RANDOMNESS PROPERTIES OF CRYPTOGRAPHIC HASH FUNCTIONS

Approved by:

Dr. Theodore Manikas

Dr. Jennifer Dworak

Dr. Eric Larson

Dr. Suku Nair

RANDOMNESS PROPERTIES OF CRYPTOGRAPHIC HASH FUNCTIONS

A Thesis Presented to the Graduate Faculty of the

Lyle College: School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Masters of Science

with a

Major in Computer Engineering

by

Micah A. Thornton

(B.S. Statistics, Southern Methodist University, 2017)
(B.S. Computer Engineering, Southern Methodist University, 2017)

-

ACKNOWLEDGMENTS

I want to thank my advisor and friend Dr. Theodore Manikas, for his patience and assistance in the preparation of this work. I would like to thank my family for all of their love and support. I would also like to thank the members of my Committee.

Thornton , Micah A. B.S. Statistics, Southern Methodist University, 2017
B.S. Computer Engineering, Southern Methodist University, 2017

Randomness Properties of Cryptographic Hash Functions

_

Advisor: Professor Theodore Manikas

Masters of Science degree conferred -

Thesis completed -

The work in this thesis seeks to answer the following: Assuming a cryptographic hash is being used to increase the apparent randomness of a data set, is it possible to formulate metrics to choose the best hash for this purpose? Towards this ends standard metrics provided by the ENT utility (entropy and serial correlation) were analyzed in conjunction with different cryptographic hash functions, and the results of several statistical analysis on the metrics are presented. The work in this thesis concludes that the hypothesis holds, and suitable metrics were formulated and verified. As a side experiment, a new entropy extractor was formulated and tested.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

For Isaac

R.I.P (1995-2017)

Chapter 1

INTRODUCTION

1.1. Overview

A Cryptographic hash function is a function that is easy to compute over the members of the domain, but is sufficiently hard (non-poly time) to compute the inverse on the members of the co-domain of the function. The number of collisions in the co-domain (i.e. hashes of different values that are the same) must also be minimized for the creation of a useful cryptographic hash. The body of work presented in this thesis examines the randomness properties of cryptographic hashes by monitoring their effect on random values. In recent years there has been an up tick in the amount of research that has gone into the development of Cryptographic hashing algorithms. Likely owing to competitions to name a successor to the SHA-2 family [15] of hash functions [27]. The existence of cryptographic hash functions has also been shown to be necessary and sufficient for the existence of pseudo-random number generators. [19]

A good starting place is to examine the extant random number generators. This will give us a feeling for the manners in which cryptographic hash functions and random number generators differ. It can also illuminate the similarities random number generators and cryptographic hash functions have. We will then move into the statistical analysis of randomness properties produced by cryptographic hash functions.

The driving hypothesis behind the work presented in this thesis is the following: Assuming a cryptographic hash is being used to increase the apparent randomness of a data set, it is possible to formulate metrics to choose the best hash for this purpose.

The conclusion was that the hypothesis holds, and suitable metrics are formulated and verified.

1.2. Random Number Generators

A Random Number Generator (RNG) is any device or algorithm that allows the user to consistently generate numbers that are independent from one another. Because most of the devices we build are deterministic however, we must attempt to simulate this randomness, and to generate numbers that are seemingly independent but that are truly related through some complex algorithm that is the constitution of the RNG. We call Random Number Generators that are inherently deterministic behind the scenes Pseudo-Random Number Generators (PRNGs).

This definition of a pseudo random number generator extends to all such generators implemented using software or hardware algorithms on modern computers. There also exists a group of random number generators that are known as true random number generators or TRNGs, these are generators that rely on natural processes that can produce a seemingly non-deterministic set of values. An example of a TRNG is a device that captures measurements of radiation from americium. The radiation is captured by a sensor and then some deterministic process is applied to the captured data in order to return a number. This number is calculated as a function of the non-deterministic data which implies that it inherently is non-deterministic.

The work discussed in this thesis is relevant to all RNGs, and helps to provide some context for their improvement in light of the generalized leftover hash lemma. It is worth noting that we are not attempting to compete with true random number generators, and that hashing the output of pseudorandom generators is simply a manner in which randomness qualities can be improved. We are then using the improvement of randomness qualities as a feature on which to classify hash functions, and potentially to describe classes of hash functions. The work in this thesis serves as the groundwork of differentiation, and as such analysis of variance and multiple comparison post-hoc procedures are used to determine whether there are statistically significant differences in the mean improvement of qualities based on the particular hash function that is used.

1.2.1. Linear Feedback Shift Register (LFSR)

One such example of a pseudo-random number generator is known as a linear feedback shift register. Because of their simple and cheap design, these devices are often used when a hardware PRNG is desired. The manner in which an LFSR works is simple, it contains a bank of memory elements that is n bits long. Several of the bits in the device are exclusive ored, and shifted into the leftmost position. This is why we use the word 'feedback' in the name. Every cycle, the contents of the register are shifted one bit position and the bit that is shifted in is calculated based on the exclusive or of several of the positions in the memory bank, these positions are known as the taps of the LFSR.

If the taps are chosen in a very specific way depending on the register length and a primal polynomial (That is, it cannot be further reduced into constituent polynomials, hence it is primal like a prime number), then the LFSR is considered a maximal-length LFSR. A maximal-length LFSR is a LFSR that does not repeat any of its internal values for $2^n - 1$ cycles. Typically when creating a random number generator there is an understanding that the LFSR will be maximal length. If the LFSR is not maximal length then it is possible that the internal value contained within the memory elements of the LFSR will repeat. This repitition causes the stream of numbers produced to appear less random, and indicates that there is a bias towards the number which occurs more than once.

For an LFSR there is temporal correlation, it is obvious to an observer that the output of the LFSR is shifted to left once per cycle. To obfuscate the shifting behavior when using LFSRs as random number generators, one possibility is to wait for the LFSR to cycle all previous bits before capturing the next value. The work in this thesis suggests that there is a different and far less time consuming technique that can be applied in order to achieve a comparable increase in the apparent randomness of the LFSR, which involves using cryptographic hash functions as a post-step.

1.2.2. Linear Congruential Generators (LCG)

The linear congruential generator is a device first proposed in the 1940's by Lehmer. Lehmer defined a method where a stream of random integers could be calculated using a mathematical formula. The formula takes advantage of modular arithmetic to form a function whose outputs are periodic, when the input is a seed applied to the function one time at the beginning. In much the same manner as Moore and Mealy machines, the LCG, and other random number generators for that matter, are high-bred contraptions, that use principles from both types of machines to determine the next output. For instance, there is a single external input at the beginning of the computation called the seed. The seed is then iterated upon, and the internal components of the device are updated, producing the next value. In this way the random number generator is part Mealy machine and part Moore machine. The LCG uses principles that are discussed in detail in Knuth's seminal work on the matter called The Art of Computer Programming: Semi-Numeric Algorithms. [21]

1.2.3. Generalized Feedback Shift Registers (GFSR)

Along the same vein as a linear feedback shift register, the general feedback shift register is at its core a shift register with taps. The taps are given according to certain properties that help to provide the longest possible string of pseudorandom numbers. The taps in a GFSR are not fed back through an addition modulo-two, or exclusive-or gate, they are instead fed back through other fundamental gates. The key difference between an LFSR and a GFSR is that the GFSR has gates other than and in addition to the gates that are contained in the LFSR, and hence can be used to realize non-linear functions.

1.2.4. True Random versus Pseudo-Random Number Generators

Although the topic of the existence of truly random numbers is strongly debated, that does not prevent their use in modern science and mathematics. If a RNG device generates a stream of numbers that can be accurately predicted given all prior information (RNG seed, algorithm) than the device is said to be a pseudo-random number generator as opposed to a true random number generator. The deterministic nature of computers (on which pseudo-random generators run) implies that the pseudo-random generators are not truly random.

In practice, most random number generators that are based on natural phenomenon are considered true random number generators. Those devices that are a pure and simple mechanical process applied to an initial value (a seed) are considered Pseudo-Random Number Generators (PRNGs).

I mentioned briefly above that we design computers in a manner where they are very strongly deterministic, that is not to say that all computers are designed this way, probabilistic and quantum computers are two examples of devices that do not compute in a deterministic manner. This statement was meant to be indicative of the modern personal computer. In short for now though, most modern computers do in-fact consume random numbers at a standard rate due to Operating System security enhancements.

This discussion would not be complete without a mention and explanation of entropy. Entropy is a concept that originally comes from thermodynamics. Essentially, in the sciences, the word entropy defines energy that is given off in a reaction in a form where it cannot be utilized, and hence is lost to the atmosphere. By using random numbers we are actually using the *random* energy floating through the universe to allow for better security, or simulation results. Entropy is discussed further in a mathematical sense later, but at its core is a measure of how much information is contained by a value, or in a list of values.

The concept of true randomness has plagued the minds of philosophers for centuries, as is evidenced by the scientific reaction to the theory of determinism presented in Laplace's Demon. At the very early turn of the nineteenth century, French mathematician Pierre-Simon Laplace postulated the existence of a theoretical entity in his work *Essai philosophique sur les probabilités*. Later known as 'Laplace's Demon', the postulated entity was one of vast capabilities. No information is beyond this entities grasp, it knows all information while simultaneously possessing the ability to analyze them and predict the outcome of any process.

The thought behind this relies on the infallibility of the physical laws of the universe (known and unknown), as a source of analysis for the entity. The heavily related philosophy of determinism in the universe has profound and far reaching implications, in some extreme cases one may even see this as evidence of no free-will. The notion of determinism underpinning Laplace's Demon sparked a breadth of responses from the scientific community, one such response was given in 2008 by David Wolpert, where he responded with a Cantorian set theory proof that for any such entity to exist it would inherently contain the universe within.

What it means to be *truly random* is also a subject of debate, in *A Primer on Pseudorandom Generators*, Oded Goldreich presents three of the leading arguments on what it means for something to truly be random.

The first notion he presents is that of Claude Shannon, which handles frequency distributions to arrive at a characteristic distribution; the uniform. In the second theory presented by Goldreich, he invokes concepts from Kolmogorov complexity theory. In this theory it is said that the complexity of an object can be assigned a metric in terms of the shortest computer program that can recreate that same object. Hence, the longer the program needed to recreate the object, the more complex it is.

It is due to both of these concepts that many of the tests for randomness exist, and what allows us to measure random number generators. Goldreich presents a third argument, which is relative to the observer. In this argument he states that an event is considered only random if a person does not have sufficient computational abilities to predict the outcome ahead of time. From this perspective, the roll of a dice may not be considered random if one could measure its trajectory and predict the outcome using a powerful computer before it lands. This interpretation is also slightly absurd in some instances however, as it may suggest for instance that the outcome of the question 'What is the derivative of x^2?' is truly random with respect to a person to whom the question was unanswerable.

The work presented in this thesis makes use of all three definitions of randomness in a manner to ascertain whether applying a certain procedure to extant generators is beneficial in producing a seemingly more random stream of numbers.

1.2.5. Cryptographically Secure Pseudo-Random Number Generators

In recent times there has been a shift to using only what are known as Cryptographically Secure Pseudorandom Number Generators (CSPRNGs), which imposes additional criteria on pseudorandom number generators that causes them to be deemed secure enough for use in cryptographic applications. There are two primary distinctions for the CSPRNGs are the satisfaction of the following requirements.

1. The random number generator must satisfactorily pass a suite of statistical tests designed to assess random number generators. This requirement was shown to be equivalent to the requirement of passing the Yao test in 1982. The Yao test, also known as the 'next-bit test', which states simply: To an attacker examining a pseudorandom number generator for which the first i bits of output are known, it must be impossible with reasonable computational power to predict the (i+1)st element.

2. In much the same manner as the PRNG must show forward security with the first point, it must show backwards security in that given any arbitrary sequence of bits after the first n bits, non of the previous bits can be predicted.

If a pseudorandom generator is to meet both of the above criteria then it is deemed a cryptographically secure pseudorandom number generator or CSPRNG.

1.3. The definition of Random

Before proceeding into the body of the research that was performed and will be presented in this thesis, a theoretical discussion of the nature of randomness should be undertaken to show the philosohpical underpinnings of the work that is presented herein. Defining the word 'random' is a very difficult if not impossible task. If you consider the essence of what the word stands for one might equate it with synonyms such as disorder, chaos, entropy, and dissolution. However at the heart of the issue these words become almost as immaterial as we struggle to helplessly define these words in the context of what is true in nature and reality.

The Oxford English dictionary defines the word random as "Governed by or involving equal chances for each of the actual or hypothetical members of a population; produced or obtained by a process of this kind (and therefore completely unpredictable in detail)." The word itself is actually derived from the old French: '*randon*' which meant great speed. Interestingly the word made its way into middle English in its current form but meaning ' an impetuous headlong rush'. Finally in modern English, the noun takes its familiar meaning as shown above. Although no definition of the word can truly describe the essence of what it means, there is much to be gained by looking at where the word came from, the deepest roots of the word are actually Germanic. In german, the root word *Der Rand* is an edge, border, or verge.

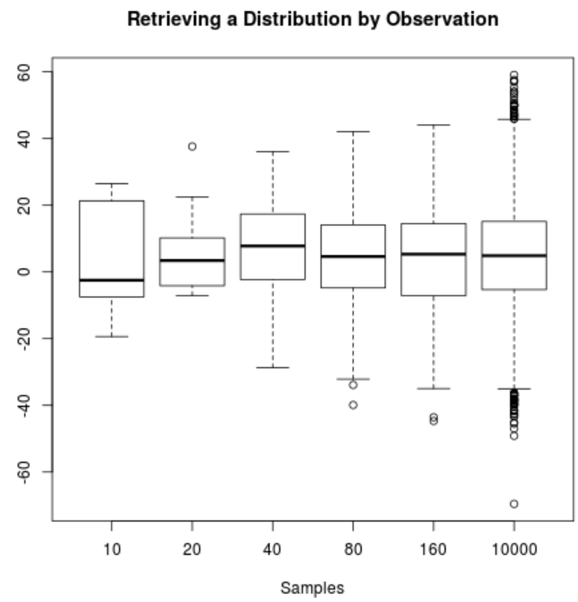
Because of their necessity in a wide variety of different applications (security and simulation are not the least of which) random number generation techniques have been refined over a long period of time into their current distilled state. Since time before written history human beings have had an innate respect for and complicated relationship with random numbers. As evidenced by ancient artifacts used by shamans, such as bone disks, and Yarrow sticks.

It is interesting that those in the village who were allowed to handle the use of random numbers in predictions were also some of the most respected. This indicates that there has been some sort of internal reverence for random numbers instilled in human beings. This reverence is still latent in modern day applications. For instance, in many security applications a random number generator is used to create a key. We use these random numbers to secure our most important information. Another major modern usage of random numbers is in testing and simulation, we use random numbers all the time for Monte Carlo simulations, as well as creating truly random samples.

As we have gained more knowledge as a species we have found many new ways of exploiting random numbers in scientific pursuits through the field of statistics. We now know that certain phenomena, both man-made and natural can be shown to follow certain distributions. This was really a natural step when it came to making random observations. After enough time, and with enough observations certain phenomenon seem to produce results which are oddly predictable. As is seen in the Figure 1.1 below, with enough observations, the distribution of values attained becomes more recognizable.

By examining the figure above we note that each subsequent sampling distribution more closely approximates the next sampling distribution, than the prior did

Figure 1.1. Drawing samples Reveals an underlying distribution



it (excluding 10 and 10000 samples). The data in the above plot was taken from the R-language built in RNG. This and many other modern language generators are based upon the Mersenne Twister discussed in the last chapter. At this point it is important to note that one can specify a distribution from which to draw random data in R. For the above example the data was taken from a normal distribution with mean 5 and standard deviation 15.

This distribution was completely specified by Carl Gauss. Formally stated, the frequencies of occurrence for values obtained from normally distributed process with known mean (μ), and variance (σ^2) are given by the following function of said value:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1.1}$$

Taking the above equation and inserting our known mean of 5, and standard deviation of 15 we arrive at the characterizing equation for the frequencies of occurrence of seemingly random values from Figure 1.1.

$$f(x) = \frac{1}{15\sqrt{2\pi}} e^{-\frac{(x-5)^2}{450}} \tag{1.2}$$

The function f(x) is known as the probability density function (pdf), and assumes a measure known as the relative frequency, when supplied with a given value. The relative frequency (a value between 0 and 1) indicates the probability of occurrence of the specified value. When enough observations are made that the distribution can be characterized by an underlying pdf there are many implications, especially in the realm of security.

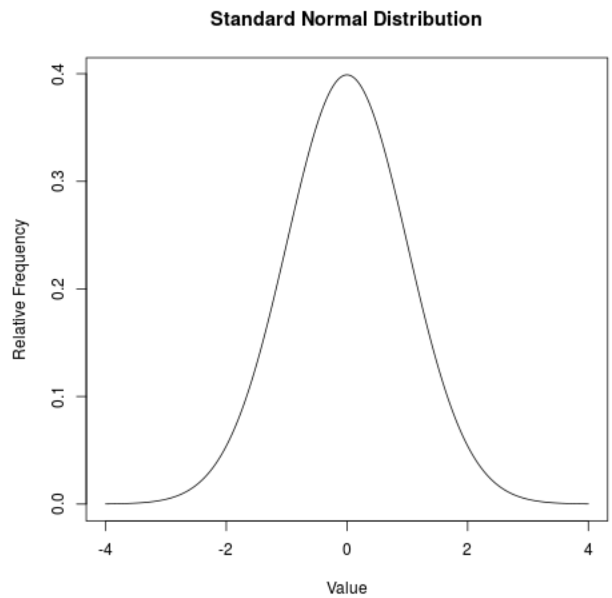
Because in distributions like the one above, certain values (discrete) or ranges of values (continuous) are more likely to occur than others we can become confident about what the next observation will be. The standard normal pdf is given in Figure 1.2.

So certain random process are not nearly as random as they may at first appear, in fact we can say with confidence how many times a certain value will appear given a sample size. If we were constructing a random number generator, we would want the user to be unable to predict, or in any way be confident about the value that will be observed. A somewhat natural way to do this would be to allow multiple values to have the same frequency of occurrence. An example of such a distribution is shown in Figure 1.3

Another natural extension would be to have three or more values with the same probability of occurrence. Figure 1.4 is technically a trimodal distribution, however we note that there is a flattening between the first two modes.

