Pseudorandom Number Generators Using Multiple Sources of Entropy

by

Gautam Srivastava

B.Sc., Briar Cliff University, 2004

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Pseudorandom Number Generators Using Multiple Sources of Entropy

Gautam Srivastava

Department of Computer Science

University of Victoria

Supervisory Committee

Dr. Venkatesh Srinivasan (Department of Computer Science)

Supervisor

Dr. Bruce Kapron (Department of Computer Science)

Supervisor

Dr. Valerie King (Department of Computer Science)

Departmental Member

Dr. Issa Traore (Department of Electrical and Computer Engineering)

External Examiner

Supervisory Committee

Dr. Venkatesh Srinivasan, Supervisor (Department of Computer Science)

Dr. Bruce Kapron, Supervisor (Department of Computer Science)

Dr. Valerie King, Departmental Member (Department of Computer Science)

Dr. Issa Traore, External Examiner (Department of Electrical and Computer Engineering)

Abstract

Randomness is an important part of computer science. A large group of work, both in theoretical and practical computer science, is dedicated to the study of whether true 'randomness' is necessary for a variety of applications and protocols to work. One of the main uses for randomness is in the generation of keys, used as a security measure for many cryptographic protocols.

The main measure of randomness is achieved by looking at entropy, a measure of the disorder of a system. Nature is able to provide us with many sources that are high in entropy. However, many cryptographic protocols need sources of randomness that are stronger (higher in entropy) than what is present naturally to ensure security. Therefore, a gap exists between what is available in Nature, and what is necessary for provable security.

This paper looks to bridge this gap. Research in pseudorandom number generation has gone on for decades. However, many of the past constructions were lacking in either documentation or provable security of their methods. The need for a pseudorandom number generator (PRG) with provable security and strong documentation is evident.

A new construction of a PRG is introduced. The new construction, labeled XRNG, looks to encompass recent research in the field of extractors along with previously known research in the field of pseudorandom number generation. Extractors, as the name suggests, looks to extract close to random information from high entropy sources.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my Supervisors, Dr. Venkatesh Srinivasan and Dr. Bruce Kapron for all their support and contnued motivation to pursue the avenues I did. They have been an ispiration from the first time I met them. I would also like to thank my mother, Dr. Rekha Srivastava, father, Dr. Hari Srivastava, and my sister, Sapna Srivastava for their continued support in all the ways they have done since I was little. I love you all dearly.

Thanks should also be given to my Theory Group at Uvic, as well as the CSC Department staff as without them I would be lost when it comes to Uvic regulations. Lastly, my Supervisory Committe members, Dr. Valerie King and Dr. Issa Traore for their insight into my topic and thesis itself.

1 Introduction

Randomness is an important facet of computer science. A large group of work, both in theoretical and practical computer science, is dedicated to the study of whether true 'randomness' is necessary for a variety of applications and protocols. It is a known fact that traditional cryptographic protocols were designed and assumed to be used with sources that output unbiased and independent random bits known as perfect random sources (truly random) [9]. In nature, more close to random (high entropy) sources occur than those that are truly random. One of the underlying questions that remain from this is if perfect randomness is an absolute necessity for provable security.

Randomness is essential for key generation, one of the main security components of many cryptographic protocols. If an inadequate source of randomness is used to generate such keys, the entire protocol's security could be jeopardized. From this fact, the generation of such random data becomes of the utmost importance and the design and implementation of proper pseudorandom number generation becomes a major priority. This is not a new avenue of research. The study of pseudorandom generators dates back to early work in the topic some 50 years ago by Von Neumann. The problem lies in the gap that exists between what seems to be necessary for cryptographic protocols (true randomness) and what is available in our environment on a consistent basis (high entropy sources). So, is there a method or protocol that can help bridge the gap between what is necessary and what is available? This thesis looks to answer this question.

In this paper, a new pseudorandom number generator construction is presented. Its main motivation comes from the work of Barak and Halevi in their paper discussing the use of single source extractor accompanied by a block cipher to create a robust pseudorandom generator [2]. This paper looks to further those ideas by examining the case where multiple sources of entropy are available. In such situations, the use of a simple multiple source extractor can be used to generate output that is close to random. Using this output as the seed file to a block cipher, it will be shown that the output of this system is both close to random and very high in entropy.

