrupts	32646241	32646570	32646907	32647196	32647394	32647585	32647759	32647938	32648143	32648406
scpustats.syscalls	0	0	0	0	0	0	0	0	0	0
scpustats.ctx_switches	75075959	75076482	75077126	75077759	75078166	75078568	75078967	75079360	75079786	75080366
scpustats.temperature.nouveau.0.0										
scpustats.temperature.nouveau.0.1	118.4	118.4	118.4	118.4	118.4	118.4	118.4	118.4	116.6	118.4
scpustats.temperature.nouveau.0.2	203.0	203.0	203.0	203.0	203.0	203.0	203.0	203.0	203.0	203.0
scpustats.temperature.nouveau.0.3	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0
scpustats.temperature.acpitz.0.0										
scpustats.temperature.acpitz.0.1	120.2	120.2	120.2	120.2	120.2	120.2	120.2	120.2	120.2	120.2
scpustats.temperature.acpitz.0.2	None	None	None	None	None	None	None	None	None	None
scpustats.temperature.acpitz.0.3	None	None	None	None	None	None	None	None	None	None
scpustats.temperature.coretemp.0.0	Core 0	Core 0	Core 0	Core 0	Core 0	Core 0	Core 0	Core 0	Core 0	Core 0
scpustats.temperature.coretemp.0.1	105.8	104.0	104.0	104.0	104.0	104.0	104.0	104.0	104.0	104.0
scpustats.temperature.coretemp.0.2	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0
scpustats.temperature.coretemp.0.3	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0
scpustats.temperature.coretemp.1.0	Core 1	Core 1	Core 1	Core 1	Core 1	Core 1	Core 1	Core 1	Core 1	Core 1
scpustats.temperature.coretemp.1.1	105.8	104.0	104.0	104.0	104.0	105.8	105.8	105.8	105.8	105.8
scpustats.temperature.coretemp.1.2	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0
scpustats.temperature.coretemp.1.3	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0	221.0
svmem.available	2684878848	2684878848	2684891136	2684899328	2684899328	2684899328	2684899328	2684899328	2684899328	2684899328
svmem.used	1117835264	1117835264	1117835264	1117835264	1117835264	1117835264	1117835264	1117835264	1117835264	1117835264
svmem.cached	1331859456	1331859456	1331859456	1331859456	1331859456	1331859456	1331859456	1331859456	1331859456	1331859456
svmem.percent	35.2	35.2	35.2	35.2	35.2	35.2	35.2	35.2	35.2	35.2
svmem.free	1271947264	1271947264	1271947264	1271947264	1271947264	1271947264	1271947264	1271947264	1271947264	1271947264
svmem.inactive	662401024	662401024	662409216	662409216	662409216	662409216	662409216	662409216	662409216	662409216

Table 4.1: Tabulation of CPU statistics within a 10 second window for observation (Idle Device)

As per the table, it is obvious that the following parameters demonstrate a rapid change.

1. Number of Context Switches
2. Number of Hardware Interrupts
3. Number of Software Interrupts

Other metrics such as CPU core temperatures, disk usage and all are demonstrating no changes at all in most cases. The same was observable in other windows also for the static/idle system. So, it has concluded to utilise a combination of these variables as the inputs to the seed generator.

4.2.1 Floating Point Conversion Algorithm

The author has taken note on the fact that the floating point algorithm is quite simple to implement and execute. More importantly, the algorithm could be implemented in such a way that it takes arbitrary values for its primary parameters, the exponent (E) and significand (m). Hence, this algorithm was implemented using python, to take a decimal number as a string and to convert it to a floating point number of arbitrary exponent and mantissa size. Python was chosen as the language to implement, primarily due to its expressive syntax and native support for certain operations on arrays and lists such as map-reduce functions, *for-each* loop, type conversions and string manipulation operations. The algorithm which was initially implemented is as the code listing outlined below (see Listing 9).

```
    # Resolution of float
f = n1[1]
f = [int(e.encode('ascii')) for e in f]

zero = [0] * len(f)

sig = []

decPosFound = decPos > -1

while True:
  if f == zero:
    break

  [of, f] = self.__arrayTimes2(f)

  if not decPosFound:
    if of == 1:
      decPosFound = True
    else:
      decPos -= 1

  if len(sig) >= (m - decPos):
    break

  sig.append(of)
```

Listing 4: Mantissa Derivation Subroutine of Floating Point Converter

The intention of implementing this algorithm was to closely analyse the behaviour of the bit strings of the numbers when they are converted to floating point. Upon the analysis, the mantissa conversion routine was identified to be a suitable component, which could be used at the core of the proposed algorithm for generation. This routine is as outlined below (listing 4).

4.2.2 Proposed Algorithm - FloatRAND

The proposed algorithm implements the mantissa conversion routine at its core, to generate the next bit. All that is provided by this algorithm is a mechanism to utilise a microscopic value (hereafter referred to as seed) to generate random bit stream which are of high statistical quality. This is precisely a state machine that depends on the inputs given and is a One-to-One mapping between the seed and the bit string generated. For an instance,

$$0.33_{10} = 0.01010100011110101 11\ldots_2$$

where the seed 0.33 into this algorithm will produce 010101000111... as the random bit stream. The algorithm is designed in a manner that it takes two arbitrary integers S_1 and S_2, both of which are 6-7 digits long and the ratio (i.e. S_1/S_2) of these are taken as the seed. Now, given a quotient, there are infinitely many number of possible combinations for the numerator and denominator, which all of them are correct. Therefore it is computationally infeasible to determine the exact numerator and denominator pair used to obtain the given quotient. The only possibility is doing an exhaustive brute force on all possible combinations and querying someone who knows the answer who in fact is the challenger himself. If the challenger is not willing to disclose the answer, there is no way that it could be verified. Hence this can be regarded as a non-deterministic problem.

Algorithmic Details

Initialisation of the algorithm requires two seeds which are about 6-7 digits long (could be grater) and approximately similar in size. Quotient of these two seeds are used by the core routine to generate bits. Quotient should be obtained to a precision about 20 digits in order to generate bits at the optimum. Larger quotients will generate bit strings which demonstrate better statistical quality, at the cost of lower generation speeds. Precision below 10 digits are not recommended due to the low statistical quality and existence of obvious patterns.

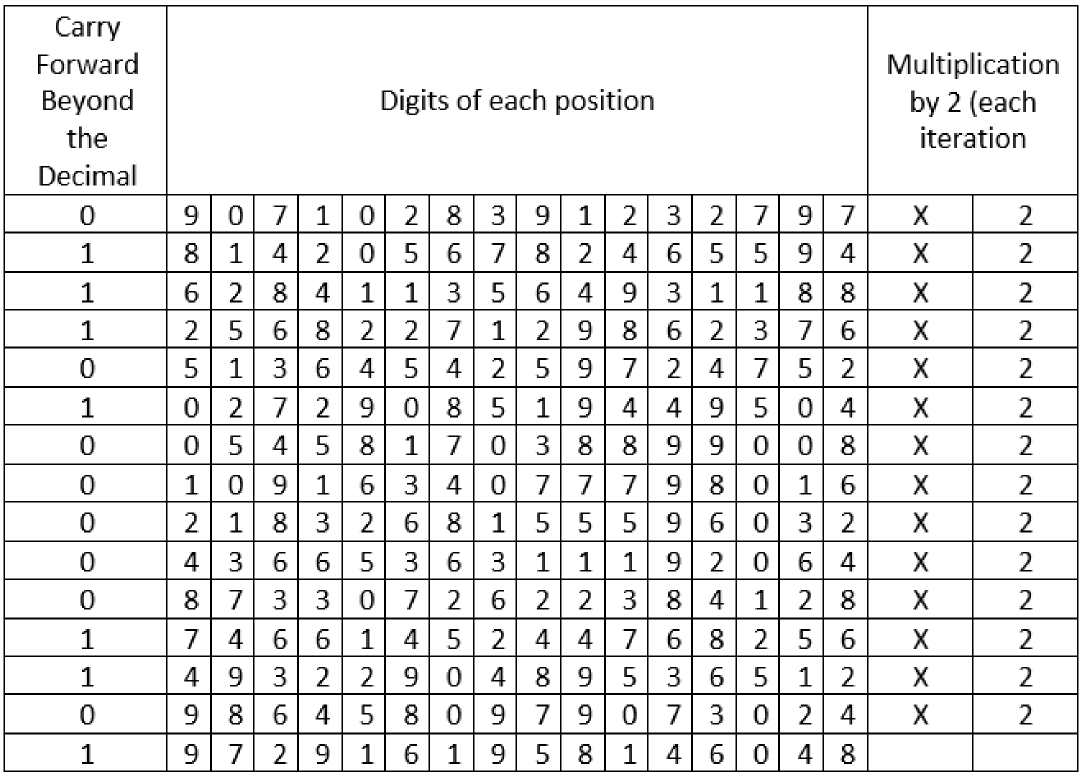
The fractional portion of the quotient is pushed into the seed buffer and then the seed buffer is repeatedly multiplied by two (2) and the overflow of the multiplication will be the next bit. Hence the multiplication of any decimal digit is always less than 20, the overflow will always be either 1 or 0. The generation stops when either the desired length is reached or all digits in the seed buffer becomes 0. This could be elaborated with an example as follows.

Assume two microscopic observations S_1 and S_2, which yielded the following values.

$$S_1 = 23728192$$

$$S_2 = 26158216$$

The quotient S_1/S_2 would result in 0.9071028391232797, which the fraction could be represented as a binary string using the following routine, graphically depicted as below (figure 4.1).



Carry Forward Beyond the Decimal	Digits of each position																Multiplication by 2 (each iteration	
0	9	0	7	1	0	2	8	3	9	1	2	3	2	7	9	7	X	2
1	8	1	4	2	0	5	6	7	8	2	4	6	5	5	9	4	X	2
1	6	2	8	4	1	1	3	5	6	4	9	3	1	1	8	8	X	2
1	2	5	6	8	2	2	7	1	2	9	8	6	2	3	7	6	X	2
0	5	1	3	6	4	5	4	2	5	9	7	2	4	7	5	2	X	2
1	0	2	7	2	9	0	8	5	1	9	4	4	9	5	0	4	X	2
0	0	5	4	5	8	1	7	0	3	8	8	9	9	0	0	8	X	2
0	1	0	9	1	6	3	4	0	7	7	7	9	8	0	1	6	X	2
0	2	1	8	3	2	6	8	1	5	5	5	9	6	0	3	2	X	2
0	4	3	6	6	5	3	6	3	1	1	1	9	2	0	6	4	X	2
0	8	7	3	3	0	7	2	6	2	2	3	8	4	1	2	8	X	2
1	7	4	6	6	1	4	5	2	4	4	7	6	8	2	5	6	X	2
1	4	9	3	2	2	9	0	4	8	9	5	3	6	5	1	2	X	2
0	9	8	6	4	5	8	0	9	7	9	0	7	3	0	2	4	X	2
1	9	7	2	9	1	6	1	9	5	8	1	4	6	0	4	8		

Figure 4.1: FloatRAND - Generation

In the figure provided, it is clearly visible that each iteration of generating the mantissa is resulting in either 1 or 0 which could be represented as a bit. This repeated for n times will yield a bit string of n bits long. Given the seeds S_1 and S_2 are chosen from random sources, the algorithm provides adequate statistical qualities of randomness which are desired by most of the applications in the personal domain. If the sources of randomness chosen are proven to have the quality attributes of randomness discussed above, the generator shall have the desired security as well, due to the non-deterministic nature of the core.