This gives us many values whose frequencies of occurrence are very similar, this is a desirable property in random number generators. The ultimate conclusion of this

Figure 1.2. Standard Normal Distribution for Reference



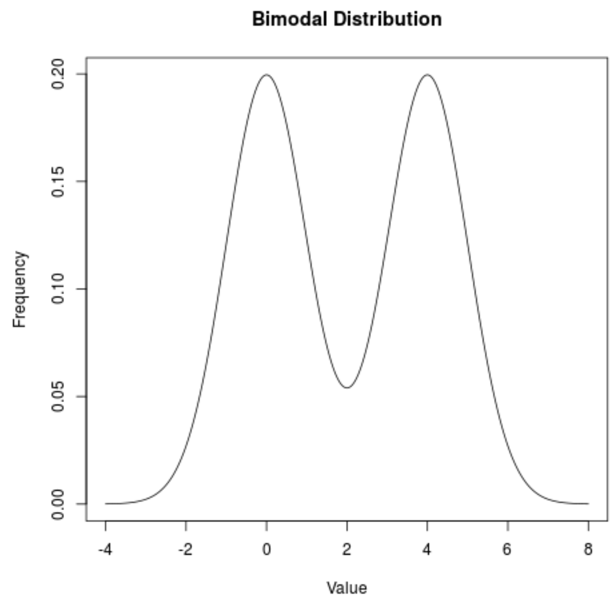
line of reasoning would be to construct a device which produced a range of numbers with equal probability. Such a distribution is characterized by an entirely flat pdf, as shown in Figure 1.5.

The uniform distribution is characterized by two values, the minimum and maximum values. With the minimum (a) and maximum (b) values, the distribution is characterized by the simple pdf:

$$f(x) = \frac{1}{b - a} \tag{1.3}$$

This concept is a hugely motivating factor behind the design of modern RNGs, and is also a key component in the design of tests and analysis techniques for random number generators. In the following section we begin an extensive review of some of the most, and least successful RNGs, as well as modern techniques. This will help to

Figure 1.3. Bimodal Normal Distribution for Reference

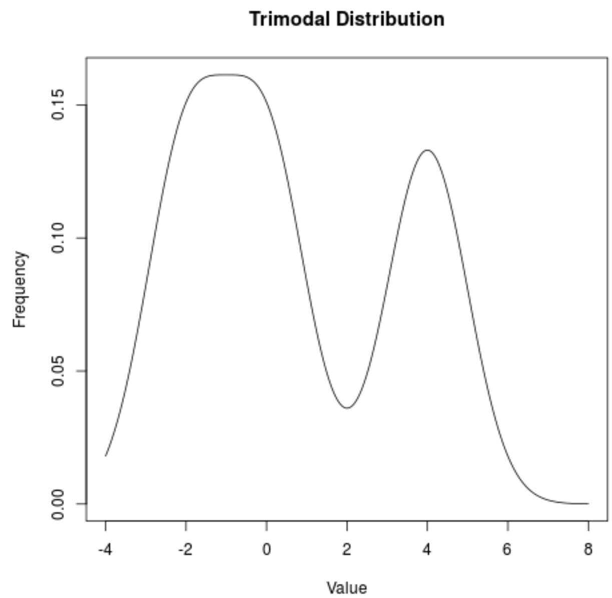


set the mood and contrast for the methodology provided in this thesis which aims at shedding light on a technique that may improve certain RNG test scores.

1.3.1. Lack of Knowledge

In *A Primer on Pseudorandom Generators* Oded Goldreich discussed three principle ways that randomness could be defined. In his book Goldreich mentions that the first theory of randomness has to deal with the quantification of *uncertainty*. The first theory presented is attributed to the famous computer scientist and mathematician Claude E. Shannon. The second theory of randomness has to do with the efficient representation of information. Some of the key researchers behind this theory were Solomonoff and Kolmogorov. The last theory presented by Goldreich as the central theory of his book presents randomness as a concept which is relative to a specific

Figure 1.4. Trimodal Normal Distribution for Reference

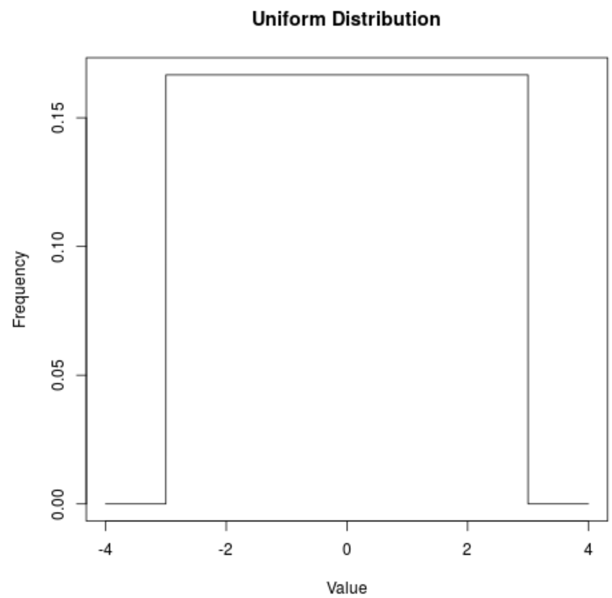


observers abilities. We discuss each of these three philosophies of randomness in turn, starting with the first here.

The first definition is attributed to a lack of knowledge about a system. This definition of randomness is related to the concept of a random variable. Random variables from statistics are defined as variables that have valuations for which the chances are distributed according to some function of parameters. For instance, one of the most famous probability distributions is the Gaussian, or Normal distribution, its frequency distribution (function attributing chances dependent on the parameters $\mu = 0, \sigma = 1$) is shown in Figure 1.6.

The first definition of randomness deals exclusively with distributions in the manner that they appear above. It is important to remember that in this approach the definition of pure randomness is dependent on a specific distribution. A distribution

Figure 1.5. Uniform Distribution for Reference

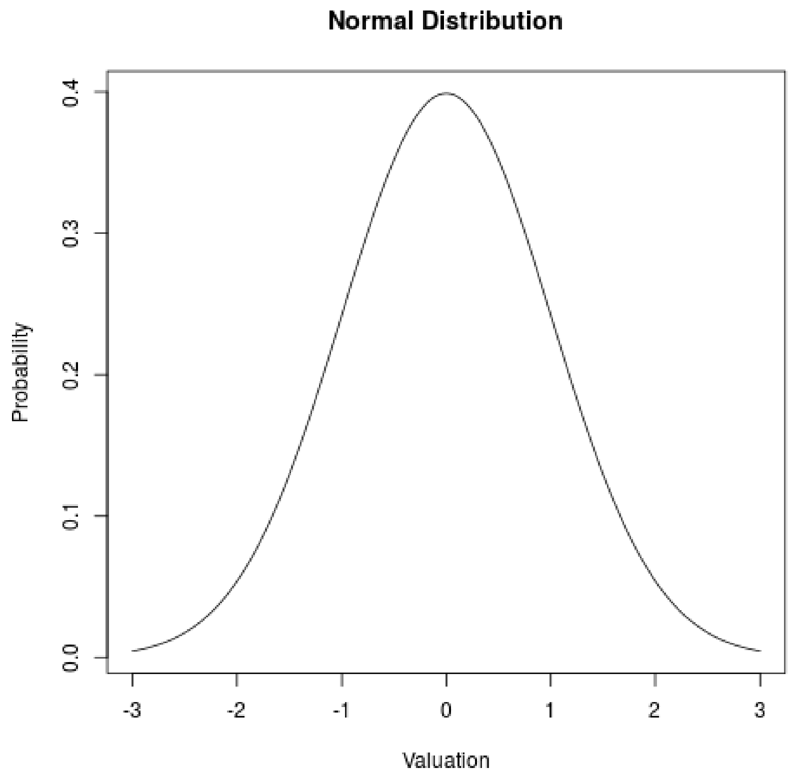


where the chances of encountering any specific valuation over a given range are equivalent, which is known as the uniform distribution. Hence we consider a value to be perfectly random when there are equal chances that it will be valuated across a range of values (This is further discussed in Chapter 2).

1.3.2. Lack of Structure

The second definition is pioneered by Kolmogorov and Solomonoff, and is defined by a lack of structure. Goldreich notes that an equivalent representation of this notion of randomness has to do with the most tersely and densely packed description of an object. This concept is related to the notion of compression, as a compressed file is a representation of the uncompressed file in a less consumptive form. According to this notion of randomness, the more trite a description is, the less random it is.

Figure 1.6. Probability Density Function for Normally Distributed Random Variables



To conceptualize this notion, consider two different random number generators that are seeded by integers of different lengths. The length of the integer that the first generator is supplied is 5, whereas that of the second is 9 digits. Because the number of values with 9 digits is much larger than the number of values with only 5 digits, we conclude that there is less information conveyed by the nine digit value. The Kolmogorov Complexity of an object is the length of the shortest possible description of the object. For instance, we can describe the 9 digit value in a string with length nine, whereas the five digit value can be described in a string of length five. Hence we conceede that the Kolmogorov complexity of the nine digit value is higher than that of the five digit value, unless the nine digit value can be encoded or described in a manner which is shorter in length than the five digit value can be, for instance consider the value 999999999, we can encode this in English by saying "9 9's" which has Kolmogorov complexity equivalent to that of a five digit value. Hence it becomes totally necessairy to agree upon a specific language before comparing two objects in terms of their Kolmogorov complexity. Kolmogorov would consider an object to be random if and only if the length of the program which produces the object is equivalent to that of the object itself.

1.3.3. Lack of Capability

The third and final definition posed by Goldreich in his book deals with a lack of capability. Lack of capability is somewhat fundamentally different from the other two approaches that were mentioned. Instead of looking at what we know about an event, or what we could possibly know about an event, this definition has to do with what we can precompute about an event, and is posed in the form of a scenario presented by Goldreich which occurs in four stages, the situation in each stage of the scenario is the same, Alice flips a coin, Bob predicts a result, and then the outcome of the flip is observed. If we wished to represent this notion mathematically we may do so as follows:

let the outcome of Alice's coin flips be an observable process, with observations $A = \{a_1, a_2, ..., a_n\}$ corresponding to times $T_A = \{t_1, t_2, ...t_n\}$ (or any other arbitrary continuous sequencer i.e. $t_{n+1} = t_n + \delta$). If we imagine a situation where Alice is continuously flipping a coin and observing the result immediately, then consider Bob's *prediction announcement* events $B = \{b_1, b_2, ..., b_n\}$ such that b_1 is Bob's prediction of a_1 which occurs at some offset of t_1 which is defined as ϵ_1. Imagine the time deltas between observation and prediction are given as $\varepsilon = \{\epsilon_1, \epsilon_2, ..., \epsilon_n\}$. Furthermore, imagine that in terms of all possible knowable information about the coin at a particular instant in time such as the velocity, and orientation of the coin, and the drag of the environment, there is some fraction of this knowledge which is 'known' to Bob (or at least stored in memory for easy recall). The proportion of the information which is known to Bob over all knowable information about the system is a continuous (non-differentiable) function of time. The function is non-differentiable because Bob goes from states of not having knowledge about a variable to having it and vice versa. Let the amount of knowledge that Bob has about a system be given by the function $\kappa_B(A) = \frac{\Phi_B(A)}{\Phi(A)}$. In this context the function $\Phi(A)$ represents the amount of information knowable about A in terms of information units i.e. bits. The function $\Phi_B(A)$ on the other hand represents the information about A which is also known to B. What we are left with is a standardized value indicating B's familiarity with A. So the function $\kappa_B(A)$ gives the value of familiarity Learner B has with process A. For example consider the following:

Imagine that the accurate determination of a coin-flip could be made with 64 bits of information. For instance, say 16 bits representing angular momentum, distance from surface, coin diameter, and orientation each. Now at any one particular moment you may only observe two attributes, while leaving the others unchanged, with the exception of a brief period of time when the third attribute is exposed and can be measured without affecting the fourth. if B knows about A $\frac{48}{64} = \frac{3}{4}$ of the possible information about A then it is more likely to succeed in its prediction of event A, if and only if it can process that information before the next observation of process A. Now for instance, we would expect that Bob's prediction of each events outcome would be correct 50% of the time, otherwise it would follow that Bob could predict the future (if this value does not tend towards 50% then we say that Bob is using some additional information available to him about Alice's coin flips $q_i = q(a_i)$ to make prediction b_i). Furthermore, q_i is a percentage of the total possible information that is knowable about Alice's coin flips at a given point in time which is known to Bob. There must then be some (either constant or dynamic) processing rate of B upon the information known about process A, for simplicity let this value have the variable α. Even if Bob knows a significant portion of the Data available about process A, how do we know if it is enough to provide an answer, what is the probability that the answer provided will be correct, and will it happen before the actual event. The earliest before the event occurrence would be the most desirable possible outcome.

1.4. Structure

This thesis is organized as follows: in the introduction, the driving hypothesis is introduced, and conclusion is briefly mentioned. Next a brief introduction to random number generators is given, as well as a description of several popular methods. Finally a philosophical approach to the nature of randomness is given, and parallels between randomness metrics are drawn. In chapter 2 the work that is drawn on in this body of research are presented and discussed, as well as research that is related to that in this thesis. In chapter 3 the methodology for producing the initial random values is discussed (as a side experiment a new extractor is introduced) as well as the manner in which metrics are taken and analyzed. In chapter 4 the results of the experiment as well as the statistical testing and analysis is performed on the results. In the final chapter, the conclusion of the work presented herein is given, as well as next steps for the continuation of this work.

Chapter 2

RELATED WORK

Recall that the there are a plethora of cryptographic hash functions. They are further subdivided into families of hashes. For example, in the SHA-2 family there are six variants, which are differentiated by the length of the produced digest (They are 224, 256, 384, 512, 512/224, and 512/256) [15]. We wish to examine the properties of these functions indirectly by measuring their effects on random values. The manner in which random numbers are generated and the random number generators are tested can be applied to values both before and after the values are hashed in order to determine how well the qualities are improved by a hash function. In order to determine if there is improvement in the randomness qualities of strings after they are hashed we must really answer the question 'What is randomness'. As was discussed in the preceeding chapter there are many seperate schools of thought on this matter [12] [24] [31]. One such metric that is inspected is the entropy of a random stream [31]. Others include the results of the Chi-Square goodness of fit metrics as well as the serial correlation also introduced by Knuth [21]. In this work, the results are compared to those for the Hotbits generator which was created at Fourmilabs. [37]

[10] suggests the following guidelines for regular hash functions: that the hash is easy to compute, and that it is hard (ie. not polynomial time) to find two values with the same hash. These properties are precisely why hash functions are of interest in the context of random number generation. Cryptographic hash functions are defined by Rogaway [29], and others [34] [1]. The properties of cryptographic hash functions require that the output appears somewhat indistinguishable from random values.

The values produced by cryptographic hash functions are generally called ‘hashes’ or ‘digests’ and must satisfy several properties to be considered ‘sufficiently random’. In 1986 Webster provided a definition of the Strict Avalanche Criterion (SAC) [40]. The criterion is a measure of two fundamental properties for cryptographic hash functions: confusion and diffusion. The concepts of the confusion and diffusion measure can be extended to cryptographic hashing functions [9], and were originally proposed by Claude E. Shannon in his 1949 work ‘Communication Theory of Secrecy Systems’ [32]. The measure of confusion which was originally applied to cryptographic ciphers by Shannon and references the dependency structure between the key and the cipher text. The more dependent the cipher text is on the key the better the confusion measure is. Diffusion is a metric that quantifies the dependency of the cipher text on the plain text. It contrasts the confusion in this manner, as it does not take into account any contrast in key and cipher text. The SAC is met if the following statement holds: for a change as small as one bit in the input to a cryptographic hash function, every bit in the output has an equal likelihood of either flipping or remaining unchanged. Very recently (just last year) a statistical procedure for quickly determining whether a hash function met the SAC was discovered [26]. It is meeting the SAC which causes the output of a cryptographic hash function to appear very random. Indeed hash functions are frequently used in the post processing of generated random numbers, due to their properties for improving the quality of the produced numbers. Sklavos et. al offered a VLSI implementation of their Pseudo-Random Number Generator (PRNG) which is based entirely on an initial condition and feedback loop using SHA [33]. Yu-Hua Wang proposed a new random generator based on random noise in 2005, which uses a thermal signal as the input to the cryptographic hash function SHA-2 (512) [39]. In Yu-Hua Wang’s work, he provides several statistics on the produced random values both before using the SHA-2 (512) hash, and after using it. The work done by Yu-Hua Wang closely parallels the work in this thesis, where different cryptographic hashes are compared by examining differences in randomness metrics across a plurality of hashes. Chia-Jung Lee examined the extraction of two seperate entropy sources . In 2007 Bang-Ju Wang proposed a novel random number generator which used a backwards propogating neural network, and SHA-2 (512 bits) for post processing [38]. Łoza et. al used the cryptographic hash SHA-256 to post-process values they produced using uniformly sampled ring oscillators [25]. Herrewege and Verbauwhede invented a very lightweight PRNG which works by taking advantage of the Keccak hash [2] [3] in order to both extract entropy as well as to generate the random values [18]. The list of random number generators which include a hashing phase goes on and on. It quickly becomes apparent that there is some underlying value to using these 'one-way functions' (hashes), and indeed in the seminal 1985 work by L. A. Levin it was shown that the existence of a 'one-way function' such as a hash was a necessary and sufficient condition for the existence of a pseudo-random generator defined over the function [23]. Previously it had only been shown that the existence of a one-way function under several assumptions was a sufficient condition for producing purely random bit strings [4]. In 1984 Blum further contributed to the theory of random number generation by describing a cryptographically secure pseudo random number generator in the following manner: given all prior output of the random number generator, the probability for an analyzer of any amount of power to predict the next bit correctly is fifty percent [6]. Another recent development in random number generation has to do with the extraction from multiple different sources, and then recombining them [22]. The multiple sources of randomness are then recombined using the generalized leftover hash lemma originally introduced by impagliazzo [19]. The leftover hash lemma is of central importance to this work as it essentially states that any hash function applied to a weak source of randomness will produce values that are random on a uniform distribution. In 1984 M. Blum also extended a previous contribution by Von Neumann [36] which allows one to turn a biased coin into a fair one, for any arbitrary number of bits.[5] More recently the interest in quantum computation has pushed random number theorists to consider potential vulnerabilities implied by the use of quantum state vectors [35]. The work contained therein is relevant as a generalized leftover hash lemma is shown to be robust in a quantum environment.