1.1 Motivation

Pseudorandomness has been a field whose study has led to many breakthroughs in the world of cryptography and computer security. Almost every operating system and programming language seem to have their own built-in pseudorandom number generator. This points towards the necessity and importance of good quality pseudorandom numbers and secure pseudorandom generators (PRG).

The first part of research behind this thesis looked into some of the existing pseudorandom generators in use today and trying to analyze such PRG's with statistical testing. Of the PRGs looked at, the main flaw that reappeared was the lack of both documentation and provable security available for these PRG's. This was disconcerting taking into account that these PRGs were meant to be used to generate keys and passwords to be used for systems with the utmost security needs. The need for a cryptographically secure pseudorandom generator whose security could be shown not only using statistical testing, but could also be theoretically shown.

Another inspiration are the newer results in the field of extractors (Chapter 2.2). A fairly new avenue of research, extractors allow entropy to be "extracted" from high min-entropy sources to give an output that is close to Uniform. Extractors have been heavily researched lately, however not many models or designs have incorporated them. The efficiency and uniform output of extractors made the incorporation of them in our model easy to justify. The problem arose in the dilemma of "'quality versus quantity"'. Although extractors give output that is very close to truly random (quality), the speed at which bits can be delivered often is much lower than what is needed for high-speed key generation and other practical applications (Figure 1).

To compensate for this, we need to take these short, close to uniform output of the extractor and expand it into usable output is clearly evident. This procedure, of taking a close to random input and being able to "'blow up"' its entropy into a long enough stream to be usable in practical applications takes us into the field of Pseudorandom Number Generation (Figure 2).

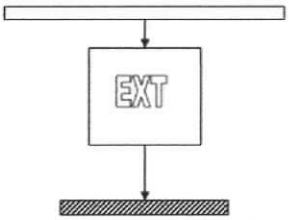

Figure 1: Concept of Extractor

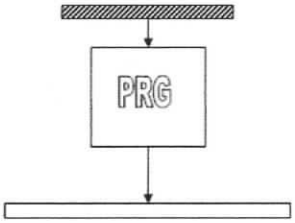

Figure 2: Concept of Pseudorandom Number Generator

From here, the main stipulation we place on such PRGs is that their output is not distinguishable from a truly random seed by any observer whose computation time can be bounded. This entire concept, of taking sources from the environment, "'compressing"' the entropy available from them into a shorter string, then taking this close to random string and expanding it into a usable output, is the main notion that motivates this thesis.

1.2 Relation to Previous Work and Models

The main influence and motivation for this new model comes from the work of Ferguson and Schneier's Fortuna algorithm [9] as well as Barak and Halevi's PRNG model [2]. The Fortuna algorithm, and to some lengths /dev/random as well, were the first to advocate the separation of the entropy collection and the random number generation phases of a good PRNG. Fortuna looks to collect entropy for entropy sources and store such information in pools, using these pools as the seed file to the PRNG when it is needed. To apply some form of extraction between the entropy pools, and the 256-bit seed file, a hash function such as SHA-256 can be used as an extraction

function to hash a large string down to one of the required 256-bit length.

The more recent work of Barak and Halevi [2] gives a strong and sound definition of the security necessary for a PRNG as well as a good construction. This thesis, similar to the Fortuna algorithm, suggests the separation of the entropy collection process from the generation of pseudorandom number generation for which the entropy will be needed. The entropy collection process is done using a simple single (entropy) source extraction function whose output is used to seed a cryptographically based pseudorandom generator *G*. The state of the entire PRNG is based on the seed file to *G*. The design uses an entropy extractor that uses system events (similar to Fortuna and /dev/random) as input to extract random bits from. The output of the extractor is then used to update the current state of *G*. From there, G, which can be a block cipher or any cryptographically secure encryption function, is used to create the pseudo random output of the entire system.