Known Weaknesses

However, the following weaknesses were identified at the initial stage.

- Seed and the bit string has a one to one mapping. Hence, an attacker who has the output bit string can generate the seed, but not the random variables.
- Seeds with too many zeros, less number of digits (digit count below 7) are generating strings which are of poor quality
- Performance depends on the computational resources.
- Serial - Each iteration of the core generates only one bit

Chapter 5

Evaluation

This chapter is dedicated to elaborate the various aspects which are related to evaluation process. Primarily it is focused on the implementation and execution details of the NIST statistical test suite, followed by the test invocation examples. Along with the NIST suite, the Initial Random String Test and the results are also elaborated here, along with the interpretation of the results. Then, the various test results and their interpretations along with the possible conclusions are derived towards the latter half of the chapter.

5.1 Initial Random String Test - Observations

As discussed previously, the evaluation on this test is two fold. The results and the interpretation are as elaborated in the subsections below. The results enumerated here are for 1000 samples.

5.1.1 Visual Inspection

The visual inspection on the images was done primarily taking the following concerns into account.

1. Perceived Similarity with the TV noise image
2. Absence of any obvious patterns
3. Plain white/black patches which are obviously large

The results and the interpretation of the visual inspection could be itemised per each appended digit up to 8th image, as below (Figure 5.1).

- **1 Digit** - When it is only one digit, except for '5', all other digits demonstrated a pattern with alternating lines. The image generated by the seed 0.8 is given in the figure below (Figure 5.1).

- **2-4 Digits** - A relatively small patch which appears to be uneven is repeated over the whole image. Bitmap generated by the seed with 2-3 digit appears to demonstrate some check/diagonal pattern. The bitmap generated with 3-digit seed is as depicted in figure 5.2

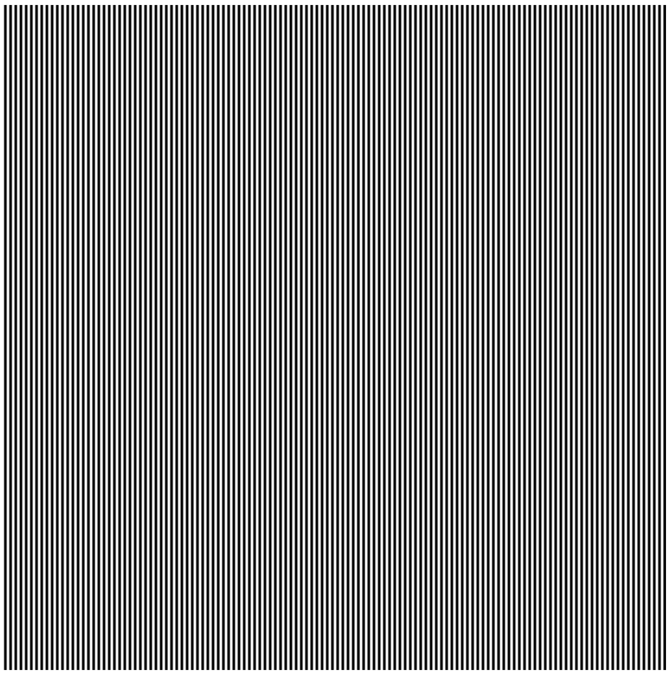


Figure 5.1: Bitmap of Random String - 1 Digit Seed

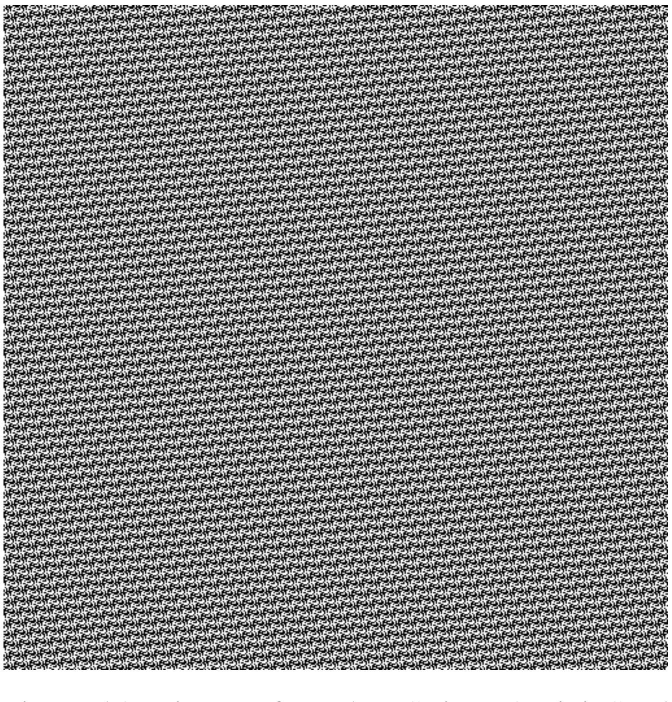


Figure 5.2: Bitmap of Random String - 3 Digit Seed

- **5-6 Digits** - A relatively large patch which appears to be uneven is repeated over the whole image. Since the patch is large, the number of times that the pattern is repeated has become lesser. Pattern still appears to be somewhat diagonal. Figure 5.3 (left) indicates the bitmap generated with a seed of 5 digits.

- **7 Digits** - The pattern has become much less obvious in the bitmap generated with 7 digits seed. Figure 5.3 (right) indicates the bitmap generated with a seed

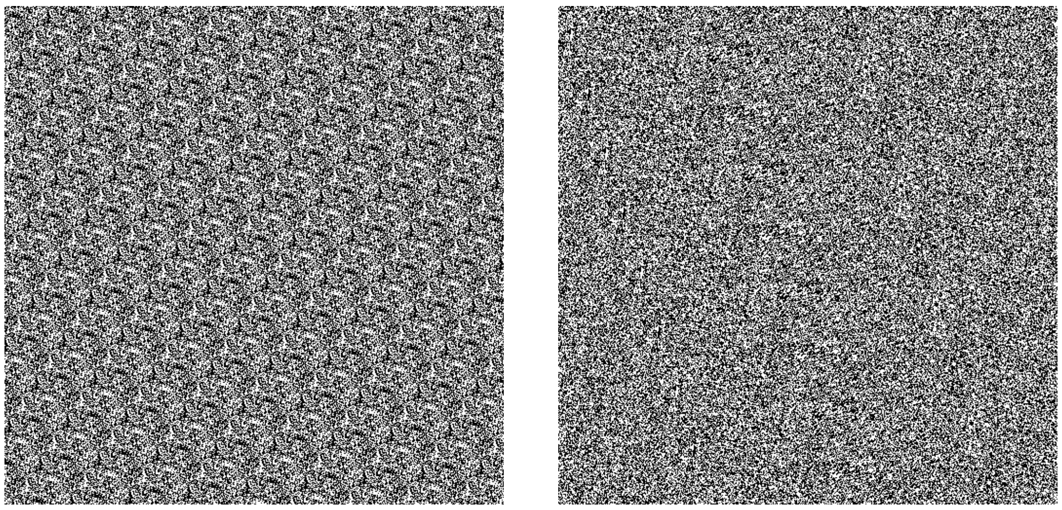

(a) 5 Digit Seed (b) 7 Digit Seed

Figure 5.3: Bitmap of Random String - 5 Digit Seed (Left) and 7 Digit Seed (Right)

of 7 digits. However, upon closer examination, some pattern could be observed. This gets obvious when the dimensions of the bitmap are increased.

- **8 Digits and above** - No obvious patterns are visible beyond 8 digits. However, it has not tested if there exists a pattern as the number of bits generated by the generator is increased. This is to be further tested in the next iterations of development. Figure 5.4 indicates the bitmap generated with a seed of 8 digits, while figure 5.5 depicts the bitmap of the bit string generated with 10 digit seed.