The comparison of hashing algorithms for use in indexing was completed by R. Jain in 1992 [20]. That work parallels this in that different hash functions are compared based on metrics which indicate how well they perform, in this case the metric was the cardinality of a trace of address references. In 1979, universal classes of hash functions were considered [7]. Several of these classes lend themselves nicely to the application of random number generation, and as such were studied in more detail as generators which require smaller seeds [16]. Chor showed in 1985 that a weak source of randomness can indeed be used to generate almost uniformly random numbers [8]. In 1986, Goldreich et. al showed how to construct random functions using one-way functions such as hashes [28]. Recently a similar undertaking by Zhandry in the realm of quantum random functions has been presented [42]. Goldreich also discussed the existence of pseudo-random generators using one-way functions such as hashes [13]. In 1982 Yao introduced a logical test, as well as multiple manner of construction for pseudo-random generators using 'trap-door functions' [41]. In 1988 Shamir produced a text detailing how to produce cryptographically secure random number generators [30].

A treatment of the majority of the random generators that are frequently used was given by Donald Knuth, in his second volume on the Art of Programming entitled 'Semi-Numeric Algorithms' [21]. Very recently a genetic algorithm was used to evolve random values using fitness criteria reported by the ENT linux utility [17].

The body of work on cryptographic hash functions and their relationship to pseudo-random number generators was discussed in detail in this chapter, and the results of several of the papers discussed herein were vastly motivating factors for the use of randomness metrics to characterize cryptographic hash functions. Previous research has not examined the 14 cryptographic hash functions which are analyzed in this thesis, and has not used randomness metrics to make determinations on which hash should be used. In this thesis these metrics are determined and verified.

Chapter 3

METHODOLOGY

In this research, the methodology consists simply of applying statistical analysis to the randomness metrics of the same values after they have been hashed using different cryptographic hash functions. The use of parametric and nonparametric methods for group location tests such as the Welch, Brown-Forsythe, and Kruskal-Wallis rank sum test were applied. As well as Dunn's test for a post-hoc multiple comparison procedure. For testing the assumptions of the ANOVA, the Shapiro-Wilks test of normality was applied, and Levene's test for homoscedasticity was used.

As a side experiment a new entropy extractor based on the inter-packet delays on a network was introduced. The extractor was used to produce values which were then experimented on by applying the hash functions and monitoring the changes in randomness metrics such as entropy, serial correlation, and the chi-squared uniformity test results.

3.1. ENTROPY

3.1.1. Derivation

Entropy is defined by the amount of disorder in a given system. The definition extends to bitstrings in the capacity It is defined mathematically by the following formula:

$$H(X) = E[I(X)] = E[-\ln(P(x))] \tag{3.1}$$

The formula is not that intuitive at first, but it can be derived in an *intuitive* fashion. We will first discuss the self information of an event. The self-information of an event is the amount of information that is construed by the event having occured. It stands to reason that events which occur with a low frequency can tell you more about a particular system. For instance, if a fair six-sided die was rolled 100 times, we would expect each of the 6 sides to be face up with roughly equal proportions. Thus, rolling a one with the dice contains no more information than rolling a six would.

However, if we imagine a six-sided dice that is not fair, and causes six to be face up 50% more frequently than one, then we gain more information about the die by observing a one, than observing a six. Because an event having occured conveys some information about itself, an event that occurs less frequently inherently conveys more information about itself than an event which occurs frequently. With this initial hurdle in mind, we can begin deriving the mathematics of the self-information of an event. Let us first attempt to define the self information of an event as follows:

$$I(A) = \frac{1}{P(A)} \tag{3.2}$$

We define the information of event A as the inverse of the probability that A occured. At first glance this seems to be a good measure of the self information, as it increases proportionally with the probability that an event occurs, the smaller the probability of the event occuring, the larger the self information. However this definition does not satisfy one crucial requirement for self information. Plainly stated, the self information of two events intersection should be equivalent to the self information of the first event cummulated with the self information of the second event. In other words we want the following to hold:

$$I(A \cap B) = I(A) + I(B) \tag{3.3}$$

We know from probability theory that the intersection of two probabilities is expressed as below:

$$P(A \cap B) = P(A) \cdot P(B) \tag{3.4}$$

Therefore, using the measure of self information that we initially suggested we would arrive at the following predicament:

$$I(A \cap B) = \frac{1}{P(A) \cdot P(B)} \neq I(A) + I(B) = \frac{P(A) + P(B)}{P(A) \cdot P(B)} \tag{3.5}$$

As our original definition of the self information of A does not satisfy this equation, we are forced to concieve of another function that will indeed satisfy this property of additive self information.

$$I(A) = ln(\frac{1}{P(A)}) = -ln(P(A)) \tag{3.6}$$

This definition of the self-information of a given event beautifully satisfies the additive self-information property that was neglected in our first definition with the same properties of being large when the probability is small, and vice-versa.

$$I(A \cap B) = -ln(P(A) \cdot P(B)) = I(A) + I(B) = -[ln(P(A)) + ln(P(B))] \tag{3.7}$$

Ergo, we can define the self-information of an event as is given in 1.13. This isn't the end of the story however, as we have simply defined the self information of an event, not the entropy. The entropy of an event is a value that is very closely related to the self information of the event occuring, but is not defined exactly the same way. Note that because the self-information is defined based on a probability density function it itself is indeed a random variable. Therefore we can take the expected value of the information of A, just as we can the random variable A itself. We call the expected value of the self-information of an event, the entropy of that event.

$$H(A) = E(I(A)) \tag{3.8}$$

3.1.2. Extraction

Sources of entropy are frequently gathered by modern random number generators. As was stated in the introduction the Linux random number generator extracts entropy from interrupts, disk accesses, keyboard input, and more.

One obvious metric to examine when attempting to extract entropy timing. For instance, when extracting entropy from the keyboard, one possible method for doing so is the careful examination of the wait times between key presses. The slight variations in key presses may not be a *truly* random data source (I.e. the distribution may not be uniform), but there are many methods for transforming known distributions into uniform ones, one such method is the first of, what we claim, are three ways to extract entropy from a random stream.

3.1.2.1. A Posteriori Coin-Flip

The principle behind this method of entropy extraction relies on a known data set size. After the data is collected, the median value of the distribution is calculated. Since by definition the median is a measure of center, calculated as the middle element of the data set, exactly one half of the retrieved values will be above the median and the other will be below. Obviously, this extends to the use of other measures of center, and even potentially trend models such as a Linear regression, ARMA and other forecasting techniques.

The A Posteriori Coin-Flip occurs after your entire data set has been collected and a valid trend seeking line (whether flat as with the mean or median, or sloped), begin at the first data point (this is the first coin-flip) if it lies above the calculated measure of center (MOC) record one, or heads as the result of the first coin-flip. Continue until all data points have been assigned a bit. This bitstring is an example of extracting entropy from a system if you are given knowledge about the data system ahead of time. Mathematically this notion can be expressed in the following way.

Definition 1 A Posteriori Coin-Flip

$$\text{Given } X \text{ such that } X = \{x_1, x_2, x_3, ..., x_n\}$$

$$Q_2 = \{x | P(X > x) = P(X < x) = 0.5\}$$

$$R_\psi(x_i) = r_i = \begin{cases} 1 & x_i > Q_2 \\ 0 & x_i < Q_2 \end{cases}$$

Hence, the entropy is extracted into the binary value: $r_1 r_2 r_3 r_4 ... r_n$

In this definition we are using the median as the MOC (represented by Q_2), however this can easily be replaced with any other MOC. As with any method there is a short pros and cons list for using the A Posteriori Coin-Flip. One obvious con of the definition in the manner we present it with a static MOC, as opposed to a dynamic MOC such as a time series ARMA model, there will tend to be an obvious correlation between values close together. Of course, the better the entropy source, the higher quality the extracted entropy will be. A Pro of using this method is that we can force a uniform distribution on the output string. A further benefit of using this method is that the MOC must only be calculated one time (after all of the data is collected)

A mathematical proof that this method (using the median) will extract the maximum theoretical entropy is given here:

Proof A Posteriori Coin-Flip maximizes Shannon Entropy

Given a *supposedly* random sample

$$X = \{x_1 \in \mathbb{R}, x_2 \in \mathbb{R}, x_3 \in \mathbb{R}, ..., x_n \in \mathbb{R}\}$$

We define the random variable α in terms of the median (or second quartile) of X

$$\alpha : \mathbb{R} \to \mathbb{B}$$

$$P(\alpha = 0) = p_0(\alpha) = \frac{|\{x|x < Q_2(X)\}|}{|X|} = \frac{1}{2}$$
$$P(\alpha = 1) = p_1(\alpha) = \frac{|\{x|x > Q_2(X)\}|}{|X|} = \frac{1}{2}$$

The formula for the entropy of a string of Bernoulli trials (or a 'bitstring') is given:

$$H(p_0(b), p_1(b)) = -(p_0(b)\log_2(p_0(b)) + p_1(b)\log_2(p_1(b)))$$

We can maximize the Entropy function as so:

$$\nabla H(p_0, p_1) = \Big(\frac{\partial H}{\partial p_0}, \frac{\partial H}{\partial p_1}\Big) = \Big(-\frac{\ln(p_0) + 1}{\ln(2)}, -\frac{\ln(p_1) + 1}{\ln(2)}\Big)$$

Maximizing we find

$$\frac{-\ln(p_0) - 1}{\ln(2)} = 0 \Longrightarrow \ln(p_0) = -1 \Longrightarrow p_0 = \frac{1}{e}$$
$$\frac{-\ln(p_1) - 1}{\ln(2)} = 0 \Longrightarrow \ln(p_1) = -1 \Longrightarrow p_1 = \frac{1}{e}$$

This seemingly odd result is because there is an *inherent* dependence among these two values, expressed mathematically as $p_0 + p_1 = 1$, in our first maximization attempt, we neglected to account for the hard-restraint $p_0 + p_1 = 1$ In constraining the original optimization we have the following system:

$$\frac{-\ln(p_1) - 1}{\ln(2)} = 0 = \frac{-\ln(p_0) - 1}{\ln(2)}$$

$$p_1 = 1 - p_0$$

$$\frac{-\ln(1-p_0)-1}{\ln(2)} = \frac{-\ln(p_0)-1}{\ln(2)} \Longrightarrow 1 - p_0 = p_0$$

$$\Longrightarrow p_0 = 0.5 \Longrightarrow p_1 = 1 - 0.5 = 0.5$$

Because $p_0(\alpha) = p_1(\alpha) = 0.5$ by definition, we have maximized the entropy function for the constraint $p_1 + p_0 = 1$. ∎

3.1.2.2. Extemporaneous Coin-Flip

As the name implies, the extemporaneous coin-flip occurs simultaneously as new data is available. The general procedure is very similar to that of the A Posteriori coin-flip, however in this method the measure of center is recomputed with every new sample that becomes available. In this method we start with an initial guess as to the measure of center (This could be a MOC from a previous entropy capture, or an educated guess, or simply the first value observed). As a side note, in both this and the A Priori Coin-Flip method, the first random bit retrieved can and should be thrown away in certain circumstances.

As with the previous method, each time new data becomes available it is compared to the MOC and a bit is generated depending on whether the point lies above or below. Unlike the previous example, the MOC changes with every new data point available. in the following definition the chosen MOC is again the median, however again this can easily be replaced with whichever statistic is desired.

Definition 2 Extemporaneous Coin-Flip

$$\text{Given a series of sets } X = \{X_1, X_2, X_3, ..., X_n\}$$

$$\text{Such that } X_1 \subset X_2 \subset X_3 \subset ... \subset X_n$$

$$\{\forall i \in \mathbb{N} | 0 < i \leq n\} \Longrightarrow |X_i| + 1 = |X_{i-1}|$$

Given an initial guess g_0 to serve as the MOC (before the first data point is collected) We can express our updating median as a function of data sets in the following manner.

$$Q_2(X_i) = \{x|P(X_i > x) = P(X_i < x) = 0.5\}$$

$$R_\phi(x_i) = r_i = \begin{cases} 1 & x_i > Q_2(X_i) \\ 0 & x_i < Q_2(X_i) \end{cases}$$

As with the previous definition the entropy generated can be utilized by the bit-stream given by:

$$r_1 r_2 r_3 r_4 \ldots r_n$$

As with the A Posteriori method, there are some benefits, and draw backs to using this method for entropy extraction. In this method the measure of center must be recomputed with each subsequent data point gathered, this could waste valuable CPU cycles. The primary benefit of using this method is that after the new MOC is calculated, the old can be discarded, as well as the immediate discarding of all previous data, as it is analyzed simultaneously as it is gathered. This method is also easy to implement on systems where you would like to be constantly analyzing data (such as massive server banks connected to Muller-Geiger tubes and old fire detector parts).

3.1.2.3. A Priori Coin-Flip

The last of what we claim are three entropy extraction methodologies presented in this work we again require some guess or ‘A priori’ knowledge in order to effectively use the first bit as a truly random coin flip. This method is performed in exactly the same manner as the previous, with one key difference. In this method, the MOC that is used must determine the outcome of the coin-flip **before** it is updated with the new information. The new value must be used to update the MOC after the previous coin-flip, both the previous data value and previous MOC can be immediately discarded after the new MOC is generated. As it is essentially the same method as above simply time-lagged by one observation the definition is very similar with the difference highlighted below

Definition 3 A Priori Coin-Flip

$$\text{Given a series of sets } X = \{X_1, X_2, X_3, ..., X_n\}$$
$$\text{Such that } X_1 \subset X_2 \subset X_3 \subset ... \subset X_n$$
$$\{\forall i \in \mathbb{N} | 0 \leq i \leq n\} \Longrightarrow |X_i| + 1 = |X_{i-1}|$$

Given an initial guess g_0 to serve as the MOC (before the first data point is collected) We can express our updating median as a function of data sets in the following manner.

$$Q_2(X_i) = \begin{cases} \{x | P(X_i > x) = P(X_i < x) = 0.5\} & i \neq 1 \\ g_0 & i = 0 \end{cases}$$

$$R_\pi(x_i) = r_i = \begin{cases} 1 & x_i > Q_2(X_{i-1}) \\ 0 & x_i < Q_2(X_{i-1}) \end{cases}$$

The entropy is in the bit string:

$$r_1 r_2 r_3 r_4 ... r_n$$

Corollary 3.1.1 *This method can easily be extended ad inf. by using MOC's from more than one observation in the past, then updating with a lag value greater than one.*

Corollary 3.1.2 *When using certain methods for the MOC, such as forecasting models for time series, prior data must be kept for the recalculation of the MOC, however, a sliding window could also be used in this case to save memory.*

Many of the pros and cons for using this method correspond with what was said about the Extemporaneous Coin-Flip. In this method, unlike the last, the new MOC does not need to be calculated before the result of the next coin-flip can be determined. This is a great method to use when events are rare, and there is a long wait time before the next observation. When using this method in the given scenario, the wait time can be used as compute time for the updated MOC.

It is essential to note that in all of the above methods the entropy extracted from the random stream should not be used directly (I.e. do not directly use the bit strings calculated as your random values), they should instead be used as inputs to pseudo-random number generators, conglomerated with multiple other sources of entropy with a mixing function, or as we will further explore in this work use pseudo-random functions such as cryptographic primitives like hash functions and HMACs (potentially even using the random value a key in a keyed HMAC). One could even potentially use a good compression algorithm to produce a valuable random number.

3.2. RANDOM SOURCE: NETWORK

In this section we attempt to extract entropy from an real-life random stream, that almost any modern computer has access to. We propose the extraction of entropy from simple packet capture data. This notion extends much further than the manner in which it is presented here, and can be applied to far more sophisticated broadcast data (I.e. low band AM, Snow on television). In this case, we take a very clearly deliberate and logical approach to extracting entropy.

We briefly mentioned in the last section that a straightforward approach to extracting entropy from a random stream is to investigate the timings of the events. In this specific application we examine the timing data for packets using a standard packet capture tool (Wireshark). Once a given packet capture was completed we computed the retrieved entropy bit strings using the A Posteriori method described in the previous section.

With our packet capture results, timing data was given in terms of seconds (with 5 digits of significance). As each time was initially reported in the elapsed time since the beginning of the capture, we first needed to subtract each next arrival time from the prior arrival time. In this manner each of the packets received would be assigned a time delta, (or in other words, the time stamp of the packet received after the current packet minus the current packet's time stamp. We can formally define this in the following manner.

Definition 4 Extracting Entropy from Packet Timing Data

let t_i represent the i^{th} time stamp given in seconds, since the beginning of the packet capture.

$$\delta_i = t_{i+1} - t_i$$

Note that this necessarily excludes the generation of the terminal δ_i (because t_{i+1} has not been observed yet). Suppose that in a given packet capture, n packets are captured, then

$$\Delta = \{\delta_1, \delta_2, .., \delta_{n-1}\}$$

We can then define a probability space over the set Δ

$$P(\overline{\Delta} < x) = \frac{|\{\delta_i \in \Delta | \delta_i < x\}|}{|\Delta|}$$

$$Q_2 = \{\delta | P(\overline{\Delta} < \delta) = P(\overline{\Delta} > \delta) = 0.5\}$$

We then simply perform the A Posteriori Coin-Flip discussed in the previous section

$$R_{\psi}(\delta_i) = r_i = \begin{cases} 1 & \delta_i > Q_2 \\ 0 & \delta_i < Q_2 \end{cases}$$

Chapter 4

RESULTS

Recall that a new extractor was introduced which worked by examining the inter-packet delays on a network. First the results of this side experiment are shown, followed by the measurement of randomness qualities before and after being hashed by a series of 14 of the most popular cryptographic hashes. These measurements are then analyzed, and finally claims and conclusions are drawn based around the analysis of the metrics.

