Validation of Pseudo Random Number Generators through Graphical Analysis

Submitted in partial fulfilment
of the requirements of the degree
Bachelor of Science (Honours)
in the Department of Computer Science
at Rhodes University

Andrew Cronwright
7th November 2005

Acknowledgements

I would like to thank my supervisor, Barry Irwin, for his support and enthusiasm throughout my project.
I must acknowledge the financial and technical support of this project of Telkom SA, Business Connexion, Comverse SA, and Verso Technologies through the Telkom Centre of Excellence at Rhodes University.

Abstract

In a many computer applications, pseudo random number generators (PRNGs) are used, their uses range from producing unique keys for security purposes to creating random samples for physics experiments. The use of a poor PRNG in these applications can lead to highly undesired results and/or consequences. This project will highlight how one can go about visually testing a PRNG, or any suspect random number sequence, to decide whether it is suitable or not. It will describe a graphical method of testing a PRNG which can draw attention to problems within a PRNG using methods that can be faster and easier than statistical testing. It will draw specific attention to the Transmission Control Protocol (TCP) Initial Sequence Number generator and the issues surrounding its level of randomness and need for randomness in terms of security.

Table of Contents

List of Figures 6
Chapter 1 Introduction 7
  1.1. Pseudo Random Number Generators defined 7
  1.2. Overview of this report 9
Chapter 2 Problem Domain and Methodology 11
  2.1. True random numbers 11
  2.2. Pseudo Random Numbers 12
  2.3. Current PRNGs 13
  2.4. Qualities a good PRNG must possess 15
  2.5. Selecting a Seed 17
  2.6. Testing a PRNG 17
  2.7. Summary 21
Chapter 3 Uses and Testing of PRNGs 22
  3.1. Applications of PRNG 22
    3.1.1. Cryptography 22
    3.1.2. Network Security 23
    3.1.3. Monte Carlo/Statistical (physical) simulations 27
  3.2. Testing of a PRNG 28
  3.3. Summary 31
Chapter 4 System Design and Implementation 32
  4.1. Randomness in three dimensions 32
  4.2. Colour, an extra dimension 36
  4.3. Construction and Testing of a hardware RNG 37
  4.4. Design features / considerations 39
  4.5. Implementation details 40
  4.6. Application 40
  4.7. Summary 42
Chapter 5 Results 43
  5.1. Operating Systems 43
    5.1.1. Windows 95 43
    5.1.2. Windows 98 45
    5.1.3. Windows 98 Second Edition (SE) 46
    5.1.4. Windows ME 46
    5.1.5. Windows 2003 Server 48
    5.1.6. Windows XP 49
    5.1.7. Cisco 54
    5.1.8. Fedora Core 3 56
    5.1.9. FreeBSD 57
    5.1.10. Solaris 59
    5.1.11. DSL 61
    5.1.12. VoIP Phone 62
  5.2. Hardware RNG Results 63
  5.3. Summary 67
Chapter 6 Conclusion 69
  6.1. Summary of Findings 69

6.2. Future Work 70
References 71
Appendix A 74
Appendix B 75

List of Figures

Figure 1 – IBM's RANDU 13
Figure 2 – Example of ISN attack 24
Figure 3 – Monte Carlo Estimate of Pi 28
Figure 4 – Phase Space control image 34
Figure 5 – Lattice Space control image 35
Figure 6 – Noise Sphere control image 35
Figure 7 – HSV colour mapping 36
Figure 8 – Noise Sphere control image with colour 36
Figure 9 – Voltage fluctuations on a resistor 38
Figure 10 - Circuit Diagram of hardware RNG 39
Figure 11 – Class diagram 40
Figure 12 - Application 42
Figure 13 – Windows 95 Phase Space 44
Figure 14 – Windows 95 Lattice Space 45
Figure 15 - Windows 98 Phase Space 46
Figure 16 - Windows ME Phase Space 47
Figure 17 - Windows ME Noise Sphere 47
Figure 18 - Windows 2003 Server Phase Space 48
Figure 19 - Windows 2003 Server Noise Sphere 48
Figure 20 - Windows XP original Phase Space 49
Figure 21 - Windows XP original Noise Sphere 50
Figure 22 - Windows XP SP1 Phase Space 50
Figure 23 - Windows XP SP1 Noise Sphere 51
Figure 24 - Windows XP SP2 Phase Space 52
Figure 25 - Windows XP SP2 Noise Sphere 52
Figure 26 - Window XP fully patched, Phase Space 53
Figure 27 - Window XP fully patched, Noise Sphere 53
Figure 28 – Cisco Phase Space 54
Figure 29 - Fedora Core 3 Phase Space 56
Figure 30 - Fedora Core 3 Phase Space 56
Figure 31 - FreeBSD 4.1 Phase Space 57
Figure 32 - FreeBSD 4.1 Lattice Space 57
Figure 33 - FreeBSD 4.1 Noise Sphere 58
Figure 34 - Solaris 5 Phase Space 59
Figure 35 - Solaris 9 Phase Space 60
Figure 36 - Solaris 9 Noise Sphere 60
Figure 37 - dslam Lattice Space 61
Figure 38 - dslam Noise Sphere 61
Figure 39 - VoIP phone Phase Space 62
Figure 40 – Hardware RNG Phase Space 65
Figure 41 – Hardware RNG Phase Space 66
Figure 42 – Hardware RNG Lattice Space 66
Figure 43 – Hardware RNG Noise Sphere 67

Chapter 1
Introduction

1.1. Pseudo Random Number Generators defined

The understanding of random numbers and how a number is defined to be random is essential to the understanding of this paper. In a statistical sense, there is no such thing a single random number. This is because it is impossible to perform any statistical tests on a single number. Instead it is better to define a sequence of numbers to be random, or speak of a number chosen from a random sequence of numbers. *[Knuth]*

A random number can be defined as a number from an independent set of numbers which is the output of a natural event and has a very high degree of uncertainty. The next number generated is completely independent from all previous numbers generated and is selected purely by chance. The set of numbers produced will have a given distribution *[Knuth]*. Here, it is assumed that all random sequences have a uniform distribution between zero, and one, excluding which is written as [0,1) mathematically. All possible numbers in this range has an equal probability of being selected. Randomness is easier understood as the outcome of some natural process or event. A true random number generator has an infinite period and given the same input parameters, will never produce the same output.
Secondly, a pseudo random number can be defined as a number from a set of numbers, which is the output of a mathematical function, which tries to "mimic" a true random number. This therefore makes them completely deterministic *[Jansson]*.

Since most pseudo random number generators (PRNGs) are of the form of a recursive mathematical functions (as is discussed in Chapter 2) each new *random* number generated is completely dependent on one or more previous numbers in the sequence. This contradicts one of the assumptions that every number in a random sequence is independent of all others. However, a well designed PRNG overcomes this shortfall by ensuring that the sequence produced "hides" this dependence. It does this by the chaotic nature of the generator and is discussed later in Chapter 2. Most PRNG's have a set period and given the same input parameters, will always produce the same sequence of numbers.

Pseudo random numbers are used in a wide variety of applications. In most cases where random numbers are needed in applications, such as security application, they are provided by a piece of software known as a pseudo random number generator, or PRNG. There are numerous implementations of PRNG's that are being used today, but some of them produce numbers are not statistically random or do not meet the needs of the parent application. This for example, can cause security applications, which rely on a good PRNG, to be compromised *[Schneier]*. Aside from security applications, if the PRNG is not capable of producing suitable random numbers, then this can result in poor performance of the parent application. Poor PRNG's have been an issue since the release of IBM's RANDU generator in the early 1960's. In its day, the RANDU generator was the PRNG of choice. Upon further investigation, this PRNG proved to be heavily flawed. It was designed to be computationally efficient and as a result RANDU was a flawed generator which had very non-random characteristics *[Entacher 2000]*.

Although statistical tests produce conclusive results that a sequence of numbers is, or is not random, they however do not give any idea as to how the sequence of numbers is, or is not random. Statistical testing is often a very specialised field in which a person needs to have a statistical background in order to understand why, or how a test is failing and to interpret the results. This is why a graphical method for viewing a sequence of numbers, described in Chapter 2, that may or may not be random, can be a quick and powerful method for testing whether a sequence of numbers is random. However, one must not completely discard the need for statistical testing.

This project's main focus is to discus and highlight the importance of PRNGs in computer communication. The Transmission Control Protocol (TCP) uses a unique identifying number, known as the Initial Sequence Number (ISN), which is selected at the beginning of every new TCP connection and identifies a single unique packet within in a series of packets making up a TCP connection *[RFC793]*. The ISN has, and can be an area of focus of a would be attacker due to the fact that ISNs have been easily predictable. If the next ISN can be predicted, an attacker can forge a packet and cause the receiver to either drop the TCP connection, or possibly take control of the connection and inject any data into a packet. Hijacking a connection is however fairly trivial if the hacker is directly in between, or has access to the traffic between the two communicating parties and can "sniff" the packets. If this can happen, the hijacker can immediately find out what the sequence number is and forge the packet.

This project will deliver a tool together with a method with which anyone can use (without extensive background knowledge in Statistics) to graphically evaluate sequences of numbers for patterns and/or randomness. Specific attention will be paid to common TCP/IP stacks and will result in an evaluation of these stacks using standard statistical tests and the tools developed in this project.

1.2. Overview of this report

Chapter 2 deals with and explains the concept of randomness. How randomness is obtained in a deterministic environment such as the computer. Common pseudo random number generators' (PRNG) methods are explained together with an introduction to the mathematics behind them. A graphical method of testing [pseudo] random sequences will be introduced and explained.

Chapter 3 shows and explains some of the areas where pseudo random number generators' are critical and what the effects are of using a poor PRNG are. Statistical tests used to test for randomness are explained as well.

Chapter 4 shows how and applies chapters two and three and discusses how randomness can be identified graphically. A design for a hardware random number generator is discussed and implemented. This chapter also discusses the design and implementation of the application developed for this paper. It shows the class diagram together with some implementation details. The application's GUI is also briefly discussed

Chapter 5 discusses the initial sequence numbers captured and graphically analyses them. The hardware random number generator's results are shown and tested together with some common PRNG's in use.

Chapter 6 concludes this paper with an explanation of the findings and concluding comments together with possible extensions to the work.

Chapter 2
Problem Domain and Methodology

2.1. True random numbers

The need for true random numbers is needed in many applications, this has resulted in specialized hardware being built to produce a sequence of true random numbers instead of using a software method of producing random like number sequences.

There are many natural occurring events that exhibit true random behaviour. The world of quantum physics is characterized by purely random events as well as natural decay of radioactive material are just two examples of many, truly random events. These processes, although very good sources of random data, are not practical for many computing needs. There are however more practical ways of generating true random data sequences.

One such method is by allowing a current to flow thought a resistor. Thermal agitation of free electrons causes small voltage fluctuations. It is this random noise that circuit designers try to minimise as much as possible *[Ghausi]*. If the amplitude of the voltage fluctuations across the resistor are made large enough to use practically, a true random number generator could be constructed *[Connor]*. This can be used, together with a comparator and a microprocessor to feed a sequence of "random" bits into the PC via the COM port. This is explained in detail in Chapter 4. This sequence of bits will of course have to be tested statistically to make sure that it does actually exhibit random properties. Another possible way to generate random data is to connect an antenna to an amplifier and measure the noise picked up. This is very similar to measuring the noise from a resistor, except that the sequence collected will have to be analyzed using Fourier analysis to check that there are no dominating frequencies in the data. This does however have its drawbacks, if the antenna is placed next to any device emitting an electro-magnetic field (which all electronic devices do), the data collected will be skewed and hence non-random. To try and reduce this effect, one could pass the random bits through a hashing function (MD5 or

SHA). Summing up all that has been said above; if a set of uniformly distributed numbers is selected in a truly random manner, then each possible number has an equal probability of being selected.

2.2. Pseudo Random Numbers

Obtaining random numbers from a physical source can often be impractical in many applications such as portable web applications. This led to the development of a mathematical method to create a sequence of numbers that could mimic true random numbers. Because a mathematical source is not a true source of random numbers, it is called a pseudo random number. This is due to the mathematical function being completely deterministic and hence, non-random. In the 1940's von Neuman developed the first mathematical algorithm to create random numbers. This was known as the middle-square method, and while it could produce seemingly random number sequences, it quickly proved to be a very poor source of pseudo random numbers. These methods of producing pseudo random numbers are known as pseudo random number generators or PRNG for short.

A key feature of a PRNG is that it should be chaotic. Even though they are completely deterministic mathematical functions, their output must be very erratic and non-predictable to an onlooker. Chaitin *[Chaitin]* and Kolmogorov have shown that to produce a random sequence by means of a computer program, then the program has to be approximately the same length as the random sequence itself. This therefore leads to the fact that a true random sequence can never be produced by a computer program. This is because a true random sequence of numbers can be infinite in length and never repeat themselves after a set period. PRNGs however do begin to repeat themselves after a set period which leads to the fact that infinite space is needed if one wants a computer program to produce a true random data sequence.

Hardware random number generators are commercially available which in some applications are essential. But in many applications issues due to availability, performance and especially portability, make the use of a hardware random number generator an impractical option. PRNGs also have the defect that in a set of uniformly distributed numbers selected by a PRNG, as is discussed above, each possible number does not have an equal probability of being selected. This is due to the process of generating pseudo random numbers is flawed by the method of using the previous result in generating the next number. Hence giving rise to its deterministic nature.

2.3. Current PRNGs

The first reliable PRNG algorithm was proposed by D. H. Lehmer in 1949, called the Linear Congruential Generator (or Linear Congruential Method, LCM). This method has been one of the most well known and widely used methods for generating random sequences. However, this method is not without flaws. It is well know that the sequence generated forms a lattice structure in 3-space. 3-space are three sets of coordinates (3-tuple) plotted against three sets of axis. This concept can be extended to 4,5… n-space. One of the most famous poor PRNG's was IBM's RANDU which did not have a full period and it had some extremely non-random characteristics *[Entacher 1998]*. Knuth described it as ``really horrible". *[Knuth pg. 173]*

Figure 1 shows what one of the problems with this PRNG was. All points fall onto a finite number of planes, which has been a problem of many subsequent PRNGs. This is a problem because a random sequence is supposed to be evenly distributed over a defined interval. Figure 1 clearly shows that the sequence is not evenly distributed over the space it lies in.

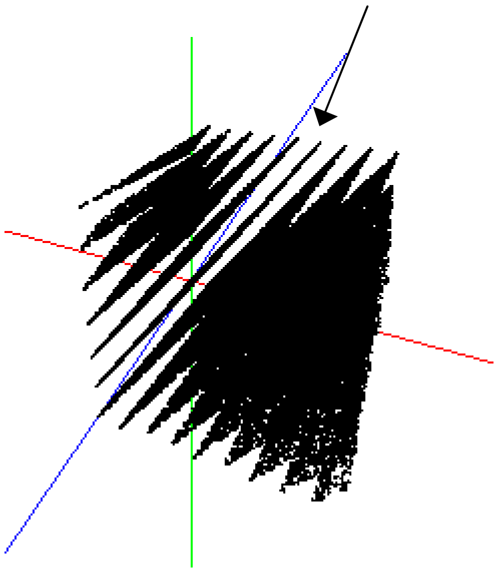


Figure 1 – IBM's RANDU

"It is well-known that all linear congruent generators (LGC) suffer from the inherent flaw that, in 3-space for example, the points (Z_i, Z_{i+1}, Z_{i+2}), (Z_{i+1}, Z_{i+2}, Z_{i+3}), (Z_{i+2}, Z_{i+3}, Z_{i+3}) all fall on a finite – and possibly small – number of parallel (hyper)planes." *[Park pg. 1197]*. The LCG is of the form of a recursive method and is as follows:

$$x_{n+1} = (a.x_n + c) \bmod m$$

where *m* is the modulus; *a* is the multiplier; *c* is the increment and X_0 is the starting value, or seed. It has also been shown that it is possible, by observing some of the output from an LCG, to recover all the parameters of an LCG in polynomial time. This makes an LCG PRNG unsuitable for any security applications *[Bellare]*.
If a, c and m are chosen correctly, it is possible for the generator to have a maximum period of length m *[Knuth]*. Many other types of PRNGs are based on the LCM as will be shown.

Lagged Fibonacci Generators

$X_i = (a_1X_{i-1} + \dots + a_rX_{i-r}) \bmod m$ <u>or</u>

$X_i = (X_{i-r} \oplus X_{i-s}) \bmod m$

Where, $\oplus$ stands short for one of the binary operations + (addition), - (subtraction), x (multiply), or the exclusive-or operation (XOR)
This is essentially the same as the LCG except that it has been extended to deal with a longer "lag", different types of operators and also more terms.

Shift Register Generators

$X_i = X_{i-p} \oplus X_{i-q}$

R250 generator uses: $X_i = Xi_{-250} . X_{i-103}$

$\oplus$ is the XOR function.

A very popular shift register generator is the Mersenne Twister, developed in 1997 by Makota Matsumoto and Takuji Nishimura from Keio University, this algorithm produces a sequence of $2^{19937} - 1$ numbers and has 623-dimensional equidistribution (compared to the 5-dimensional equidistribution of LCM generators). The name, Mersenne Twister, is actually derived from the fact that it has a Mersenne-prime period. A Mersenne-prime is a prime number defined as $(2^n – 1)$. This generator is not suitable for security applications because it is possible to analyze the output and recognize the numbers as being non-random, or reconstruct the internal state of the PRNG. The developers of this algorithm do however advise that a secure hashing function be used with the output or a simple linear transformation to help get around this issue. It is, however, a very good PRNG for other applications such as Monte Carlo simulations due to its good statistical properties. This PRNG is fast becoming the PRNG of choice for such application. *[Matsumoto]*

As stated earlier, all numbers produced by a PRNG are dependent on previous numbers produced, hence the fact that they are deterministic. The LCM and similar type PRNGs (attempt to) hide this shortfall by using the *mod* operator. If one obtained a number from a pseudo random sequence and also knew the inner workings of one of these generators, it would still be almost impossible to calculate previous numbers in the sequence. This is because the *mod* operator disregards some of the seeding value which results in the chaotic nature of the PRNG.

For example given a seed value, $x_n = 1234$, $a = 10^5$, $c = 0$, and $m = 10^6$. By using the LCM method: $x_{n+1} = (a.x_n + c) \bmod m$, we are left with a result of 400 000. As one can see, we are left with a result that has “lost” some of its data. This is the reason why most PRNG’s either use the mod operator, or one of the binary logic operators. There are of course brute force methods to recover the seed *[Ferguson]*, but with a good choice of seed, a, c and m values, this task can be made tedious. It is this feature that makes PRNG’s chaotic.

2.4. Qualities a good PRNG must possess

A good PRNG can be designed and built following a few simple guide lines. This however does not make the actual task of designed such an algorithm a menial task.
These six characteristics are as follows:

- Since processing power is not limited to the extent as it was a few decades ago, PRNG algorithms must still be short and efficient. This will allow pseudo random numbers to be generated in only a few clock cycles to allow the processor to continue with the main calling function or program *[Jansson, Atreya]*.

- Since PRNG's are of the form of a mathematical function, it is noted that they will, at some stage, begin to repeat themselves *[Park]*. It is this period of a PRNG that must be as long as possible *[Jansson, Atreya]*.
- There are statistical tests in use that can test the possibility of randomness with high levels of accuracy. The sequence produced from a PRNG should be checked against these tests, and pass them *[Jansson]*.
- By analysing the outputs of a PRNG, it should not be possible to predict the next number that will be generated *[Atreya]*.
- A random number sequence, in its binary representation, must have, on average, an equal amount of 1's and 0's. Furthermore, there must be no noticeable patterns in the bit string. *[Atreya]*.
- A PRNG must be seeded with a value, and given the same value, the same sequence of numbers must be produced (this is especially important for Monte Carlo simulations discussed later). For systems where the PRNG must behave in a more random manner, the seed must not be known or must not be able to be calculated. In this aspect, it is important that the seed contain a high level on entropy. "Entropy is the measure of uncertainty" … "The more entropy we have in an event, the more random the event is expected to be. Keys used for encryption are expected to be extremely random in nature and thus have high entropy levels" *[Atreya pg. 5]*. This is why many of the cryptographic PRNGs rely on many techniques to create a seed value from known random events on a computer system (interrupts, key presses, etc.).