1.3 Brief History of Extractors

Before introducing our model, a quick look into the research on the topic of extractors that has occurred until now is pertinent. The main parameters that are measured when looking at an extractor are **min-entropy threshold**, **seed length**, and **output length**. The **min-entropy threshold** helps describe the minimum amount of uncertainty that is present in a source. This helps regulate the amount of uncertainty that is required by a particular construction to ensure that its output is close to truly random. Moreover, the **seed length** is the necessary input to the extractor that must be provided to extract the **output length** in uniform bits. These three parameters need to be optimized to obtain a construction that is both feasible in efficiency (runs in polynomial time) and quality (output is close to random).

1.3.1 The Initial Construction

The first extractor construction was put forward by Impagliazzo, Levin and Luby [13]. There extractor was based on the work of Carter-Wegman in the field of universal hash functions. Hash functions, which will be described later on, are functions that take an arbitrarily long input string and from it produce a fixed length output (usually shorter). While this extractor gives an output that is close to random, its seed length is nowhere close to optimal. The only reason such an extractor makes any sense to use in a practical application is the fact that the output given is very close to truly random.

Improvements were slowly made on this initial construction using hash functions. In [13], it was shown that universal hash functions can be replaced with pairwise independent hash functions which in turn end up shortening the requirements on seed length. Most initial research in the field was geared towards constructions that shortened the seed length.

1.3.2 Recent Work

Of the major events in the field of extractors, none had more impact than the work of Trevisan [25]. He was able to take research in the field of pseudorandom number generators and apply them to the field of extractors. He noticed that the hard functions used to produce pseudorandom generators can be used to produce extractors. Hard functions are those that are easy to compute but hard to invert. They rely solely on the notion that one-way functions exist. As previously mentioned, a pseudorandom generator is a procedure that elongates a short seed of truly random bits into a longer output of "'pseudorandom"' bits.

Trevisan observed that certain pseudorandom constructions use hard functions in a "'black box"' sense. Thus, if we can think of these hard functions as an additional input to the generator, an extractor can be created from this.

Example

Let $n = 2^l$, construct a function

$$E : \{0,1\}^n \times \{0,1\}^d \rightarrow \{0,1\}^m$$

where d can be defined as a short, truly random external seed to the construction.

such that whenever x is the truth table of a hard function $f : \{0,1\}^l \rightarrow \{0,1\}$, the distribution $E(x, \bullet)$ cannot be distinguished from uniform by circuits of size m. Functions E with this property can be considered pseudo-random generator schemes.

Example

Work has continued in the field of extractors. One of the simplest constructions of an extractor was put forward by Von Neumann. Called the Von Neumann corrector, it is in all actuality an extractor. Two consecutive bits are examined. If the bits examined are the same, then both bits are ignored. If the bits examined are different, of the two bits examined the first bit is used as output. Its main purpose is to remove any bias that may exist within a source. For a given source, a Von Neumann corrector may have the following output.

Table 1: Von Neumann Corrector

Entropy Source	10	11	00	10	10	01
XOR Output	1			1	1	0

One of the main advantages of the Von Neumann corrector is its ability to detect situation of long periods of zeros or ones. Using a corrector, there are obvious losses that occur from the number of bit inputted by an entropy source and the number of bits outputted. A Von Neumann corrector deletes approximately $\frac{3}{4}$ of the original sequence of bits.

In literature, Extractors are often referred to as TRNG's, which stands for True Random Number Generators. True random number generators (TRNG) differ in one major way than other generators, they require a source of entropy. The source of entropy is usually an external physical process that has a measurable value that can be quantified within a given range. For example,

looking at the previous discussed psychical process of a Geiger counter, the amount of radiation detected by a Geiger counter at any given time in a normal setting can be considered random. Such a phenomenon can be used as the external source of entropy for a TRNG.