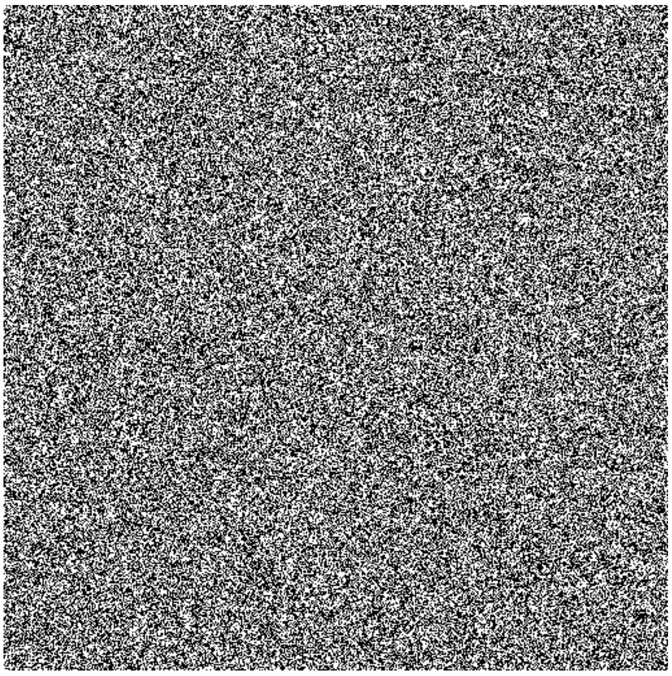


Figure 5.4: Bitmap of Random String - 8 Digit Seed

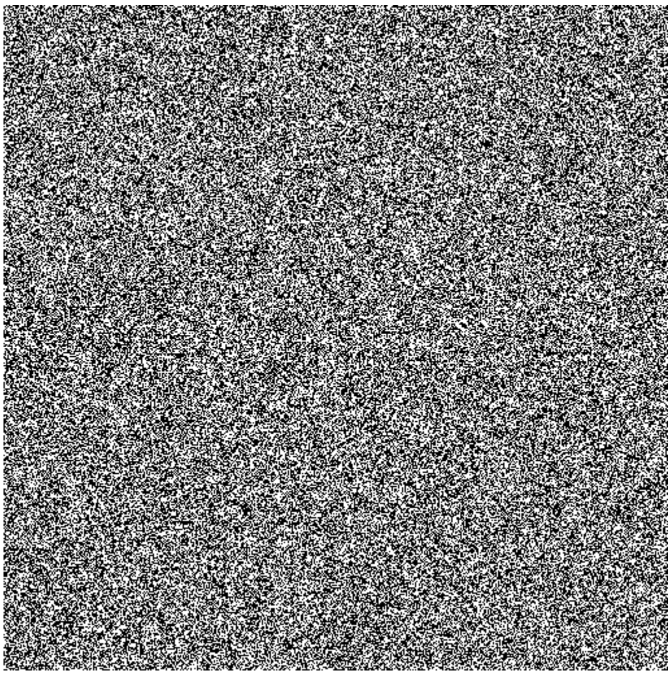
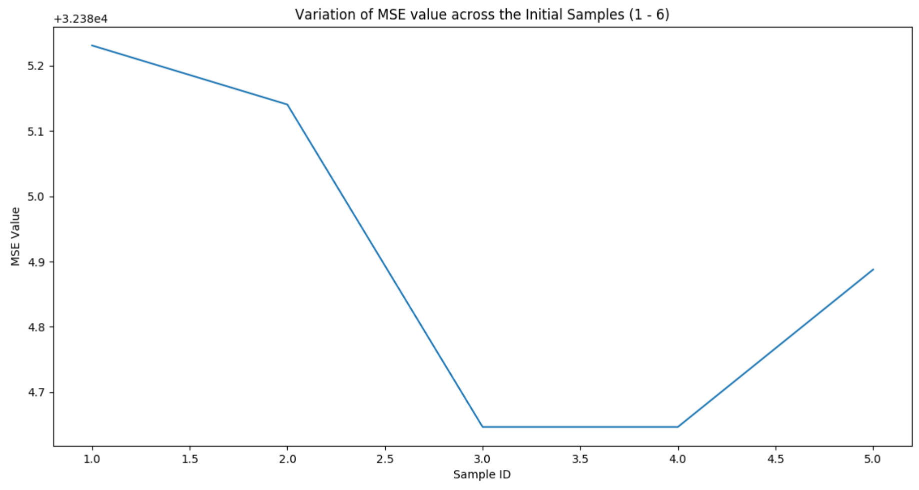


Figure 5.5: Bitmap of Random String - 10 Digit Seed

5.1.2 Statistical Testing on Bitmaps

Then, each generated bitmap is compared against the bitmap B of TV noise. For this comparison, the MSE and the SSIM values for each of the sample against B were computed and the variation of the MSE values are plotted against the sample number, into a two dimensional line chart as depicted in figure 5.6.



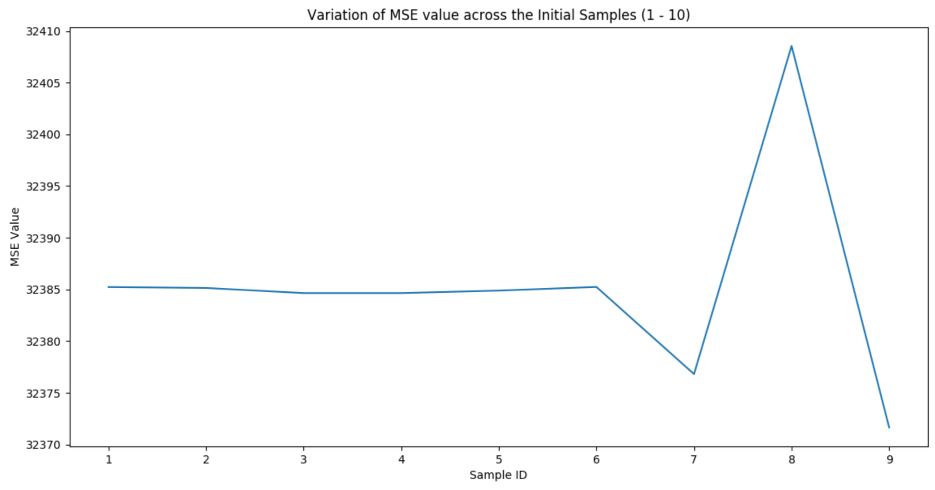


Figure 5.6: Variation of the MSE of the Samples in Initial Random String Test (Up to 10 Digits)

When these are observed, it is quite evident that the variation of the MSE for the first 6 samples are not that high. Yet, beyond that point, the MSE starts to vary quite rapidly and the variation is not demonstrating any patterns. This is visible in the graphs in figure 5.7.

These observations are falling in line with the observations of the visual inspection outlined above. The observations were quite similar when the same test was conducted on other samples of randomly chosen strings. Further, each of the bit strings was tested with the NIST suite and a summary report for each random string tested is generated.

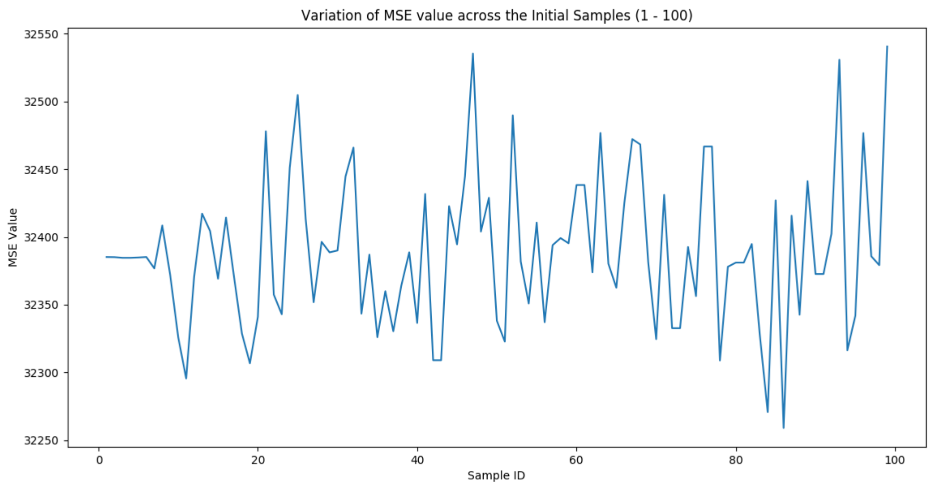


Figure 5.7: Variation of the MSE of the Samples in Initial Random String Test

The test reports correspond to sample 0001 and 0500 are attached as appendix F and appendix G respectively. The test results were summarised as follows.

- Total number of instances, passed and skipped of each instance was summarised as mentioned below.

```
{
    "_id" : "SAMPLE_0001",
    "passed" : 6,
    "total" : 188,
    "passRate" : 0.0319148936170213
}
```

- Then, the passRate was plotted against the sample id in a 2D plot (figure 5.8).

Upon closer observation, it is evident that the initial samples are showing very low pass rates and beyond sample bearing the ID 6, the pass rates of the tests exceed 80% of pass rate and settle to vary almost linearly around 90%. Test which are dropping the pass rate close to 80% are the tests that which the instances Random Excursion Test and Random Excursion Test (Variant) were skipped due to the inadequate number of cycles in the random walk of each bit string. This could be better observed in the graphs below (figure 5.9)

5.1.3 Performance during Testing

For generating 10^6 bits, the processing time has demonstrated variations which are as summarised and enumerated below, in the second itemised list. The generation timings were measured in a device with the following configuration.

- **Processor** - Intel Core i5-6400 CPU @ 2.70GHz $\times$ 4
- **RAM** - 7.7GiB

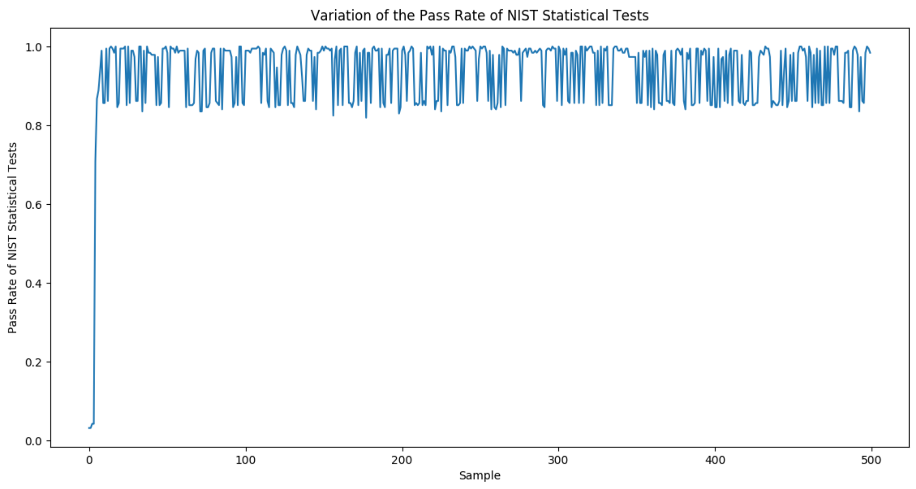


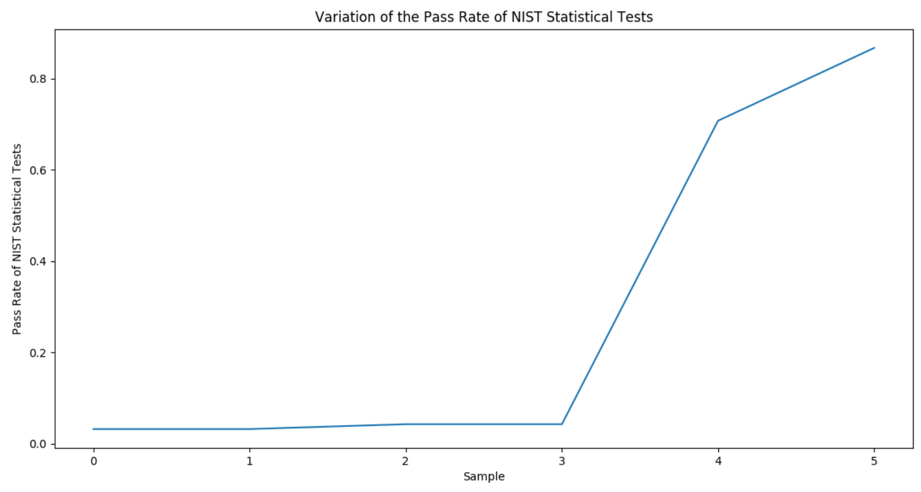
Figure 5.8: Variation of the Pass Rate of NIST Test Suite of Samples in Initial Random String Test (Samples 1 - 500)

- **Graphics** - Intel HD Graphics 530 (Skylake GT2)
- **OS** - Ubuntu 16.04 LTS

Observations

- During the earliest stage (i.e. between 1-10 digits), the generation time was well below a second.
- Up to 20 digits of seed, the generation time of the random string was close to, but less than a second.
- Between 20 to 500 digits, the generation time was rapidly increasing and the average generation time lies around five to seven seconds.
- Above 500 digits, the generation becomes quite slow, and towards the latter stage, the generation time has exceeded 10 seconds.

As per the observation enumerated above, the period for repetitions which could cause obvious patterns, rises as the seed length grows. So, longer the seed, better the generation would be. Nevertheless, at the same time, the processing time complexity also rise as the number of digits in the seed is increased. It has a time complexity of $O(n \cdot l)$, where n is the number of digits and l is the length of the bit stream. Hence, the optimum combination of n and l is dependent on the requirement. However, the test combination used during the study (i.e. $n = 20$ and $l = 10^6$) appeared to be sufficiently performant in terms of volume and statistical quality.