2.5. Selecting a Seed

Selecting a seed number for a PRNG can be a very important process in the correct operation of a PRNG. In some applications it is acceptable to use a predefined seeding value or, for example the time of day. In security applications, this method of seeding a PRNG is not so simple. If the seeding value can be discovered, the entire security of an application can be broken *[Ferguson]*. A good seeding value is one which can not be calculated or discovered, the only possible way a secure seeding value should be able to be discovered, is by using a brute force approach. Since the seeding value is often a 32 bit integer, this makes a brute force approach a tedious affair, if the generator has a long period. Also, if the PRNG is given a new seed value at regular intervals, this can increase the security many fold *[Atreya]*.

But how is a good seeding value chosen? A good secure seed should contain a high degree of entropy. This means that the seed should essentially be a true random number. This can prove to be a difficult affair for a computer. But since a computer is exposed to a "random" external environment (a lot of unpredictability, or entropy exists in the environment), in the form of a network, a user etc., creating a true random seed can possibly be performed. Certain PRNGs have what is known as an entropy pool. This is essentially a program that monitors these random, external sources and pools the data. When a seed is needed, the PRNG obtains one from the entropy pool. The external sources can be:

1) The time intervals between keystroke of the user.
2) Time intervals between network traffic received.
3) The sound card can be used as a source of entropy.

These are just a few of the possible sources; the more of these sources used concurrently, the more entropy the seed will contain *[Kelsey]*.

2.6. Testing a PRNG

The need for testing a PRNG makes it possible to be able to conclusively say that a specific PRNG is good or bad in terms of cryptographic strength and / or statistical strength. Many statistical tests have been developed to test sequences of random numbers. One of the most useful methods is the Chi-square test of independence

[Knuth]. The Chi-square test can be used to test the dependence of a given sequence of numbers. This is especially important in this case as random numbers, by definition, are supposed to be completely independent. Thus, if the Chi-square test returns a result saying that a given sequence of numbers is dependent, one can confidently say that the sequence is not random.

The spectral test, formulated by Coveyou and MacPherson (1967), is probably one of the best and well known tests for LCM PRNGs. The pseudorandom sequences produced by these generators, when plotted in 3-space using (Z_i, Z_{i+1}, Z_{i+2}), $(Z_{i+1}, Z_{i+2}, Z_{i+3})$, $(Z_{i+2}, Z_{i+3}, Z_{i+3})$… as coordinates for the points, all fall on a finite number of parallel hyperplanes. The spectral test finds the maximal distance d_s between adjacent hyperplanes. This method of viewing sequences has consequently been implemented in the visualisation tool developed for this project. This test can only be applied to LCM generators and thus provides no direct comparison to any other PRNG using other techniques. Knuth describes this test as being especially significant because all known good LCM PRNGs pass it, but even more significant because all known bad LCM PRNGs fail the test *[Golder]*.

This is why statistical tests must be used to test the output of various PRNG so that a direct comparison can be made. The National Institute of Science and Technology (NIST) provide a statistical test suite for testing PRNG for cryptographic applications. This test suite runs various, intensive tests, on the output of PRNG, in bit array form. A full description of the NIST test suite is discussed later in Chapter 3.

The human eye is the most advanced pattern recognition "device". It is therefore obvious to use a method to plot pseudo random data on a set of axis. If one knows what a good known data source looks like, then this can then be used as a reference for other pseudo-random numbers in question *[Chambers]*. There are three such methods that will be implemented to plot random/pseudo-random data. One such method is the one that Michael Zalewski used in his paper titled "Strange Attractors and TCP/IP Sequence Number Analysis". This method transforms 1-dimensional data sequences into 3-dimensional data to be visually analysed:

If *s[n]* is a set of random numbers and *n* represents the nth random number in *s*, then the transformation to 3-space is as follows.

X[n] = s[n-2] – s[n-3]
Y[n] = s[n-1] – s[n - 2]
Z[n] = s[n] – s[n-1]

Equation 1 – Phase Space

This technique is known as delayed coordinates and can be explained as a comb being passed through a set of numbers to pick out, or identify, any patterns in the data. This can be extended to as many dimensions as one likes also known as n-dimensional space. Three dimensions are used so that it is easy to plot on set of three axes. An extra dimension is possible in the form of displaying time data. This will show how the sequence is being generated and whether there is some pattern in the generation process. Also, by modifying the lag it is possible to find dependencies in the data that is not immediately obvious *[Hoglund]*.

Another very similar method as the one above is to check how dependent a number in a sequence is from its predecessor *[Bowman]*.

If *s[n]* is a set of random numbers and *n* represents the nth random number in *s*, then the transformation to 3-space is as follows.

X[n] = s[n]
Y[n] = s[n-1]
Z[n] = s[n-2]

Equation 2 – Lattice Space

This can, like the above method, be extended to as many dimension as one likes *[Atkinson]*. Plotting sequences using this method allows one to visualise lattice structures, if there are any, in the sequence. Lattice structures will be a result of any dependencies in the data. If the numbers in the sequence are completely independent of each other, then one can expect all the points to lie in no apparent order in a three dimensional cube. The above methods have all been plotting numbers in Cartesian coordinates. Cartesian coordinates is a coordinate system in which each axis in perpendicular to each other.

The third method which will be used to plot a sequence of numbers will be using spherical coordinates *[Pickover]*. Spherical coordinates is a coordinate system that describes r, the radius, or the distance from to origin to a point, φ, the polar angle or the angle of separation from the z-axis and θ to be the angle of separation in the x-y plane from the x-axis. This method, much like the method above can be used to highlight problems of independence of a pseudo-random number sequence. Instead of mapping the above 3-Tuple method to Cartesian coordinates, it is mapped to spherical coordinates using the following transformation:

If *s[n]* is a set of random numbers and *n* represents the nth random number in *s*, then the transformation to 3-space is as follows.

$$\theta[n] = 2 * PI * s[n-2]$$
$$\varphi[n] = PI * s[n-1]$$
$$r[n] = \sqrt{(\, s[n] \,)}$$

$$X[n] = r * Cos(\theta) * Sin(\varphi)$$
$$Y[n] = r * Sin(\theta) * Sin(\varphi)$$
$$Z[n] = r * Cos(\varphi)$$

Equation 3 – Noise Sphere

Equation 3, also known as a Noise Sphere, is very similar to the Lattice Space, so similar information will be displayed. It might however highlight some flaw or dependence that is not easily visible using the Lattice Space. If a function is increasing by using linear increments, the Lattice Space will often result in a straight line in 3-space. This is particularly important in the results section of this paper. The Noise Sphere however, will show that random increments are being used as the points will appear to be more randomly distributed. This is because the transformation is much more volatile to minor changes compared to the Lattice Space transformation because it is a linear transformation. The Noise Sphere transformation is a non-linear transformation because of the *cosine* and *sine* functions.

The reason for using these graphical methods is to provide a very quick mechanism to test random sequences. This can however be extended to testing many suspect random events in general. For example, the results of lotto numbers are supposed to be random. These results can be displayed graphically, and if any attractors appear in the image, then it is possible to say that there is a pattern in the results. An attractor can be defined as a set of points in the phase space which attract all other points in that space [*Eric W. Weisstein*, http://www.mathworld.wolfram.com/Attractor.html]. These graphical methods are not meant to replace statistical testing; they merely provide a different method of quickly testing for randomness. Statistical tests should always be used when objective results are required. Any results obtained using these graphical methods alone can be very subjective and open to interpretation.

2.7. Summary

This chapter has described true randomness and how a computer can generate random-like numbers. Methods of generating pseudo random numbers together with the issues surrounding the construction of a PRNG have been discussed. Three visual methods of testing "random" (or sequences of numbers) numbers in three space were introduced and explained.

The following chapter will introduce and explain the importance of PRNG's and some applications of PRNG's. The testing of random numbers will also be explained

Chapter 3
Uses and Testing of PRNGs

3.1. Applications of PRNG

3.1.1. Cryptography

This area of computer science is probably the most critical when it comes to the need of good random number sources. A poor source of predictable pseudo random numbers would lead to the break down of the entire cryptographic process, leaving ones encrypted data open to any malicious activity. In cryptographic terms, it is important that the PRNG produces pseudo-random numbers that are statistically indistinguishable from true random numbers. In many other applications it is not too critical if it is found out exactly how a sequence was generated. But in cryptography special care must be taken at all stages of the sequence generation process.

Issues such a seeding, reseeding, entropy and generation process must all be considered and dealt with *[Kelsey]*. In the design of the Yarrow cryptographic PRNG all these issues were dealt with resulting in an extremely safe cryptographic PRNG. Yarrow uses a very “intelligent” system for generating seeds whereby it has a system monitor that captures data with known levels of entropy. This can include keyboard events, mouse movement events and even “into using the random fluctuations in hard-disk access time caused by turbulence inside the enclosure” *[Ferguson]*.

Another PRNG that has very good cryptographic properties is the Blum-Blum-Shub generator. This generator is very similar to the LCG PRNG, except it is of the form:

$$\boxed{x_{n+1} = x_n^2 \bmod M}$$

Equation 4 – Blum-Blum-Shub

where M is the product of two large distinct prime numbers. The output is the least significant bit (or parity) of x_{n+1}. What makes this PRNG so cryptographically strong is the fact that its parameters are not polynomial time computable. It has been shown that trying to crack this PRNG is as hard as trying to break RSA public-key encryption *[Blum]*.

In 1995 the security offered in the Netscape Navigator web browser was compromised. The success of this attack was due to the fact that it was aimed at the PRNG *[Schneier]*. More specifically it was aimed at the algorithm used to seed the PRNG. The algorithm implemented to create a seed value depended on three values, namely: the time of day, the process ID, and the parent process ID. These values can be easily predicted by an adversary. Once the seed was known, assuming the attacker knew the PRNG algorithm (which he/she did), the internal state of the PRNG could be recreated making the attack fairly trivial *[Dobb's]*.

3.1.2. Network Security

Over the recent few years network security has become one of the most important areas of focus of a network administrators. If the network security can be broken, an intruder can obtain sensitive information or perform malicious activities. The implementation of Transmission Control Protocol (TCP) requires that an Initial Sequence Number (ISN) be selected at the beginning of every new TCP connection. The ISN must also be unique as to avoid an overlap of connections which would result in packets of data being reassembled in the wrong order *[CIAC]*. During a connection, if multiple packets are sent, the original ISN is incremented sequentially.

The original implementation of TCP used a simple linear method of generating ISN's. A 32 bit clock was used which was incremented every four microseconds. The value of this clock was then used as an ISN when needed. This led to ISN's to begin repeating after only 4.55 hours *[RFC 793]*. A security flaw was discovered in the TCP/IP protocol suite, initially using 4.2BSD Unix, that if an intruder could cause a packet to be dropped, sent by a host machine to a connecting machine and the intruder could forge a return packet in which the correct ISN is guessed, the intruder could gain control of the connection and data may be injected *[Morris]*.

A typical TCP connection is created using the three way handshake. Below shows how the three way handshake normally should take place. Following this, data transmission can take place.

Three way handshake.

$C \rightarrow S$:SYN(ISN_C)
$S \rightarrow C$:SYN(ISN_S), ACK(ISN_C)
$C \rightarrow S$:ACK(ISN_C)
Data transmission can now take place

C, the client, selects an ISN and sends it to S, the server. S receives this, and selects its own ISN and sends it together with an acknowledgement to C. C then sends an acknowledgement of this back to S, after which data transmission can take place.

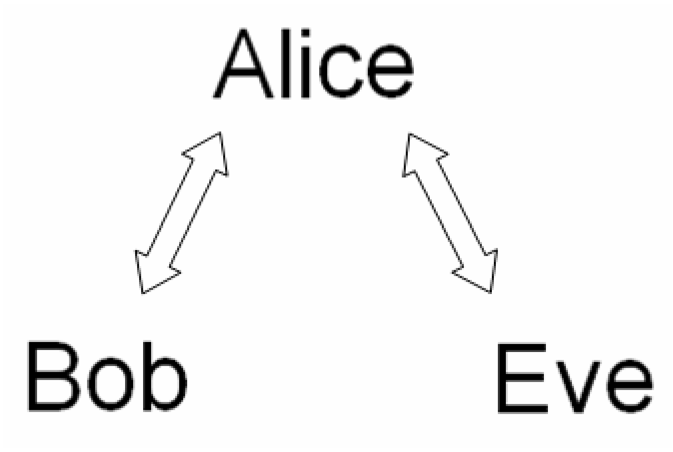


Figure 2 – Example of ISN attack

In a connection between Alice, the client and Bob, the trusted host packets will only be accepted if, amongst other reasons, they have the correct ISN. If an intruder, Eve (eavesdropper), forges a packet so that Alice “sees” it as coming from Bob and is able to predict the correct ISN_S, it is possible for Eve to send any data in the packet with the possibility that Alice executes code that was in the malicious packet. This type of attack is known as spoofing. Together with this discovered flaw and the original implementation of the

TCP ISN generator, an attacker could very quickly and easily predict the next ISN and successfully launch an attack.

In 2001 Michael Zalewski released a paper highlighting obvious flaws in the then current operating systems' TCP/IP implementation. In this paper a method of plotting the ISNs was described which allowed one to visualise the ISNs. This method resulted in attractors forming which would allow any hacker to exploit this fact and obtain control of the TCP/IP connection with relevant ease *[Michael]*. The first method of plotting sequences of numbers in a Phase Space, described in this paper is the method that was used in Michael Zalewski's paper.

Another type of attack is the TCP reset attack. If two machines, host A and host B have an established TCP connection. Host C (a malicious host) can cause the TCP connection to be terminated by forging a packet from one host (A or B) to the other and setting the reset (RST) bit. If enough bandwidth is available to C, then it is easy enough to try all possible ISNs or a smaller subset if the generation algorithm is known. But in most cases bandwidth is limited resulting in the fact that host C must also be able to predict the correct ISN that is being used in the connection for the forged packet to be accepted by one of the hosts. This is an easy task if host C is directly in between, and can "sniff" packets, the communicating hosts. But in most cases, host C cannot do this. So if either hosts are using a system that has predictable ISNs, host C can easily predict the ISN and terminate the connection.

It has been shown that the more random an ISN is chosen, the less likely it is for this type of attack to occur. This is because an attacker has to guess the next ISN by monitoring previous ISN's and discover the ISN generation method. His paper shows how easy it can be to create a set of ISNs which can be used to create a spoofing set. By implementing a PRNG in the generation of ISN's results in the prediction of ISN's to become a lot more tedious and/or difficult.

Proposed fixes to this problem has been to use a function to generate ISN which can be of the following form:

ISN = M + F(localhostIP + localport + remotehost + remoteportIP)
or
ISN = M + R(t)
or
ISN = R(t)

Where: M is the standard 32 bit, four microsecond timer
F is a function which introduces some randomness or unpredictability (as suggested in RFC 1948) and R(t) is a PRNG complying with RFC 1750.

Equation 5

It is essential that the function F is not computable by an outside source, or it would allow an attacker to still predict the ISN's. It has been recommended that the function F be some secure cryptographic hashing function that takes [localhostIP, localport, remotehost, remoteportIP] together with some random data. The inclusion of random data is to ensure that there is and element of unpredictability in the ISN. Although, when a sample of ISN's, created according to RFC 1948, is plotted using Lattice Space, the resulting image will suggest that the ISN generator is using a completely linear ISN generator as described in RFC 793. The Phase Space and Noise Sphere will highlight the fact that the increment used between ISN's is linear or not. This is where the need for a good PRNG is necessary and the use of a poor PRNG would make this method of cause the implementation of TCP/IP providing a false sense of security *[RFC 1948]*. It is however noted that using random increments to increase the ISN over time will not, according to the central limit theorem, introduce enough variance in the ISN's *[CERT]*.

Another proposal has been to use a PRNG to generate the ISN, with no use of localhostIP, localport, remotehost and remoteportIP. The PRNG used for this method should be in accordance with RFC 1750. Although, problems can arise using a PRNG as the ISN generator. If the generator, by "chance", happens to generate the same number in close succession, then it is possible for packets to become intermixed and the data stream to be incorrectly reassembled.

3.1.3. Monte Carlo/Statistical (physical) simulations

Scientific computing has, since 1945, relied on random numbers to solve many complex problems that cannot be solved analytically. The worlds first super computer, MANIAC, used Monte Carlo methods to solve intense numerical problems for the design of the atomic bomb. The maths behind Monte Carlo is too advanced for the needs of this paper. However, the driving force behind Monte Carlo simulations are good random numbers. As has been said earlier, obtaining true random numbers can be a difficult affair. Hence the need for a good PRNG. The only requirement here is that the PRNG exhibits good statistical properties and that the same set of pseudo-random values can be reproduced multiple times using the same initial seed value every time (this allows for results to be validated) *[Liu]*. The integral, which is essential to many scientific problems, can be solved computationally by mean of Monte Carlo simulation.

Given:

$I = \int_D g(x)dx$, can be approximated by computational methods if we have independent and uniformly distributed [pseudo] random numbers over D using the following identity:

$\breve{I}_m = 1/m\{g(x^{(1)}) + \ldots + g(x^{(m)})\}$, where $x^{(1)} \ldots x^{(m)}$ are the random values.

Now if $m \rightarrow$ infinity, then $\breve{I} \rightarrow I$. This is the basic principal behind the Monte Carlo method and more advanced methods do exist *[Liu]*.

This is the basic method behind Monte Carlo, and as one can see if a poor PRNG is used (one which yields poor statistical results), the results from this simulation can be disastrous *[Buslenko]*.

An example of this can be shown by using the Monte Carlo method to find an estimate value of PI. This is done by generating random (x,y) coordinates in the range [0,1). If the point has a distance greater than one, it lies outside the quarter circle of radius one, it does not get counted. Otherwise, all other point get counted.

Now: total points = n

Points inside circle = x

Estimated PI = 4 * x / n

This is a crude example, but it does show how poor pseudo random numbers can lead to a breakdown in a physicists experiments. Figure 3 shows how Pi can be estimated using this method.

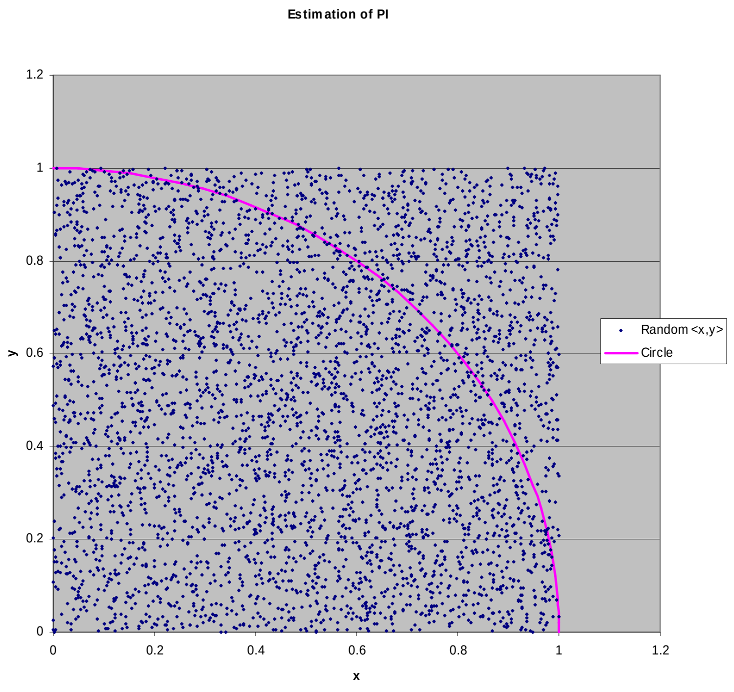


Figure 3 – Monte Carlo Estimate of Pi

The better the PRNG used and the larger the random data sample is, the more reliable the results will be.