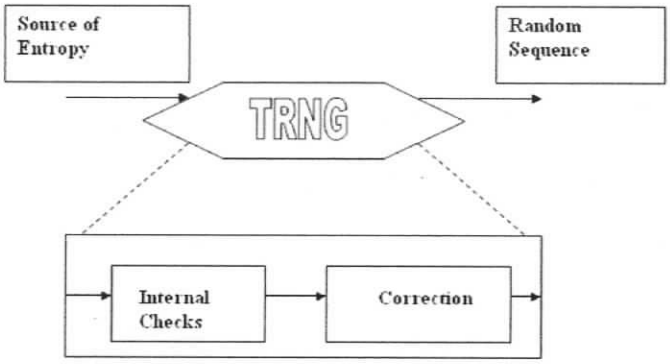

Figure 3: Design of TRNG

1.4 Brief History of Pseudorandom Generators

Random numbers are an essential part of cryptography. All of the cryptographic protocols in commercial use today rely on the generation of random numbers as part of the protocol's security. The use of random numbers in generating passwords and encrypting data is critical to the security of information being transmitted on a daily basis.

The problem arises when it becomes necessary to generate random numbers. It is not as easy a task as it may seem. The generation of random numbers is a research topic that has been looked at for a very long time. It may seem odd that in most cases, to generate random numbers, a computer, that is the most precise and deterministic of tools, is used. For all intents and purposes, numbers generated by a computer that appear random can be called pseudorandom numbers, or having the appearance of randomness. The term "random", is most of the time reserved for outputs of intrinsically random physical processes, such as the elapsed time between clicks of a Geiger counter when placed near a sample of a radioactive element [16].

Until recently, a computer's CPU had no direct way to generate random numbers. The introduction of Intel's Pentium III introduced for the first time a hardware random number generator that uses thermal noise to, in the words of Intel, "generate high-quality random and non-deterministic numbers." Prior to the Pentium III, add-on boards or external inputs were needed. The problem lies in the deterministic nature of a computer program. Given a certain input, a program will always follow the same path to completion. That makes it impossible to generate random numbers using an algorithm. For all that is known, it may be the case that it is impossible to generate random numbers altogether. While the ticks on a Geiger counter may seem random, more in depth study of the science behind the phenomenon might prove one day not to be random at all.

1.4.1 Timeline

Since the late 1940's, randomness and its offsets have been constantly researched and debated. One of these offsets lies in the study of pseudorandom generators. Due to the lack of known sources of randomness that can be deemed truly random (close to the Uniform distribution), a method or procedure becomes necessary to create such output. But why is true randomness such a necessity? Why not rely on sources that have some randomness that make them unpredictable? This is a tough question, and the heart of the answer lies in how you look at the question. First, it is a known fact that many of the cryptographic applications and algorithms that have been designed over the years that need randomness to operate have always been under the assumption that a truly random source is present at their disposal. A recent paper [8] looked at how these algorithms handle being given a source of imperfect randomness, with less than positive results.

Secondly, we must take our focus away from the user of such a system and focus our attention on someone that is trying to compromise our system (adversary). The whole notion of pseudorandom generation relies on the fact that an adversary, running in an efficient amount of time (say polynomial time), will not be able to distinguish a pseudorandom source from a truly random one.

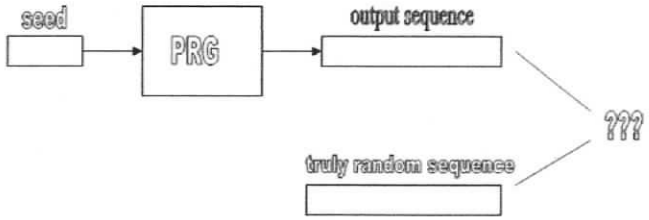

Figure 4: Goals of a PRNG

Take for example, two streams of binary data on a TV screen in front of our adversary. One of these streams, is truly random (from where we got it, I can not tell). The other, is pseudorandom and was generated by an algorithm we are currently working on. Our adversary, when looking at, analyzing, running tests on these two streams of data, will not be able to distinguish (tell which one is truly random and which one is pseudo random) between the two streams within an efficient amount of time (Figure 4).

This notion may not be present in just any weak source that occurs in nature, so we look to pseudorandom generation with some form of provable security to give us as users of the system the peace of mind to use our system with ease.