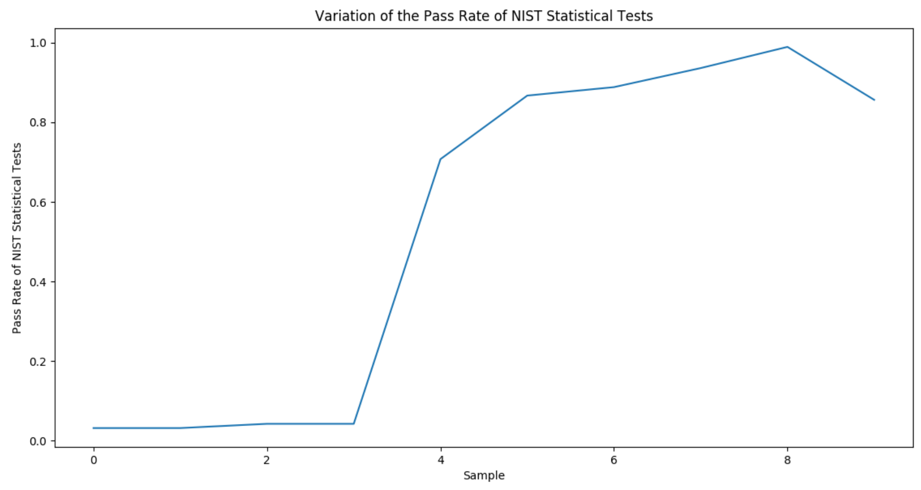


Figure 5.9: Variation of the Pass Rate of NIST Test Suite of Samples in Initial Random String Test (Samples 1 - 6)

5.2 Visual Inspections

Further, the data sequences chosen were visually inspected by plotting them on graphs against the timestamp they were obtained. This was conducted for the both IDLE and WORKING systems. Obtained plots are as graphically depicted below. Figure 5.10 depicts the variation of the total number of hardware interrupts count while figure 5.11 and 5.12 depicts the variations of total number of software counts and the ratio between hardware and software counts respectively.

These plots suggest that the variation of the differences between each observation is not following any deterministic pattern. That is, total number of interrupts or soft interrupts during the time $t+1$ is determined by the system environment and has no correlation whatsoever to the same counts at the time t. Even though their variation appears to be varying close to linear, obtaining pin point values would be extremely hard for an outside observer, and the implementation the proposed algorithm requires

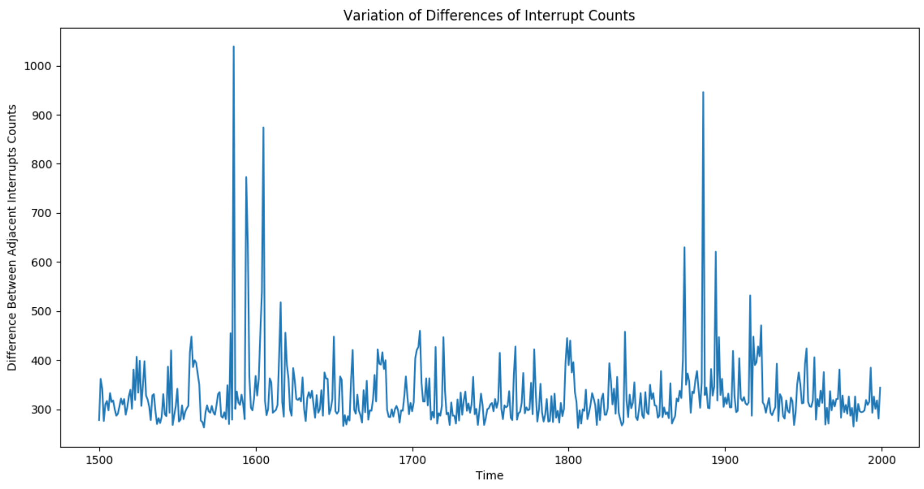


Figure 5.10: Variation of the Total Hardware Interrupts Count (Between 1500th and 2000th seconds - IDLE System)

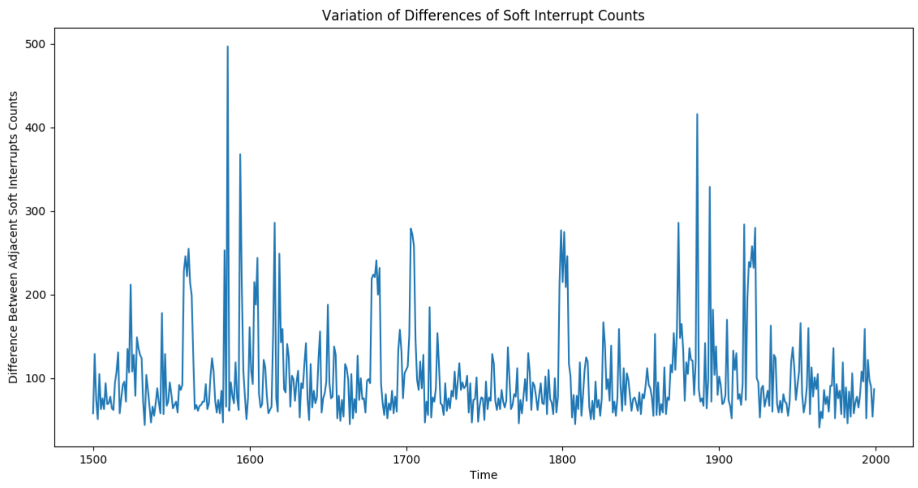


Figure 5.11: Variation of the Total Software Interrupts Count (Between 1500th and 2000th seconds - IDLE System)

to have a 100% match with the seed, in order to successfully compromise the state. This becomes even mote complex, as appearing in the figures 5.14 and 5.13 which plot the variation of the interrupts count (top), soft interrupts count (bottom) and the variation of the ratio of the two metrics respectively for the WORKING system.

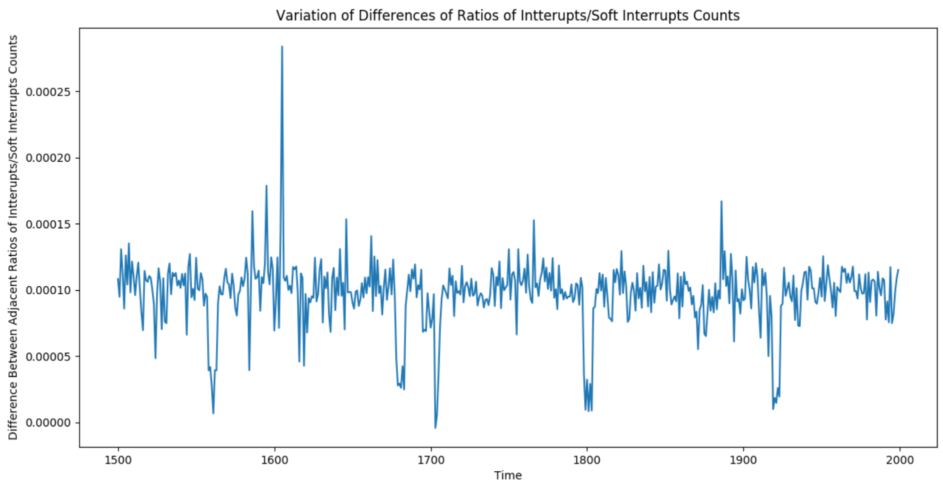


Figure 5.12: Variation of the Ratio between the Hardware and Software Interrupts Count (Between 1500th and 2000th seconds - IDLE System)

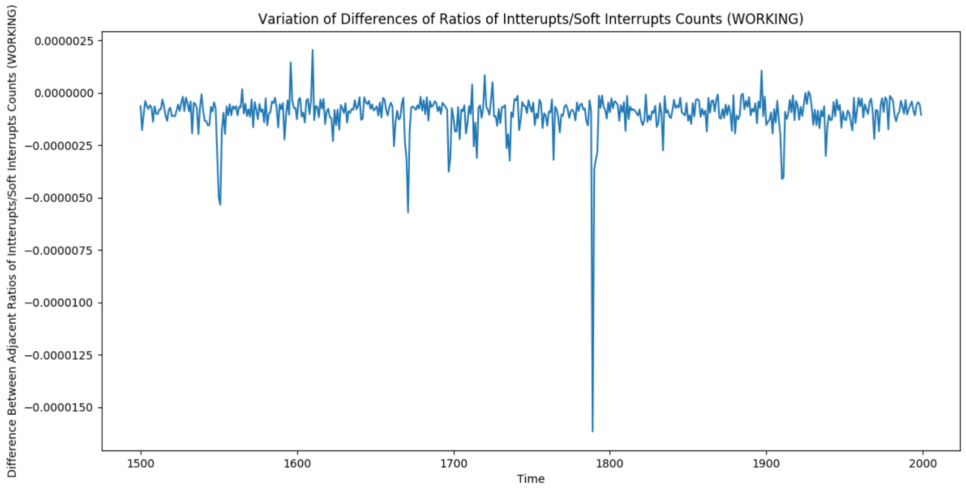
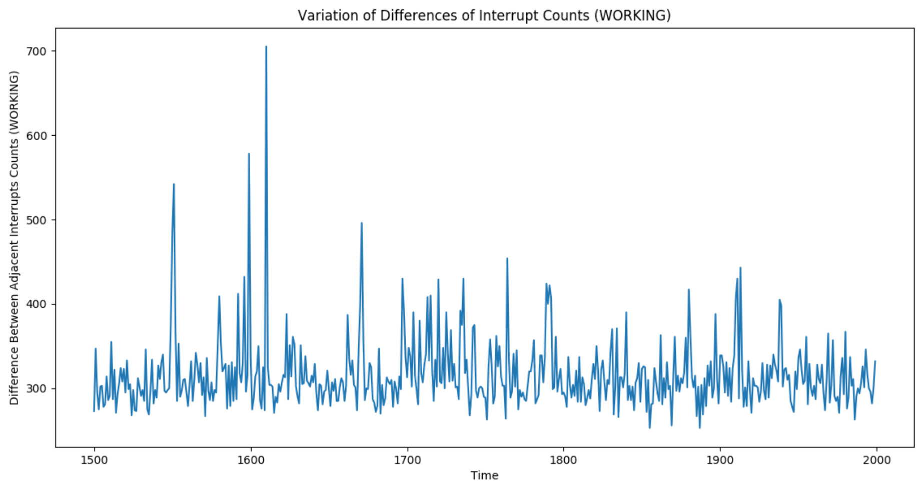


Figure 5.13: Variation of the Ratio between Total Hardware and Software Interrupts Count (Between 1500th and 2000th seconds - WORKING System)

5.3 Test Results

Afterwards, the data sequences were used as seeds for the generator and 10^6 bit long bit sequences were generated from each seed pair. Then each of the bit string generated was tested with the NIST statistical test suite. For these tests, only a sample of observations were used from each of the IDLE and WORKING system. This constraint was imposed due to the long time which was consumed. For each bit string of 10^6 bits, it takes little over 1 minute to complete the entire test suite and each test produces a test summary, which is as given in appendix H.