3.2. Testing of a PRNG

Testing for randomness can prove to be a tedious process. Many statistical tests have been developed for this process. For the needs of this project a statistical test suite developed by the National Institute of Science and Technology (NIST) was used. It was initially developed for testing PRNG's for cryptographic applications and consists of 16 common statistical tests. All data tested had to either be in bits in Ascii format or hex digits in binary format. Table 1 gives a brief overview of all the tests performed in the NIST test suite.

Test	**Description**
Frequency (Monobit) Test	This tests the assumption that on average, the binary stream should consist of 50% 0's and 50% 1's, which is to be expected from a truly random sequence.
Frequency test within a Block	The test sequence is divided up into smaller blocks, and each block is tested as above.
Runs Test	This test looks at the number of consecutive 0's and consecutive 1's and considers whether the oscillation is too fast or too slow.
Test for the Longest Run of Ones in a Block	The test sequence is divided up into smaller blocks, and each block is tested as above.
Binary Matrix Rank Test	The test sequence is divided up into smaller blocks. Each block is tested for linear dependence against the original test sequence.
Discrete Fourier Transform (spectral) Test	This tests for any period features (repeating substrings) in the test sequence.
Non-overlapping Template Matching Test	The Non-overlapping Template Matching test determines whether there are too many occurrences of predefined aperiodic patterns.
Overlapping Template Matching Test	This test reject sequences which show too many (or few) occurrences of m-runs of 1's.
Maurer's "Universal Statistical" Test	This test is closely related to the per bit entropy of a random sequence. It is mainly significant if testing whether sequences are cryptographically strong. It rejects sequences that can be compressed.
Lempel-Ziv Compression Test	This test compresses the test sequence and determines whether compression in statistically significant.
Linear Complexity Test	This test uses linear complexity to test for randomness. I.e. whether the sequence is complex enough to be considered random.

Serial Test	This test is a battery of methods which tests the uniformity of distributions of patterns on given lengths.
Approximate Entropy Test	This test looks for repeating patterns of small sequences within the main sequence.
Cumulative Sums (Cusum) Test	This test determines whether sum of partial sequences occurring in the sequence is too large, or too small.
Random Excursions Test	The test is used to determine whether the number of visits to a state within a random walk is greater than what one would expect for a random sequence.
Random Excursions Variant Test	Similar to the test above, this test is to detect deviations from the distribution of the number of visits of a random walk to a certain state.

Table 1 – Description of NIST tests *[NIST]*

Another test program used to test sequences of numbers for randomness was used. Ent, A Pseudorandom Number Sequence Test Program applies a smaller set of statistical tests than the NIST test suite, but it returns actual test results and not probabilities which the NIST test suite does. This can in useful when a quick, direct comparison between sources was needed. Ent applies the following tests to a data file *[Walker]*

Entropy	This essentially returns how dense the information is, or random the data is. The higher the entropy of a file, the less it can be compressed
Chi-square Test	This is exactly the same as is explained in the NIST test suite and the test results returned shall not be used.
Arithmetic Mean	The arithmetic mean of a random data sequence bounded [0,1) should be 0.5
Monte Carlo Value for Pi	A way to test a PRNG is to calculate the value of Pi by means of a Monte Carlo simulation and then compare this value to a known value of Pi.

Serial Correlation Coefficient	This tests how much each byte of data is dependent on the previous byte. The closer the result is to zero, the more independent each byte is. A good random sequence should have a result very near to zero.

Table 2 – Description of Ent tests

3.3. Summary

This chapter has shown how different applications have different needs in terms of PRNG's. Cryptographic applications mainly require a high level of entropy in their "random" numbers. The TCP ISN generator should use some sort PRNG, or some sort of randomness in the ISN generator. ISN attacks were also discussed and explained. The importance of PRNG's in physical applications was highlighted, especially the need for the PRNG to produce statistically good pseudo random numbers. The NIST test suite was introduced and explained, as well as a smaller test, the Ent test.

The next chapter shall introduce a method of visually testing sequences of numbers for randomness in three dimensions. How to identify randomness in the images will be explained. A hardware device to generate true random numbers will be explained as well as its implementation details. Finally, the application developed for this project will be explained.

Chapter 4
System Design and Implementation

4.1. Randomness in three dimensions

A problem with the graphical method of identifying “randomness” has been “what is randomness in an image?”. To identify a good random source one needs some sort of control image with which a possible random source should be compared to. To find a good source of random data, the NIST test package was used on multiple pseudo random sources. The reason for using a pseudo random source was because the test data would be readily available and of a high “random” quality. The /dev/urandom device in the Linux[1] system was found to be a very good source of data to construct control images with which one can compare to. While the /dev/random device collects and stores an entropy pool by timing the difference in keyboard interrupts, it can be very slow. A major difference between the two devices is that /dev/random provides truly random cryptographically strong random numbers. The /dev/urandom device is simply a PRNG which does return statistically random output. A large data sample of 16 streams of 1000000 bit each was taken using the /dev/urandom device. Ent returned the following results:

Test	**Result**
Entropy	1.000000 bits per bit
Optimum compression	Reduce file by 0%
Arithmetic mean	0.5000 (0.5 = random)
Monte Carlo value for Pi	3.142649923 (error 0.03 percent).
Serial correlation coefficient	0.000012 (totally uncorrelated = 0.0)

Table 3 – Ent results for /dev/urandom

As can be seen from the Ent test results all the values are either equal to, or very nearly equal to what they should be for a true random sequence. The entropy of 1 bit

[1] Kernel version 2.6.10-1.741_FC3smp

per bit is the maximum entropy possible. The optimum compression of 0% means that the file cannot be compressed. This is due to the fact that truly random files cannot be compressed. The arithmetic of 0.5 is what is to be expected from a random source in the range [0, 1)

The NIST test results are as follows:

Test	**Result**
Frequency (Monobit) Test	SUCCESS
Frequency test within a Block	SUCCESS
Runs Test	SUCCESS
Test for the Longest Run of Ones in a Block	SUCCESS
Binary Matrix Rank Test	SUCCESS
Discrete Fourier Transform (spectral) Test	SUCCESS
Non-overlapping Template Matching Test	SUCCESS
Overlapping Template Matching Test	SUCCESS
Maurer's "Universal Statistical" Test	SUCCESS
Lempel-Ziv Compression Test	0% compression
Linear Complexity Test	SUCCESS
Serial Test	SUCCESS
Approximate Entropy Test	SUCCESS
Cumulative Sums (Cusum) Test	SUCCESS
Random Excursions Test	SUCCESS
Random Excursions Variant Test	SUCCESS

Table 4 – Results of /dev/urandom device

The results from both the Ent test and the NIST test suite are conclusive evidence that the randomness of the /dev/urandom device is adequate for the construction of the control images.

Figure 4 is an example of what a very good PRNG will produce using this graphical method of analysis plotted in Phase Space. The images shown here are all created using the program developed for the purposes of this project. Using the Phase Space transformation on the sequence of random numbers, the shape in Figure 4 below is what is to be expected. Figure 5 show what Lattice Space looks like for this random sequence. Figure 6 is an example of the Noise Sphere for the random sequence. This method will (or should) create a sphere of random points in three dimensions with no obvious attractor forming. The band of points clustered around the z-axis is due to the spherical coordinate's transformation and must not be confused with an attractor.

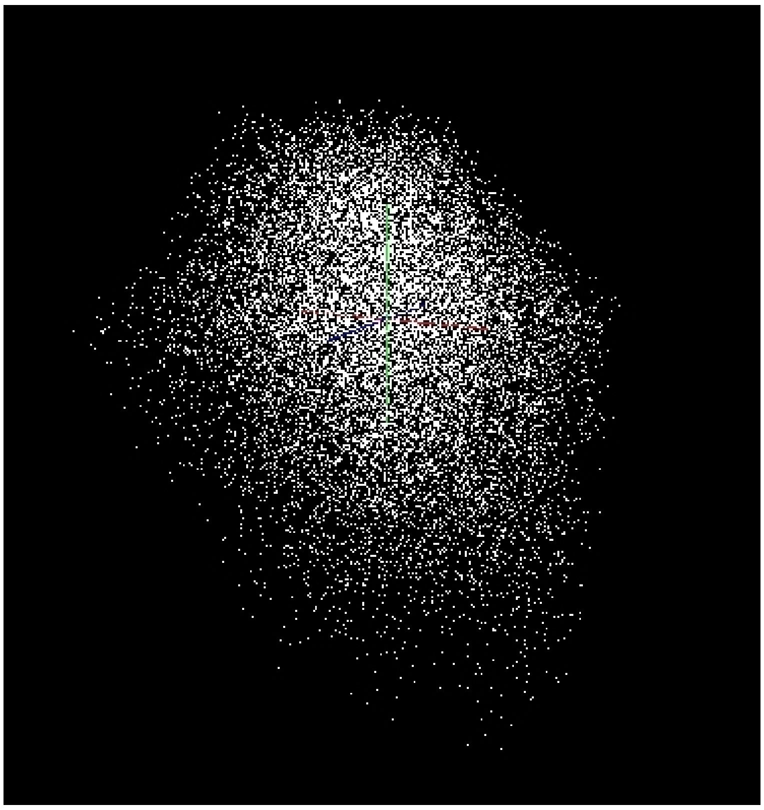


Figure 4 – Phase Space control image

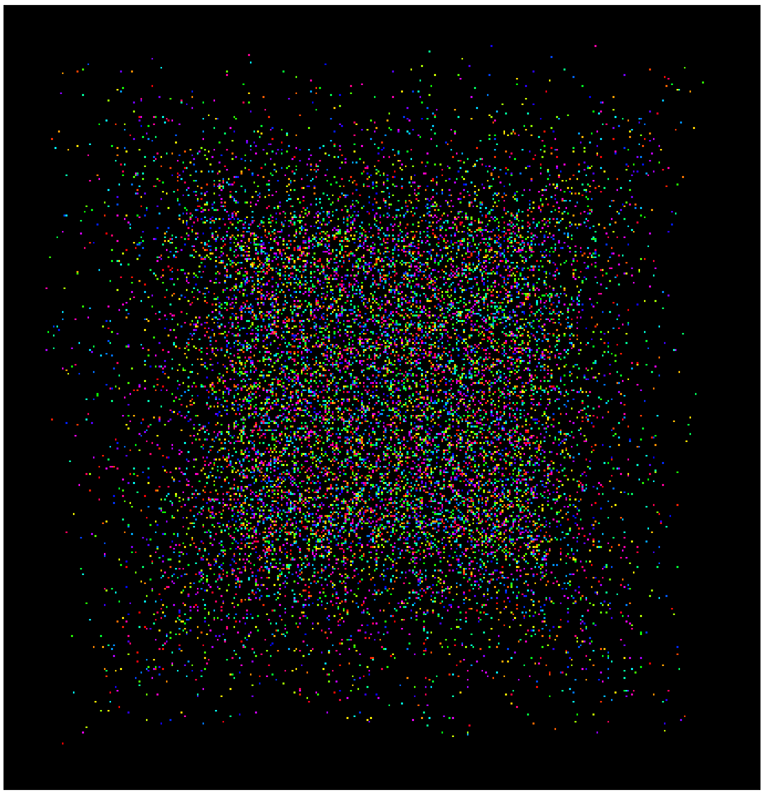


Figure 5 – Lattice Space control image

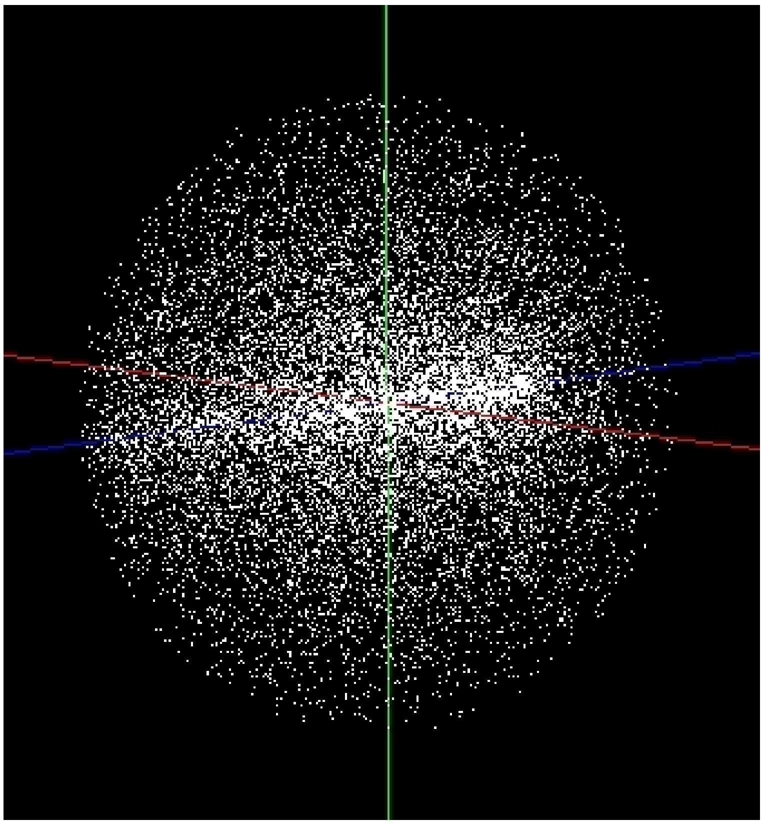


Figure 6 – Noise Sphere control image

4.2. Colour, an extra dimension

The use of simple black point plotted on a white background (or vise versa) does not give any indication of "how" the numbers were generated. The use of colour can introduce an extra dimension and hence give some indication of how the data sequence was generated. Two methods were implemented to give some insight into the data.

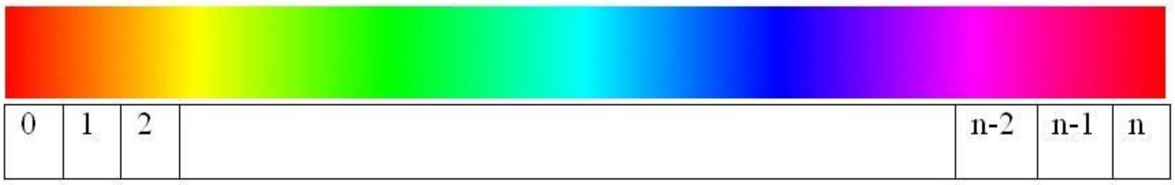


Figure 7 – HSV colour mapping

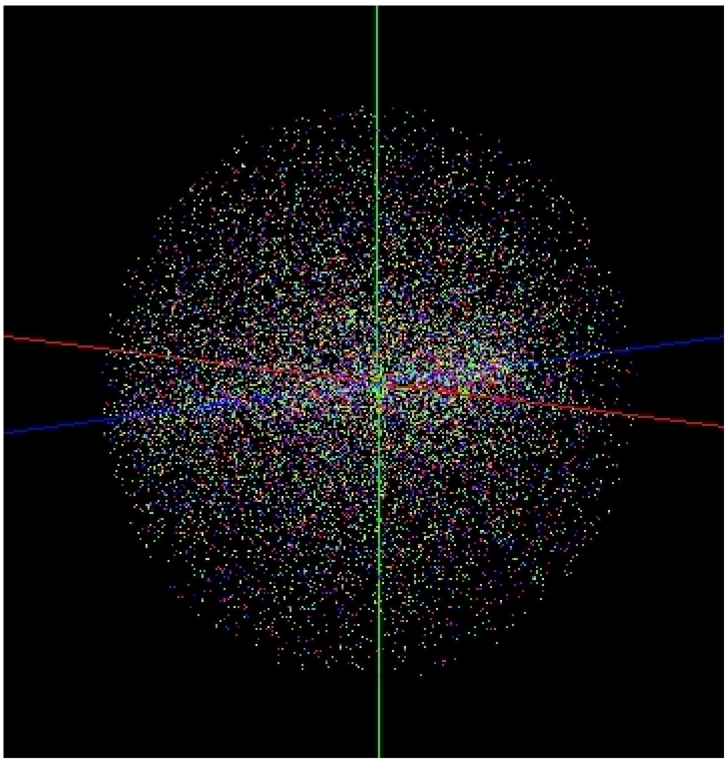


Figure 8 – Noise Sphere control image with colour

The first method used was to take the sequence of numbers and by assigning consecutive colours (in a continuous defined spectrum) to the consecutive numbers (in a temporal sense) in the sequence. The colour model used was the HSV model. Figure 7 shows how the HSV colour model is mapped to the sequence of numbers.

Red is assigned to the first element in the array and successive assigned to successive array elements, ending at red being assigned to the last element in the array. This was because it made constructing a clear, continuous colour spectrum much easier than using the RGB colour model. This is because it is very difficult to create a smooth continuous spectrum of colours using the RGB colour model. By doing this one can see whether the sequence was created in a specific order or if it was created in some kind of random order.

A second method of adding colour that was used, was to take the sequence of numbers, and divide each and every number but the maximum number in the sequence. This will result in the sequence being in the range (0,1]. Then using the HSV colour model, colours were assigned to numbers in the sequence according to their size. This method of adding colour shows whether the sequence of numbers is well distributed over the entire range. It also highlights the fact that every number in a random sequence has equal probability of being selected.

Figure 8 is an example of the Noise Sphere using the temporal method of adding colour. The colour is very well distributed with no attractors developing and is what is to be expected. The Phase Space transformation will produce images like these with the colour being randomly distributed. If any non-randomness exists in the sequence to be tested, then attractors will result in the images. The use of colour is also a good method to identify non-random features in the sequence. If the colours in the image are clumped together (all the reds together for example) then this shows some linearity (or pattern) in the sequence.

4.3. Construction and Testing of a hardware RNG

The use of a physical device to generate random numbers can possibly be one of the most secure methods of generating random numbers for computational needs. It is not, however, suited for all applications. Monte Carlo experiments, for example, should not be performed using a physical random number generator. This is due to the fact that the same sequence of numbers can never be obtained again which will result in an experiment never being replicated exactly. This will have major implications when results need to be verified by colleagues. Instead, a good PRNG should be used with very good statistical properties. But for purposes of security and cryptography, the use of a physical device is desirable. This is because a hardware device, if properly constructed, will have a high degree of entropy.

Hardware devices range from Geiger counters counting the decay of a radio-active sample to devices that amplify and sample the noise on a resistor. The focus here will be on a device that amplifies and samples the noise on a resistor. This noise is known as Johnson noise and arises due to thermal agitation of electrons in a conductor which cause small random voltage fluctuations in the signal *[Ghausi]*. The noise produced is evenly distributed over all frequencies and is known as flatband noise, or white noise. *[Israelsohn]*. To ensure that other common frequencies, like the 50Hz mains frequency, don't interfere and cause the "random" source to be skewed, one needs to use filters to band-limit the noise produced. Figure 9 shows the output of the voltage fluctuation across a resistor as a function of time.