So the question remains, Why can we not just rely on sources that have some randomness that make them unpredictable? Truly random sources, that produce say bit strings of length n, have the property the each string will be outputted with probability $\frac{1}{2^n}$. The problem arises in weak random sources because they do not draw from a the entire space occupied by the 2^n strings. Let us say a weak random source outputs from a subset size k with probability $\frac{1}{2^k}$ and each string outside the subset with probability 0. By doing this, clearly, an adversary looking at such a source can after time start ruling out certain strings that have never occurred. This is an advantage we are not comfortable giving to an adversary. Take for example an algorithm known as a one-time pad. The one-time pad (OTP) is an encryption algorithm where the plaintext combined with a random key or "pad" that is as long as the plaintext and used only once. The condition is placed on the key to be

Figure 5: Design of DRNG

truly random. If the key used was not truly random and say a weak source was used, the advantage given to an efficient adversary by not using a truly random key may be enough to compromise a system making use of the one-time pad.

In literature, pseudorandom number generators of the type discussed are referred to as DRNG's, which stands for Deterministic Random Number Generators. One of the most important qualities of a DRNG is that the sequence of numbers generated is predetermined from an equation. DRNG's are given an input value, known as a seed, to start where the sequence of pseudorandom numbers is generated from. Different sequences can be generated using different seeds. However complex the math and algorithms are behind the output of a DRNG, given the same seed and internal states, a DRNG will produce the same output. In general, DRNGs are software based. Being software based, all a DRNG needs is a seed value to start producing a pseudorandom sequence.

1.4.2 Older Constructions

One of the oldest and most popular methods for a very long time for generating random numbers was the **linear congruential generator**. In the early years of pseudorandom generators, not much was known about methods to substantiate the effectiveness of these generators. Simple statistical tests, such as the Runs test, which in a nutshell counts the numbers of zeros and ones in a sequence and looks for bias, would be used to test the validity of such generators. These early year generators were able to pass the tests with flying colours, and at the time their effectiveness as PRGs was based on this. However, as time progressed, it was quickly understood that such tests are easily fooled and not as accurate as some of them let on. So, as more intricate tests became available (Maurer's

Test, DieHard Test Suite), linear congruential generators became a thing of the past.

Example

Let us look at a simple Linear congruential generator. LCGs are defined by the recurrence relation:

$$V_{j+1} = (A \times V_j + B)\, mod M \tag{1.1}$$

Where V_n is the sequence of random values and A, B and M are generator-specific integer constants. *mod* is the modulo operation.

The period of a general LCG is at most M, and in most cases less than that. Take for example the Mersenne twister, which both runs faster than and generates higher-quality deviates than almost any LCG, only LCGs with M equal to a power of 2, most often $M = 2^{32}$ or $M = 2^{64}$, make sense at all. These are the fastest-evaluated of all random number generators; a common Mersenne twister implementation uses it to generate seed data. Numerical Recipes in C advocates a generator of this form with:

$A = 1664525$, $B = 1013904223$, $M = 2^{32}$

1.4.3 Factors Determining Quality of a RNG

Different cryptographic needs can determine which type of RNG will best fit a certain intended application. Most RNGs in cryptosystems have the same general qualities that allow them to be implemented. A RNG must be secure enough to be considered cryptographically strong. In other words, a RNG must be able to produce its random numbers under the scrutiny of an adversary without giving away any information as to how the sequences generated are produced. However, the true challenge lies in the ability to quantify the true "quality" of an RNG.

One of the most important qualities of a random number generator is its ability to give equiprobable random numbers. In a similar manner that flipping a coin should give heads and tails with equal probability, the output of a random number generator should produce a random number within its range that is equally probable. In other words, there should not be a bias towards certain values within the range of values an RNG is drawing from. Take for example an adversary that views a PRNG for a long period of time. After some time, this adversary notices that some values appear with less frequency and other values with much more. This type of advantage that an adversary can attain should be avoided in PRNGs. Statistical testing can be used to determine whether an RNG outputs values within a reasonably small deviation from truly random.

Another important quality for a random number generator to have is that of indeterministic values. In other words, previous numbers generated by the RNG should not influence the numbers that are to be generated in the future. If given a long list of previous values outputted by a RNG in a given crypto-system, future outputs of the RNG should not be able to be guessed using the list. If the RNG was used for encryption, all future key values could easily be determined using information that the RNG had already outputted and the entire crypto-system would be compromised.