4.1. Packet Captures

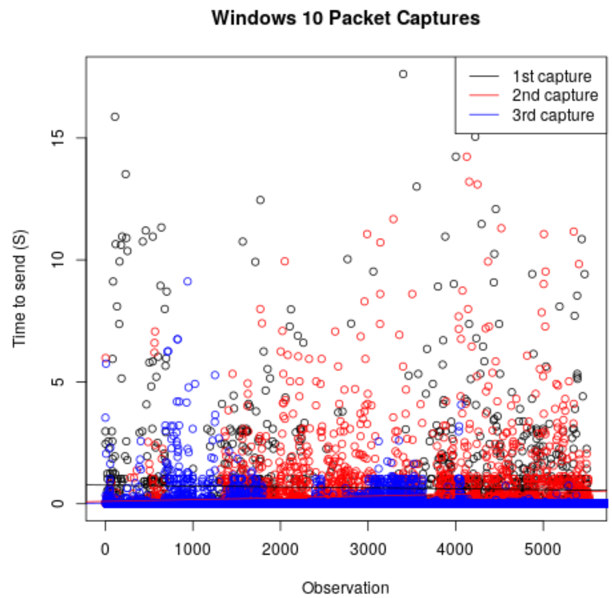
Packet captures of sizes ranging from 5000-50000 packets were run at three different times, and over multiple networks on three different operating systems (Mac OSX, Windows 10, Ubuntu Linux 16.10).

A very quick inspection of the plots reveals that there is temporal correlation in the data at this point, however, we are not worried about the quality of the random numbers produced at this stage in their creation. As was stated in Section III, we are not going to simply use the produced value, we are more likely to use the gathered entropy to seed a deterministic pseudorandom generator.

We leave it to future work to perform more sophisticated trend modeling, in this analysis we will simply be using the median to decide the outcome of the A Posteriori Coin-Flip. The medians were calculated and nine different entropic bit strings were collected.

As we are merely concerned with having values which are able to be improved through the application of a cryptographic hash function we are not really interested

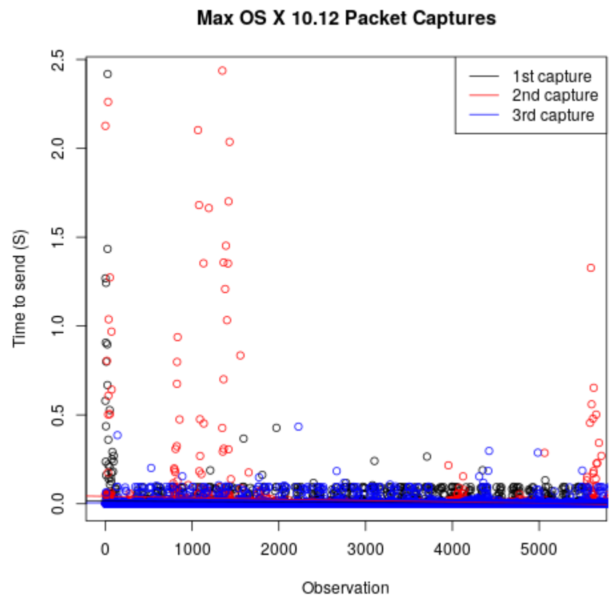
Figure 4.1. Windows Packet Capture Inter Packet Delays



in the resulting characteristics of these produced random strings being very good. The use of inter-network packet delays as a weak source of randomness should be considered in this context as a side experiment. The primary utility of the provided values simply has to do with the amount by which they can be improved when utilizing different Cryptographic hash functions. In this work we investigate several different families of hash functions, and several different randomness metrics, to determine what properties are improved by which hash functions.

Note that we will be using nine values, which correspond to the nine packet captures that were performed. As the captures were on different networks, and different operating systems, it is fairly safe to say that the sources are independent of one another. As shown in the Appendix, the ENT utility was run on each of the nine strings, and the measures (such as entropy and serial correlation) are reported. The values

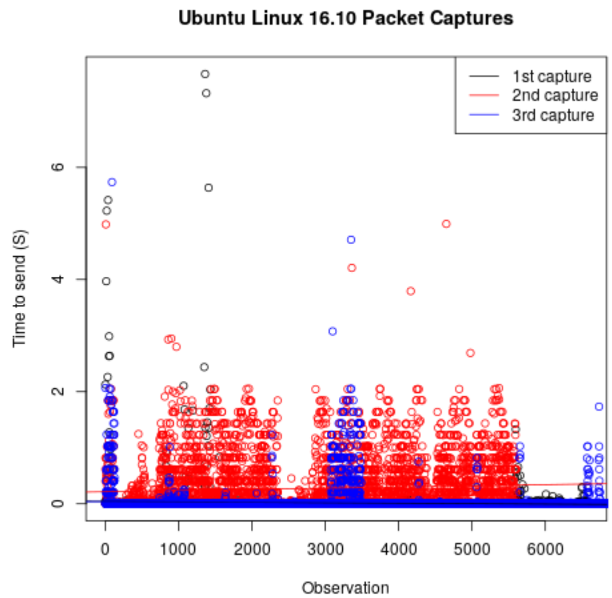
Figure 4.2. Mac OSX Packet Capture Inter Packet Delays



are then broken up into chunks of the same size as the output of the corresponding hash function, and each chunk is hashed individually then concatenated. The ENT utility is then also used on the hashed values, and the quality measures are recorded.

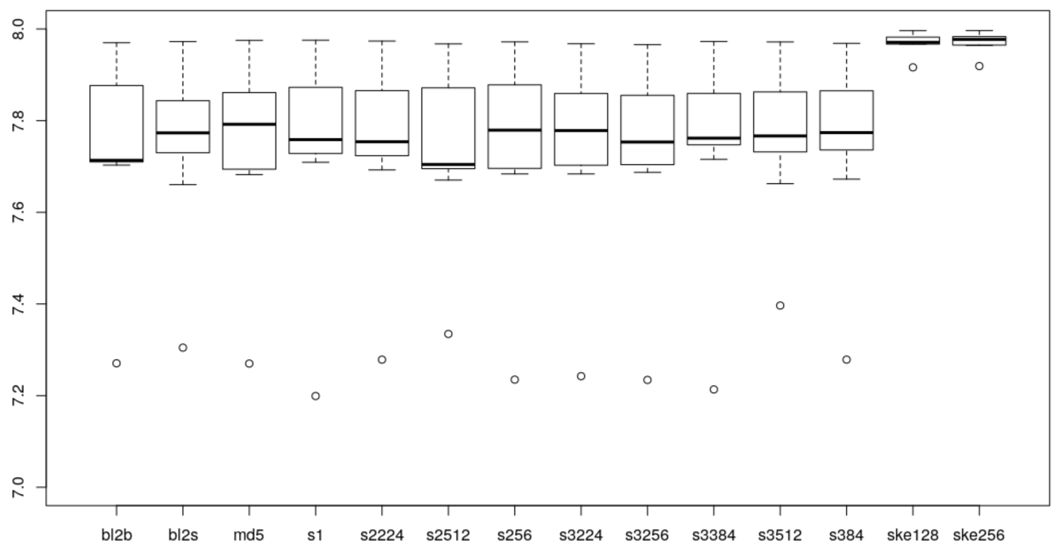
Taking the difference in the quality metrics before and after hashing for each of the nine values (for each hash) allows a quantification of the improvement of the random value dependent on hash. All the parameter deltas are grouped by hash, and the variance is Analyzed to determine if there is any mean pair of differences that are statistically lower or higher than each other. Then the post-hoc multiple comparison procedure involving Dunn Confidence intervals was performed, giving statistical evidence that certain classes of Cryptographic hash functions tend to have better metric improvement. For a complete listing of the metrics and data that were produced for each of the nine strings, consult the appendix. Before performing an

Figure 4.3. Linux Packet Capture Inter Packet Delays



analysis of variance on the data, we should inspect it visually in the form of box plots. The first box plot represents one of the first metrics that we are examining, the entropy. The entropy is considered over ensembles of 8 bit strings, indicating that there are $2^8 = 256$ symbols in the alphabet. Figure 4.4 shows the entropy distributions of each of the 14 hash functions that are being compared. Recall that there are nine different strings per hash, giving a total of 126 total strings that are being analyzed. As is becoming readily apparent from the plot, there appears to be vaster improvement of Entropy in the case of the shake hash function. This is an important intermediate result as it indicates that an ANOVA may be necessary to determine if there is any statistical difference in the means of the entropies produced by different hashes.

Figure 4.4. Entropy in bits/byte for hashed strings (n = 9 per hash)

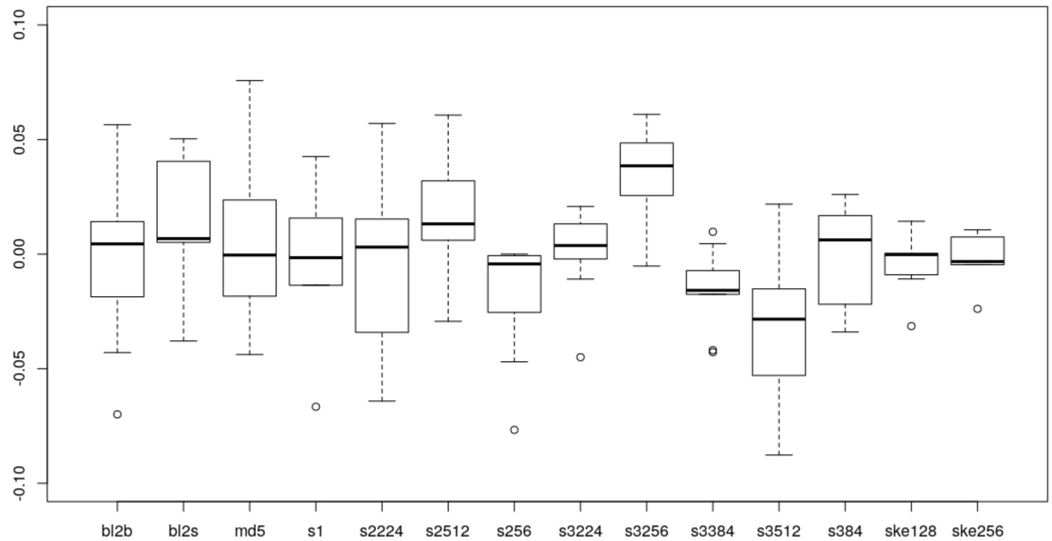


Similarly by inspecting Figure 4.5 we notice that there appears to be major differences among the means of the different groups. This indicates that it would be good to perform an ANOVA as well as a post hoc procedure in order to see which pairs of mean differences are significant. In order to perform an ANOVA there are several statistical assumptions that must first be addressed.

- The data are normally distributed (tested with the Shapiro Test/ qq plots)
- The data are homoscedastic (have equal variances, levene's test)
- The data are independent within subgroups (This is assumed)

One final boxplot to inspect is that which relates to the Chi-Square test statistic for the data. The test statistic should fall on the middle of the interval for truly random data, which in this case is 256. Figure 4.6 shows yet again that there are potentially statistically significant differences among the different hash functions.

Figure 4.5. Serial Correlation for hashed strings (n = 9 per hash)



We must first address the assumptions for each of the variables we wish to examine. We will examine the entropy statistic first, Figure 4.7 shows the qq-plot of Entropies across subgroups. As is fairly apparent from the figure, the data do not seem to follow a normal distribution. We can formally verify this notion by using the shapiro test. The null hypothesis of the Shapiro test is that the data come from a normally distributed population.

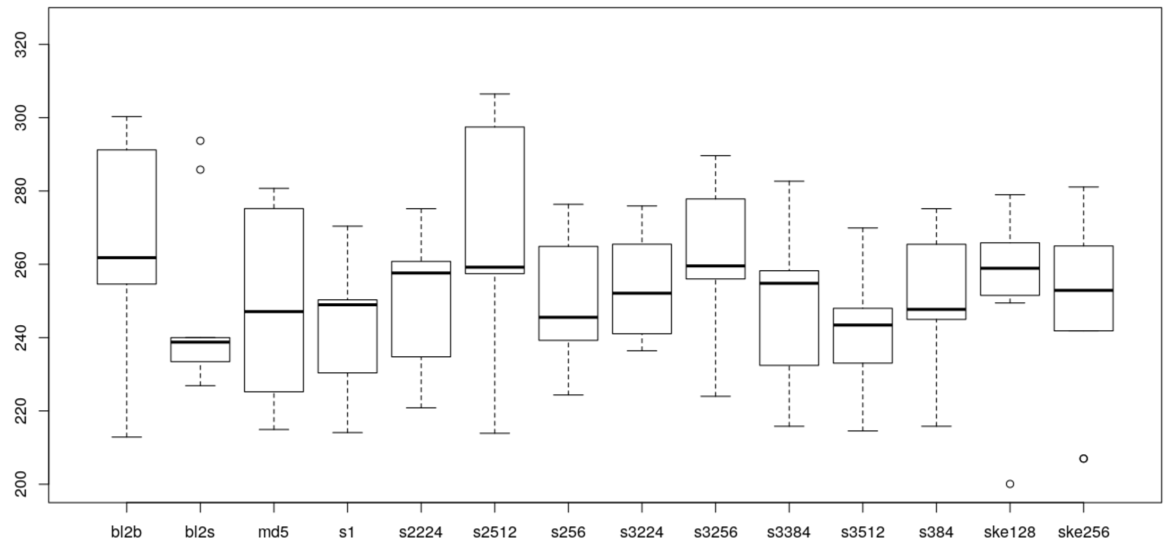
Table 4.1 indicates the results of the Shapiro-Wilks test for normality.

Table 4.1. Shapiro-Wilks Test for Normality of Entropies

W	0.81796
p-val	3.418e-11**

Note that we must reject the null hypothesis of normality in this case in favor of the alternative that the data do not come from a normal distribution. This means that

Figure 4.6. Chi-Squared for hashed strings (n = 9 per hash)



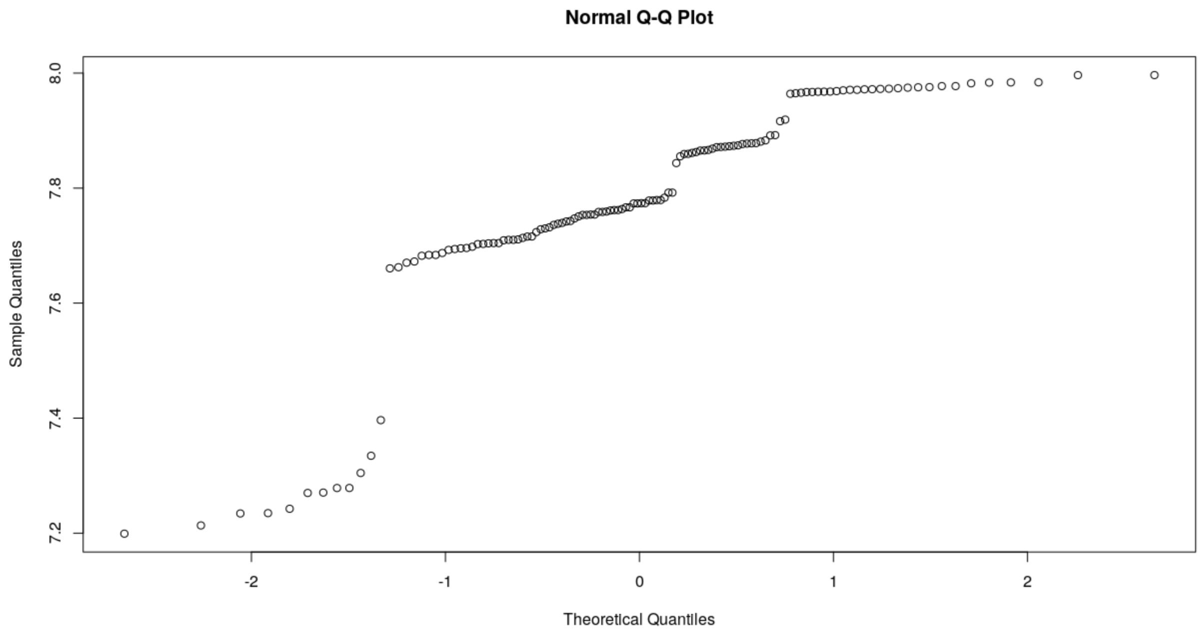
we cannot perform a regular One-Way ANOVA on the data (well we can technically, because ANOVA is robust to the normality assumption, with only a slight drawback on the type I error rate.) We will use the Kruskal-Wallis H test instead 4.2, as it does not require the working assumption that the data come from a normal distribution.

Table 4.2. Kruskal-Wallis Test for equivalent population Entropies

χ^2	38.57
p-val	0.0002341**

Note that the p-value is below our significance level of 0.05 which indicates that we should reject the null hypothesis of the Kruskal-Wallis H test which is that the Entropies among the different groups come from populations which are distributed differently. I used the Dunn multiple comparison procedure in this package [11]. The

Figure 4.7. qq-plot of Entropy



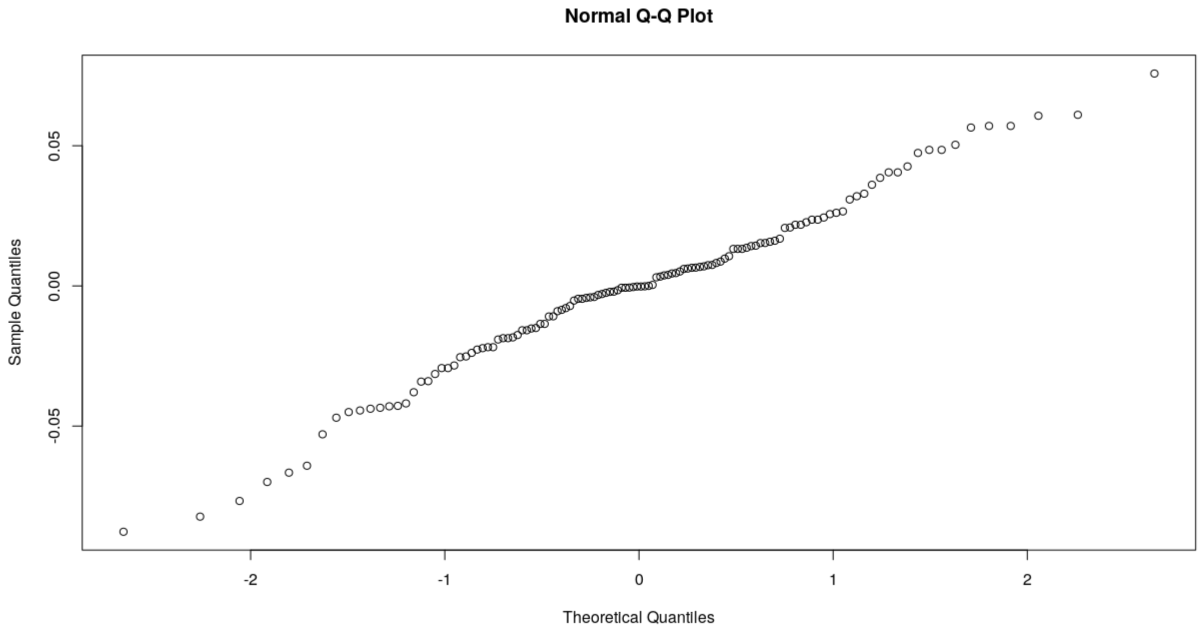
results of the post-hoc procedure are shown in the appendix.