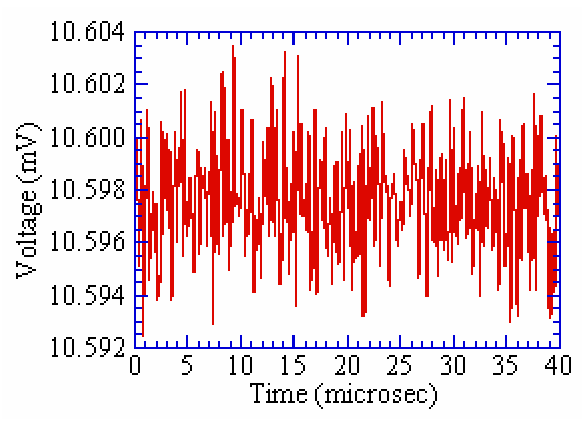


Figure 9 – Voltage fluctuations on a resistor

For the purposes of this project, it was necessary to try and implement a hardware random number generator. Figure 10 shows the circuit design of the generator. The way this generator works is by amplifying the noise on a resistor. The output is then used together with a comparator to produce a binary stream of data. The resistor labelled R1 was initially used, but the results obtained were not promising. This could be because the required amplification used to amplify the noise on the resistor is so large that any other possible noise introduced by outside interference can potentially dominate, which seemed to be the case here. This is explained in Chapter 6.

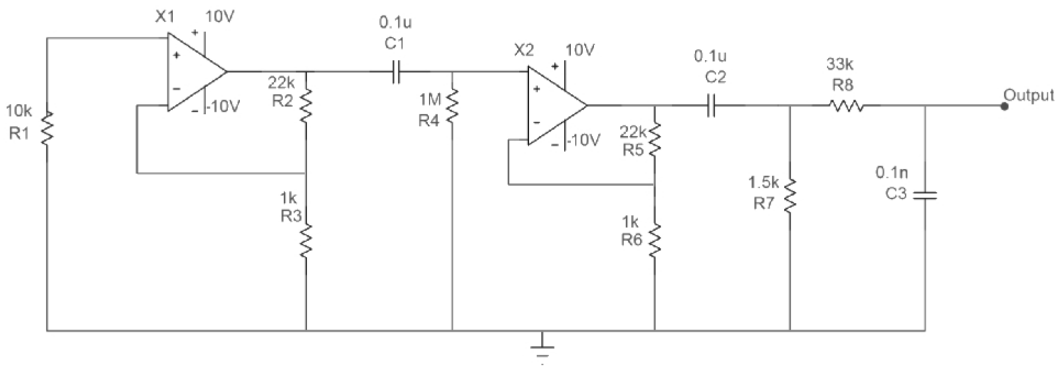


Figure 10 - Circuit Diagram of hardware RNG

The resistor was then replaced with an antenna to capture free space noise. The results obtained using this method were much more promising and showed signs of randomness. It must be stated now that this is a very crude hardware RNG and that while precautions were taken to eliminate external noise, noise was still introduced which skewed the results. The test results are explained and shown in Chapter 6.

4.4. Design features / considerations

The implementation of this application was relatively simple. Figure 11 shows the class diagram for the implementation of the application used to graphically analyse the sequences. Table 5 gives a brief overview of the classes and their functions.

Class	**Use**
easyform	This is the GUI for the application.
glisn	This class is the core class of the application. All button presses and openGL drawing calls begin from this class
Mesh	This class is used to read, create and plot the sequences of numbers.
Timer	This class is used to get frame rate information.

Table 5 - Class diagram explanation

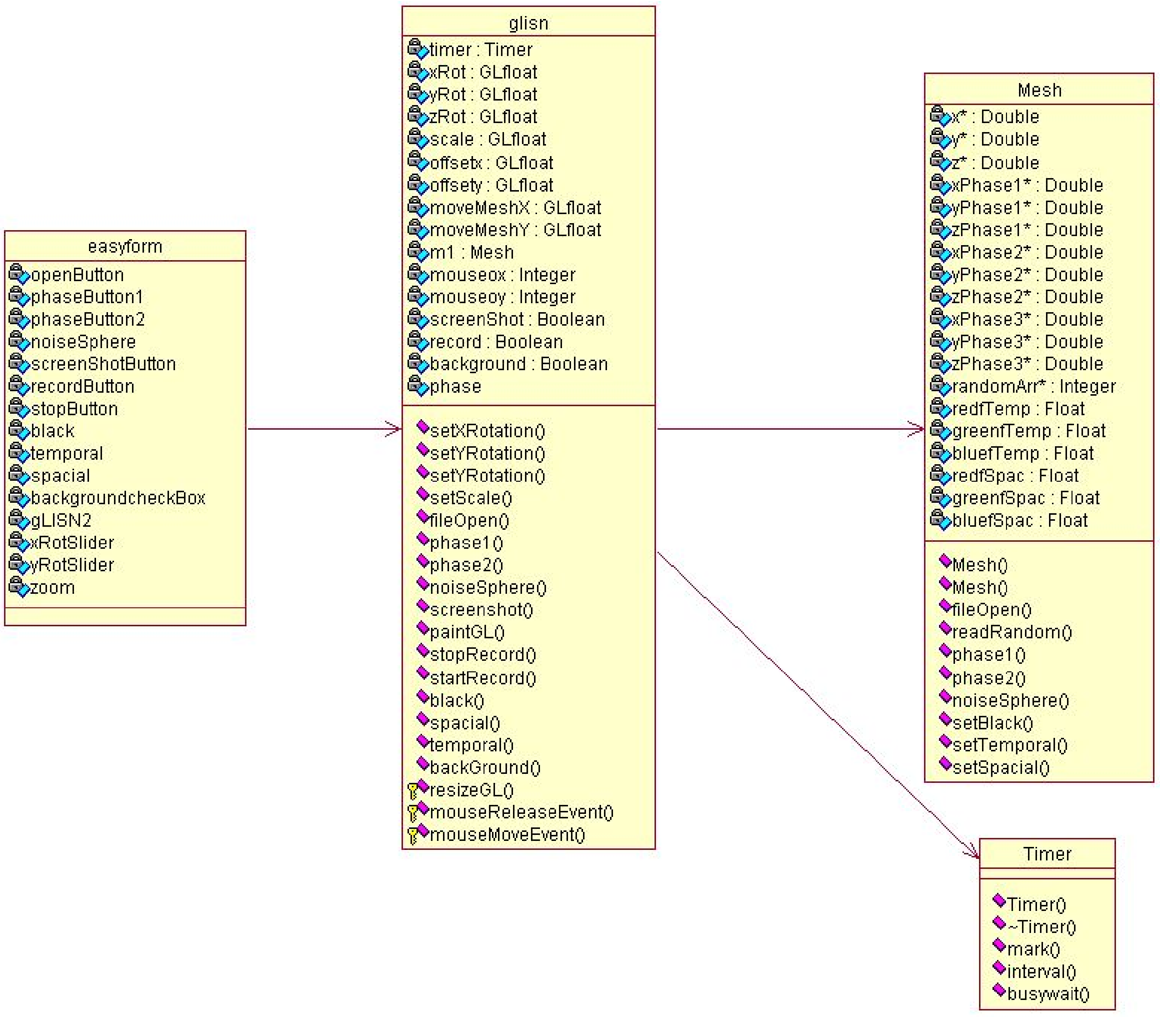


Figure 11 – Class diagram

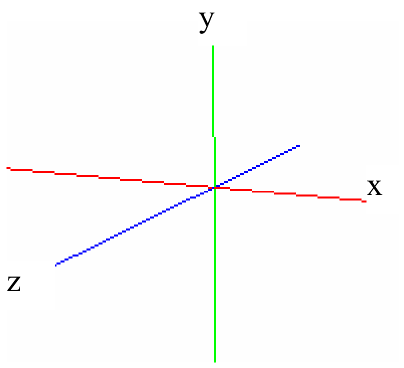
4.5. Implementation details

The implementation of this application was done on the Linux platform using QT Designer. The C++ language together with the OpenGL library was used to handle all the graphical need. Care had to be taken when reading the sequences of numbers in from the files. It was decided that all numbers be read in as `unsigned int.` This decision came about after the fact that ISNs are unsigned 32-bit integers.

4.6. Application

The application itself is fairly simply to use, with Figure 12 being a screenshot of it. The main visualisation display has the three axis in red, green and blue. They are as follows:

Red axis	-	x- axis
Green axis	-	y- axis
Blue axis	-	z – axis



- Phase 1 button will plot the coordinates using the Phase Space.
- Phase 2 button will plot the coordinates using the Lattice Space.
- Noise Sphere button will plot the coordinates using the Noise Sphere.
- The Screenshot button will take a screenshot of whatever is currently being displayed in the visualisation window.
- Record and Stop buttons are used to capture sequences of screenshot which can be used to create video clips. Creating video clips proved to be a difficult process. This is because each frame had to be saved to disk, causing the frame rate to drop dramatically resulting in it being very difficult to navigate around the image.
- The three radio buttons, Black, Temporal and Spatial, select the method of adding colour as is discussed in Section 4.2.
- The background checkbox selects whether the background of the main visualisation window is either black or white.
- The zoom slider scales the image either up or down.
- The x and y sliders allow for fine adjustment when rotating the image around the relevant axes.

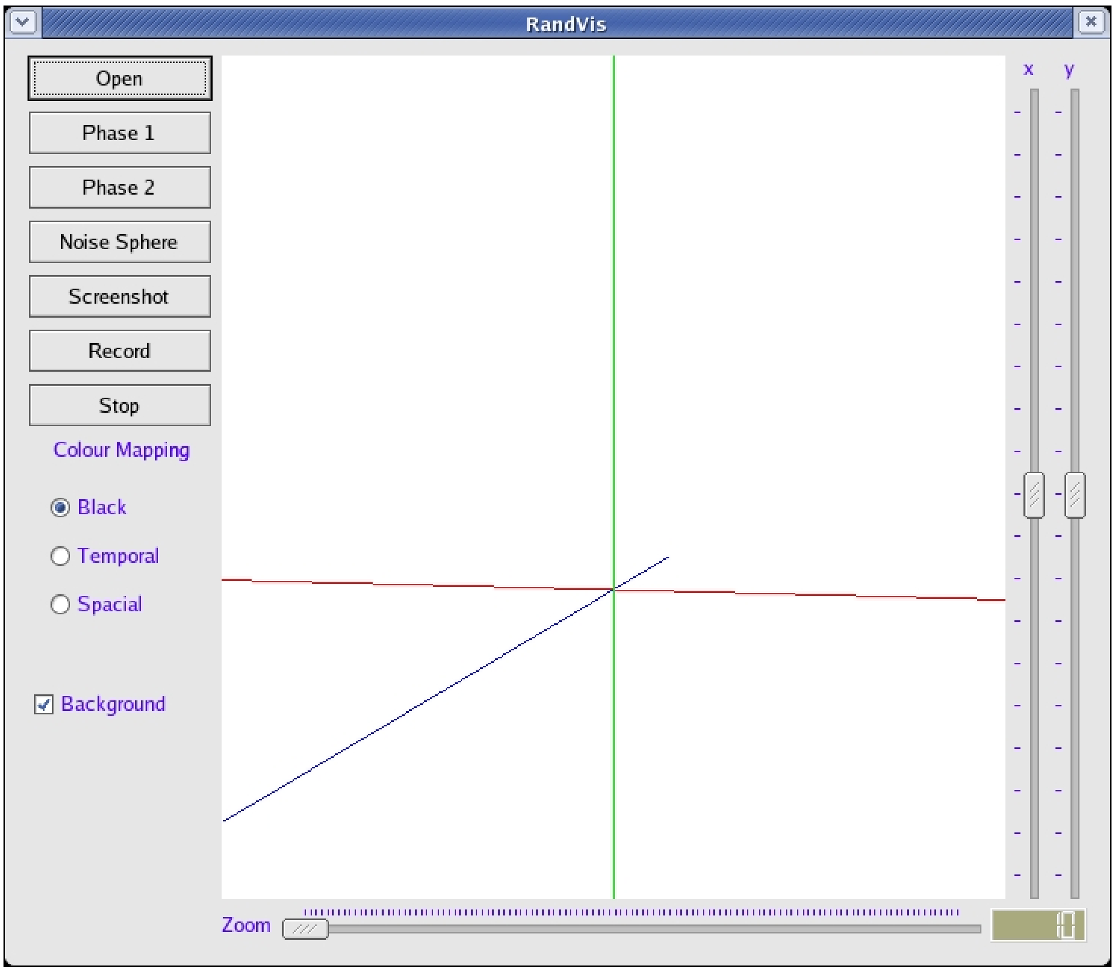


Figure 12 - Application

4.7. Summary

This chapter has shown what randomness should look like in three dimensions with the use of control images. The random number sequence was obtained from a statistically good source, the /dev/urandom device. Colour was shown how it can be used to add an extra dimension to the data. Two methods of adding colour were explained. An implementation of a hardware RNG was shown and some theory behind it was explained. Finally the design of the application developed for this project was shown together with the application itself.

The next chapter will show the results of the ISN's of different operating systems tested. The hardware RNG will also be tested and evaluated.

Chapter 5
Results

5.1. Operating Systems

Multiple operating systems were tested to try and determine the amount of randomness in its ISN generator. This will give an indication as to how secure they are to ISN attacks. The ISNs were "harvested" using a Perl script specifically written by Tom Vandepoel to obtain TCP ISNs. Usage of ISNProber is as follows:

```
Single host mode:
isnprober [options] <ip>|<ip:port>
        where <ip> is the IP address of the target machine
        and <ip:port> is the port on the target machine
```

Text Box 1 – Use of ISNProber

The use of different source ports and/or different target ports will most likely produce an ISN sequence that appears to be random due to Equation 4. For the purposes of this paper, it was decided to use a single source port and a single target port. This was because an attack aimed at the ISN generator is most likely to occur in this fashion.

It was found that any patterns in the data began to clearly appear after about 10000 points were captured. To make sure that any patterns in the ISN's would be identified, larger data samples were captured. It was therefore decided that 35000 ISN's was a good number of ISN's to capture and test. Also, 35000 ISN's would allow the NIST test use suite to be used.

5.1.1. Windows 95

The Windows 95 ISN generator appears to be very weak. Figure 13, shows an image of the ISN's plotted in Phase Space. As can be seen majority of the points are clumped together in a region and not distributed according to the control image, Figure 6. The colour in the image is also not well distributed implying that the ISNs were created in some sequential order. In the event of an attacker attempting a spoofing attack, Windows 95 offers very little protection against ISN attacks which can result in ISN attacks more likely than not, succeeding. Figure 14 is an image of the ISN sequence plotted in Lattice Space. This image shows the linear nature of the ISN generator. RFC 793 recommends that the ISN be generated using a 32 bit clock, as seems to be the case here *[RFC 793]*. This linear method of ISN generation can especially be seen in the smooth change in colour in Figure 14. Although Figure 14 suggests that no other information is being used to create the ISN, due to its very linear nature, Figure 13 shows that the difference between ISNs in not a constant. RFC 1948 recommends that the [localIP, localport, remoteIP, remoteport] be used to introduce some sort of unpredictability in the ISN generator. But RFC 1948 does also suggest that some randomness be introduced in the random increment. One can conclude, from these images, that the security offered by Windows 95 by its ISN generator, is very weak. When plotted using the Noise Sphere no useful information was produced.

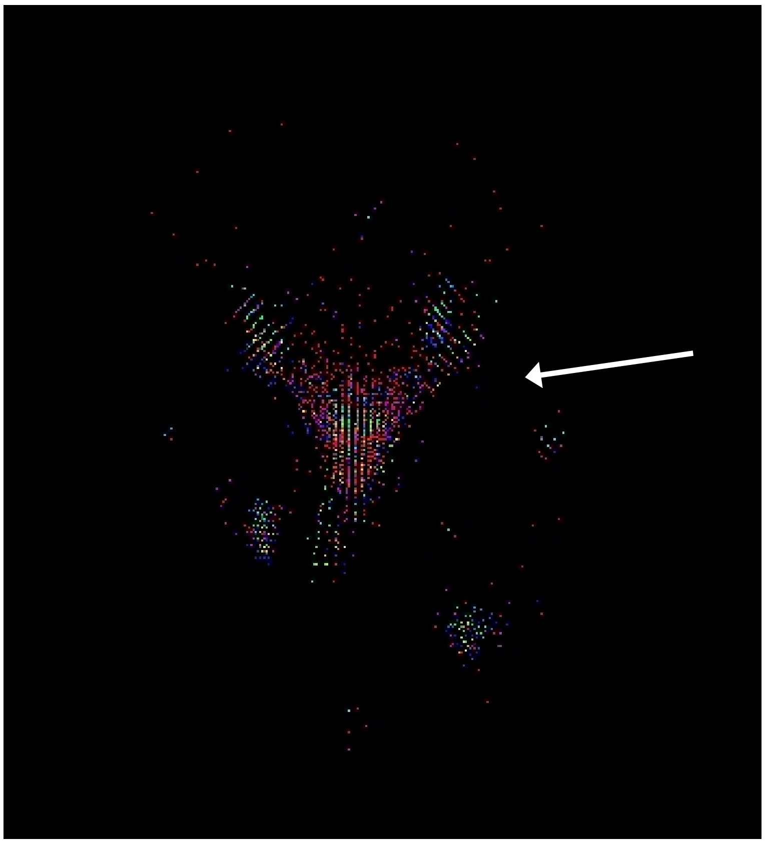


Figure 13 – Windows 95 Phase Space

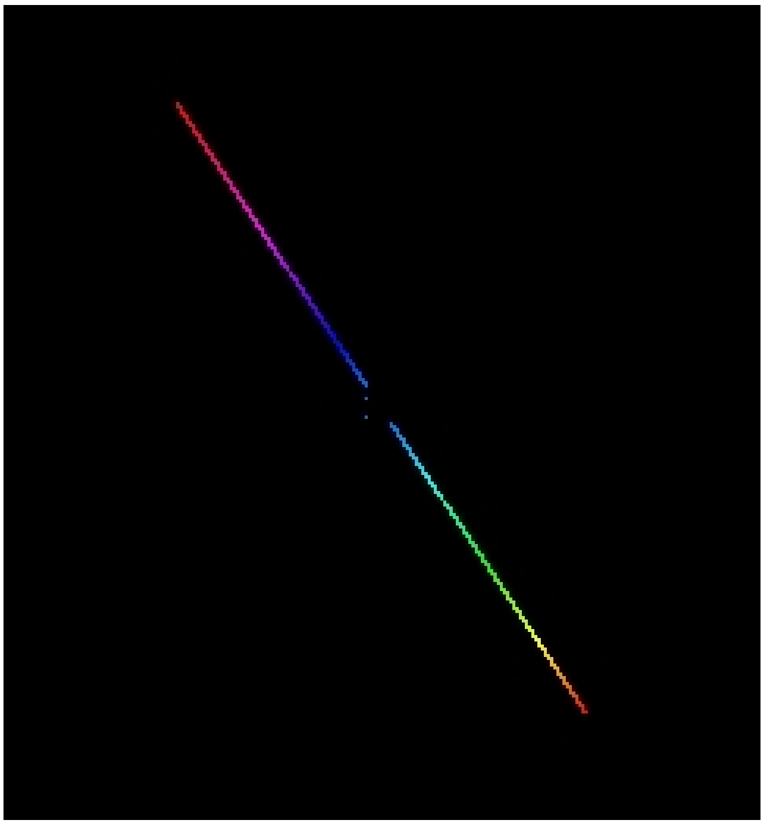


Figure 14 – Windows 95 Lattice Space

5.1.2. Windows 98

Figure 15 shows the ISN's of the Windows 98 operating system plotted in Phase Space. As can be seen there is some difference in terms of structure when compared to the Windows 95 OS. Majority of the points lie in a small region of space, which will allow an attacker to create a set of ISNs to be used in a spoofing attack fairly easily. The Lattice Space and Noise Sphere images exhibit the exact same linear structure as Windows 95. Obviously, this leads to the fact that Windows 98 provides no better security than Windows 95 and uses a very similar IP stack.