For bench marking, two of the previously discussed algorithms were used. Mersenne



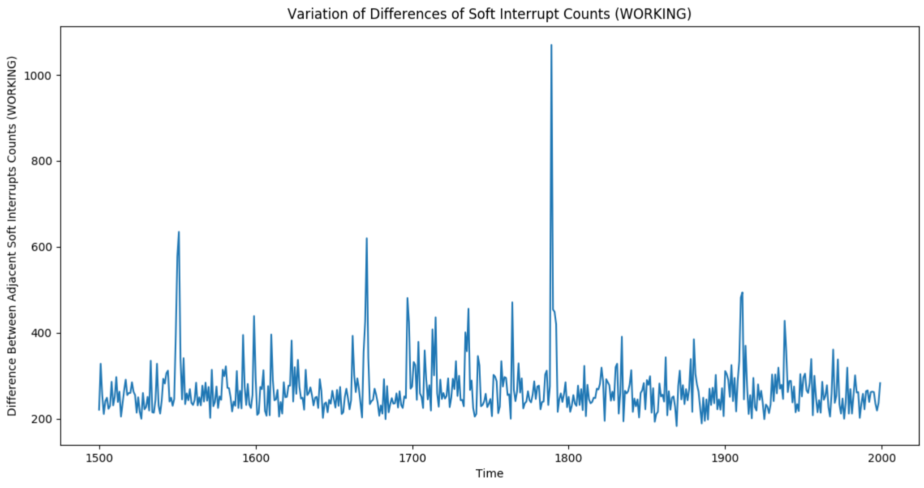


Figure 5.14: Variation of the Total Hardware and Software Interrupts Count (Between 1500th and 2000th seconds - WORKING System)

Twister (MT19937) and Xoroshiro128+ was used for bench marking and LFSR was purposely dropped from the benchmark due to the fact that the generator is already outdated. From each of MT and Xoroshiro128+, 1000 samples of random strings were generated and each of these were statistically tested using the NIST test suite.

The test suite executes multiple instances of each tests, selected from the entire set. Out of the all of 15 tests, Random Excursion Test and Random Excursion (Variant) Test was not executed on approximately 45% of the entire samples which were tested. This was due to the absence of sufficient number of random walks in the bit strings which were skipped. All of the other tests were executed. There are a total of 188 test instances from all tests, that which the number of instances executed for each test is as tabulated below (table 5.1).

The test summary was used to summarise the test results for each of the systems. Similar to the previous case of Initial Random String Test, here also the pass counts were summarised for each sample tested as follows.

Test Name	Number of Instances
Frequency	1
Block Frequency	1
Cumulative Sums	2
Runs	1
Longest Run	1
Rank	1
Spectral (DFT)	1
Non Overlapping Template	148
Overlapping Template	1
Universal	1
Approximate Entropy	1
Random Excursion	8
Random Excursion (Variant)	18
Serial	2
Linear Complexity	1
Total Number of Instances	**188**

Table 5.1: Total number of Instances per Test Class

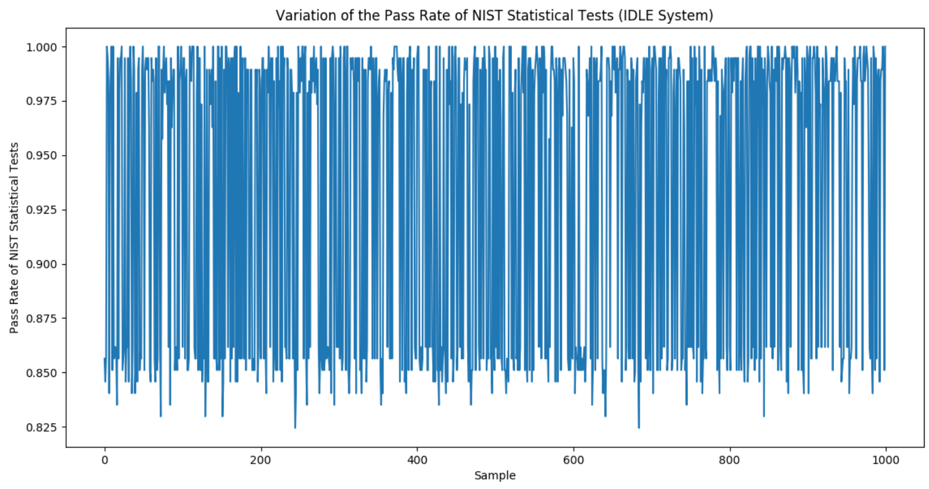
```
{
    "_id" : "2090775_1536805",
    "passed" : 161,
    "total" : 188,
    "passRate" : 0.856382978723404
}
```

Then, these summaries were plotted on 2D line graphs against their index in the sequence. The variation of the pass rates of the tests conducted for the proposed algorithm are as depicted below in figure 5.15. The top most graph plots the variations of the proposed algorithm executed on the idle system. The graph at the bottom is the variation of the pass rates of the tests conducted on the bit strings generated by the seeds extracted from the working system.

Figure 5.16 outlines the variation of the pass rates of the tests conducted on the bit strings generated with Mersenne Twister (top) and the Xoroshiro128+ (bottom).

Interpretation

Upon close examination it is evident that the pass rates of the random strings generated by the proposed algorithm are much higher compared to the pass rates of the bit strings generated by the Mersenne Twister. The proposed algorithm shows pass rates between 82% and 100% with much large variation, while Mersenne Twister demonstrates results which are between 44% and 56% with comparatively less variation. For the case of



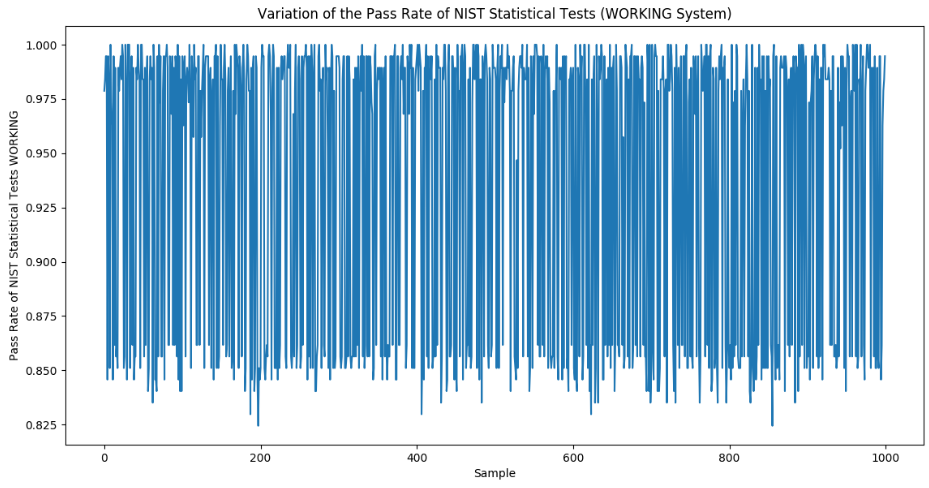


Figure 5.15: Variation of the Pass Rate of NIST Test Suite for Proposed Algorithm, MT and Xoroshiro128+

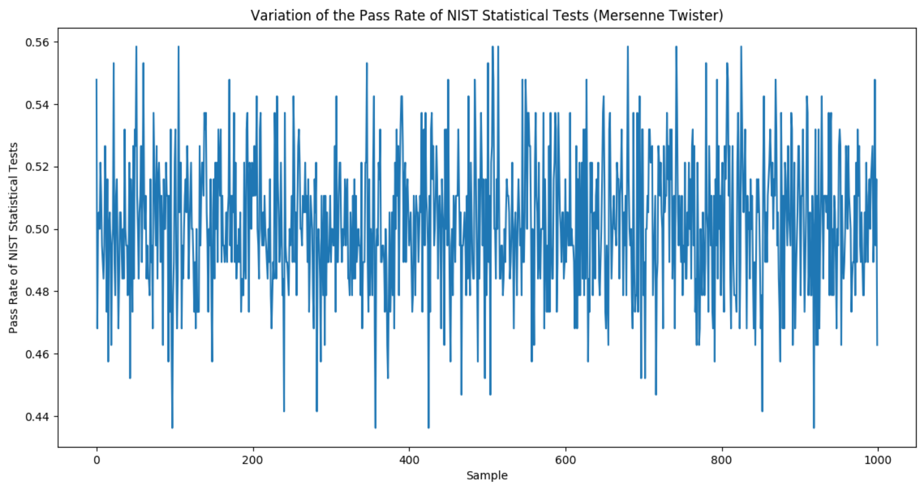
Xoroshiro128+, the pass rates vary between 82% and 99.9%, with the lower bound of variation is mostly close to 85%. This is almost similar and close to that of the proposed algorithm.

5.4 Statistical Summarising of Test Results

Then the sequences of pass rates were summarised using the mean, variance and standard deviation for each sequence. These measures were obtained for each of the system considered and they are as tabulated below (table 5.2).

Interpretation

Since the mean values for proposed algorithm and MT are 0.934809 and 0.501771 respectively, the higher mean of the proposed algorithm, suggests that the average statisti-



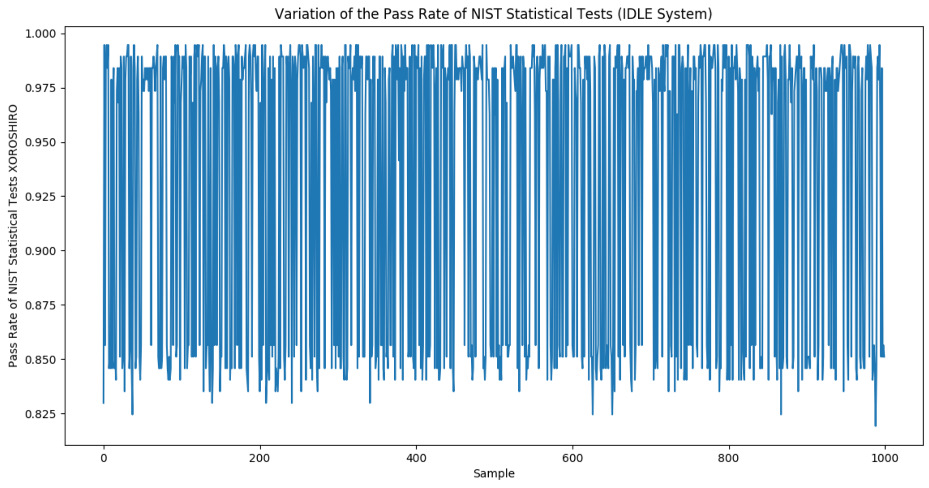


Figure 5.16: Variation of the Pass Rate of NIST Test Suite of the bit string generated by MT and Xoroshiro128+

System	**Mean**	**Variance**	**Standard Deviation**
Mersenne Twister	0.501771	0.000407	0.021690
Xoroshiro128+	0.935840	0.004422	0.066500
FloatRAND (Idle)	0.934809	0.004563	0.067552
FloatRAND (Working)	0.937500	0.004482	0.066947

Table 5.2: Summary of Statistical Measures

cal quality is much higher in the proposed algorithm than the typical MT. The difference is roughly about 43% between the two. However, the proposed algorithm demonstrates much higher standard deviation, which is almost 5% more than that of the MT. This suggests that MT demonstrates more consistency than the proposed algorithm in terms of the statistical quality. These properties are further highlighted by the graphs provided in figure 5.16.