We will now investigate the serial correlation improvement of these hashes by a similar analysis. Let us first determine what the proper variance comparison technique is. The qq plot in Figure 4.8 shows that the data appears normally distributed, the Shapiro Wilk test results indicates that the data is sufficiently normal, and hence we must use Levene's test to determine if the data are homoscedastistic. The results of Levene's test and the Shapiro-Wilk test are given in Tables 4.3 and 4.4

Table 4.3. Shapiro-Wilks Test for Normality of Serial Correlations

W	0.98486
p-val	0.1741

Figure 4.8. qq-plot of Serial Correlation



As the result of levene's test is not significant enough to reject the null hypothesis that the groups are homoscedastic we are able to apply a proper ANOVA to the serial correlations.

The results of the variance analysis as well as appropriate post-hoc procedures are provided in the appropriate appendices.

4.2. ANALYSIS

In our analysis we will simply be using the Linux utility ENT, created at Fourmilabs, which reports several statistics on the entropy contained, temporal correlations, even quality of randomness (Monte Carlo simulation of Pi). In the following paragraphs we will briefly examine and discuss the use of each of the statistics that are calculated using ENT, as well as presenting a table of the results for the entropic strings that were generated using the A Posteriori Coin-Flip method on our nine

Table 4.4. Levene's test for homoscedasticity of serial correlation

F	1.4785
p-val	.1364

packet capture timing Data vectors.

The first metric reported by ENT is its name-sake: entropy. For all intents and purposes we can think of this measure as a rough estimate as to how much information is actually contained within a bitstring. The metric of entropy was defined by Claude Shannon in his seminal work *A Mathematical Theory of Communication.* The formula he gives for approximating the entropy of a bit string is defined as:

Definition Shannon Entropy (as applied to bitstrings)
The Shannon Entropy of a bitstring is quantified in terms of the information inherently contained by the string. The derivation of this formula is given in Chapter 3. Shannon entropy is defined as:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

In a bitstring, there are two possible outcomes, either a one or a zero. Hence there are only two probabilities to deal with, $p_0 = P(\overline{X} = 0)$ and $p_1 = P(\overline{X} = 1)$ for random variable $\overline{X}$, which is distributed according to the relative frequency distribution for bits contained within the random string $X = x_1 x_2 x_3 .. x_n$. In this way, the entropy is calculated in a very similar manner to that of the way which it is extracted in the A Posteriori Coin-Flip, where we must have retrieved all information before we can provide an accurate estimate. Once we have collected all of the data, and calculated $\{p_0, p_1\}$ we can calculate the entropy of the bit string directly as:

$$H(X) = -[p_1 \log_2(p_1) + p_0 \log_2(p_0)]$$

The entropy is reported in terms of a measurement of the amount of 'bits' of information contained in a specific character, in the case of a random bit string it reports the 'bits' per bit. Hence in the *ideal* case we would see entropy approaching 1.0 (or subsequently, the number of bits used to encode the symbol)

The second statistic reported on by ENT is the optimum compression ratio. Note that, this is calculated directly from the entropy estimate. We also would like to note that because of this, it may not take more useful and modern compression techniques into account and is simply a theoretical value. The statistic is reported in relation to compression on bytes, as well as the compression for the value as a bit string. Again, thanks to Claude Shannon, a theoretical maximum value for the optimum compression ratio was quantized in terms of the entropy of the string to be compressed.

Definition Shannon's Source Coding Theorem

This theorem simply states that the length of the optimally compressed string is directly related to the entropy contained over the entire string, Note that this is not the ratio of entropy per symbol as it is reported by ent, it is technically equivalent to the entropy ratio reported by ent multiplied by the length of the string in symbols. Mathematically the bound on the minimum length of a compressed string is given as:

$$L_c < H(X) + \frac{1}{N}$$

Where L_c is the minimum length of the compressed string X of length N. The Compression ratio then simply becomes:

$$C = \frac{L_u - L_c}{L_u}$$

Where L_u and L_c are the uncompressed and compressed string lengths. Ideally, the entropy ratio is close to one bit per bit, and therefore there the difference $L_u - L_c$ will approach zero, causing the compression ratio to become zero. The theoretical nature of this calculation does not apply to different compression methods, such as run-length encoding.

The next statistical result reported on by ent are the results of a chi-squared(χ^2) test for randomness. The χ^2-test was first proposed by Karl Pearson, an extremely prolific statistician out of London England. As with any statistical hypothesis test this test is formulated by examining collected data through the use of a deterministic function of said data known as a statistic. The statistic, or test-statistic, is then compared to a critical point on the **Known Distribution** of test-statistics to ascertain whether or not a specific hypothesis should be rejected. In this way, it is similar to the Coin-Flip methods described above, however the MOC would correspond to the critical-value on the distribution of test-statistics. The key contribution of Pearson to this particular test was his proof that a test statistic (deterministic function of the data) would be distributed (in the theoretical case) according to a parameterized distribution known as the χ^2 distribution (so named as the square of the sum of normally distributed random variables follows this distribution). The χ^2 distribution is a parameterized distribution, meaning that the deterministic function providing its probability density function (PDF) is dependent on a value supplied in the instantiation of the distribution. The required value for the χ^2 distribution is given the symbol κ and known as the "degrees of freedom". Degrees of freedom are so called, because when special conditions are met, they can be calculated indirectly as a function of how much data was collected, however it should be noted that this is simply an estimate, as was proven in Kendall's Advanced theory of Statistics.

In the case of the χ^2 test, it was shown that the sum of the squared differences in frequencies between that of an observed distribution, and that of a theoretical distribution divided by the theoretical distribution, for each discretized observation bin will follow a χ^2 distribution with degrees of freedom calculated by use of the number of observation bins n, and the number of co-variates used to specify the theoretical distribution p. Again due to Shannon, it was speculated that if data were to be truly random, it would be pulled from a Uniform distribution, with two parameters (a which is the first observation,b which is the last observation). Therefore, to use the proposed test for randomness, the theoretical distribution that will be used is the uniform distribution. Hence a χ^2 test for randomness can be mathematically formulated in the following way:

Definition Chi-Square test for Randomness

Given a set of **discrete** observations $X = x_1, x_2, x_3, ..., x_n$, we wish to show:

$$(\overline{X} : X \to X) \sim U[x_1, x_n]$$

The random variable $\overline{X}$ *defined as a mapping of data points to themselves in the measure space is distributed sufficiently close to, or far away from the given uniform distribution.*

$$H_0 : \overline{X} \sim U[x_1, x_n]$$

$$H_1 : \overline{X} \sim \Lambda$$

Where Λ is an unknown distribution. A significance level α, denoting the point of the critical value (I.e. which determines *how similar* the frequencies must be to **not reject the null hypothesis**) is chosen. A standard value is $\alpha = 0.05$. Calculating the test statistic, and determining the critical point on the appropriate χ^2 distribution as shown here:

$$c_p = \{x|P(\chi^2(\kappa) < x) = \alpha = 0.05\}$$
$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - T_i)^2}{T_i}$$
$$\kappa = n - (p+1)$$

Note that the definition of how many bins exist depends strongly on the manner in which a random string is analyzed. In ent, for instance, when running in byte mode, there are $2^8 = 256$ bins, whereas when analyzing in bit mode, there are only two bins, one and zero.

Remark The ent results also report the tail probability ('it will exceed this value less than X percent of the time'). This value is also known as the p-value, if the p-value is greater than our significance level α then we fail to reject the possibility that the data does indeed come from a random distribution.

Fourmilabs, the creator of ent, suggests the following interpretations for χ^2 test results.

Table 4.5. χ^2 p-value interpretations

p-value	**Interpretation**
p-val > 99%	Almost certainly not random.
99% < p-val > 95%	The sequence is suspect.
95% < p-val > 90%	The sequence is almost suspect.
90% < p-val > 10%	The sequence is good.
10% < p-val > 5%	The sequence is almost suspect.
5% < p-val > 1%	The sequence is suspect.
p-val < 1%	Almost certainly not random.

The next parameter reported by ent is the arithmetic mean of the data, depending on whether you specify byte or bit mode in ent, this will produce an average of the binary numbers encoded in a single byte (0-255) or, simply the bits present in the file. It is calculated simply by summing every value, and dividing by the number of values present. We could further use the calculated mean as a test-statistic in a student's t-test, however it is sufficient to simply compare to the ideal means 0.5, and 127.5 for bits and bytes respectively.

The ENT measures several other characteristics of the data that is provided, but the two most important metrics in this research are the entropy and the serial correlation, as the chi-squared test results were shown to not have any statistically significant difference among the grouped values.

Chapter 5

CONCLUSION

As was stated at the beginning of this thesis, the guiding hypothesis for the work presented herein is given as: Assuming a cryptographic hash is being used to increase the apparant randomness of a data set, It is possible to formulate metrics to choose the best hash for this purpose. This body of work found suitable metrics in the form of the Entropy and Serial Correlation of effected values. The conclusion of this work is that the hypothesis holds, and suitable metrics were formulated and verified.

5.1. Future Work

The primary contribution of this work was the ability to differentiate which hashes are able to provide a boost to the desired aspects of random numbers. In the body of this work 14 of the common cryptographic hash functions are compared using statistical procedures that show greater improvement of entropy, serial correlation, and chi-squared test statistic. The logical next step would be to show that these property improving qualities of hash functions hold across multiple different 'generators'. For instance, perhaps drawing strings from the linux /dev/random utility and hashing them, as well as pulling multiple strings from true random generators to ensure that the improvement qualities of the hash functions extend across other sources as well.

Another direction to take this work is monitoring the property changes when multiple different hashes are chained together. This would allow us to determine if chaining multiple hashes together will provide any discernible advantage over using a single hash, or perhaps simply iterating through the same hash multiple times.

During the body of this work it became quickly apparent that there are a significant number of similarities between cryptographic hash functions and pseudo-random number generators. For instance, one could produce a pseudo-random number generator from any one-way function by continually iterating the hash ouput back through the original hash. [14]. Further work in the realm might seek to examine the applicability of different hash functions are random number generators by using the test suites for random generators (I.E. NIST STS and Diehard). Or on the other hand, might seek to use extant pseudo-random number generators as cryptographic hash functions with variable output length which is a multiple of the input length. The properties of diffusion and confusion as well as the strict avalanche criterion could be shown to be met for pseudo-random number generators which behave as cryptographic hash functions (i.e. producing the 'hash' of the seed).

Another potential next step would be to use a genetic algorithm to evolve cryptographic hash functions using fitness criteria that are based on the apparent randomness metrics such as entropy and serial correlation. The result of that work would be the ability to construct cryptographic hash functions with the desired ability to increase the randomness metrics of supplied values. This would allow one to construct algorithms to quickly come up with new families of cryptographic hash functions.

Appendix A

Hashed Network Data

Table A.1. Pure Entropic String ENT results

	Entropy		Arithmetic Mean		Serial Correlation		Compression		
	bits/byte	*bits/bit*	*by byte*	*by bit*	*by byte*	*by bit*	*size (b)*	*comp. size (b)*	*ratio (%)*
Windows 10									
Capture 1	6.666783	0.999995	126.5384	0.4986	-0.104782	0.207267	2200	2200	0
Capture 2	7.196086	1.000000	125.5073	0.4998	0.083004	0.209302	5504	5504	0
Capture 3	7.362089	1.000000	126.5384	0.4997	0.374569	0.232915	11472	11472	0
Mac OS X 12.10									
Capture 1	7.198828	1.000000	125.7977	0.5000	0.326609	0.070809	5536	5536	0
Capture 2	7.229747	1.000000	126.3883	0.5000	0.249999	0.119658	6552	6552	0
Capture 3	3.843544	0.980664	71.4336	0.4183	0.397400	0.018324	11680	11563	1
Ubuntu Linux 16.10									
Capture 1	7.229747	1.000000	126.3883	0.5000	0.249999	0.119658	6552	6552	0
Capture 2	6.973394	0.999999	127.8026	0.4993	0.313810	0.307581	5592	5592	0
Capture 3	5.304753	1.000000	128.1367	0.5001	-0.003104	0.682508	50080	50080	0
Averages									
Linux	6.503	1.0	127.4	0.4998	0.188971	0.3699	-	-	0
Mac	6.091	0.9936	107.87	0.4728	0.3247	0.06960	-	-	0.3333
PC	7.075	1.0	126.2	0.4994	0.11760	0.2165	-	-	0
Reference									
Hotbits	7.916369	0.999995	128.7873	0.4987	0.031555	0.000973	16320	16320	0
Ideal	8.0	1.0	127.5	0.5	0.0	0.0	-	-	0
Cross Platform Average	6.5563	0.997867	120.49	0.49067	0.21042367	0.21867	-	-	1

Table of Hashed Packet Data ENT results (by bits)

	Operating.System	Hash	Size	Entropy	Chi.Sq.	Mean	MC.Pi	Serial.Correlation
1	Linux	bl2b	6656	0.999993	0.060096	0.498498	3.217391	-0.013230
2	Linux	bl2b	5632	0.999960	0.313210	0.503729	3.247863	-0.004317
3	Linux	bl2b	50176	0.999996	0.277503	0.498824	3.058373	0.004060
4	Windows	bl2b	2560	0.999901	0.351562	0.505859	3.396226	0.018615
5	Windows	bl2b	5632	0.999923	0.597301	0.505149	2.974359	0.008418
6	Windows	bl2b	11776	0.999970	0.490489	0.496773	3.134694	0.000638
7	Mac	bl2b	5632	0.999997	0.025568	0.498935	3.111111	-0.004266
8	Mac	bl2b	6656	0.999993	0.060096	0.498498	3.217391	-0.013230
9	Mac	bl2b	11776	0.999928	1.182405	0.494990	3.118367	-0.002139
10	Linux	bl2s	6656	0.999933	0.615385	0.504808	3.304348	0.023347
11	Linux	bl2s	5632	0.999974	0.205256	0.496982	3.145299	-0.008559
12	Linux	bl2s	50176	1.000000	0.006457	0.499821	3.196172	-0.000160
13	Windows	bl2s	2304	0.999783	0.694444	0.508681	3.000000	-0.022878
14	Windows	bl2s	5632	0.999977	0.181818	0.497159	3.145299	-0.003584
15	Windows	bl2s	11520	0.999996	0.068056	0.501215	3.116667	-0.016325
16	Mac	bl2s	5632	0.999956	0.343750	0.496094	3.179487	-0.011425
17	Mac	bl2s	6656	0.999933	0.615385	0.504808	3.304348	0.023347
18	Mac	bl2s	11776	0.999992	0.135870	0.498302	3.036735	0.013915
19	Linux	md5	6656	0.999850	1.384615	0.492788	3.275362	-0.005618
20	Linux	md5	5632	0.999994	0.045455	0.498580	3.247863	-0.004980
21	Linux	md5	50176	0.999944	3.928890	0.504424	3.123445	0.006220
22	Windows	md5	2304	0.999986	0.043403	0.502170	2.750000	-0.008700
23	Windows	md5	5504	0.999790	1.605378	0.491461	3.157895	-0.006107
24	Windows	md5	11520	0.999948	0.833681	0.495747	3.100000	-0.010490
25	Mac	md5	5632	0.999934	0.517756	0.495206	3.384615	0.030451
26	Mac	md5	6656	0.999850	1.384615	0.492788	3.275362	-0.005618
27	Mac	md5	11776	0.999962	0.628057	0.496349	3.102041	-0.003111
28	Linux	s1	6560	0.999846	1.404878	0.492683	3.264706	0.005885
29	Linux	s1	5600	0.999853	1.142857	0.507143	3.137931	-0.011635
30	Linux	s1	50080	0.999993	0.511182	0.498403	3.137105	0.001667

31	Windows	s1	2240	0.999917	0.257143	0.494643	3.217391	-0.005473
32	Windows	s1	5600	0.999860	1.086429	0.493036	3.206897	0.015523
33	Windows	s1	11520	0.999969	0.501389	0.503299	3.066667	-0.005599
34	Mac	s1	5600	0.999923	0.600714	0.505179	3.275862	-0.000822
35	Mac	s1	6560	0.999846	1.404878	0.492683	3.264706	0.005885
36	Mac	s1	11680	1.000000	0.003082	0.500257	3.209877	-0.001713
37	Linux	s2224	6720	0.999998	0.021429	0.500893	3.057143	0.011306
38	Linux	s2224	5600	0.999994	0.045714	0.498571	3.275862	-0.006437
39	Linux	s2224	50176	0.999959	2.817602	0.503747	3.154067	-0.002288
40	Windows	s2224	2240	0.999792	0.644643	0.491518	3.478261	-0.019936
41	Windows	s2224	5600	0.999830	1.320714	0.507679	2.724138	0.002622
42	Windows	s2224	11648	0.999808	3.099245	0.508156	3.123967	0.004886
43	Mac	s2224	5600	0.999845	1.200714	0.507321	2.931034	0.019790
44	Mac	s2224	6720	0.999998	0.021429	0.500893	3.057143	0.011306
45	Mac	s2224	11872	0.999976	0.389488	0.497136	3.206478	-0.012163
46	Linux	s2512	6656	0.999741	2.385216	0.490535	3.275362	-0.000358
47	Linux	s2512	5632	0.999971	0.230114	0.496804	3.589744	-0.005013
48	Linux	s2512	50176	0.999998	0.121253	0.500777	3.027751	-0.006858
49	Windows	s2512	2560	0.999841	0.564063	0.492578	3.245283	-0.001783
50	Windows	s2512	5632	0.999661	2.642756	0.510831	3.418803	-0.006865
51	Windows	s2512	11776	0.999970	0.490489	0.496773	3.069388	0.006412
52	Mac	s2512	5632	0.999847	1.193892	0.492720	3.008547	-0.011578
53	Mac	s2512	6656	0.999741	2.385216	0.490535	3.275362	-0.000358
54	Mac	s2512	11776	0.999904	1.570652	0.494226	3.232653	-0.002851
55	Linux	s256	6656	0.999896	0.961538	0.493990	3.159420	0.000457
56	Linux	s256	5632	0.999987	0.102273	0.497869	2.769231	-0.019195
57	Linux	s256	50176	0.999978	1.518176	0.497250	3.234450	-0.006408
58	Windows	s256	2304	0.999783	0.694444	0.491319	3.333333	0.004908
59	Windows	s256	5632	0.999715	2.227273	0.509943	3.076923	0.022341
60	Windows	s256	11520	0.999727	4.355556	0.509722	3.116667	-0.010799
61	Mac	s256	5632	0.999999	0.006392	0.499467	3.350427	-0.004263
62	Mac	s256	6656	0.999896	0.961538	0.493990	3.159420	0.000457