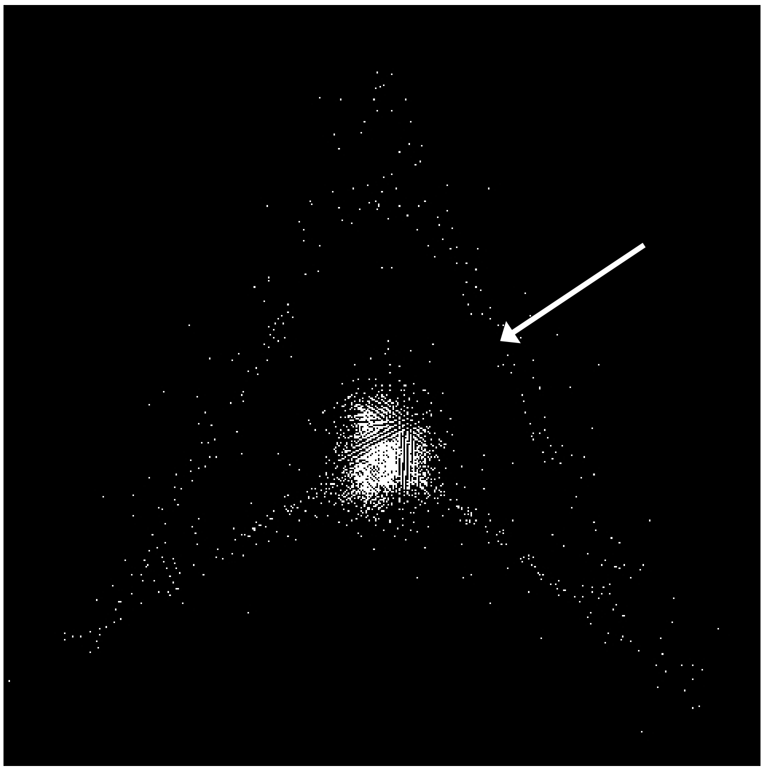


Figure 15 - Windows 98 Phase Space

5.1.3. Windows 98 Second Edition (SE)

The Windows 98SE OS tested did not reveal any differences from the original Windows 98 OS and hence the same potential vulnerabilities exist in terms of its ISN generator.

5.1.4. Windows ME

The Widows ME OS has shown a significant improvement to the previous version of Windows tested. Figure 16 shows the ISN's are creating using a bit more of a random method. Although they are still appear to be created in a linear fashion, much like Figure 14 when viewed in Lattice Space, the difference between successive ISN's is highlighted in Figure 16 to be of a much more random nature. Figure 17 shows the ISN's plotted using the Noise Sphere method. While not completely random, some degree of unpredictability is evident. The points are relatively evenly distributed in the sphere, but colour in the image suggests some linearity in the ISN generation process. This is expected and shows that the ISN stack is implementing RFC 1948 and introducing some degree of randomness.

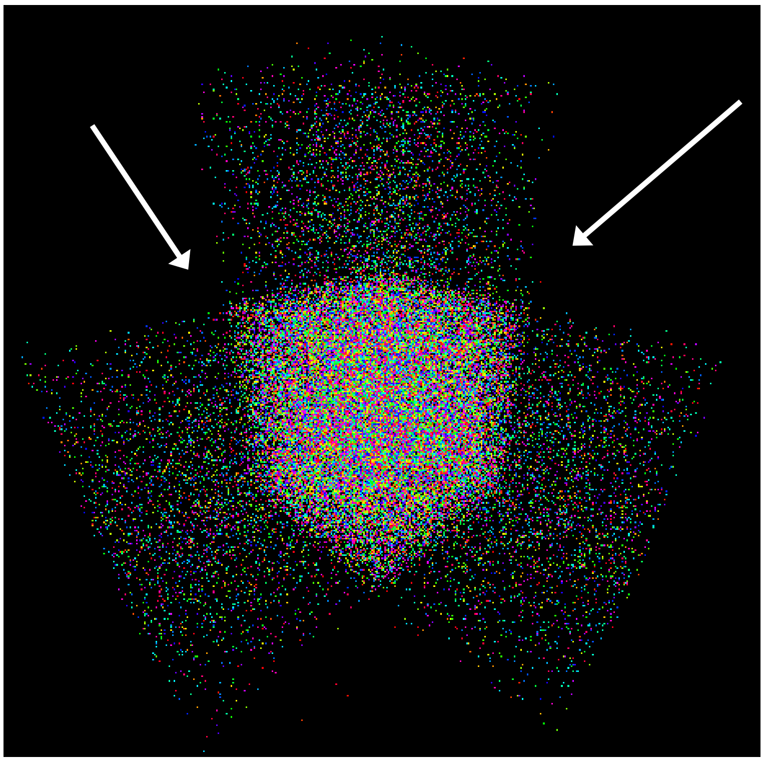


Figure 16 - Windows ME Phase Space

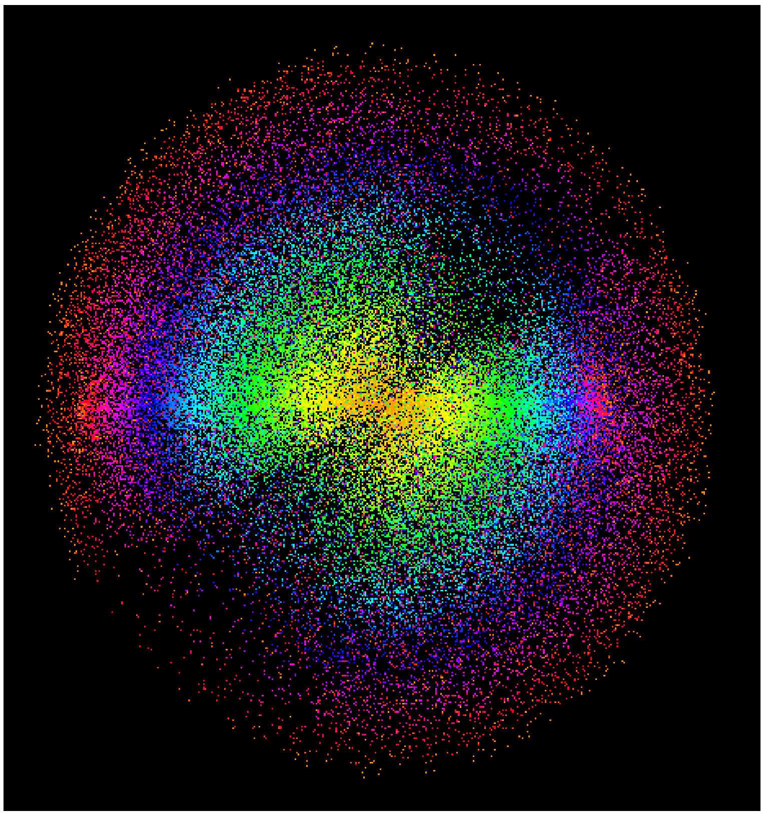


Figure 17 - Windows ME Noise Sphere

5.1.5. Windows 2003 Server

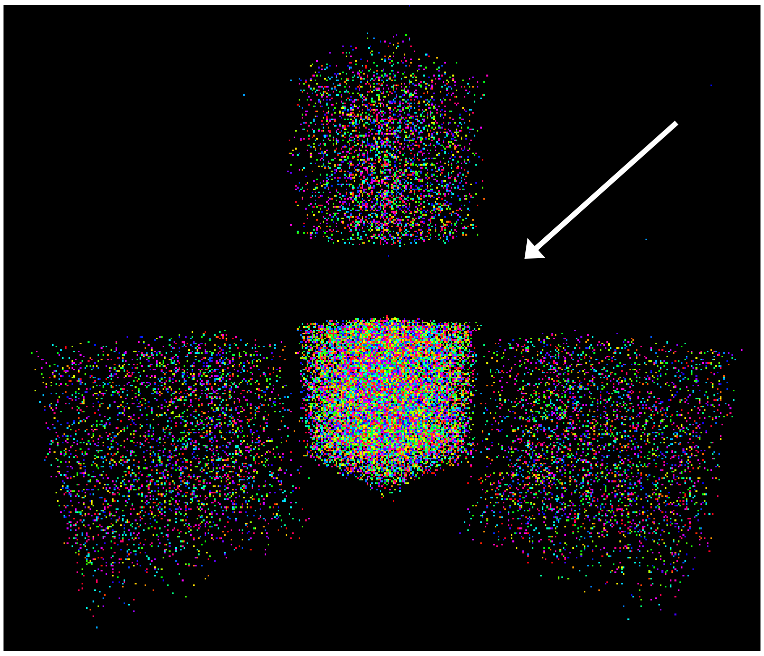


Figure 18 - Windows 2003 Server Phase Space

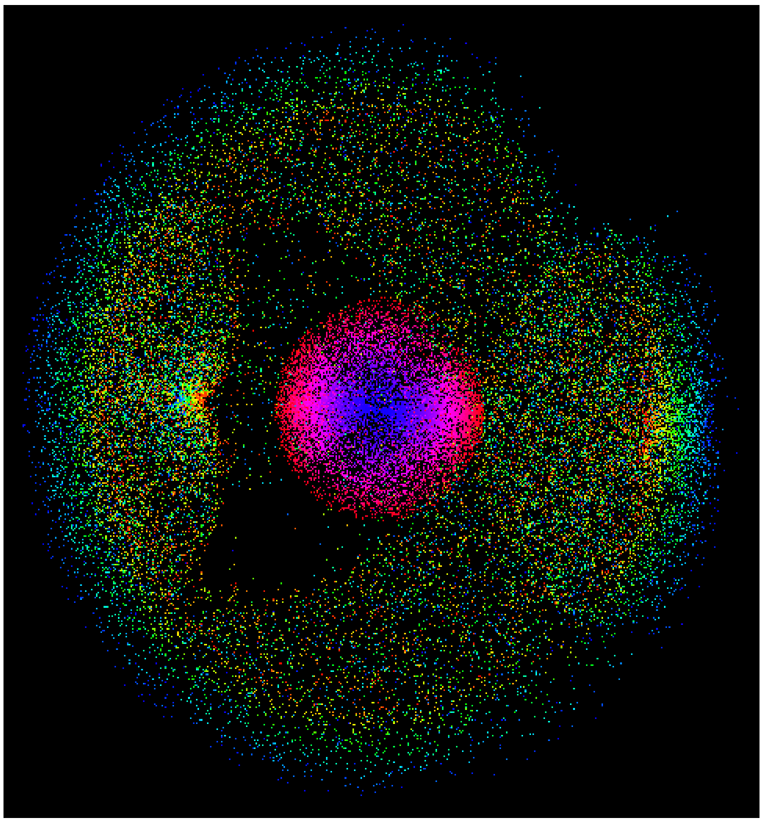


Figure 19 - Windows 2003 Server Noise Sphere

Windows 2003 Server shows a pattern very similar to that of Windows ME in Figure 18. Figure 19 is also very similar to that of Windows ME, except that the noise sphere is not as evenly distributed as Windows ME, with majority of the points lying at the centre of the sphere and on the radius. This is most likely due to the fact that these two OS's are using very similar TCP stacks.

5.1.6. Windows XP

5.1.6.1. Original

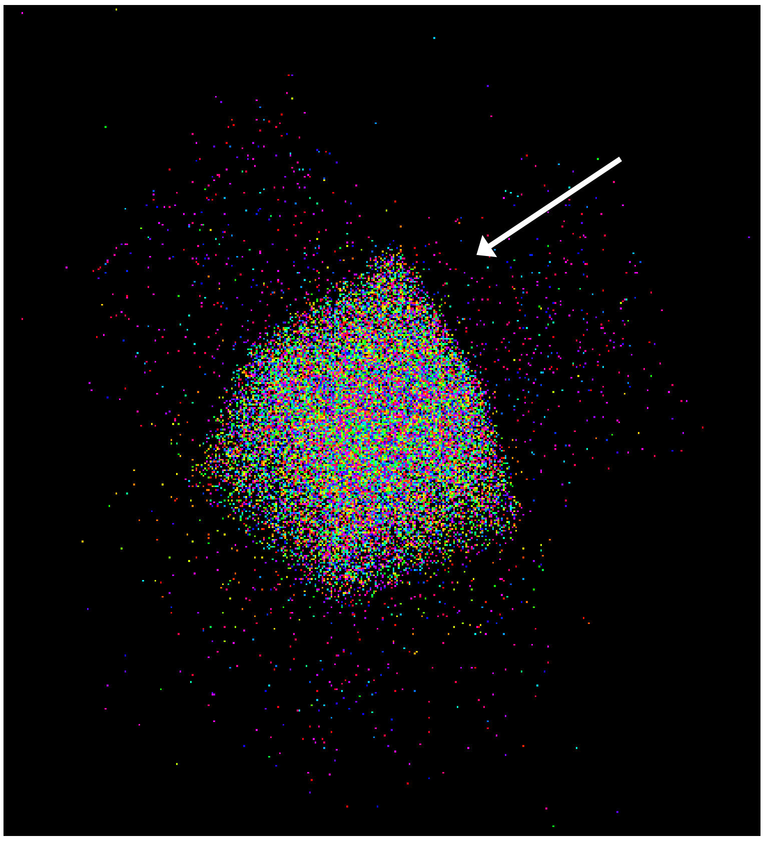


Figure 20 - Windows XP original Phase Space

Figure 20 shows the original (non-patched) Windows XP OS. The Phase Space view of the ISN's shows how the ISN's from windows XP appear to be random in the small area of space they occupy. Figure 21 show the ISN's plotted using the Noise Sphere technique. As can be seen, there is very little randomness in this image.

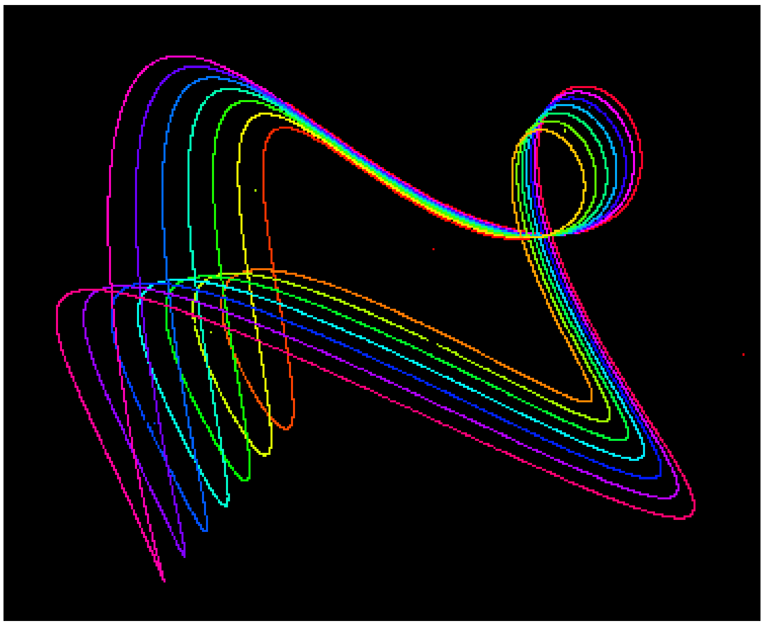


Figure 21 - Windows XP original Noise Sphere

5.1.6.2. Windows XP - Service Pack 1

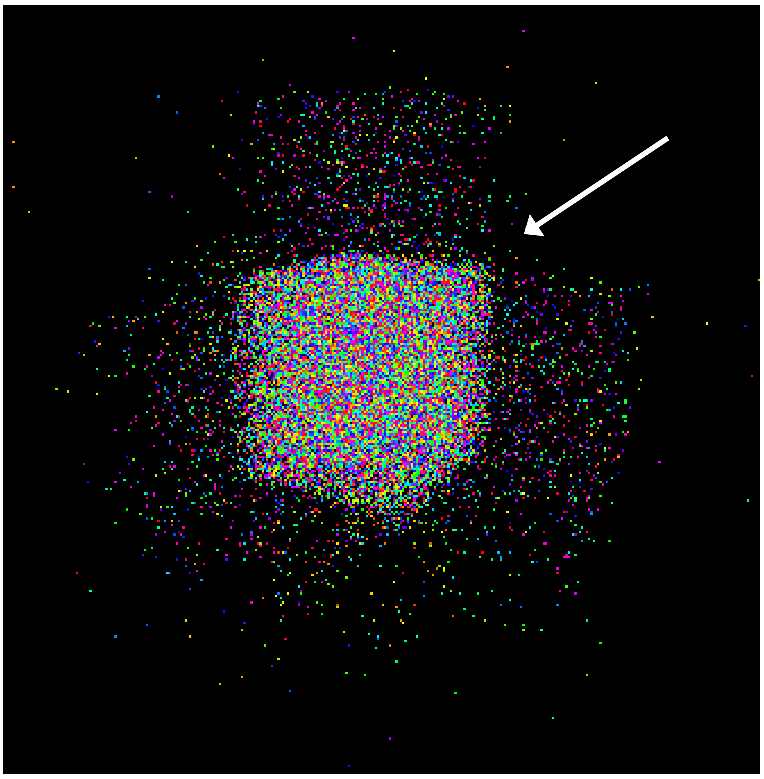


Figure 22 - Windows XP SP1 Phase Space

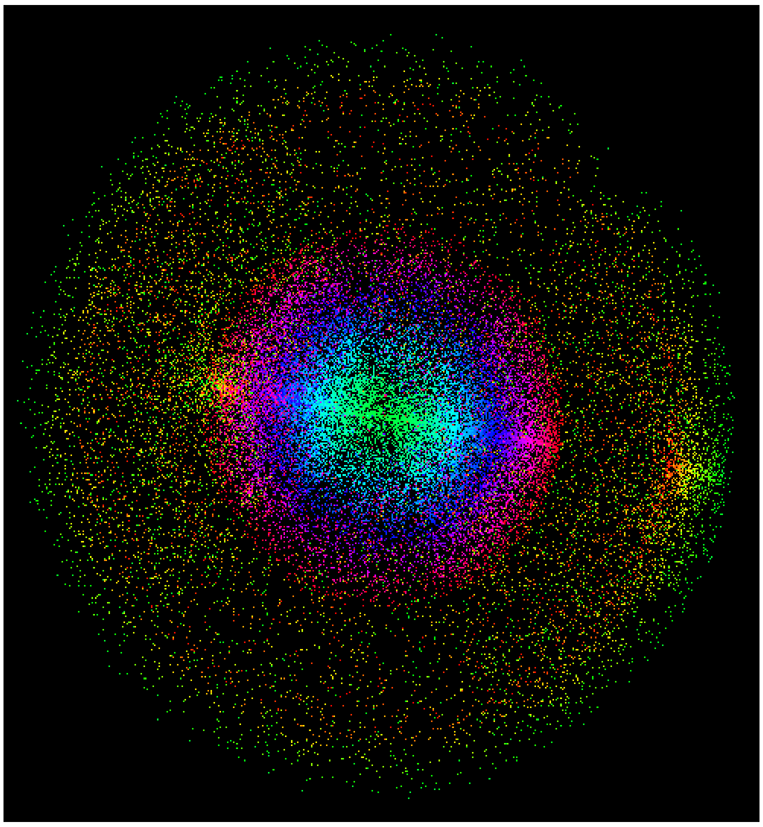


Figure 23 - Windows XP SP1 Noise Sphere

Figure 22 shows a marked improvement from the original Windows XP. The points are a bit more distributed. Figure 23 highlights the fact that the "random" increments suggested in RFC 1948 is being implements.

5.1.6.3. Windows XP - Service Pack 2

Figure 24 looks like there has been an improvement from service pack 1 as the points are even more widely distributed. But looking at Figure 25, one can see that all the points all lie on a line with no sign of random increments being used. This, as discussed earlier, has security issues involved.