Compared to the case of MT vs. proposed algorithm, the results of the comparison of Xoroshiro128+ and the proposed algorithm, are much closer. The mean values of the pass rates of Xoroshiro128+ and the proposed algorithm are 0.935840 and 0.934809 respectively, where the Xoroshiro128+ has demonstrated slightly better pass rates which is about 0.001%. When the standard deviations are compared, the standard deviation of the pass rates of Xoroshiro128+ is about 0.001% lower than that of the proposed algorithm. Hence, it could be concluded that the Xoroshiro128+ is slightly more consistent in terms of the pass rates, than the proposed algorithm. Yet, the differences are at small scales. Also it is obvious that irrespective of the chosen system (idle or working), the statistical quality of the proposed algorithm in terms of the NIST criteria remains close.

Chapter 6

Conclusion

In retrospect at the the conclusion of this research project, it is evident that random numbers play an essential role in a variety of applications. Yet, the problem ahead of the generation of random numbers is the quality of randomness of the numbers generated by the said systems.

One important aspect to note regarding the *True Randomness* is that it is impossible to attain true randomness from a finite state machines that which the conditions for the state transitions are well defined. As a rule of thumb, it is believed that ideal randomness is impractical without specialised and expensive hardware that gathers data from, for an instance, quantum events, there is no such thing as *ideally true random number generator* (i.e. a RNG that generates truly unpredictable number). Nonetheless, it is possible to reach the quality attributes of true randomness. PRNGs in the modern context are doing that and in fact, most PRNGs in use have become better at it.

Yet, they tend to have a problem of repetition of the output after a certain large number of iterations. This is known as the period of the said generator. Irrespective of how large the period is, it still repeats, making it vulnerable to certain attacks such as replay attacks. The problem lies in the core concept, which is to deterministically change the seed for the generator to get different numbers. This in almost all the cases causes the output to repeat after a known number of cycles. This is the reason why a PRNG usually requires a seed provider which ideally should be a TRNG. However, since such seeds are not quite common and even capturing the commonly available random variables from the surrounding is expensive, this has become and exhaustive matter.

This study takes a slightly different approach to resolve the problem. Instead of deterministically changing a seed, this uses a microscopic random variable as a seed and utilises a commonly available representation as a generator core. The proposed algorithm's randomness therefore is entirely dependent on the randomness of the seed. As long as the variations of the seed is random, the output will also be random. This highlights the fact that the source of seed should have the attributes of randomness, instead of the algorithm. And also this study have focused on identifying the possible sources of seeds which has the required attributes of randomness. Further, the identified sources were statistically tested using the NIST statistical test suite, which is a widely accepted standard across the industry.

6.1 Lessons Learnt

First and foremost, as a postgraduate student, this was a great opportunity for me to practice the knowledge and skills on various subject areas, I have gathered during the academic tenure for the masters and even the bachelors tenure. It was quite interesting on discovering various practical aspects of the theoretical knowledge gathered. More specifically, exposure to this research enabled me to closely investigate an important aspect of cryptography and many other different applications, and attempt to investigate the feasibility of a different approach to solve the said problem. Also, I could expose myself for certain important libraries, tools and techniques on analysing and summarising relatively large sets of data and utilising these summaries in various decision making.

The knowledge and practical exposure I have gained in conducting a scientific investigation on determining the success of a proposed solution for a given problem, is also worth noting about. It was a monumental contributor to broaden my knowledge envelop on scientifically analyse and investigate a proposed solution and determine is success, failure and determine probes for further research.

Apart from that, this was also a great opportunity to understand and practice the important aspects of systematically presenting and communicating the findings of an academic research. it was possible to get exposed to a variety of important aspects including, but not limited to academic writing styles, reviewing existing literature, determining methodologies of analysis, evaluation and conclusion, documenting the research project within the academic standards and regulations and so forth. I beseech that this exposure will be a prominent guideline in further academic studies.

6.2 Critical Evaluation

Initially it was intended to explore and identify the possible sources of randomness within the system of a personal computing device and its surroundings. For this, the system environment and the surrounding was monitored. Monitoring external environment was restricted only to the network interfaces, due to the requirement of additional expensive and less portable hardware that would otherwise violate the constraints of cost and portability. Under this investigation, it was able to determine and establish that certain CPU variables such as hardware and software interrupt counts and context switches vary in a manner which is non deterministic external to the system. This was the same basis of the Linux random generator. However, only a limited set of variables were analysed and the analysis on the external data sources were abandoned due to the excessive volume. These could be analysed in future developments.

Next objective was to identify and establish transformation strategies which is identified as *Distillation*, which are required to improve the complexity and volume. This objective resulted in the design and implementation of a transformation algorithm known as `FloatRAND`. This is more of a bit generation algorithm based on a pair of random seeds. Also it was set as an objective to identify and establish the encryption strategies which are required to meet the cryptographic security requirements of the said bit string. This was determined to be an optional stage and the whole purpose of this step is to eliminate the One-to-One mapping between the generated bit string and the source seed. Purpose of this step is to provide an added layer of transparency, which will provide additional security.

Next goal was to identify evaluation strategies to assess the quality of the outputs and to assess the quality of the outputs based on the complexity and volume requirements in order to determine the performance of the system. This was conducted using a bench mark test with the use of the Statistical Test Suite for evaluating CSPRNGs, provided by NIST. The tests were conducted for two other widely used PRNG algorithms namely Mersenne Twister and the Xoroshiro128+, aside of the proposed algorithm. Tests were conducted on a number of samples from each algorithm and the pass rates were summarised. The summaries of the results were tallied using the mean and standard deviation of the pass rates and also the pass rates were visually depicted and compared for any visible observations. Further, some constraints of the proposed algorithm was also determined with the use of the same statistical test suite and visualisations of the variations of the random variables chosen.

Upon retrospect of the research outcomes against the objectives that were initially set, it is fair to conclude that the research study as intended has contributed to expand the horizons of the problem of random number generation. While it has supported to establish some of the set rules on random number generation, it also has enabled a new possible approach of generating numbers which are close to ideal randomness.

6.3 Future Work

In order to further develop this model, the following probes could be used.

- Find better random variables from the system environment - For the time being, the system uses only CPU metrics as the source of entropy. However, there could be other measures and sources that could be obtained from within the environment of the system. Feasibility of such could be assessed.

- Introduce security hardening to eliminate the mapping between the seed and the random string - Even though a suggestion to mitigate this problem is already provided, the feasibility and the performance of such needs to be assessed. Alternatively, using a different approach to solve the problem also could be introduced, in the due course.

- Derive routines for parallel operations (such as seed slicing) - Another inherent weakness of this system is it currently operates only on serial mode. It is required to derive ways and means of utilising this algorithm to generate a block of bits at once, other than collecting in into a buffer. One alternative could be multiple generators with sliced seed.

- Test further to determine the limits - Apart from the currently known limits, this system needs to be further tested to determine if there are any vulnerabilities or weaknesses which would cause flaws in using this in the production state. Such limits and constraints could be investigated for, in a future work.

- Consider and evaluate the feasibility of using network interfaces as a source of entropy - Even though this was initially considered in the scope of the study, it has been moved to future work, due to the load of data. That investigation could also be suggested as a future probe.

Appendix A

Floating Point Converter - Python

```
class Float(object):
  def __init__(self, l, m, n):
    super(Float, self).__init__()

    n = str(n)
    n1 = n.split('.')

    # Resolution of sign and integer
    n = int(n1[0])
    n = '{0:b}'.format(n)
    n = n.split('-')
    s = 1 if not n[0] else 0
    n = n[1] if not n[0] else n[0]
    n = [int(e.encode('ascii')) for e in n]

    e = l - m - 1;

    # Resolution of exponent
    decPos = len(n) - 1


    if len(n) == 1 and n[0] == 0:
      decPos -= 1

    # Resolution of float
    f = n1[1]
    f = [int(e.encode('ascii')) for e in f]

    zero = [0] * len(f)

    sig = []

    decPosFound = decPos > -1

    while True:
      if f == zero:
        break

      [of, f] = self.__arrayTimes2(f)

      if not decPosFound:
        if of == 1:
          decPosFound = True
        else:
          decPos -= 1

      if len(sig) >= (m - decPos):
        break

      sig.append(of)

    n = n[1:] + sig[sig.index(1):]

    n += [0] * (m - len(n))
    decPos += ((2 ** e) >> 1) - 1
    exp = [int(e.encode('ascii')) for e in ('{0:b}'.format(decPos))]

    self.n = [s] + exp + n

  def __arrayTimes2(self, arr):
    of = 0
    for i in reversed(range(len(arr))):
      x = of + (arr[i] * 2)
      of = x // 10
      arr[i] = x % 10
    return [of, arr]
```