63	Mac	s256	11776	0.999880	1.961957	0.493546	3.151020	-0.005602
64	Linux	s3224	6720	0.999946	0.500595	0.504315	3.057143	0.013617
65	Linux	s3224	5600	0.999669	2.571429	0.489286	3.103448	0.019550
66	Linux	s3224	50176	0.999914	5.985013	0.494539	3.146411	-0.001953
67	Windows	s3224	2240	0.999998	0.007143	0.499107	3.652174	-0.017860
68	Windows	s3224	5600	0.999979	0.160714	0.502679	3.241379	-0.015029
69	Windows	s3224	11648	0.999936	1.038805	0.495278	3.206612	-0.006271
70	Mac	s3224	5600	0.999669	2.571429	0.510714	3.172414	0.007401
71	Mac	s3224	6720	0.999946	0.500595	0.504315	3.057143	0.013617
72	Mac	s3224	11872	0.999999	0.008423	0.499579	3.028340	0.003705
73	Linux	s3256	6656	0.999850	1.384615	0.507212	3.275362	-0.001410
74	Linux	s3256	5632	0.999980	0.159801	0.502663	2.905983	0.002102
75	Linux	s3256	50176	0.999988	0.829401	0.497967	3.211483	-0.006075
76	Windows	s3256	2304	0.999804	0.626736	0.491753	2.916667	-0.012428
77	Windows	s3256	5632	0.999980	0.159801	0.497337	2.632479	0.002813
78	Windows	s3256	11520	0.999978	0.355556	0.502778	2.900000	0.006219
79	Mac	s3256	5632	0.999934	0.517756	0.504794	3.145299	-0.029924
80	Mac	s3256	6656	0.999850	1.384615	0.507212	3.275362	-0.001410
81	Mac	s3256	11776	0.999983	0.285666	0.497537	3.248980	-0.010215
82	Linux	s3384	6912	1.000000	0.000579	0.500145	3.194444	0.017361
83	Linux	s3384	5760	0.999765	1.877778	0.490972	2.966667	-0.003105
84	Linux	s3384	50304	0.999958	2.900843	0.503797	3.171756	0.001612
85	Windows	s3384	2304	0.999893	0.340278	0.493924	3.500000	0.020689
86	Windows	s3384	5760	1.000000	0.000000	0.500000	3.266667	0.005556
87	Windows	s3384	11520	0.999996	0.068056	0.498785	3.300000	-0.018409
88	Mac	s3384	5760	0.999727	2.177778	0.490278	3.200000	0.004485
89	Mac	s3384	6912	1.000000	0.000579	0.500145	3.194444	0.017361
90	Mac	s3384	11904	0.999996	0.065860	0.498824	3.145161	0.001003
91	Linux	s3512	6656	0.999998	0.021635	0.500901	3.043478	0.007208
92	Linux	s3512	5632	0.999247	5.881392	0.516158	3.111111	0.011041
93	Linux	s3512	50176	0.999997	0.191406	0.499023	3.184689	0.001591
94	Windows	s3512	2560	0.999989	0.039062	0.501953	3.169811	-0.000015

95	Windows	s3512	5632	0.999974	0.205256	0.503018	3.111111	-0.000036
96	Windows	s3512	11776	1.000000	0.000340	0.499915	3.363265	-0.004755
97	Mac	s3512	5632	0.999967	0.256392	0.503374	3.247863	-0.005728
98	Mac	s3512	6656	0.999998	0.021635	0.500901	3.043478	0.007208
99	Mac	s3512	11776	0.999883	1.910666	0.506369	3.053061	-0.002540
100	Linux	s384	6912	0.999978	0.208912	0.502749	3.361111	-0.000030
101	Linux	s384	5760	0.999854	1.167361	0.507118	3.000000	-0.000203
102	Linux	s384	50304	0.999985	1.015347	0.497754	3.152672	0.010874
103	Windows	s384	2304	0.999760	0.765625	0.490885	3.333333	-0.009016
104	Windows	s384	5760	0.999893	0.850694	0.506076	2.800000	0.008187
105	Windows	s384	11520	0.999905	1.512500	0.494271	3.133333	0.006467
106	Mac	s384	5760	0.999993	0.056250	0.498437	3.133333	-0.002788
107	Mac	s384	6912	0.999978	0.208912	0.502749	3.361111	-0.000030
108	Mac	s384	11904	0.999999	0.008401	0.500420	3.080645	-0.008065
109	Linux	ske128	53248	0.999998	0.126277	0.500770	3.184851	-0.006388
110	Linux	ske128	45056	0.999986	0.852628	0.502175	3.087420	0.000070
111	Linux	ske128	401408	0.999993	3.671566	0.501512	3.097823	-0.002929
112	Windows	ske128	18432	0.999917	2.126953	0.505371	3.041667	-0.005975
113	Windows	ske128	44032	0.999998	0.093023	0.500727	3.097056	0.003450
114	Windows	ske128	92160	0.999999	0.117361	0.500564	3.104167	-0.003387
115	Mac	ske128	45056	0.999999	0.079901	0.500666	3.223881	-0.007726
116	Mac	ske128	53248	0.999998	0.126277	0.500770	3.184851	-0.006388
117	Mac	ske128	94208	0.999986	1.854662	0.502218	3.080530	-0.001463
118	Linux	ske256	53248	0.999989	0.844050	0.498009	3.065825	0.001036
119	Linux	ske256	45056	0.999984	1.016424	0.497625	3.159915	0.000421
120	Linux	ske256	401408	0.999996	2.000000	0.501116	3.106912	0.000324
121	Windows	ske256	18432	0.999915	2.170139	0.505425	3.166667	0.004874
122	Windows	ske256	45056	0.999998	0.096680	0.499268	3.070362	-0.002843
123	Windows	ske256	92160	0.999999	0.108507	0.499457	3.133333	-0.003039
124	Mac	ske256	45056	0.999976	1.523526	0.502907	3.113006	-0.001099
125	Mac	ske256	53248	0.999989	0.844050	0.498009	3.065825	0.001036

126	Mac	ske256	94208	0.999996	0.495245	0.498854	3.233435	-0.002680

Table of Hashed Packet Data ENT results (by Bytes)

	Operating.System	Hash	Size	Entropy	Chi.Sq.	Mean	MC.Pi	Serial.Correlation
1	Linux	bl2b	832	7.710039	300.307692	127.264423	3.217391	-0.018633
2	Linux	bl2b	704	7.702965	261.818182	126.428977	3.247863	0.014207
3	Linux	bl2b	6272	7.969971	254.938776	127.346301	3.058373	0.007395
4	Windows	bl2b	320	7.270584	291.200000	121.150000	3.396226	0.056477
5	Windows	bl2b	704	7.715849	242.181818	129.509943	2.974359	-0.069928
6	Windows	bl2b	1472	7.892107	212.869565	127.847826	3.134694	0.016110
7	Mac	bl2b	704	7.713464	267.636364	129.460227	3.111111	-0.042946
8	Mac	bl2b	832	7.710039	300.307692	127.264423	3.217391	-0.018633
9	Mac	bl2b	1472	7.876650	254.608696	127.368886	3.118367	0.004450
10	Linux	bl2s	832	7.773328	238.769231	125.485577	3.304348	0.040503
11	Linux	bl2s	704	7.738005	226.909091	127.433239	3.145299	0.006780
12	Linux	bl2s	6272	7.972498	240.000000	127.052136	3.196172	0.005147
13	Windows	bl2s	288	7.304656	238.222222	128.576389	3.000000	0.050329
14	Windows	bl2s	704	7.660487	285.818182	130.575284	3.145299	-0.025152
15	Windows	bl2s	1440	7.843532	293.688889	126.505556	3.116667	0.006463
16	Mac	bl2s	704	7.729896	233.454545	125.386364	3.179487	0.030814
17	Mac	bl2s	832	7.773328	238.769231	125.485577	3.304348	0.040503
18	Mac	bl2s	1472	7.877831	232.000000	129.487092	3.036735	-0.037924
19	Linux	md5	832	7.792125	225.230769	125.064904	3.275362	0.023621
20	Linux	md5	704	7.750837	219.636364	127.099432	3.247863	-0.018354
21	Linux	md5	6272	7.975274	214.938776	129.631696	3.123445	-0.000403
22	Windows	md5	288	7.269933	247.111111	126.934028	2.750000	-0.043805
23	Windows	md5	688	7.694080	280.186047	127.311047	3.157895	0.075736
24	Windows	md5	1440	7.861282	275.200000	125.158333	3.100000	0.013610
25	Mac	md5	704	7.682353	280.727273	125.585227	3.384615	-0.002881
26	Mac	md5	832	7.792125	225.230769	125.064904	3.275362	0.023621
27	Mac	md5	1472	7.872019	252.869565	127.814538	3.102041	-0.019107

28	Linux	s1	820	7.758565	248.956098	129.296341	3.264706	-0.013552
29	Linux	s1	700	7.783396	185.760000	129.557143	3.137931	-0.001530
30	Linux	s1	6260	7.975602	214.100958	127.829393	3.137105	-0.000585
31	Windows	s1	280	7.199130	270.400000	124.496429	3.217391	-0.066604
32	Windows	s1	700	7.709263	263.291429	127.067143	3.206897	0.042612
33	Windows	s1	1440	7.874314	250.311111	127.495139	3.066667	-0.002471
34	Mac	s1	700	7.728470	230.377143	128.402857	3.275862	0.015743
35	Mac	s1	820	7.758565	248.956098	129.296341	3.264706	-0.013552
36	Mac	s1	1460	7.872787	250.290411	125.832877	3.209877	0.026565
37	Linux	s2224	840	7.753958	260.800000	128.400000	3.057143	0.057046
38	Linux	s2224	700	7.742857	220.868571	128.710000	3.275862	0.015311
39	Linux	s2224	6272	7.973649	227.265306	127.926977	3.154067	0.006535
40	Windows	s2224	280	7.278507	241.142857	123.903571	3.478261	-0.043458
41	Windows	s2224	700	7.723549	234.765714	130.487143	2.724138	-0.064134
42	Windows	s2224	1456	7.865446	275.164835	131.181319	3.123967	0.003073
43	Mac	s2224	700	7.692566	272.068571	131.611429	2.931034	-0.034132
44	Mac	s2224	840	7.753958	260.800000	128.400000	3.057143	0.057046
45	Mac	s2224	1484	7.868655	257.628032	126.528976	3.206478	-0.004081
46	Linux	s2512	832	7.704390	306.461538	126.534856	3.275362	-0.029305
47	Linux	s2512	704	7.670309	297.454545	123.305398	3.589744	0.006054
48	Linux	s2512	6272	7.967540	278.530612	129.594228	3.027751	0.013194
49	Windows	s2512	320	7.334577	259.200000	125.846875	3.245283	0.032896
50	Windows	s2512	704	7.695319	257.454545	127.291193	3.418803	0.060673
51	Windows	s2512	1472	7.871521	248.347826	128.235054	3.069388	0.031987
52	Mac	s2512	704	7.698028	257.454545	127.694602	3.008547	0.006978
53	Mac	s2512	832	7.704390	306.461538	126.534856	3.275362	-0.029305
54	Mac	s2512	1472	7.891616	213.913043	128.177310	3.232653	0.022711
55	Linux	s256	832	7.779267	245.538462	126.610577	3.159420	-0.000649
56	Linux	s256	704	7.710811	245.090909	129.092330	2.769231	0.000002
57	Linux	s256	6272	7.971919	239.265306	126.325733	3.234450	-0.002037
58	Windows	s256	288	7.234864	264.888889	130.118056	3.333333	-0.047004
59	Windows	s256	704	7.683751	276.363636	129.930398	3.076923	-0.076702

60	Windows	s256	1440	7.880827	232.177778	132.816667	3.116667	-0.004271
61	Mac	s256	704	7.695856	273.454545	125.764205	3.350427	-0.022197
62	Mac	s256	832	7.779267	245.538462	126.610577	3.159420	-0.000649
63	Mac	s256	1472	7.878207	224.347826	124.204484	3.151020	-0.025419
64	Linux	s3224	840	7.778400	236.419048	128.988095	3.057143	0.013207
65	Linux	s3224	700	7.702655	259.634286	124.245714	3.103448	-0.044994
66	Linux	s3224	6272	7.967804	275.918367	125.329401	3.146411	-0.000142
67	Windows	s3224	280	7.242436	252.114286	123.835714	3.652174	-0.010893
68	Windows	s3224	700	7.739539	247.200000	129.010000	3.241379	0.020837
69	Windows	s3224	1456	7.859315	270.593407	126.365385	3.206612	0.008682
70	Mac	s3224	700	7.683715	265.485714	129.378571	3.172414	-0.002125
71	Mac	s3224	840	7.778400	236.419048	128.988095	3.057143	0.013207
72	Mac	s3224	1484	7.877415	241.067385	127.932615	3.028340	0.003752
73	Linux	s3256	832	7.753414	256.000000	128.602163	3.275362	0.048506
74	Linux	s3256	704	7.741961	224.000000	131.028409	2.905983	0.047420
75	Linux	s3256	6272	7.965849	289.632653	126.299107	3.211483	0.020687
76	Windows	s3256	288	7.234292	259.555556	128.000000	2.916667	0.061016
77	Windows	s3256	704	7.703964	264.727273	127.936080	2.632479	-0.005221
78	Windows	s3256	1440	7.866361	254.222222	130.946528	2.900000	0.025603
79	Mac	s3256	704	7.687196	277.818182	128.656250	3.145299	0.036072
80	Mac	s3256	832	7.753414	256.000000	128.602163	3.275362	0.048506
81	Mac	s3256	1472	7.855335	288.000000	128.259511	3.248980	0.038541
82	Linux	s3384	864	7.761851	254.814815	128.918981	3.194444	-0.015815
83	Linux	s3384	720	7.715698	252.800000	127.777778	2.966667	-0.041907
84	Linux	s3384	6288	7.972837	232.427481	128.986641	3.171756	0.004582
85	Windows	s3384	288	7.213459	282.666667	124.416667	3.500000	-0.015012
86	Windows	s3384	720	7.747280	223.644444	129.375000	3.266667	-0.042779
87	Windows	s3384	1440	7.859569	267.377778	127.314583	3.300000	-0.017518
88	Mac	s3384	720	7.763615	215.822222	124.181944	3.200000	-0.007164
89	Mac	s3384	864	7.761851	254.814815	128.918981	3.194444	-0.015815
90	Mac	s3384	1488	7.871211	258.236559	125.281586	3.145161	0.009742
91	Linux	s3512	832	7.766552	248.000000	128.135817	3.043478	0.021811

92	Linux	s3512	704	7.662472	268.363636	131.238636	3.111111	-0.052930
93	Linux	s3512	6272	7.971745	243.428571	126.980230	3.184689	-0.015177
94	Windows	s3512	320	7.396562	222.400000	123.125000	3.169811	-0.082289
95	Windows	s3512	704	7.731868	242.181818	130.046875	3.111111	-0.087689
96	Windows	s3512	1472	7.882993	233.043478	125.650136	3.363265	-0.028381
97	Mac	s3512	704	7.761054	214.545455	126.816761	3.247863	-0.022706
98	Mac	s3512	832	7.766552	248.000000	128.135817	3.043478	0.021811
99	Mac	s3512	1472	7.862705	269.913043	125.770380	3.053061	-0.044422
100	Linux	s384	864	7.773845	247.703704	128.181713	3.361111	-0.021854
101	Linux	s384	720	7.736049	244.977778	133.411111	3.000000	0.016837
102	Linux	s384	6288	7.968660	275.175573	126.286896	3.152672	-0.008539
103	Windows	s384	288	7.278580	254.222222	121.520833	3.333333	0.024395
104	Windows	s384	720	7.759210	215.822222	128.675000	2.800000	0.015283
105	Windows	s384	1440	7.873597	243.555556	127.036111	3.133333	0.026074
106	Mac	s384	720	7.672383	269.866667	126.418056	3.133333	-0.033970
107	Mac	s384	864	7.773845	247.703704	128.181713	3.361111	-0.021854
108	Mac	s384	1488	7.865237	265.462366	128.364919	3.080645	0.006206
109	Linux	ske128	6656	7.970905	265.846154	126.759465	3.184851	-0.000201
110	Linux	ske128	5632	7.967123	258.909091	127.199929	3.087420	0.014356
111	Linux	ske128	50176	7.996414	249.459184	128.226244	3.097823	0.003339
112	Windows	ske128	2304	7.916286	264.666667	128.543837	3.041667	-0.031410
113	Windows	ske128	5504	7.967025	251.534884	127.953852	3.097056	-0.010840
114	Windows	ske128	11520	7.982189	278.977778	127.773003	3.104167	-0.007903
115	Mac	ske128	5632	7.974622	200.090909	127.500888	3.223881	0.000357
116	Mac	ske128	6656	7.970905	265.846154	126.759465	3.184851	-0.000201
117	Mac	ske128	11776	7.983905	257.739130	127.984800	3.080530	-0.009015
118	Linux	ske256	6656	7.977339	207.000000	127.836088	3.065825	-0.004594
119	Linux	ske256	5632	7.964023	281.090909	125.890980	3.159915	0.010597
120	Linux	ske256	50176	7.996496	241.836735	127.928890	3.106912	0.003930
121	Windows	ske256	2304	7.919073	252.888889	127.405382	3.166667	0.007476
122	Windows	ske256	5632	7.967695	247.545455	128.561435	3.070362	0.008183
123	Windows	ske256	11520	7.983745	259.955556	127.478125	3.133333	-0.003942

124	Mac	ske256	5632	7.964904	271.000000	128.811435	3.113006	-0.023878
125	Mac	ske256	6656	7.977339	207.000000	127.836088	3.065825	-0.004594
126	Mac	ske256	11776	7.983456	265.000000	126.689963	3.233435	-0.003222