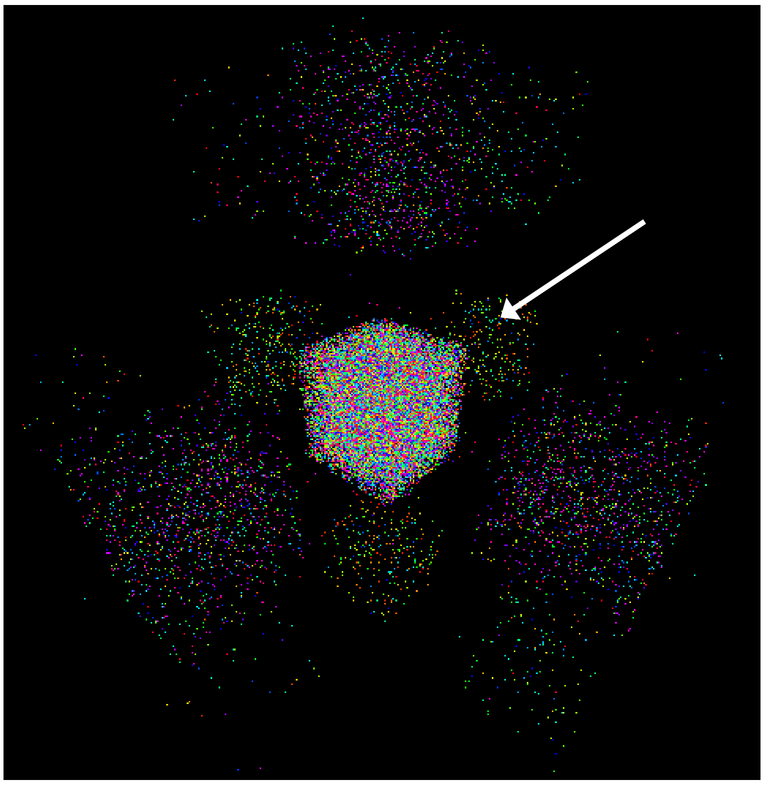


Figure 24 - Windows XP SP2 Phase Space

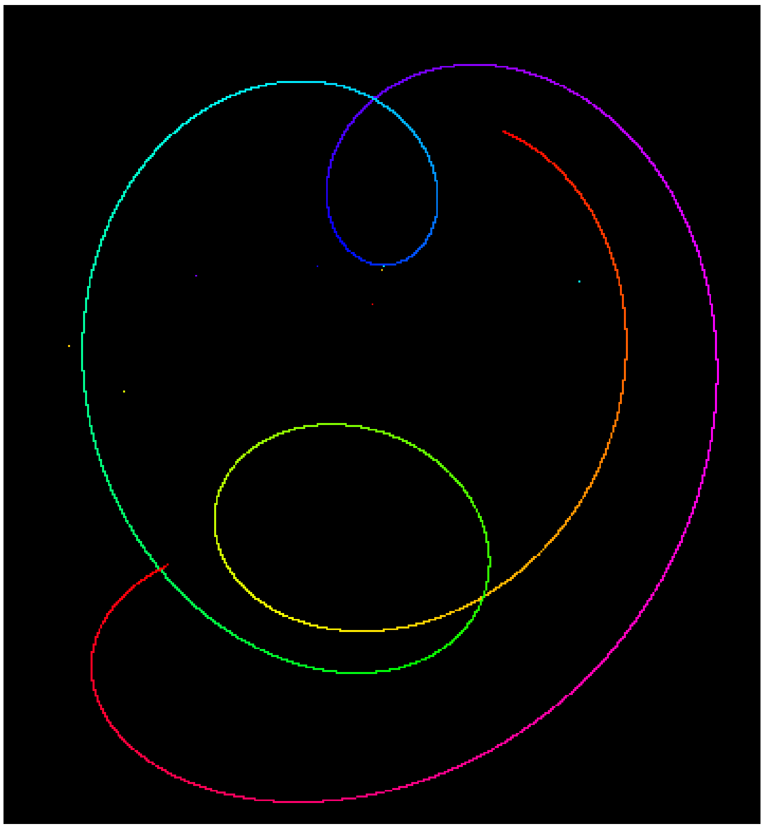


Figure 25 - Windows XP SP2 Noise Sphere

5.1.6.4. Current

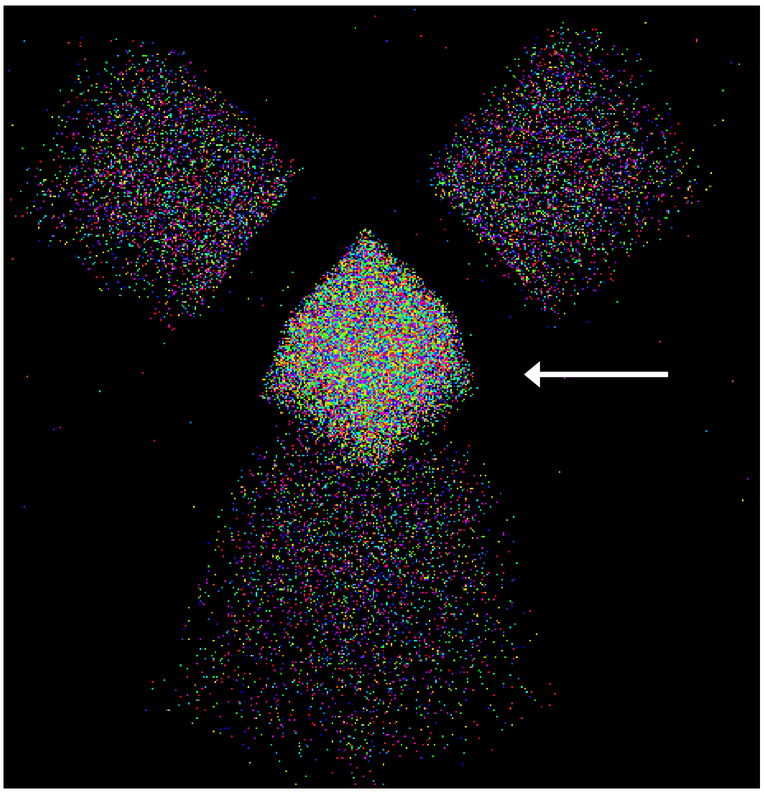


Figure 26 - Window XP fully patched, Phase Space

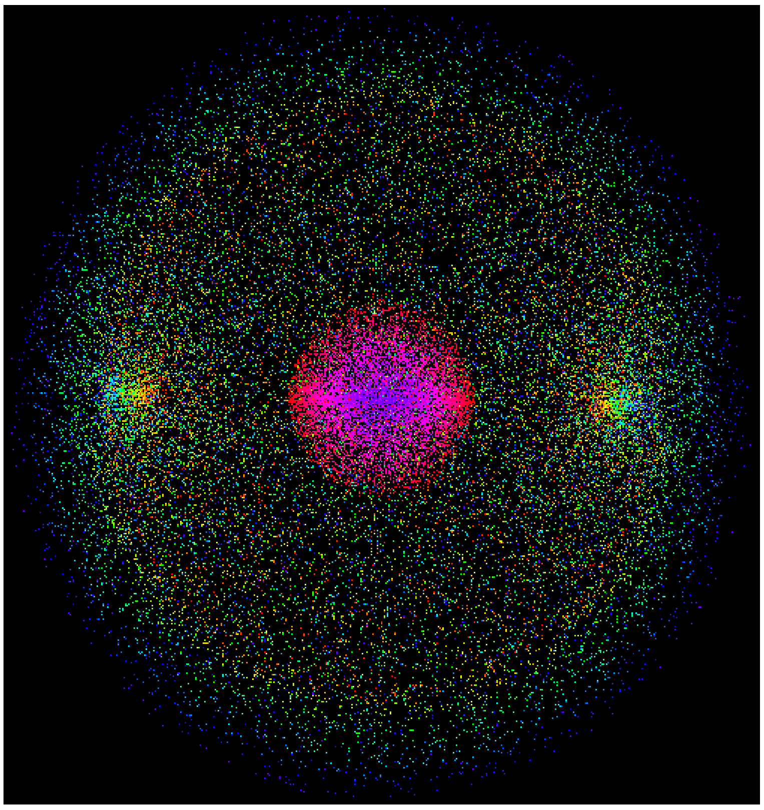


Figure 27 - Window XP fully patched, Noise Sphere

This Windows system tested was fully patched as of 4 November 2005. It is very interesting to note that both these images are simlar to Windows XP with service pack 1. Figure 26, a Phase Space plot of the ISN's, once again shows the clumping of majority of the points in an area of space. Although, as the use of colour shows us, it does appears to have some degree of randomness in it. Figure 27, a Noise Sphere plot of the ISN's, shows how the points look randomly distributed over the sphere, although majority of the points lie at the centre of the sphere and on the radius.

5.1.7. Cisco

Figure 28 shows the ISN's of a Cisco router running IOS 12.1. This image is very closely matched to the control image, Figure 4. The "randomness" of the ISNs produced show that the ISN generator implemented is a good quality PRNG. This is confirmed in a software patch released by Cisco updating the software to use a PRNG as the ISN generator. The ISN generator also has a window in which previous ISNs are stored and used to compare newly generated ISNs with. If the newly generated ISN is too close to previous ISNs, it is discarded and a new ISN is generated. This is to ensure that and overlapping of ISNs. Equations 2 and 3 produce images very similar to the control figures 5 and 6 respectively *[Cisco]*.

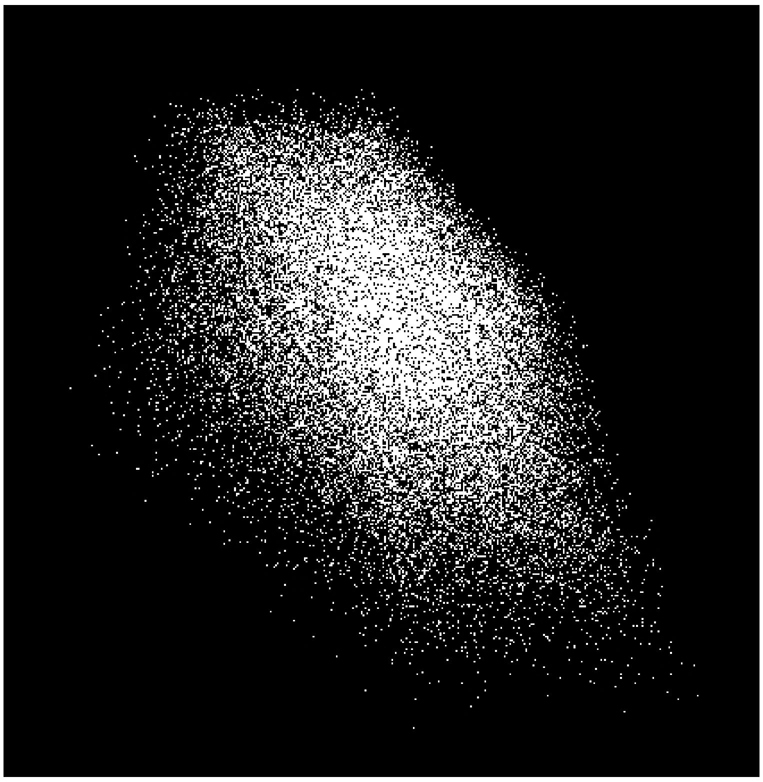


Figure 28 – Cisco Phase Space

Due to the graphical analysis of the Cisco ISN's and the similarities to that of that "random" control image, it was decided that NIST test suite be used to test the ISN sequence. The results are as follows:

Test	**Result**
Frequency (Monobit) Test	SUCCESS
Frequency test within a Block	SUCCESS
Runs Test	SUCCESS
Test for the Longest Run of Ones in a Block	SUCCESS
Binary Matrix Rank Test	SUCCESS
Discrete Fourier Transform (spectral) Test	SUCCESS
Non-overlapping Template Matching Test	SUCCESS
Overlapping Template Matching Test	SUCCESS
Maurer's "Universal Statistical" Test	SUCCESS
Lempel-Ziv Compression Test	0% compression
Linear Complexity Test	SUCCESS
Serial Test	FAILURE
Approximate Entropy Test	FAILURE
Cumulative Sums (Cusum) Test	SUCCESS
Random Excursions Test	N/A
Random Excursions Variant Test	N/A

Table 6

The Random Excursions Test and Random Excursions Variant Test could not be performed because the NIST test suite returned the following error:
"WARNING: TEST NOT APPLICABLE. THERE ARE AN INSUFFICIENT NUMBER OF CYCLES."

As can be seen form the results in Table 6, the Cisco router implements a very good ISN generator. The Cisco router obviously complies with RFC 1750 by implementing a good PRNG as its ISN generator.

5.1.8. Fedora Core 3

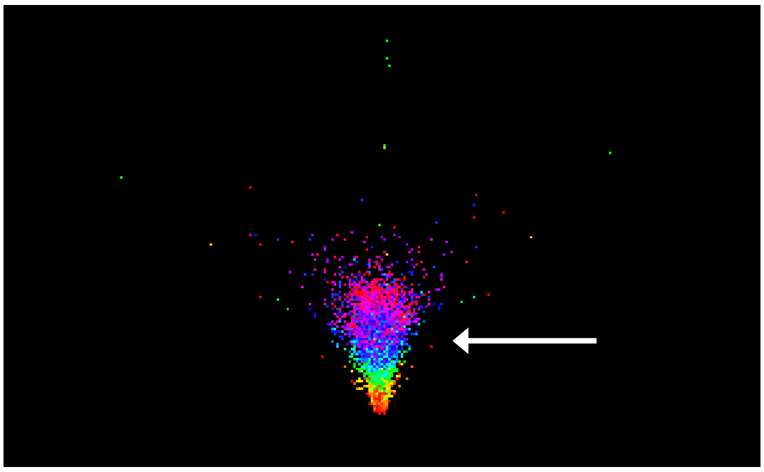


Figure 29 - Fedora Core 3 Phase Space

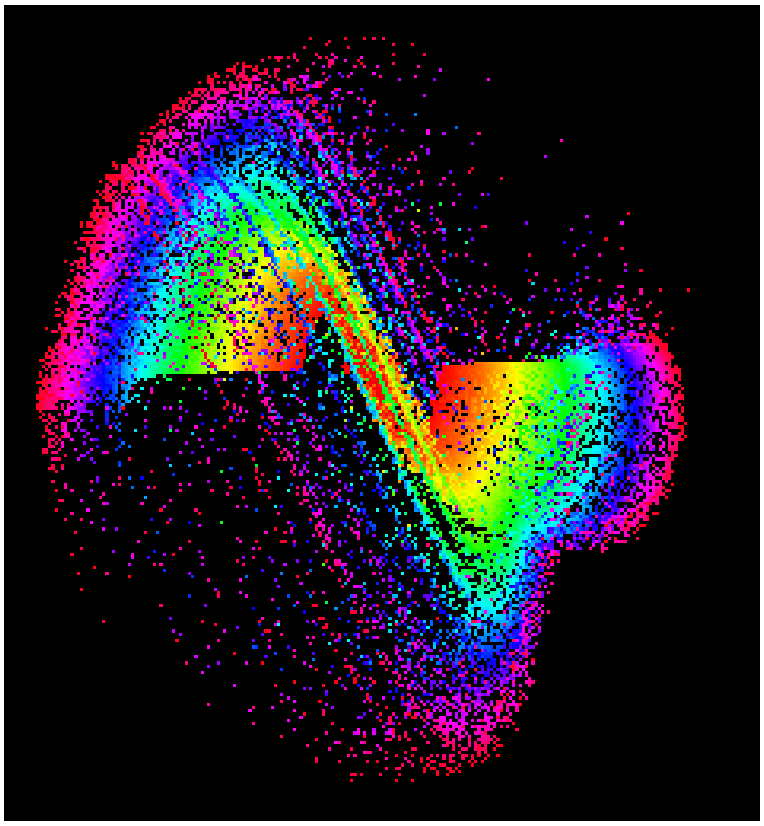


Figure 30 - Fedora Core 3 Phase Space

Fedora Core 3 uses kernel version 2.6.10-1.741. The TCP stack exists inside the Linux kernel, so most, if not all, Linux distributions using this kernel will produce the same ISN's. As can be seen in Figure 29, the differences between successive ISN's all lie in a small region of space. Once again, the Lattice Space image shows that the ISN generator is RFC 1948 compliant. Figure 30 show that there is randomness introduced into the ISN generator, but some structure is still evident. The fact that all the points lie in such a small region of space in Figure 29 shows that if an attack on the ISN generator had to take place, a spoofing set could be built up quite quickly.

5.1.9. FreeBSD

5.1.9.1. FreeBSD 4.1

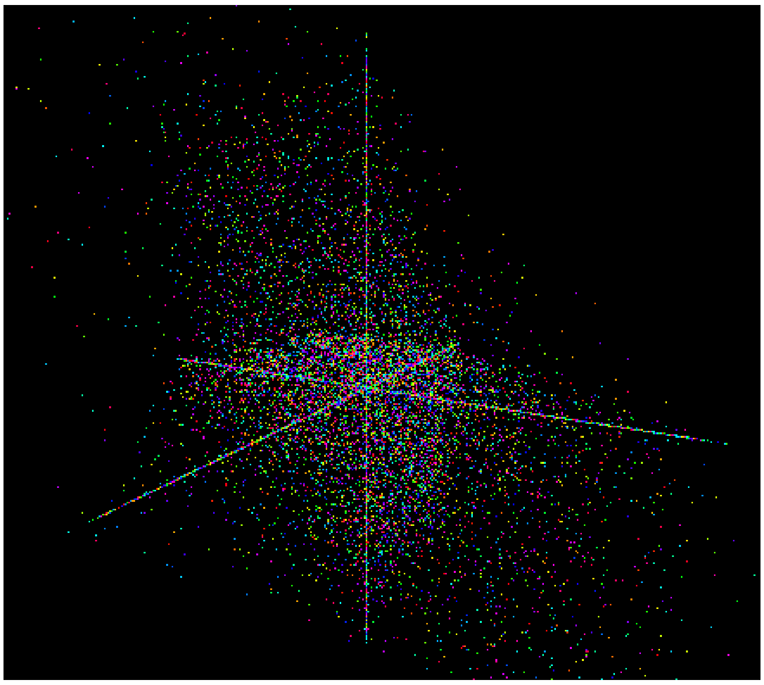


Figure 31 - FreeBSD 4.1 Phase Space

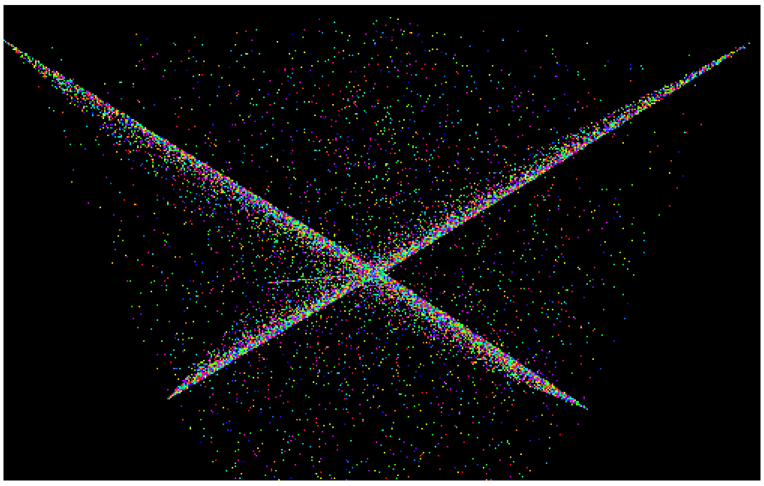


Figure 32 - FreeBSD 4.1 Lattice Space

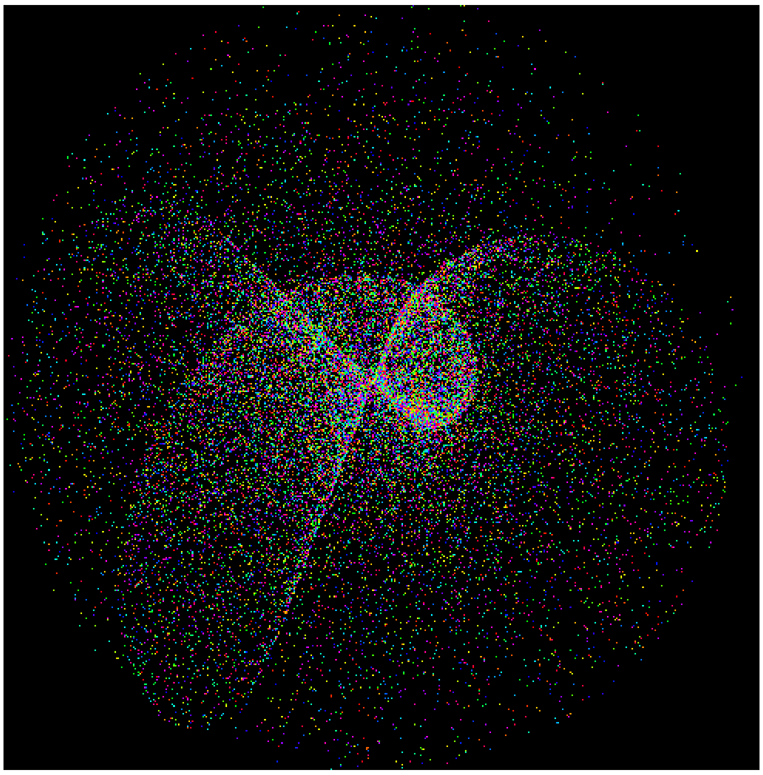


Figure 33 - FreeBSD 4.1 Noise Sphere

As can be seen from Figure 31, all the points lie on the x-y, y-z and x-z planes. While this seems to be a problem, all the points on the planes appear to be randomly distributed. Looking at Figure 32, majority of the points lie on two planes, although points lying on these two planes are evenly distributed with no appearance of attractors forming. Figure 33 once again shows how majority of the points lie on a plane.