Listing 5: Python implementation of the `Float` class

Appendix B

CPU Metrics Collector Routine

```python
#!/usr/bin/python3

import os
import sys
import threading
import time
import psutil
import json
import cutil
import re

StartTime = None
fileNameStr = None
fileName = None
fSource = None


def grab():
  global fileNameStr
  global fSource

  fileName = str(fileNameStr) + ".txt"
  if fSource.closed:
    fSource = open(fileName, "a")

  if fileNameStr != str(cutil.epochHour()):
    fSource.close()
    print(fileName, " is closing...")
    fileNameStr = str(cutil.epochHour())
    fileName = fileNameStr + ".txt"
    fSource = open(fileName, "a")
    print("DONE! Started logging in a new file ", fileName)

  metrics = dict()

  # Retrieval of CPU stats
  res = psutil.cpu_stats()
  resType = type(res).__name__
  data = dict(res._asdict())
  data["temperature"] = psutil.sensors_temperatures(1)  # TODO: Fahrenheit?
  ↪   Or Celsius
  metrics[resType] = data

  # Retrieval of Memory Stats
  res = psutil.virtual_memory()
  resType = type(res).__name__
  data = dict(res._asdict())
  metrics[resType] = data

  # Retrieval of Disk Usage Stats
  res = psutil.disk_usage("/")
  resType = type(res).__name__
  data = dict(res._asdict())
  metrics[resType] = data
```

```
    fSource.write(json.dumps(metrics) + "\n")


def getConnections():
    for con in psutil.net_connections():
        try:
            yield [psutil.Process(pid=con.pid).name(), con]
        except psutil.AccessDenied:
            continue


def toSeconds(timeStr):
    if re.match(r'^[\d]+[hHmMsS]$', timeStr, re.M | re.I) is None:
        raise ValueError("Invalid input string.")
    timeUnit = (timeStr[-1])
    timeAmt = int(timeStr.replace(timeUnit, ""))
    timeUnit = timeUnit.lower()

    seconds = {
        'h': lambda timeAmt: timeAmt * 60 * 60,
        'm': lambda timeAmt: timeAmt * 60,
        's': lambda timeAmt: timeAmt
    }[timeUnit](timeAmt)

    return seconds


class SetInterval:
    def __init__(self, interval, action):
        self.interval = interval
        self.action = action
        self.stopEvent = threading.Event()
        thread = threading.Thread(target=self.__setInterval)
        thread.start()

    def __setInterval(self):
        nextTime = time.time() + self.interval
        while not self.stopEvent.wait(nextTime - time.time()):
            nextTime += self.interval
            self.action()

    def cancel(self):
        self.stopEvent.set()


def main():
    global StartTime
    global fileNameStr
    global fileName
    global fSource

    interval = int(sys.argv[1])
    cycles = toSeconds(sys.argv[2])

    StartTime = time.time()
    fileNameStr = str(cutil.epochHour())
    fileName = fileNameStr + ".txt"
    fSource = open(fileName, "a")

    # start action every {interval} seconds
    inter = SetInterval(interval, grab)
    print('Commenced reading datasources : {:.1f}s'.format(time.time()))

    # will stop interval in {cycles} seconds
    t = threading.Timer(cycles, inter.cancel)
    t.start()
```

```
if __name__ == '__main__':
  try:
    main()
  except (KeyboardInterrupt, SystemExit):
    print('Shutting down...')
    try:
      sys.exit(0)
    except SystemExit:
      os._exit(0)

  except IndexError as e:
    print("USAGE:python grab.py <<Interval>> <<Duration in H|M|S>>")
```

Listing 6: Python implementation of the CPU Data capturing routine

Appendix C

CPU Metrics Data Insertion Routine

```python
#!/usr/bin/python3
from pymongo import MongoClient
import os
from os import listdir
from os.path import isfile, join
import json
from loggin import log

log("Connecting to Database...")
client = MongoClient('mongodb://127.0.0.1:27017/', connect=False)
db = client.db_mis_research
# col = db.internalMetricsIdle
# col = db.internalMetricsWorking

txtDir = './txt'

fileNames = [f for f in listdir(txtDir) if isfile(join(txtDir, f))]

for filename in fileNames:
  epochHour = int((filename.split(".txt"))[0])
  path = '{}/{}'.format(txtDir, filename)
  with open(path) as fp:
    line = fp.readline()
    cnt = 0
    while line:
      docId = epochHour + cnt
      extObj = col.find_one({'_id':docId})

      obj = json.loads(line.strip())

      if extObj != None:
        dupLog = open('./logs/dupes/{}.dup.log'.format(epochHour), 'a+')
        logEntry = '{{"OBJ":{}, "EXT_OBJ":{}}}\n'.format(json.dumps(obj),
         ↪  json.dumps(extObj))
        dupLog.write(logEntry)
        dupLog.close()

        log('Duplicate entry found for {} when processing {}\n', docId,
         ↪  path)
        cnt += 1
        continue
        # os._exit(0)

      obj['_id'] = docId
      oId = col.insert_one(obj).inserted_id
      # print('Processed {}. Created Document with ID {}'.format(path,
       ↪  oId))
      line = fp.readline()
      cnt += 1
    log("{}::DONE! Processed {} documents!\n", path, cnt)

log("DONE!")
```

Listing 7: Python implementation of data insertion to a MongoDB database

Appendix D

Sequence Differentiation Routine

```
#!/usr/bin/python3

import multiprocessing
from multiprocessing import Pool
from pymongo import MongoClient
import os
from os import listdir
from os.path import isfile, join
import json
import matplotlib.pyplot as plt
import threading
import datetime
from gridfs import GridFS
import numpy
from loggin import log

def nowStr():
  return str(datetime.datetime.now())

def now():
  return datetime.datetime.now()

def f(x):
  return x["scpustats"]["interrupts"]

client = MongoClient('mongodb://127.0.0.1:27017/', connect=False)
db = client.db_mis_research
fs = GridFS(db)

col = db.internalMetricsIdle


project = {'scpustats.interrupts':1, 'scpustats.soft_interrupts':1,
 ↪  'scpustats.syscalls':1, 'scpustats.ctx_switches':1}

log('Commenced reading databases...')

res = col.find({}, project)
scpustatsList = list(res)

log("Extraction to a list of dictionaries is completed. Starting collecting
 ↪  interrupts counts...")

interruptsCounts = []

def getInterrupts(x):
  return x["scpustats"]["interrupts"]

def getSoftInterrupts(x):
  return x["scpustats"]["soft_interrupts"]

def getRatios(x):
  intrs = x["scpustats"]["interrupts"]
  softIntrs = x["scpustats"]["soft_interrupts"]
```

```
    return intrs / softIntrs

  with Pool(5) as p:
    interruptsCounts = p.map(getInterrupts, scpustatsList)
    softInterrupCounts = p.map(getSoftInterrupts, scpustatsList)
    ratios = p.map(getRatios, scpustatsList)

    log("Extraction of interrupts counts is complete.")

    # Interrupts
    y = list(numpy.diff(interruptsCounts))
    x = list(range(len(y)))

    oId = "PLT_ADJ_INTERRUPTS_DIFFERENCES_IDLE"

    colPlots = db.plots
    fileId = fs.put(str([x, y]), encoding='utf-8')
    log('Inserted plot file data into "db_mis_research.fs.files" with ID {}',
     ↪  fileId)

    xLabel = 'Time'
    yLabel = 'Difference Between Adjacent Interrupts Counts'
    title = 'Variation of Differences of Interrupt Counts'

    obj = {"_id": oId, "name": "Difference Between Adjacent Interrupts
     ↪  Counts", "fileId": fileId, "xLabel": xLabel, "yLabel": yLabel,
     ↪  "title": title}

    oId = colPlots.insert_one(obj).inserted_id
    log('Inserted plot metadata into "db_mis_research.plots" with ID {}',
     ↪  oId)

    # Soft Interrupts
    y = list(numpy.diff(softInterrupCounts))
    x = list(range(len(y)))

    oId = "PLT_ADJ_SOFT_INTERRUPTS_DIFFERENCES_IDLE"

    colPlots = db.plots
    fileId = fs.put(str([x, y]), encoding='utf-8')
    log('Inserted plot file data into "db_mis_research.fs.files" with ID {}',
     ↪  fileId)

    xLabel = 'Time'
    yLabel = 'Difference Between Adjacent Soft Interrupts Counts'
    title = 'Variation of Differences of Soft Interrupt Counts'

    obj = {"_id": oId, "name": "Difference Between Adjacent Soft Interrupts
     ↪  Counts", "fileId": fileId, "xLabel": xLabel, "yLabel": yLabel,
     ↪  "title": title}

    oId = colPlots.insert_one(obj).inserted_id
    log('Inserted plot metadata into "db_mis_research.plots" with ID {}',
     ↪  oId)

    # Interrupts / Soft Interrupts
    y = list(numpy.diff(ratios))
    x = list(range(len(y)))

    oId = "PLT_ADJ_INTERRUPTS_RATIOS_DIFFERENCES_IDLE"

    colPlots = db.plots
    fileId = fs.put(str([x, y]), encoding='utf-8')
    log('Inserted plot file data into "db_mis_research.fs.files" with ID {}',
     ↪  fileId)

    xLabel = 'Time'
    yLabel = 'Difference Between Adjacent Ratios of Intterupts/Soft
     ↪  Interrupts Counts'
```

```
title = 'Variation of Differences of Ratios of Intterupts/Soft Interrupts
 ↪   Counts'

obj = {"_id": oId, "name": "Difference Between Adjacent Soft Interrupts
 ↪   Counts", "fileId": fileId, "xLabel": xLabel, "yLabel": yLabel,
 ↪   "title": title}

oId = colPlots.insert_one(obj).inserted_id
log('Inserted plot metadata into "db_mis_research.plots" with ID {}',
 ↪   oId)

client.close();

log('Display of plot closed! COMPLETE!')
```

Listing 8: Python implementation of the routine to differentiate the sequence data

Appendix E

View Plot Routine

This routine was used to generate the plots of the linear difference data on CPU metrics.

```python
#!/usr/bin/python3

from pymongo import MongoClient
import os
import matplotlib.pyplot as plt
import datetime
from gridfs import GridFS
from bson import ObjectId
import ast
import sys

def now():
  return str(datetime.datetime.now())

def log(message, *args):
  message = message.format(*args)
  print('{}: {}'.format(now(), message))

def main():
  log("Connecting to DB...")
  client = MongoClient('mongodb://127.0.0.1:27017/', connect=False)
  db = client.db_mis_research
  fs = GridFS(db)
  plots = db.plots

  pltId = sys.argv[1]#"PLT_ADJ_INTERRUPTS_DIFFERENCES"
  plot = plots.find_one({'_id':pltId})
  fileId = plot['fileId']

  file = fs.get(ObjectId(fileId)).read().decode('utf-8')
  log("CONNECTED! Files found and retrieved!")

  obj = ast.literal_eval(file)

  try:
    offset = int(sys.argv[2])
  except IndexError:
    offset = 0

  try:
    limit = int(sys.argv[3])
  except IndexError:
    limit = len(obj[0])


  plt.plot(obj[0][offset:(offset + limit)], obj[1][offset:(offset +
   ↪  limit)])

  plt.xlabel(plot['xLabel'])
  plt.ylabel(plot['yLabel'])

  plt.title(plot['title'])
  plt.show()
```

```
    client.close()

  if __name__ == '__main__':
    try:
      main()
    except (KeyboardInterrupt, SystemExit):
      print('Shutting down...')
      try:
        sys.exit(0)
      except SystemExit:
        os._exit(0)

    except IndexError as e:
      print("USAGE:view_plot.py <<PlotID>> [<<int offset>> [<<int limit>>]]")

    except TypeError as e:
      print("{} was not found in the DB.".format(sys.argv[1]))
```

Listing 9: Python implementation for plotting linear differences of CPU metrics stored in the database

Appendix F

Sample of NIST Test Summary - Initial (S0001)

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <randbits.txt>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  0   0   0   0   0   0   0   0   0   1    ----      1/1       Frequency
  0   0   0   0   0   0   0   0   0   1    ----      1/1       BlockFrequency
  0   0   0   0   0   0   0   0   0   1    ----      1/1       CumulativeSums
  0   0   0   0   0   0   0   0   0   1    ----      1/1       CumulativeSums
  0   0   0   0   0   0   0   0   0   1    ----      1/1       Runs
  1   0   0   0   0   0   0   0   0   0    ----      0/1       LongestRun
  1   0   0   0   0   0   0   0   0   0    ----      0/1       Rank
  1   0   0   0   0   0   0   0   0   0    ----      0/1       FFT
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      0/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----     0/1      NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     NonOverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     OverlappingTemplate
  1  0  0  0  0  0  0  0  0  0   ----     0/1     Universal
  1  0  0  0  0  0  0  0  0  0   ----     0/1     ApproximateEntropy
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursions
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  1   ----     1/1     RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  0  0  0  0  0  0  0  0  0  0   ----   ------   RandomExcursionsVariant
  1  0  0  0  0  0  0  0  0  0   ----     0/1     Serial
  1  0  0  0  0  0  0  0  0  0   ----     0/1     Serial
  1  0  0  0  0  0  0  0  0  0   ----     0/1     LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 0 for a
```

```
sample size = 1 binary sequences.