Appendix B

Kruskal-Wallis Dunn Confidence Intervals for mean differences

Entropy

```
  Kruskal-Wallis rank sum test

data: x and group
Kruskal-Wallis chi-squared = 38.5698, df = 13, p-value = 0


                              Comparison of x by group
                                   (No adjustment)
Col Mean-|
Row Mean |       bl2b       bl2s        md5         s1      s2224      s2512
---------+------------------------------------------------------------------
    bl2s |  -0.284006
         |     0.3882
         |
     md5 |  -0.329189  -0.045182
         |     0.3710     0.4820
         |
      s1 |  -0.490557  -0.206550  -0.161367
         |     0.3119     0.4182     0.4359
         |
   s2224 |  -0.187186   0.096820   0.142003   0.303371
         |     0.4258     0.4614     0.4435     0.3808
         |
   s2512 |   0.309825   0.593832   0.639015   0.800383   0.497012
         |     0.3783     0.2763     0.2614     0.2117     0.3096
         |
    s256 |  -0.329189  -0.045182   0.000000   0.161367  -0.142003  -0.639015
         |     0.3710     0.4820     0.5000     0.4359     0.4435     0.2614
         |
   s3224 |  -0.213005   0.071001   0.116184   0.277552  -0.025818  -0.522831
```

```
           |     0.4157     0.4717     0.4538     0.3907     0.4897     0.3005
           |
     s3256 |   0.083911   0.367918   0.413101   0.574468   0.271097  -0.225914
           |     0.4666     0.3565     0.3398     0.2828     0.3932     0.4106
           |
     s3384 |  -0.438919  -0.154912  -0.109729   0.051637  -0.251733  -0.748745
           |     0.3304     0.4384     0.4563     0.4794     0.4006     0.2270
           |
     s3512 |  -0.406646  -0.122639  -0.077456   0.083911  -0.219459  -0.716472
           |     0.3421     0.4512     0.4691     0.4666     0.4131     0.2368
           |
      s384 |  -0.387282  -0.103275  -0.058092   0.103275  -0.200095  -0.697108
           |     0.3493     0.4589     0.4768     0.4589     0.4207     0.2429
           |
    ske128 |  -3.504904  -3.220897  -3.175714  -3.014346  -3.317718  -3.814730
           |     0.0002     0.0006     0.0007     0.0013     0.0005     0.0001
           |
    ske256 |  -3.537177  -3.253171  -3.207988  -3.046620  -3.349991  -3.847003
           |     0.0002     0.0006     0.0007     0.0012     0.0004     0.0001
Col Mean-|
Row Mean |       s256      s3224      s3256      s3384      s3512       s384
---------+------------------------------------------------------------------
     s3224 |   0.116184
           |     0.4538
           |
     s3256 |   0.413101   0.296916
           |     0.3398     0.3833
           |
     s3384 |  -0.109729  -0.225914  -0.522831
           |     0.4563     0.4106     0.3005
           |
     s3512 |  -0.077456  -0.193641  -0.490557   0.032273
           |     0.4691     0.4232     0.3119     0.4871
           |
      s384 |  -0.058092  -0.174277  -0.471193   0.051637   0.019364
           |     0.4768     0.4308     0.3188     0.4794     0.4923
           |
    ske128 |  -3.175714  -3.291899  -3.588815  -3.065984  -3.098258  -3.117622
           |     0.0007     0.0005     0.0002     0.0011     0.0010     0.0009
           |
```

```
  ske256 |  -3.207988   -3.324172   -3.621089   -3.098258   -3.130531   -3.149895
         |     0.0007      0.0004      0.0001      0.0010      0.0009      0.0008
Col Mean-|
Row Mean |     ske128
---------+-----------
  ske256 |  -0.032273
         |     0.4871
```

Serial Correlation

```
  Kruskal-Wallis rank sum test

data: x and group
Kruskal-Wallis chi-squared = 30.198, df = 13, p-value = 0


                          Comparison of x by group
                               (No adjustment)
Col Mean-|
Row Mean |       bl2b        bl2s         md5          s1       s2224       s2512
---------+------------------------------------------------------------------------
    bl2s |  -1.207029
         |     0.1137
         |
     md5 |  -0.464738    0.742291
         |     0.3211      0.2290
         |
      s1 |  -0.232369    0.974660    0.232369
         |     0.4081      0.1649      0.4081
         |
   s2224 |  -0.193641    1.013388    0.271097    0.038728
         |     0.4232      0.1554      0.3932      0.4846
         |
   s2512 |  -1.219939   -0.012909   -0.755200   -0.987569   -1.026297
         |     0.1112      0.4949      0.2251      0.1617      0.1524
         |
    s256 |   1.213484    2.420514    1.678223    1.445853    1.407125    2.433423
```

```
         |     0.1125     0.0077     0.0467     0.0741     0.0797     0.0075
         |
   s3224 |  -0.387282   0.819747   0.077456  -0.154912  -0.193641   0.832656
         |     0.3493     0.2062     0.4691     0.4384     0.4232     0.2025
         |
   s3256 |  -2.672247  -1.465217  -2.207508  -2.439878  -2.478606  -1.452308
         |     0.0038     0.0714     0.0136     0.0073     0.0066     0.0732
         |
   s3384 |   1.065026   2.272055   1.529764   1.297395   1.258667   2.284965
         |     0.1434     0.0115     0.0630     0.0972     0.1041     0.0112
         |
   s3512 |   1.516855   2.723885   1.981594   1.749224   1.710496   2.736794
         |     0.0647     0.0032     0.0238     0.0401     0.0436     0.0031
         |
    s384 |  -0.361463   0.845566   0.103275  -0.129094  -0.167822   0.858475
         |     0.3589     0.1989     0.4589     0.4486     0.4334     0.1953
         |
  ske128 |   0.238824   1.445853   0.703562   0.471193   0.432465   1.458763
         |     0.4056     0.0741     0.2409     0.3188     0.3327     0.0723
         |
  ske256 |  -0.051637   1.155392   0.413101   0.180731   0.142003   1.168301
         |     0.4794     0.1240     0.3398     0.4283     0.4435     0.1213
Col Mean-|
Row Mean |       s256      s3224      s3256      s3384      s3512       s384
---------+------------------------------------------------------------------
   s3224 |  -1.600766
         |     0.0547
         |
   s3256 |  -3.885732  -2.284965
         |     0.0001     0.0112
         |
   s3384 |  -0.148458   1.452308   3.737273
         |     0.4410     0.0732     0.0001
         |
   s3512 |   0.303371   1.904137   4.189103   0.451829
         |     0.3808     0.0284     0.0000     0.3257
         |
    s384 |  -1.574947   0.025818   2.310784  -1.426489  -1.878318
         |     0.0576     0.4897     0.0104     0.0769     0.0302
         |
```

```
  ske128 |  -0.974660   0.626106   2.911071  -0.826202  -1.278031   0.600287
         |     0.1649     0.2656     0.0018     0.2043     0.1006     0.2742
         |
  ske256 |  -1.265122   0.335644   2.620609  -1.116663  -1.568493   0.309825
         |     0.1029     0.3686     0.0044     0.1321     0.0584     0.3783
Col Mean-|
Row Mean |     ske128
---------+-----------
  ske256 |  -0.290461
         |     0.3857
```

Chi-Squared Test Statistic

```
  Kruskal-Wallis rank sum test

data: x and group
Kruskal-Wallis chi-squared = 13.7721, df = 13, p-value = 0.39


                           Comparison of x by group
                                (No adjustment)
Col Mean-|
Row Mean |       bl2b       bl2s        md5         s1      s2224      s2512
---------+------------------------------------------------------------------
    bl2s |   1.875102
         |     0.0304
         |
     md5 |   1.468454  -0.406648
         |     0.0710     0.3421
         |
      s1 |   1.842829  -0.032273   0.374375
         |     0.0327     0.4871     0.3541
         |
   s2224 |   1.165081  -0.710021  -0.303372  -0.677748
         |     0.1220     0.2388     0.3808     0.2490
         |
   s2512 |  -0.261417  -2.136519  -1.729871  -2.104246  -1.426498
```

```
             |   0.3969     0.0163     0.0418     0.0177     0.0769
             |
        s256 |  1.403906  -0.471196  -0.064547  -0.438922   0.238825   1.665323
             |   0.0802     0.3188     0.4743     0.3304     0.4056     0.0479
             |
       s3224 |  0.926255  -0.948847  -0.542198  -0.916573  -0.238825   1.187672
             |   0.1772     0.1713     0.2938     0.1797     0.4056     0.1175
             |
       s3256 | -0.064547  -1.939650  -1.533001  -1.907376  -1.229628   0.196869
             |   0.4743     0.0262     0.0626     0.0282     0.1094     0.4220
             |
       s3384 |  1.239310  -0.635792  -0.229143  -0.603518   0.074229   1.500727
             |   0.1076     0.2625     0.4094     0.2731     0.4704     0.0667
             |
       s3512 |  1.936422   0.061320   0.467968   0.093593   0.771341   2.197840
             |   0.0264     0.4756     0.3199     0.4627     0.2203     0.0140
             |
        s384 |  1.113443  -0.761659  -0.355010  -0.729385  -0.051637   1.374860
             |   0.1328     0.2231     0.3613     0.2329     0.4794     0.0846
             |
      ske128 |  0.455059  -1.420043  -1.013394  -1.387769  -0.710021   0.716476
             |   0.3245     0.0778     0.1554     0.0826     0.2388     0.2368
             |
      ske256 |  1.177990  -0.697112  -0.290463  -0.664838   0.012909   1.439407
             |   0.1194     0.2429     0.3857     0.2531     0.4949     0.0750
Col Mean-|
Row Mean |      s256      s3224      s3256      s3384      s3512       s384
---------+------------------------------------------------------------------
       s3224 | -0.477650
             |   0.3164
             |
       s3256 | -1.468454  -0.990803
             |   0.0710     0.1609
             |
       s3384 | -0.164595   0.313055   1.303858
             |   0.4346     0.3771     0.0961
             |
       s3512 |  0.532516   1.010167   2.000970   0.697112
             |   0.2972     0.1562     0.0227     0.2429
             |
```

```
    s384 |  -0.290463   0.187187   1.177990  -0.125867  -0.822979
         |     0.3857     0.4258     0.1194     0.4499     0.2053
         |
  ske128 |  -0.948847  -0.471196   0.519606  -0.784251  -1.481363  -0.658383
         |     0.1713     0.3188     0.3017     0.2164     0.0693     0.2551
         |
  ske256 |  -0.225916   0.251734   1.242538  -0.061320  -0.758432   0.064547
         |     0.4106     0.4006     0.1070     0.4756     0.2241     0.4743
Col Mean-|
Row Mean |     ske128
---------+-----------
  ske256 |   0.722931
         |     0.2349
```

Appendix C

Tukey Honestly Significant Differences

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = dfB$Serial.Correlation ~ dfB$Hash)

                 Df  Sum Sq    Mean Sq F value   Pr(>F)
dfB$Hash         13  0.03032  0.0023321    2.938  0.00104 **
Residuals       112  0.08891  0.0007938


$`dfB$Hash`
                        diff          lwr           upr     p adj
bl2s-bl2b       0.0187737778 -0.026766335  0.0643138905 0.9784332
md5-bl2b        0.0115043333 -0.034035779  0.0570444460 0.9998304
s1-bl2b         0.0042363333 -0.041303779  0.0497764460 1.0000000
s2224-bl2b      0.0049674444 -0.040572668  0.0505075572 1.0000000
s2512-bl2b      0.0185982222 -0.026941890  0.0641383349 0.9800751
s256-bl2b      -0.0141583333 -0.059698446  0.0313817794 0.9984545
s3224-bl2b      0.0058924444 -0.039647668  0.0514325572 0.9999999
s3256-bl2b      0.0414034444 -0.004136668  0.0869435572 0.1158168
s3384-bl2b     -0.0100205556 -0.055560668  0.0355195572 0.9999644
s3512-bl2b     -0.0264967778 -0.072036890  0.0190433349 0.7674090
s384-bl2b       0.0060087778 -0.039531335  0.0515488905 0.9999999
ske128-bl2b     0.0011092222 -0.044430890  0.0466493349 1.0000000
ske256-bl2b     0.0046063333 -0.040933779  0.0501464460 1.0000000
md5-bl2s       -0.0072694444 -0.052809557  0.0382706683 0.9999992
s1-bl2s        -0.0145374444 -0.060077557  0.0310026683 0.9979816
s2224-bl2s     -0.0138063333 -0.059346446  0.0317337794 0.9988056
s2512-bl2s     -0.0001755556 -0.045715668  0.0453645572 1.0000000
s256-bl2s      -0.0329321111 -0.078472224  0.0126080016 0.4315838
s3224-bl2s     -0.0128813333 -0.058421446  0.0326587794 0.9994217
s3256-bl2s      0.0226296667 -0.022910446  0.0681697794 0.9103458
s3384-bl2s     -0.0287943333 -0.074334446  0.0167457794 0.6528779
```

```
s3512-bl2s      -0.0452705556 -0.090810668 0.0002695572 0.0529957
s384-bl2s       -0.0127650000 -0.058305113 0.0327751127 0.9994749
ske128-bl2s     -0.0176645556 -0.063204668 0.0278755572 0.9872403
ske256-bl2s     -0.0141674444 -0.059707557 0.0313726683 0.9984444
s1-md5          -0.0072680000 -0.052808113 0.0382721127 0.9999992
s2224-md5       -0.0065368889 -0.052077002 0.0390032238 0.9999998
s2512-md5        0.0070938889 -0.038446224 0.0526340016 0.9999994
s256-md5        -0.0256626667 -0.071202779 0.0198774460 0.8043930
s3224-md5       -0.0056118889 -0.051152002 0.0399282238 1.0000000
s3256-md5        0.0298991111 -0.015641002 0.0754392238 0.5937301
s3384-md5       -0.0215248889 -0.067065002 0.0240152238 0.9369658
s3512-md5       -0.0380011111 -0.083541224 0.0075390016 0.2107740
s384-md5        -0.0054955556 -0.051035668 0.0400445572 1.0000000
ske128-md5      -0.0103951111 -0.055935224 0.0351450016 0.9999457
ske256-md5      -0.0068980000 -0.052438113 0.0386421127 0.9999996
s2224-s1         0.0007311111 -0.044809002 0.0462712238 1.0000000
s2512-s1         0.0143618889 -0.031178224 0.0599020016 0.9982140
s256-s1         -0.0183946667 -0.063934779 0.0271454460 0.9818554
s3224-s1         0.0016561111 -0.043884002 0.0471962238 1.0000000
s3256-s1         0.0371671111 -0.008373002 0.0827072238 0.2407367
s3384-s1        -0.0142568889 -0.059797002 0.0312832238 0.9983417
s3512-s1        -0.0307331111 -0.076273224 0.0148070016 0.5485149
s384-s1          0.0017724444 -0.043767668 0.0473125572 1.0000000
ske128-s1       -0.0031271111 -0.048667224 0.0424130016 1.0000000
ske256-s1        0.0003700000 -0.045170113 0.0459101127 1.0000000
s2512-s2224      0.0136307778 -0.031909335 0.0591708905 0.9989535
s256-s2224      -0.0191257778 -0.064665890 0.0264143349 0.9748280
s3224-s2224      0.0009250000 -0.044615113 0.0464651127 1.0000000
s3256-s2224      0.0364360000 -0.009104113 0.0819761127 0.2692148
s3384-s2224     -0.0149880000 -0.060528113 0.0305521127 0.9972652
s3512-s2224     -0.0314642222 -0.077004335 0.0140758905 0.5090041
s384-s2224       0.0010413333 -0.044498779 0.0465814460 1.0000000
ske128-s2224    -0.0038582222 -0.049398335 0.0416818905 1.0000000
ske256-s2224    -0.0003611111 -0.045901224 0.0451790016 1.0000000
s256-s2512      -0.0327565556 -0.078296668 0.0127835572 0.4406425
s3224-s2512     -0.0127057778 -0.058245890 0.0328343349 0.9995003
s3256-s2512      0.0228052222 -0.022734890 0.0683453349 0.9055285
s3384-s2512     -0.0286187778 -0.074158890 0.0169213349 0.6621147
s3512-s2512     -0.0450950000 -0.090635113 0.0004451127 0.0550289
s384-s2512      -0.0125894444 -0.058129557 0.0329506683 0.9995471
```

```
ske128-s2512   -0.0174890000 -0.063029113  0.0280511127 0.9883224
ske256-s2512   -0.0139918889 -0.059532002  0.0315482238 0.9986302
s3224-s256      0.0200507778 -0.025489335  0.0655908905 0.9631623
s3256-s256      0.0555617778  0.010021665  0.1011018905 0.0042898
s3384-s256      0.0041377778 -0.041402335  0.0496778905 1.0000000
s3512-s256     -0.0123384444 -0.057878557  0.0332016683 0.9996354
s384-s256       0.0201671111 -0.025373002  0.0657072238 0.9614513
ske128-s256     0.0152675556 -0.030272557  0.0608076683 0.9967211
ske256-s256     0.0187646667 -0.026775446  0.0643047794 0.9785209
s3256-s3224     0.0355110000 -0.010029113  0.0810511127 0.3081264
s3384-s3224    -0.0159130000 -0.061453113  0.0296271127 0.9951117
s3512-s3224    -0.0323892222 -0.077929335  0.0131508905 0.4597982
s384-s3224      0.0001163333 -0.045423779  0.0456564460 1.0000000
ske128-s3224   -0.0047832222 -0.050323335  0.0407568905 1.0000000
ske256-s3224   -0.0012861111 -0.046826224  0.0442540016 1.0000000
s3384-s3256    -0.0514240000 -0.096964113 -0.0058838873 0.0126122
s3512-s3256    -0.0679002222 -0.113440335 -0.0223601095 0.0001123
s384-s3256     -0.0353946667 -0.080934779  0.0101454460 0.3132401
ske128-s3256   -0.0402942222 -0.085834335  0.0052458905 0.1421496
ske256-s3256   -0.0367971111 -0.082337224  0.0087430016 0.2548928
s3512-s3384    -0.0164762222 -0.062016335  0.0290638905 0.9932179
s384-s3384      0.0160293333 -0.029510779  0.0615694460 0.9947615
ske128-s3384    0.0111297778 -0.034410335  0.0566698905 0.9998826
ske256-s3384    0.0146268889 -0.030913224  0.0601670016 0.9978537
s384-s3512      0.0325055556 -0.013034557  0.0780456683 0.4537038
ske128-s3512    0.0276060000 -0.017934113  0.0731461127 0.7140526
ske256-s3512    0.0311031111 -0.014437002  0.0766432238 0.5284778
ske128-s384    -0.0048995556 -0.050439668  0.0406405572 1.0000000
ske256-s384    -0.0014024444 -0.046942557  0.0441376683 1.0000000
ske256-ske128   0.0034971111 -0.042043002  0.0490372238 1.0000000
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = dfB$Entropy ~ dfB$Hash)