5.1.9.2. FreeBSD 5.4

FreeBSD 5.4 produces very similar results as FreeBSD 4.1. This suggests that it uses the exact same TCP stack as 4.1.

5.1.9.3. FreeBSD 6.0

FreeBSD 6.0 produces very similar results as FreeBSD 4.1. This suggests that it uses the exact same TCP stack as 4.1. But this version of FreeBSD is pre-release so its ISN generator might be changed.

5.1.10.Solaris

5.1.10.1. Solaris 5

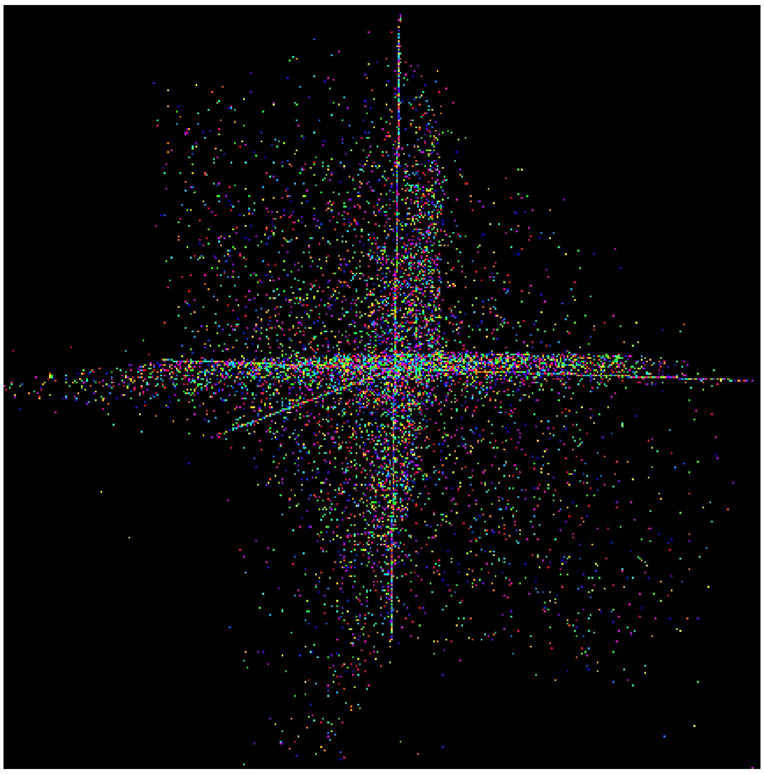


Figure 34 - Solaris 5 Phase Space

Figure 34 shows the ISN's plotted in Phase Space of the Solaris 5 operating system. As can be seen, Figure 34 is very similar to Figure 31 of the FreeBSD 4.1. Also, both the Lattice Space and Noise Sphere images are very similar to the FreeBSD OS. Solaris 5 is most likely using a very similar, if not the same, TCP stack as the FreeBSD machines.

5.1.10.2. Solaris 9

Figure 35 shows the ISN's from the Solaris 9 operating system plotted in Phase Space. The points in this image appear to be randomly distributed much like the control image, Figure 4, except that the points are grouped together in a much smaller region of space. When plotted in Lattice Space, it is found that the ISN generator is complying to RFC 1948 and using a linear method with random increments. Figure 36 shows the ISN's plotted in the Noise Sphere. From this it can be seen that there is a fair amount of randomness in the ISN generator.

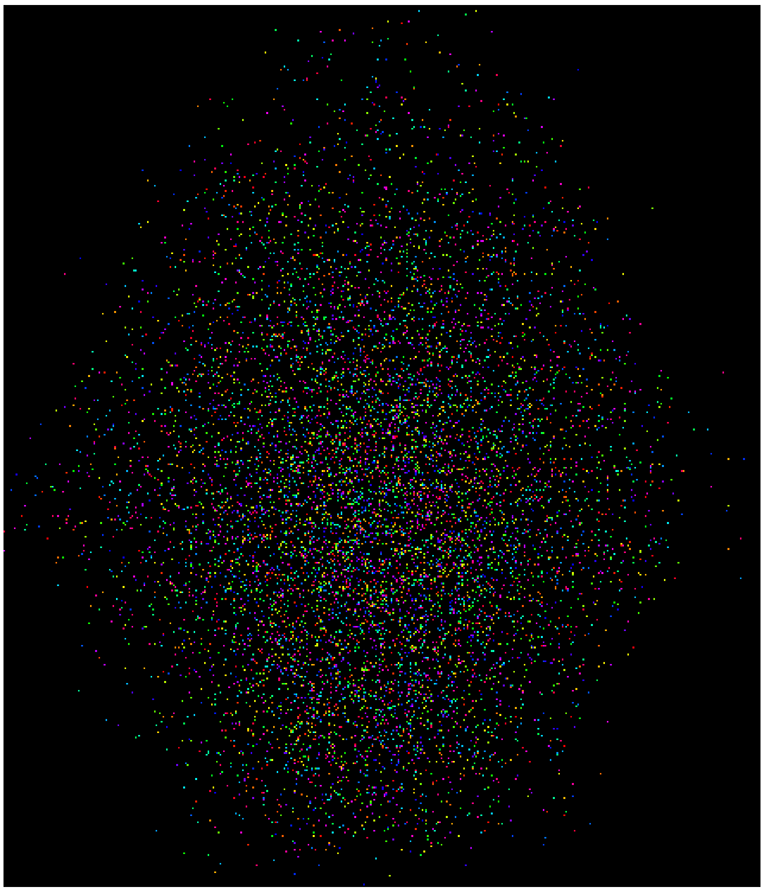


Figure 35 - Solaris 9 Phase Space

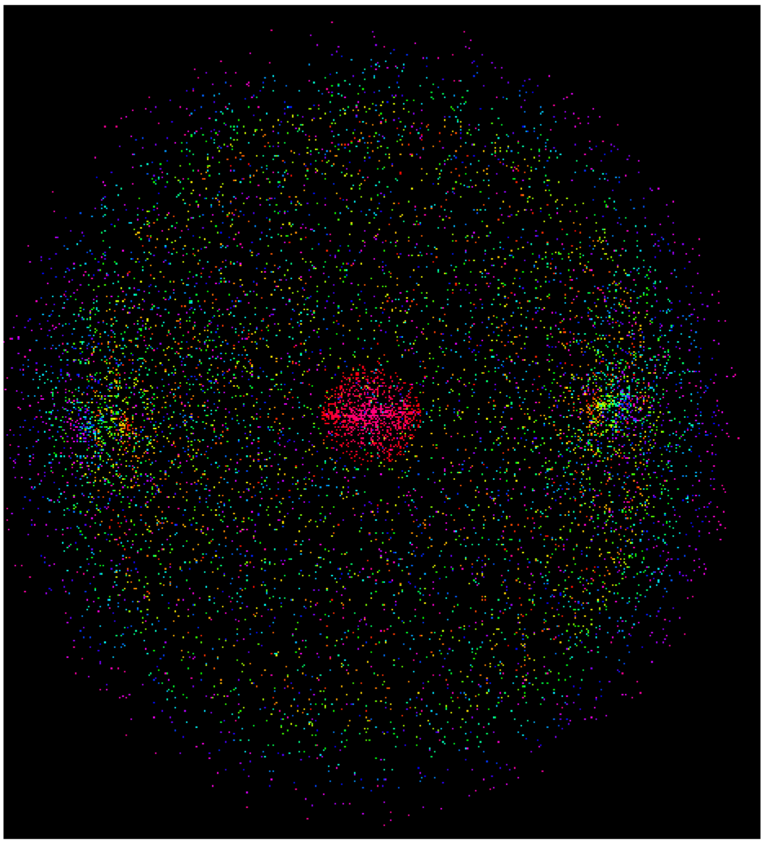


Figure 36 - Solaris 9 Noise Sphere

5.1.11. Paradyne Hotwire 8610

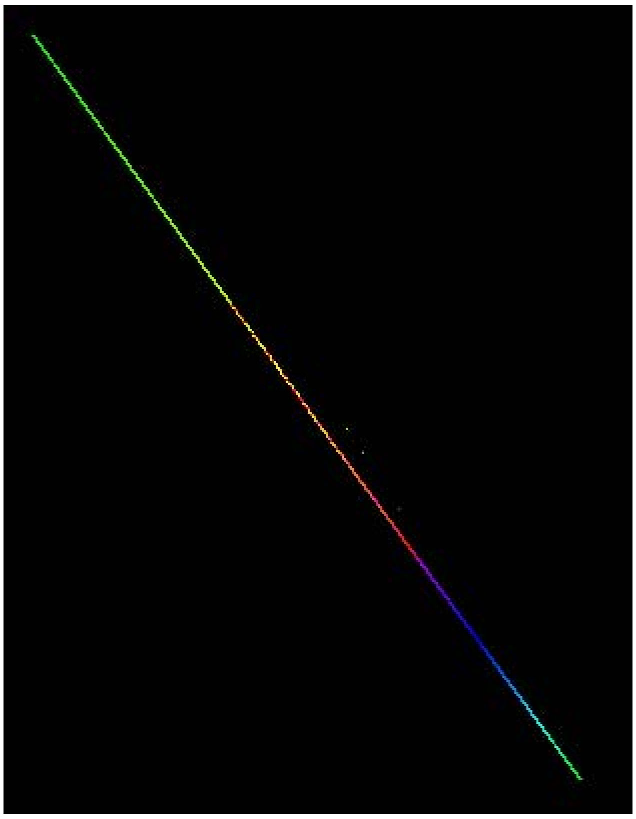


Figure 37 - dslam Lattice Space

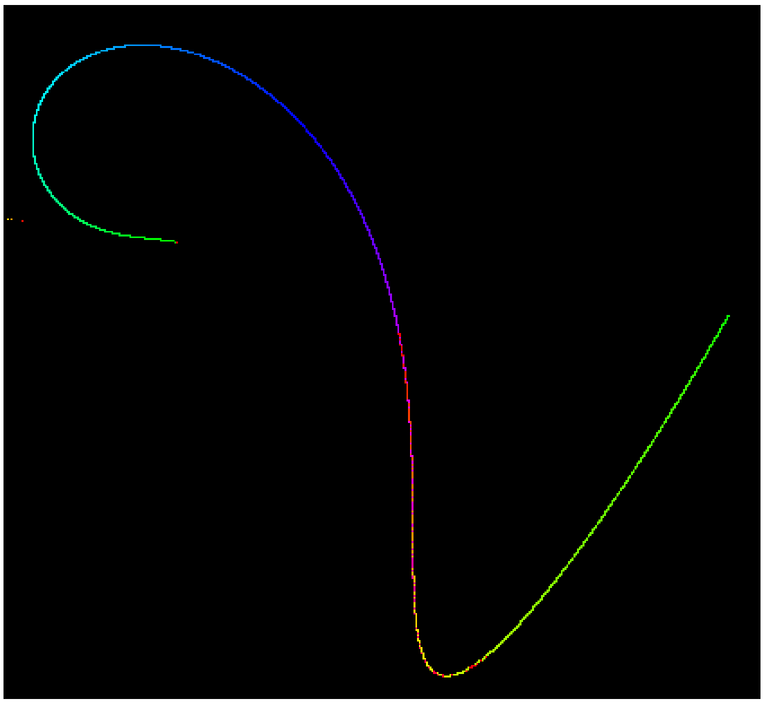


Figure 38 - dslam Noise Sphere

This device is a dslam, which basically converts Ethernet to DSL. The dslam uses a completely linear method of creating ISN's. When the ISN's are plotted in Phase Space only a single point in plotted. The ISN generator being used is obviously still complies with the original TCP recommendations in RFC 793. This can be seen in Figure 37 and Figure 38. The security offered by the ISN generator is very weak and an attacker could target this and easily cause connections to be dropped of perform a spoofing attack.

5.1.12. VoIP Phone

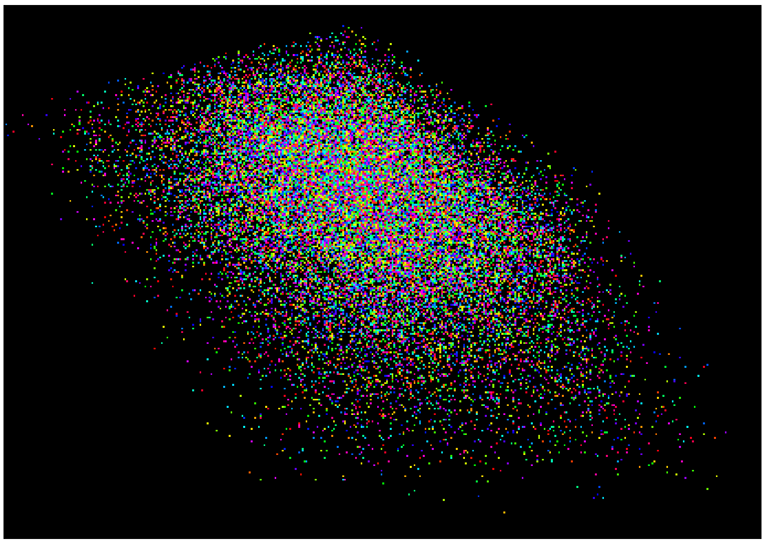


Figure 39 - VoIP phone Phase Space

As can be seen in Figure 39, the Phase Space of the ISN's captured from the VoIP phone show that the ISN generator appears to be using a PRNG. There is a close resemblance to the control image for a good random data source.

Test	**Result**
Frequency (Monobit) Test	FAILURE
Frequency test within a Block	FAILURE
Runs Test	N/A
Test for the Longest Run of Ones in a Block	FAILURE
Binary Matrix Rank Test	FAILURE
Discrete Fourier Transform (spectral) Test	FAILURE

Non-overlapping Template Matching Test	50% FAILURE / 50% SUCCESS
Overlapping Template Matching Test	FAILURE
Maurer’s “Universal Statistical” Test	FAILURE
Lempel-Ziv Compression Test	0% compression
Linear Complexity Test	SUCCESS
Serial Test	FAILURE
Approximate Entropy Test	FAILURE
Cumulative Sums (Cusum) Test	FAILURE
Random Excursions Test	N/A
Random Excursions Variant Test	N/A

Table 7 - NIST test results for VoIP phone

As can be seen in Table 7, the NIST test results show that the PRNG used in the ISN generator is not a statistically good one. This is something that is not evident in the graphical analysis of the image. It is for this case (and many cases like this) where care must be taken in making a conclusion before statistical testing takes place.

5.2. Hardware RNG Results

Table 8 are the results of the NIST test suite from the hardware RNG implemented. While the statistical results are not very promising, the images produced do shows some potential in this device.

Testing the output of hardware random number generator showed quite a few flaws in the generator. Majority of the tests failed, except for Test for the Longest Run of Ones in a Block, Binary Matrix Rank Test and Linear Complexity Test. The results returned from Ent, shown in Table 8, were:

Test	**Result**
Entropy	0.985369 bits per bit
Optimum compression	Reduce file by 1%
Arithmetic mean	0.4289 (0.5 = random)
Monte Carlo value for Pi	3.447587580 (error 9.74 percent)
Serial correlation coefficient	0.082591 (totally uncorrelated = 0.0)

Table 8 – Ent test results for hardware RNG

From the above results one can see that this hardware RNG did deliver a high amount of entropy. These results are promising in the fact that it is not too difficult to implement a hardware RNG. It is however important that the device be properly tested to ensure that sequences produced are in fact random.

Test	**Result**
Frequency (Monobit) Test	FAILURE
Frequency test within a Block	FAILURE
Runs Test	N/A
Test for the Longest Run of Ones in a Block	FAILURE
Binary Matrix Rank Test	SUCCESS
Discrete Fourier Transform (spectral) Test	FAILURE
Non-overlapping Template Matching Test	FAILURE
Overlapping Template Matching Test	FAILURE
Maurer's "Universal Statistical" Test	FAILURE
Lempel-Ziv Compression Test	1% compression
Linear Complexity Test	SUCCESS
Serial Test	FAILURE
Approximate Entropy Test	FAILURE
Cumulative Sums (Cusum) Test	FAILURE
Random Excursions Test	N/A
Random Excursions Variant Test	N/A

Table 9 – NIST test results for hardware RNG

Table 9 shows the results from the NIST test suite. These results are not very promising, but this test suite is very aggressive in the way it tests for randomness in number sequences. If there are any patterns and or correlation in the data, the tests will fail. In the images shown later, one can see that there are in fact patterns/attractor in the data showing why the NIST tests failed. These patterns in the data can come from several sources. The most probable source of noise would be from the 50Hz mains power and the fact that the tests were performed next to a computer which is a big producer of electromagnetic noise.