The minimum pass rate for the random excursion (variant) test is undefined.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Listing 10: Example of Test Summary - Initial Random String Test (Sample 0001)

Appendix G

Sample of NIST Test Summary - Initial (S0500)

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <randbits.txt>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  0   0   0   0   0   0   1   0   0   0     ----      1/1      Frequency
  0   0   0   0   0   0   0   0   1   0     ----      1/1      BlockFrequency
  0   0   0   0   0   1   0   0   0   0     ----      1/1      CumulativeSums
  0   0   0   0   0   0   1   0   0   0     ----      1/1      CumulativeSums
  0   0   0   0   0   0   1   0   0   0     ----      1/1      Runs
  0   0   0   1   0   0   0   0   0   0     ----      1/1      LongestRun
  0   0   0   0   0   1   0   0   0   0     ----      1/1      Rank
  0   0   0   0   0   0   1   0   0   0     ----      1/1      FFT
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      0/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       0/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1    ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0    ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0    ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0    ----      1/1      OverlappingTemplate
  0   0   0   1   0   0   0   0   0   0    ----      1/1      Universal
  0   0   0   0   0   0   0   0   0   1    ----      1/1      ApproximateEntropy
  0   0   0   0   0   0   0   0   1   0    ----      1/1      RandomExcursions
  0   0   0   0   0   0   0   1   0   0    ----      1/1      RandomExcursions
  0   0   0   1   0   0   0   0   0   0    ----      1/1      RandomExcursions
  0   0   0   0   0   0   1   0   0   0    ----      1/1      RandomExcursions
  1   0   0   0   0   0   0   0   0   0    ----      0/1      RandomExcursions
  0   0   1   0   0   0   0   0   0   0    ----      1/1      RandomExcursions
  0   0   0   0   1   0   0   0   0   0    ----      1/1      RandomExcursions
  0   0   0   0   0   0   1   0   0   0    ----      1/1      RandomExcursions
  0   0   0   1   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  0   0   1   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  0   0   0   0   0   0   1   0   0   0    ----      1/1      RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   1    ----      1/1      RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   1    ----      1/1      RandomExcursionsVariant
  0   0   0   0   0   0   0   1   0   0    ----      1/1      RandomExcursionsVariant
  0   0   0   0   0   0   0   0   1   0    ----      1/1      RandomExcursionsVariant
  0   0   0   0   0   0   1   0   0   0    ----      1/1      RandomExcursionsVariant
  0   0   0   0   1   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  1   0   0   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  1   0   0   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  1   0   0   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  1   0   0   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  0   1   0   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  0   0   1   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  0   1   0   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  1   0   0   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  0   0   1   0   0   0   0   0   0   0    ----      1/1      RandomExcursionsVariant
  0   0   0   1   0   0   0   0   0   0    ----      1/1      Serial
  0   0   0   1   0   0   0   0   0   0    ----      1/1      Serial
  0   0   1   0   0   0   0   0   0   0    ----      1/1      LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 0 for a
```

```
sample size = 1 binary sequences.

The minimum pass rate for the random excursion (variant) test
is approximately = 0 for a sample size = 1 binary sequences.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Listing 11: Example of Test Summary - Initial Random String Test (Sample 0500)

Appendix H

Sample of NIST Test Summary - IDLE

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <2090775_1536805.txt>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  0   0   0   0   0   0   0   1   0   0     ----       1/1       Frequency
  0   0   0   0   0   0   1   0   0   0     ----       1/1       BlockFrequency
  0   0   0   1   0   0   0   0   0   0     ----       1/1       CumulativeSums
  0   0   0   0   0   1   0   0   0   0     ----       1/1       CumulativeSums
  0   0   0   0   0   0   0   0   1   0     ----       1/1       Runs
  0   1   0   0   0   0   0   0   0   0     ----       1/1       LongestRun
  0   0   0   0   0   1   0   0   0   0     ----       1/1       Rank
  0   0   1   0   0   0   0   0   0   0     ----       1/1       FFT
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----       1/1       NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----       1/1       NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      0/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
```

```
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   1   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   1   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   0   1     ----      1/1      NonOverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   1   0   0     ----      1/1      NonOverlappingTemplate
  0   0   1   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   0   0   0   0   0   1   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  1   0   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   1   0   0   0   0   0   0   0   0     ----      1/1      NonOverlappingTemplate
  0   0   0   1   0   0   0   0   0   0     ----      1/1      OverlappingTemplate
  0   0   0   0   1   0   0   0   0   0     ----      1/1      Universal
  0   0   0   0   0   0   0   0   1   0     ----      1/1      ApproximateEntropy
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursions
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   0   0   0   0   0   0   0   0   0     ----    ------    RandomExcursionsVariant
  0   1   0   0   0   0   0   0   0   0     ----      1/1      Serial
  1   0   0   0   0   0   0   0   0   0     ----      1/1      Serial
  0   0   0   1   0   0   0   0   0   0     ----      1/1      LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 0 for a
sample size = 1 binary sequences.

The minimum pass rate for the random excursion (variant) test is undefined.
```

```
For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Listing 12: Example of Test Summary - IDLE System

Bibliography

[1] "Are the digits of pi random? researcher may hold the key." [Online]. Available: https://www2.lbl.gov/Science-Articles/Archive/pi-random.html

[2] "Epoch converter - unix timestamp converter." [Online]. Available: https://www.epochconverter.com/

[3] "List of random number generators." [Online]. Available: https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/List_of_random_number_generators.html

[4] "The lotto machines | howstuffworks." [Online]. Available: https://entertainment.howstuffworks.com/lottery3.htm

[5] "Non-deterministic algorithms." [Online]. Available: https://cs.nyu.edu/courses/spring03/G22.2560-001/nondet.html

[6] "random.c/char/drivers - kernel/git/torvalds/linux.git - linux kernel source tree." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/char/random.c

[7] "Randomness and mathematical proof by gregory j. chaitin." [Online]. Available: https://www.owlnet.rice.edu/~km9/Randomness\%20and\%20Mathematical.pdf

[8] "Random.org - introduction to randomness and random numbers." [Online]. Available: https://www.random.org/randomness/

[9] "urandom(4): kernel random number source devices - linux man page." [Online]. Available: https://linux.die.net/man/4/urandom

[10] *Feynman path integrals.* Springer Germany, 1979.

[11] S. Arora and B. Barak, *Computational Complexity: A Modern Approach.* Cambridge University Press, 20 April 2009.

[12] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management, part 1: General (revision 3)." [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-3/archive/2012-07-10

[13] E. Barker and J. Kelsey, "Recommendation for random number generation using deterministic random bit generators - revision 1." [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf

[14] W. Brown, “Random number generation in c++11.” [Online]. Available: https://isocpp.org/files/papers/n3551.pdf

[15] “Random|meaning in the cambridge english dictionary,” Cambridge University Press. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/random

[16] T. Cusick and P. Stanica, *Cryptographic Boolean Functions and Applications (Second edition).* Academic Press, 2017.

[17] D. Fleisch and J. Kregenow, *A Student’s Guide to the Mathematics of Astronomy*. Cambridge University Press, August 2013.

[18] R. W. Floyd, “Non-deterministic algorithms,” *Journal of the ACM*, vol. 14, pp. 636–644.

[19] M. Gallagher, “Random number generators in swift.” [Online]. Available: https://www.cocoawithlove.com/blog/2016/05/19/random-numbers.html

[20] T. Gowers, J. Barrow-Green, and I. Leaderk, *The Princeton Companion to Mathematics*. Princeton University Press, 18 July 2018.

[21] S. G. Krantz, *The Proof is in the Pudding: The Changing Nature of Mathematical Proof.* Springer Science and Business Media, 13 May 2011.

[22] “Random number generator algorithms - matlab randstream.list,” The Mathworks Inc. [Online]. Available: https://www.mathworks.com/help/matlab/ref/randstream.list.html

[23] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator,” *ACM Transactions on Modeling and Computer Simulation (TOMACS) - Special issue on uniform random number generation*, vol. 8, pp. 3–30.

[24] M. Matsumoto, T. Nishimura, M. Hagita, and M. Saito, “Cryptographic mersenne twister and fubuki stream/block cipher.” [Online]. Available: https://eprint.iacr.org/2005/165.pdf

[25] G. Merald, “On the accuracy of statistical procedures in microsoft excel 2010,” pp. 1095–1128.

[26] “curand :: Cuda toolkit documentation,” nVIDIA Developer Zone. [Online]. Available: https://docs.nvidia.com/cuda/curand/host-api-overview.html#generator-types

[27] “deterministic|definition of deterministic in english by oxford dictionaries,” Oxford University Press. [Online]. Available: https://en.oxforddictionaries.com/definition/deterministic

[28] "random|definition of random in english by oxford dictionaries," Oxford University Press. [Online]. Available: https://en.oxforddictionaries.com/definition/random

[29] "Php: mt_rand - manual," The PHP Groups. [Online]. Available: https://www.php.net/manual/en/function.mt-rand.php

[30] "8.6. random — generate pseudo-random numbers — python v3.2 documentation," Python Software Foundation. [Online]. Available: https://docs.python.org/release/3.2/library/random.html

[31] "9.6. random — generate pseudo-random numbers — python v2.6.8 documentation," Python Software Foundation. [Online]. Available: https://docs.python.org/release/2.6.8/library/random.html

[32] T. Ritter, "Randomness tests: A literature survey." [Online]. Available: http://www.ciphersbyritter.com/RES/RANDTEST.HTM

[33] M. Route, "Radio-flaring ultracool dwarf population synthesis," *The Astrophysical Journal*, p. 66.

[34] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, "A statistical test suite for random and pseudorandom number generators for cryptographic applications." [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf

[35] W. Schindler, "Functionality classes and evaluation methodology for deterministic random number generators," pp. 5–11. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_20_Functionality_Classes_Evaluation_Methodology_DRNG_e.pdf

[36] S. Vigna, "xoshiro / xoroshiro generators and the prng shootout." [Online]. Available: http://prng.di.unimi.it/xoroshiro128plus.c

[37] S. Vigna and D. Blackman, "Scrambled linear pseudorandom number generators." [Online]. Available: https://arxiv.org/abs/1805.01407