             Df Sum Sq Mean Sq F value Pr(>F)
dfB$Hash     13  0.835 0.06423   1.891 0.0383 *
```

```
Residuals    112   3.803  0.03396
---

$`dfB$Hash`
                      diff         lwr       upr     p adj
bl2s-bl2b     1.243256e-02 -0.28542014 0.3102852 1.0000000
md5-bl2b      1.426222e-02 -0.28359047 0.3121149 1.0000000
s1-bl2b       1.093600e-02 -0.28691669 0.3087887 1.0000000
s2224-bl2b    1.016411e-02 -0.28768858 0.3080168 1.0000000
s2512-bl2b   -2.664222e-03 -0.30051692 0.2951885 1.0000000
s256-bl2b     5.900111e-03 -0.29195258 0.3037528 1.0000000
s3224-bl2b    7.556778e-03 -0.29029592 0.3054095 1.0000000
s3256-bl2b    1.311111e-05 -0.29783958 0.2978658 1.0000000
s3384-bl2b    1.174478e-02 -0.28610792 0.3095975 1.0000000
s3512-bl2b    2.675944e-02 -0.27109325 0.3246121 1.0000000
s384-bl2b     1.552644e-02 -0.28232625 0.3133791 1.0000000
ske128-bl2b   2.408562e-01 -0.05699647 0.5387089 0.2537821
ske256-bl2b   2.413780e-01 -0.05647469 0.5392307 0.2506940
md5-bl2s      1.829667e-03 -0.29602303 0.2996824 1.0000000
s1-bl2s      -1.496556e-03 -0.29934925 0.2963561 1.0000000
s2224-bl2s   -2.268444e-03 -0.30012114 0.2955842 1.0000000
s2512-bl2s   -1.509678e-02 -0.31294947 0.2827559 1.0000000
s256-bl2s    -6.532444e-03 -0.30438514 0.2913202 1.0000000
s3224-bl2s   -4.875778e-03 -0.30272847 0.2929769 1.0000000
s3256-bl2s   -1.241944e-02 -0.31027214 0.2854332 1.0000000
s3384-bl2s   -6.877778e-04 -0.29854047 0.2971649 1.0000000
s3512-bl2s    1.432689e-02 -0.28352581 0.3121796 1.0000000
s384-bl2s     3.093889e-03 -0.29475881 0.3009466 1.0000000
ske128-bl2s   2.284237e-01 -0.06942903 0.5262764 0.3343741
ske256-bl2s   2.289454e-01 -0.06890725 0.5267981 0.3307328
s1-md5       -3.326222e-03 -0.30117892 0.2945265 1.0000000
s2224-md5    -4.098111e-03 -0.30195081 0.2937546 1.0000000
s2512-md5    -1.692644e-02 -0.31477914 0.2809262 1.0000000
s256-md5     -8.362111e-03 -0.30621481 0.2894906 1.0000000
s3224-md5    -6.705444e-03 -0.30455814 0.2911472 1.0000000
s3256-md5    -1.424911e-02 -0.31210181 0.2836036 1.0000000
s3384-md5    -2.517444e-03 -0.30037014 0.2953352 1.0000000
s3512-md5     1.249722e-02 -0.28535547 0.3103499 1.0000000
s384-md5      1.264222e-03 -0.29658847 0.2991169 1.0000000
ske128-md5    2.265940e-01 -0.07125869 0.5244467 0.3473093
```

```
ske256-md5      2.271158e-01 -0.07073692 0.5249685 0.3435944
s2224-s1       -7.718889e-04 -0.29862458 0.2970808 1.0000000
s2512-s1       -1.360022e-02 -0.31145292 0.2842525 1.0000000
s256-s1        -5.035889e-03 -0.30288858 0.2928168 1.0000000
s3224-s1       -3.379222e-03 -0.30123192 0.2944735 1.0000000
s3256-s1       -1.092289e-02 -0.30877558 0.2869298 1.0000000
s3384-s1        8.087778e-04 -0.29704392 0.2986615 1.0000000
s3512-s1        1.582344e-02 -0.28202925 0.3136761 1.0000000
s384-s1         4.590444e-03 -0.29326225 0.3024431 1.0000000
ske128-s1       2.299202e-01 -0.06793247 0.5277729 0.3239880
ske256-s1       2.304420e-01 -0.06741069 0.5282947 0.3204089
s2512-s2224    -1.282833e-02 -0.31068103 0.2850244 1.0000000
s256-s2224     -4.264000e-03 -0.30211669 0.2935887 1.0000000
s3224-s2224    -2.607333e-03 -0.30046003 0.2952454 1.0000000
s3256-s2224    -1.015100e-02 -0.30800369 0.2877017 1.0000000
s3384-s2224     1.580667e-03 -0.29627203 0.2994334 1.0000000
s3512-s2224     1.659533e-02 -0.28125736 0.3144480 1.0000000
s384-s2224      5.362333e-03 -0.29249036 0.3032150 1.0000000
ske128-s2224    2.306921e-01 -0.06716058 0.5285448 0.3187011
ske256-s2224    2.312139e-01 -0.06663881 0.5290666 0.3151546
s256-s2512      8.564333e-03 -0.28928836 0.3064170 1.0000000
s3224-s2512     1.022100e-02 -0.28763169 0.3080737 1.0000000
s3256-s2512     2.677333e-03 -0.29517536 0.3005300 1.0000000
s3384-s2512     1.440900e-02 -0.28344369 0.3122617 1.0000000
s3512-s2512     2.942367e-02 -0.26842903 0.3272764 1.0000000
s384-s2512      1.819067e-02 -0.27966203 0.3160434 1.0000000
ske128-s2512    2.435204e-01 -0.05433225 0.5413731 0.2382714
ske256-s2512    2.440422e-01 -0.05381047 0.5418949 0.2353088
s3224-s256      1.656667e-03 -0.29619603 0.2995094 1.0000000
s3256-s256     -5.887000e-03 -0.30373969 0.2919657 1.0000000
s3384-s256      5.844667e-03 -0.29200803 0.3036974 1.0000000
s3512-s256      2.085933e-02 -0.27699336 0.3187120 1.0000000
s384-s256       9.626333e-03 -0.28822636 0.3074790 1.0000000
ske128-s256     2.349561e-01 -0.06289658 0.5328088 0.2903827
ske256-s256     2.354779e-01 -0.06237481 0.5333306 0.2870231
s3256-s3224    -7.543667e-03 -0.30539636 0.2903090 1.0000000
s3384-s3224     4.188000e-03 -0.29366469 0.3020407 1.0000000
s3512-s3224     1.920267e-02 -0.27865003 0.3170554 1.0000000
s384-s3224      7.969667e-03 -0.28988303 0.3058224 1.0000000
ske128-s3224    2.332994e-01 -0.06455325 0.5311521 0.3012039
```

```
ske256-s3224     2.338212e-01  -0.06403147  0.5316739  0.2977705
s3384-s3256      1.173167e-02  -0.28612103  0.3095844  1.0000000
s3512-s3256      2.674633e-02  -0.27110636  0.3245990  1.0000000
s384-s3256       1.551333e-02  -0.28233936  0.3133660  1.0000000
ske128-s3256     2.408431e-01  -0.05700958  0.5386958  0.2538600
ske256-s3256     2.413649e-01  -0.05648781  0.5392176  0.2507713
s3512-s3384      1.501467e-02  -0.28283803  0.3128674  1.0000000
s384-s3384       3.781667e-03  -0.29407103  0.3016344  1.0000000
ske128-s3384     2.291114e-01  -0.06874125  0.5269641  0.3295789
ske256-s3384     2.296332e-01  -0.06821947  0.5274859  0.3259660
s384-s3512      -1.123300e-02  -0.30908569  0.2866197  1.0000000
ske128-s3512     2.140968e-01  -0.08375592  0.5119495  0.4417969
ske256-s3512     2.146186e-01  -0.08323414  0.5124712  0.4376686
ske128-s384      2.253298e-01  -0.07252292  0.5231825  0.3563950
ske256-s384      2.258516e-01  -0.07200114  0.5237042  0.3526307
ske256-ske128    5.217778e-04  -0.29733092  0.2983745  1.0000000
```

Appendix D

Bonferroni Multiple Comparisons

```

        Pairwise comparisons using t tests with pooled SD

data:  dfB$Entropy and dfB$Hash

       bl2b bl2s md5  s1   s2224 s2512 s256 s3224 s3256 s3384 s3512 s384 ske128
bl2s   1.00 -    -    -    -     -     -    -     -     -     -     -    -
md5    1.00 1.00 -    -    -     -     -    -     -     -     -     -    -
s1     1.00 1.00 1.00 -    -     -     -    -     -     -     -     -    -
s2224  1.00 1.00 1.00 1.00 -     -     -    -     -     -     -     -    -
s2512  1.00 1.00 1.00 1.00 1.00  -     -    -     -     -     -     -    -
s256   1.00 1.00 1.00 1.00 1.00  1.00  -    -     -     -     -     -    -
s3224  1.00 1.00 1.00 1.00 1.00  1.00  1.00 -     -     -     -     -    -
s3256  1.00 1.00 1.00 1.00 1.00  1.00  1.00 1.00  -     -     -     -    -
s3384  1.00 1.00 1.00 1.00 1.00  1.00  1.00 1.00  1.00  -     -     -    -
s3512  1.00 1.00 1.00 1.00 1.00  1.00  1.00 1.00  1.00  1.00  -     -    -
s384   1.00 1.00 1.00 1.00 1.00  1.00  1.00 1.00  1.00  1.00  1.00  -    -
ske128 0.59 0.89 0.94 0.85 0.83  0.54  0.72 0.76  0.59  0.87  1.00  0.98 -
ske256 0.58 0.87 0.93 0.83 0.81  0.53  0.71 0.75  0.58  0.85  1.00  0.96 1.00
```

```
        Pairwise comparisons using t tests with pooled SD

data:  dfB$Serial.Correlation and dfB$Hash

       bl2b    bl2s    md5     s1      s2224   s2512   s256    s3224   s3256
bl2s   1.00000 -       -       -       -       -       -       -       -
md5    1.00000 1.00000 -       -       -       -       -       -       -
s1     1.00000 1.00000 1.00000 -       -       -       -       -       -
s2224  1.00000 1.00000 1.00000 1.00000 -       -       -       -       -
s2512  1.00000 1.00000 1.00000 1.00000 1.00000 -       -       -       -
```

```
s256    1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 –       –       –
s3224   1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 –       –
s3256   0.21111 1.00000 1.00000 0.55053 0.64476 1.00000 0.00522 0.78495 –
s3384   1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 0.01658
s3512   1.00000 0.08270 0.45849 1.00000 1.00000 0.08641 1.00000 1.00000 0.00012
s384    1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 0.80441
ske128  1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 0.27324
ske256  1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 0.59652
        s3384   s3512   s384    ske128
bl2s    –       –       –       –
md5     –       –       –       –
s1      –       –       –       –
s2224   –       –       –       –
s2512   –       –       –       –
s256    –       –       –       –
s3224   –       –       –       –
s3256   –       –       –       –
s3384   –       –       –       –
s3512   1.00000 –       –       –
s384    1.00000 1.00000 –       –
ske128  1.00000 1.00000 1.00000 –
ske256  1.00000 1.00000 1.00000 1.00000
```

REFERENCES

[1] Anderson, R. The classification of hash functions, 1993.

[2] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. The Keccak reference, January 2011. http://keccak.noekeon.org/.

[3] Bertoni, G., Daemen, J., Peeters, M., Assche, G. V., and Keer, R. V. Keccak implementation overview, May 2012. http://keccak.noekeon.org/.

[4] Blum, L., Blum, M., and Shub, M. A simple unpredictable pseudo-random number generator. SIAM Journal on Computing 15, 2 (1986 submitted 1982), 364–383.

[5] Blum, M. Independent unbiased coin flips from a correlated biased source: A finite state markov chain. In 25th Annual Symposium onFoundations of Computer Science, 1984. (Oct 1984), pp. 425–433.

[6] Blum, M., and Micali, S. How to generate cryptographically strong sequences of pseudo-random bits. SIAM Journal on Computing 13, 4 (1984), 850–864.

[7] Carter, J. L., and Wegman, M. N. Universal classes of hash functions.

[8] Chor, B., and Goldreich, O. Unbiased bits from sources of weak randomness and probabilistic communication complexity. In 26th Annual Symposium on Foundations of Computer Science (sfcs 1985) (Oct 1985), pp. 429–442.

[9] Coskun, B., and Memon, N. Confusion/diffusion capabilities of some robust hash functions. In 2006 40th Annual Conference on Information Sciences and Systems (March 2006), pp. 1188–1193.

[10] Damgaard, I. A design principle for hash functions. In CRYPTO (1989), G. Brassard, Ed., vol. 435 of LNCS, Springer, pp. 416–427.

[11] Dinno, A. dunn.test: Dunn's Test of Multiple Comparisons Using Rank Sums, 2017. R package version 1.3.4.

[12] Goldreich, O. 1 introduction and preliminaries. Foundations and Trends in Theoretical Computer Science 1, 1 (April 2005).

[13] Goldreich, O., Krawczyk, H., and Luby, M. On the existence of pseudorandom generators. In [Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science (Oct 1988), pp. 12–24.

[14] H, J., Impagliazzo, R., Levin, L. A., and Luby, M. A pseudorandom generator from any one-way function. SIAM J. Comput. 28, 4 (Mar. 1999), 1364–1396.

[15] Handschuh, H. SHA Family (Secure Hash Algorithm). Springer US, Boston, MA, 2005, pp. 565–567.

[16] Hayashi, M., and Tsurumaru, T. More efficient privacy amplification with less random seeds via dual universal hash function. IEEE Transactions on Information Theory 62, 4 (April 2016), 2213–2232.

[17] Hernandez-Castro, J., and Barrero, D. F. Evolutionary generation and degeneration of randomness to assess the indepedence of the ent test battery. In 2017 IEEE Congress on Evolutionary Computation (CEC) (June 2017), pp. 1420–1427.

[18] Herrewege, A. V., and Verbauwhede, I. Software only, extremely compact, keccak-based secure prng on arm cortex-m. In 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC) (June 2014), pp. 1–6.

[19] Impagliazzo, R., Levin, L. A., and Luby, M. Pseudo-random generation from one-way functions. In Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing (New York, NY, USA, 1989), STOC '89, ACM, pp. 12–24.

[20] Jain, R. A comparison of hashing schemes for address lookup in computer networks. IEEE Transactions on Communications 40, 10 (Oct 1992), 1570–1573.

[21] Knuth, D. E. The art of computer programming., 2d ed.. ed. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co, Reading, Mass., 1973.

[22] Lee, C.-J., Lu, C.-J., Tsai, S.-C., and Tzeng, W.-G. Extracting randomness from multiple independent sources. IEEE Transactions on Information Theory 51, 6 (June 2005), 2224–2227.

[23] Levin, L. A. One-way functions and pseudorandom generators. In Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (New York, NY, USA, 1985), STOC '85, ACM, pp. 363–365.

[24] Li, M., and Vitanyi, P. M. B. Two decades of applied kolmogorov complexity: in memoriam andrei nikolaevich kolmogorov 1903-87. In [1988] Proceedings. Structure in Complexity Theory Third Annual Conference (Jun 1988), pp. 80–101.

[25] Łoza, S., and Matuszewski, L. A true random number generator using ring oscillators and sha-256 as post-processing. In 2014 International Conference on Signals and Electronic Systems (ICSES) (Sept 2014), pp. 1–4.

[26] Motara, Y. M., and Irwin, B. Sha-1 and the strict avalanche criterion. In 2016 Information Security for South Africa (ISSA) (Aug 2016), pp. 35–40.

[27] NIST. Sha-3 competition (2007-2012), 2005.

[28] Oded Goldreich, S. G., and Micali, S. How to construct random functions.

[29] Rogaway, P., and Shrimpton, T. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 371–388.

[30] Shamir, A. On the generation of cryptographically strong pseudorandom sequences. ACM Transactions on Computer Systems (TOCS) 1, 1 (1983), 38–44.

[31] Shannon, C. E. A mathematical theory of communication, 1948.

[32] Shannon, C. E. Communication theory of secrecy systems. Bell Labs Technical Journal 28, 4 (1949), 656–715.

[33] Sklavos, N., Kitsos, P., Papadomanolakis, K., and Koufopavlou, O. Random number generator architecture and vlsi implementation. In 2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353) (2002), vol. 4, pp. IV–854–IV–857 vol.4.

[34] Stinson, D. R. Some observations on the theory of cryptographic hash functions. Designs, Codes and Cryptography 38, 2 (Feb 2006), 259–277.

[35] Tomamichel, M., Schaffner, C., Smith, A., and Renner, R. Leftover hashing against quantum side information. IEEE Transactions on Information Theory 57, 8 (Aug 2011), 5524–5535.

[36] von Neumann, J. Various techniques used in connection with random digits. In National Bureau of Standards Applied Mathematics Series (1951), vol. 12, pp. 36–38.

[37] Walker, J. Hotbits: Genuine random numbers, generated by radioactive decay, 1996.

[38] Wang, B. J., Cao, H. J., Wang, Y. H., and Zhang, H. G. Random number generator of bp neural network based on sha-2 (512). In 2007 International Conference on Machine Learning and Cybernetics (Aug 2007), vol. 5, pp. 2708–2712.

[39] Wang, Y.-H., Zhang, H.-G., Shen, Z.-D., and Li, K.-S. Thermal noise random number generator based on sha-2 (512). In 2005 International Conference on Machine Learning and Cybernetics (Aug 2005), vol. 7, pp. 3970–3974 Vol. 7.

[40] Webster, A. F., and Tavares, S. E. On the Design of S-Boxes. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986, pp. 523–534.

[41] Yao, A. C. Theory and application of trapdoor functions. In Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on (1982), IEEE, pp. 80–91.

[42] Zhandry, M. How to construct quantum random functions. In 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science (Oct 2012), pp. 679–687.