The Runs Test could not be performed because the NIST test suite returned the following error:

```
“PI ESTIMATOR CRITERIA NOT MET! PI = 4.281440e-001”
```

The Random Excursions Test and Random Excursions Variant Test could not be performed because the NIST test suite returned the following error:

```
“WARNING:  TEST NOT APPLICABLE.  THERE ARE AN INSUFFICIENT
NUMBER OF CYCLES.”
```

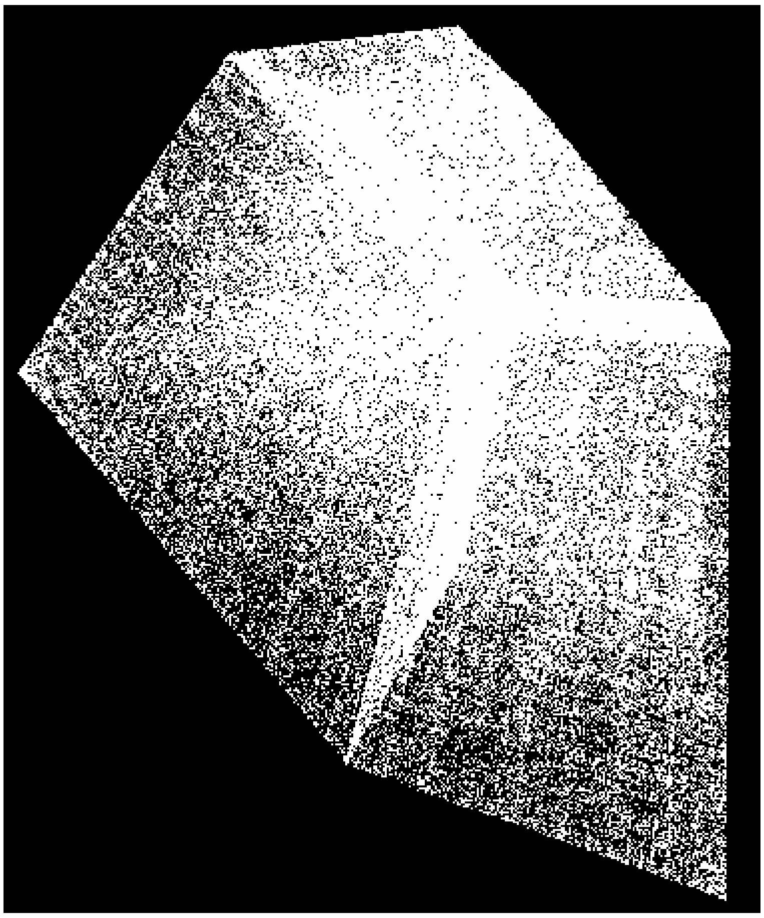


Figure 40 – Hardware RNG Phase Space

Figure 40 shows the image plotted in Phase Space. As can be seen on closer inspection in Figure 41 the points are distributed as can be expected for a true random sequence, except many points lie in planes as can be seen. The red points “clump” in the image results from the fact that these numbers were generated in close succession. For a true random sequence this should not occur. Figure 42 shows the image plotted in Lattice Space. The points in this image are fairly evenly distributed in the cube and the same red “clumps” occurs in this image.

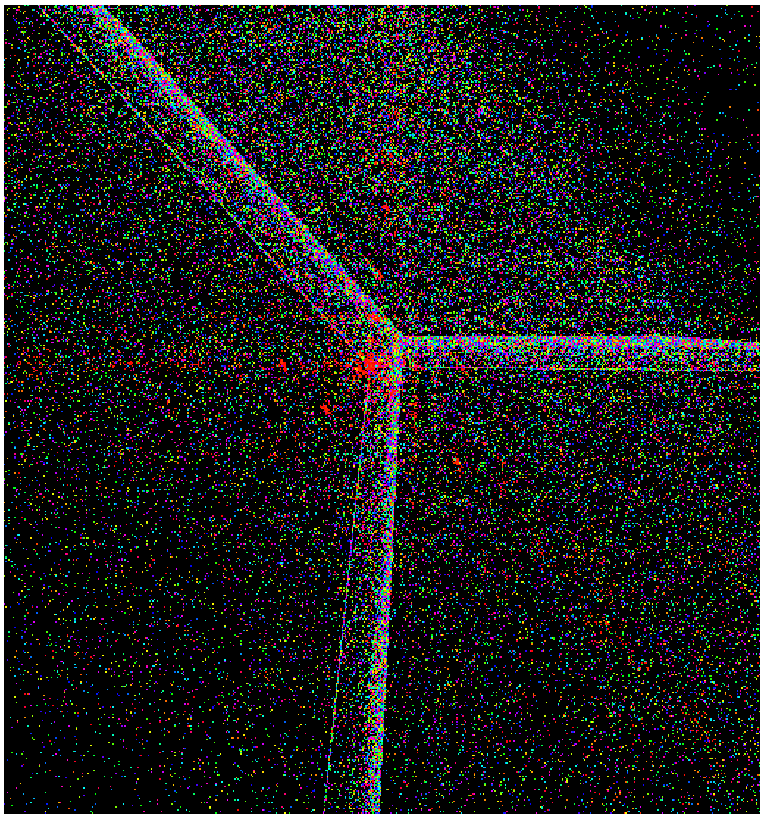


Figure 41 – Hardware RNG Phase Space

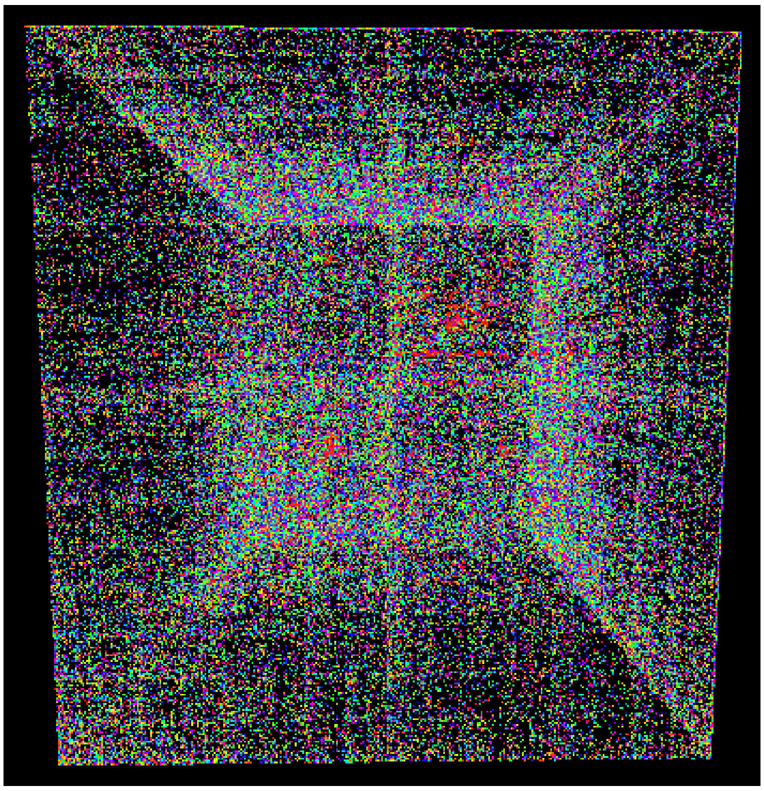


Figure 42 – Hardware RNG Lattice Space

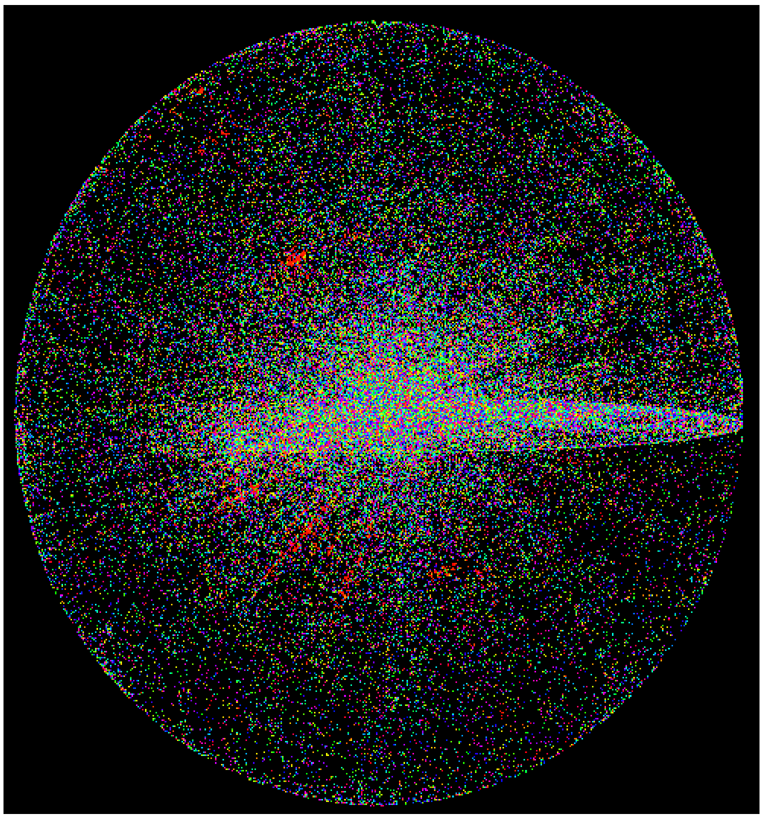


Figure 43 – Hardware RNG Noise Sphere

Figure 43 is the image plotted using the Noise Sphere method. An interesting feature highlighted by this image that is not shown in Figure 42 is that many of the points lie in a plane in the sphere. This hardware RNG has shown many signs of random features, but at the same time there are non-random features. This non-randomness is most likely a result of external interference.

5.3. Summary

This chapter has shown how it is possible to graphically test sequences of numbers for randomness. In particular, the ISN generator from several operating systems were tested. The results have shown that ISN generators complying with RFC 1948 produce ISN's that are easier to predict than the Cisco, FreeBSD and VoIP ISN generators. In most of the graphical analysis it has been shown that when attractors developed in the image, the sequence failed the statistical testing[2]. But when the image did have a very close resemblance to the control image, a definite conclusion could not be made. This was shown in the VoIP phone ISN generator. While the

[2] Appendix B contains a table of results

image did appear to be random, the ISN sequence failed the NIST test suite quite badly.

The implementation of the hardware RNG did reveal some randomness in its output, especially when one looks at the entropy in the results. Although the sequence did fail majority of the NIST tests, the tests were promising in the fact that the true RNG could easily be developed.

Chapter 6
Conclusion

6.1. Summary of Findings

PRNG’s are a critical and integral part of many applications, the use of a poor or sub-standard PRNG can result in poor performance. RFC 1750 proposes a set of guidelines to produce pseudo-random sequences for security applications. It is critical that applications that require the use of a PRNG, use a PRNG that has been shown to produce output that is suitable for the application. For example, the use of a PRNG in a cryptographic application, which is better suited in a Monte Carlo type application, would fail horribly. The security produced by the application would easily be broken.

Chapter 3 introduced a graphical method for testing and evaluating possible random sequences of numbers. This method of testing can be far more rapid than traditional statistical testing. Another advantage of using a graphical method is that the eye can detect patterns in images very quickly and accurately. None or very little background knowledge in statistics is needed when performing graphical testing. Although it is important to keep in mind that interpreting these results can be a very subjective task and different people testing the same sequence can come to different conclusions. That is why statistical tests are still an important role in testing.

The results obtained from testing the ISN’s from different operating systems showed that there is no real standard when it comes to ISN generators. The Windows generators, starting with Windows 95 showed a very linear function being used, resulting in poor security. This continued through Windows 98 and Windows 98 Second Edition. From Windows ME and onwards it can be seen the the ISN generator is complying to RFC 1948. This does result in an improvement of security in terms of preventing ISN type attacks. But the very linear nature of the method RFC 1948 does not make an ISN attack incredibly difficult. All the attacker need to find out is what state the internal clock is in and then use that state, together with his/her own random increment. The attacker would not “guess” the correct ISN immediately, but “guessing” it would be easier than guessing the ISN produced by a PRNG alone.

The Cisco ISN generator proved to very a very good PRNG. The images closely resembled those of the control images and it passed majority of the NIST test suite tests. This type of method of generating ISN’s makes it highly unlikely that a hacker would be able to predict the ISN’s, hence making the system secure, in terms of its ISN generator. The VoIP phone produced ISN’s that looked very similar to the control images yet failed most of the NIST statistical tests. This shows that when attractors and/or patterns are produced in the image, that statistical tests will most likely fail. But, when a sequence of numbers appears to be random, statistical testing should be used to avoid false positives.

The hardware RNG implemented was successful in the fact that it produced results that showed some level of randomness. Graphically it showed signs of randomness with many of the points being evenly distributed. The fact that attractors formed in the graphical analysis did show that there were signs of non-randomness. In a hardware generator, this non-randomness can be introduced by many external sources which bias the output.

6.2. Future Work

Possible extensions to this work could be to implement a method to detect clustering in the three dimensional space. This will help find and identify attractors in the graphical analysis.

Also, using the open source Linux system, implementing a PRNG as the ISN generator and conducting performance tests and comparing the results with other methods of generating ISN’s.

References

[Atkinson] - Atkinson, A. C., *Tests of Pseudo-random Numbers*, Imperial College, Applied Statistics, Vol. 29, No. 2 (1980), pages 164-171

[Atreya] - Atreya, M., RSA Security article: *Intoduction to Cryptography* (part 4): *Pseudo Random Number Generators*, November 2000

[Bellare] - Bellare M., Goldwasser S., Micciancio D., *``Pseudo-Random" Number Generation within Cryptographic Algorithms: the DSS Case,* Advances in Cryptology - Crypto 97 Proceedings, Lecture Notes in Computer Science No. 1294. Springer-Verlag, 1997.

[Blum] - Blum L., Blum M., Shub M., *A Simple Unpredictable Pseudo-Random Number Generator,* SIAM Journal on Computing, volume 15, pages 364-383, May 1986

[Bowman] - Bowman, R. L., *Evaluating Pseudo-Random Number Generators,* Computer & Graphics, Vol. 19, No. 2, Pages 315-324, 1995

[Buslenko] - Buslenko, N. P., Golenko D. I., Schreider, Y. A., Sobol', I. M., Sragovich, V. G., *The Monte Carlo Method – The Method of Statistical Trials*, 1966

[CERT] - CERT® Advisory CA-2001-09, *Statistical Weaknesses in TCP/IP Initial Sequence Numbers,* online: http://www.cert.org/advisories/CA-2001-09.html, May 2001

[Chambers] - Chambers, J. M., Cleveland, W. S., Kleiner, B., Tukey, P. A., *Graphical Methods for Data Analysis*, 1983

[Chaitin] - Chaitin, G.J., *Randomness and mathematical proof*, 1975, Science America Vol. 232, No. 47

[CIAC] - Computer Incident Advisory Capabilty , *Information Bulletin ,* http://www.ciac.org/ciac/bulletins/l-003.shtml

[Cisco] - Cisco Security Advisory, Online: http://www.cisco.com/warp/public/707/ios-tcp-isn-random-pub.shtml, October 2004

[Connor] - Connor, F. R., *Noise – Second Edition*, 1982

[Dobb's] - Dr. Dobb's Journal, *Randomness and the Netscape Browser*, January 1996, Online: http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html, January 1996

[Entacher 1998] - Entacher, K., *Bad Subsequences of Well_known Linear Congruential Pseudorandom Number Generators*, January 1998, University of Salzburg, ACM Transactions on Modelling and Computer Simulation, Volume 8, Number 1, Pages 61-70

[Entacher 2000] - Entacher, K., *Linear Congruential Generator: LCG*, Online: http://crypto.mat.sbg.ac.at/results/karl/server/node4.html, June 2000

[Eric] - Eric W. Weisstein, *Attractor, MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/Attractor.html

[Ferguson] - Ferguson, N., Schneier, B., *Practical Cryptography*, 2003, pages 155-184

[Ghausi] - Ghausi, M. S., *Electronic Circuits*, New York University, 1971

[Golder] - Golder, E. R., *Algorithm AS 98: The spectral Test for the Evaluation of Congruential Pseudo-Random Generators*, Royal Statistics Society, Applied Statistics, Vol. 25, No. 2 (1976), pages 173-180

[Herring] - Herring, C., Palmore, J. I., *SIGPLAN Notices, Vol. 24, No. 11*,

[Hoglund] - Hoglund, G., *Exploiting Software – How to break codes*, 2004

[Israelsohn] - Israelsohn, J., *Noise 101*, Technical Editor – EDN, 2004, http://www.edn.com/article/CA371088.html

[Jansson] - Jansson, B., *Random Number Generators*, 1966

[Kelsey] - Kelsey, J., Schneier, B., Ferguson, N., *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator*, Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, August 1999.

[Knuth] - Knuth, D. E., *The Art of Computer Programming*, Volume 2, 1969

[Liu] - Liu, J. S., *Monte Carlo Strategies in Scientific Computing*, Springer Series in Statistics, 2001

[Marsaglia] - Marsaglia, G., *Seeds for Random Number Generators,* Communications of the ACM, Vol. 46, No 5, May 2003

[Matsumoto] - Matsumoto M. , Nishimura T., *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, January 1998, Keio Universit, ACM Transactions on Modelling and Computer Simulation, Volume 8, Number 1, Pages 3-30

[Michael] - Michael Zalewski, *Strange Attractors and TCP/IP Sequence Number Analysis,* 2001, BindView Corporation - www.bindview.com/

[Morris] -Morris, R.T. 1985. *A Weakness in the 4.2BSD UNIX TCP/IP Software*. Technical Report No. 117, AT&T Bell Laboratories, 1985

[Network] - Network Working Group, *Randomness Recommendations for Security*, RFC 1750, December 1994

[NIST] - Rukhin, A., Soto J., Nechvatal J., Smid M., Barker E., Leigh S., Levenson M., Vangel M., Banks D., Heckert A., Dray J., Vo S., *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.,* NIST Special Publication 800-22, 2001 (with revisions dated May 15, 2001)

[Park] - Park, S. K., Miller, K. W., *Random Number Generators: Good Ones Are Hard to Find*, Communications of the ACM, October 1988, Volume 31, Number10

[Pickover] - Pickover, C.A., *Picturing Randomness on a graphics supercomputer*, IBM J. Res. Develop. 35, Pages 227-230, 1991

[RFC 793] - Information Sciences Institute University of Southern California, *Transmission Control Protocol*, RFC 793, September 1981

[RFC 1750] - Network Working Group, *Randomness Recommendations for Security*, RFC 1750, December 1994

[RFC 1948] - Network Working Group, *Defending Against Sequence Number Attacks*, RFC 1948, May 1996

[Schneier] - Schneier, B., *Yarrow PRNG – Press Release*, Online: http://www.schneier.com/yarrow-pressrel.html, 17 June 1998

[Walker] - Walker, J., *Ent - A Pseudorandom Number Sequence Test Program*, Online: http://www.fourmilab.ch/random/, October 1998

Appendix A – Glossary of Acronyms

ISN	- Initial Sequence Number
LCM	- Linear Congruential Method
LCG	- Linear Congruential Generator
PRNG	- Pseudo Random Number Generator
OS	- Operating System
RFC	- Request for Comments
RNG	- Random Number Generator

Appendix B

Test	**Hardware RNG**	**Windows 98SE**	**VoIP Phone**	**Win 2003 Server**	**Fedora Core 3**	**Windows XP SP2**	**Cisco**
Frequency (Monobit) Test	✗	✗	✗	✗	✗	✗	✓
Frequency test within a Block	✗	✗	✗	✗	✗	✗	✓
Runs Test	N/A	N/A	N/A	N/A	N/A	N/A	✓
Test for the Longest Run of Ones in a Block	✗	✗	✗	✗	✓	✓	✓
Binary Matrix Rank Test	✓	✗	✗	✗	✗	✗	✓
Discrete Fourier Transform (spectral) Test	✗	✗	✗	✗	✗	✗	✓
Non-overlapping Template Matching Test	✗	✗	✗ / ✓	✗	✗	✗	✓
Overlapping Template Matching Test	✗	✗	✗	✗	✗	✗	✓
Maurer's "Universal Statistical" Test	✗	✗	✗	✗	✗	✗	✓
Lempel-Ziv Compression Test	1% compression						0% compression
Linear Complexity Test	✓	✓	✓	✓	✓	✓	✓
Serial Test	✗	✗	✗	✗	✗	✗	✗
Approximate Entropy Test	✗	✗	✗	✗	✗	✗	✗
Cumulative Sums (Cusum) Test	✗	✗	✗	✗	✗	✗	✓
Random Excursions Test	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Random Excursions Variant Test	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 10 – Comparison of NIST results for different OS's ISN generators

✗ = Failure

✓ = Pass