1.5 Hybrid Random Number Generator

Hybrid Random Number Generators, as the name suggests, are combinations of a TRNG and a DRNG. There are two types of Hybrid RNGs. They can either be a TRNG and a DRNG XORed together, or a DRNG with the seed value needed generated by a TRNG.

A hybrid RNG created from the combination of a TRNG and a DRNG usually mixes the two outputs using an XOR, however other mixing functions can be used appropriately. As is the case with XOR corrector used in a TRNG, the XOR decreases the output by half. However, since there are two separate sources of information, the output length would equal the input length of the sequences to either the TRNG or DRNG.

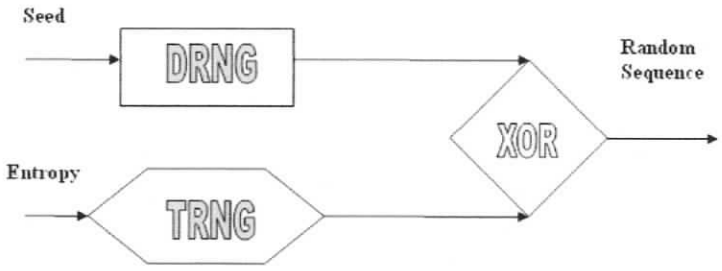

Figure 6: Hybrid RNG

One of the main advantages of this configuration is its fault tolerance. If either the DRNG or the TRNG fail for any reason, the other generator can continue to produce output until the problem is fixed. For example, if the entropy source for the TRNG becomes unreachable or is affected in such a manner such that the TRNG produces no output, the DRNG could still provide pseudorandom output.

Another advantage of the hybrid configuration is its insusceptibility to cryptographic attacks. An adversary would have to make a two-tier attack, both physical and software based. An adversary would have to be able to determine what the DRNG would provide using software and be able to prevent the TRNG from properly functioning.

Another type of hybrid RNG uses a TRNG to seed a DRNG, which we will call the XRNG. This construction is at the base of our model, which will be touched on in later Chapters.

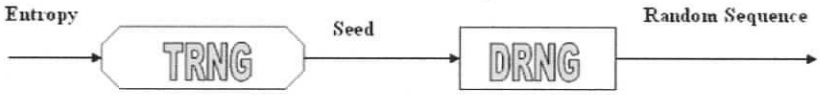

Figure 7: XRNG

The TRNG uses its own entropy source (Extractor) to provide the seed value used in the DRNG (Generator). As long as the DRNG is cryptographically secure, since the seed value to the DRNG is random, the output of this configuration will be random as well. One assumption that is made in

this configuration is that the TRNG will be used far less than the DRNG.

1.6 Outline

The remainder of this paper will be divided into chapters reviewing some background info, describing our model, and presenting our test results. **Chapter 2** will provide some background information necessary to understand our model. **Chapter 3** will introduce the Fortuna Heuristic as well as /dev/random. The new construction and model will be shown in **Chapter 4**, and simulation and test results will be introduced in **Chapter 5**.

Chapter 2 - Background

Chapter 2 introduces all the background information necessary to understand the model of Chapter 4 as well as some of the previous constructions in the field (Chapter 3). The notion of Entropy, and brief definitions and descriptions of Extractors, Block Ciphers, and Hash Functions are provided.

Chapter 3 - Fortuna Heuristic and /dev/random

Chapter 3 looks into two previous pseudorandom generators that use entropy collection as the basis of their randomness. Their similarity to our model and design are explained in this Chapter to give insight into our model. In the final part of the chapter, some initial research that was done into the quality of a PRNG, and methods for analyzing PRNGs are introduced.

Chapter 4 - Our Model

Chapter 4 introduces our model. The architecture as a whole, as well as the intricacies of each phase are given. A formal proof is provided showing the properties of the PRG.

Chapter 5 - Simulation and Test Results

Chapter 5 provides the heuristics behind the simulation and the results of the testing that was done on the input and output of our model. Min-entropy and the statistical distance from the Uniform Distribution are highlighted.

2 Background

This Chapter looks to introduce some of the background information necessary to understand our construction in a detailed manner. The two main components of the PRNG are the extractor and the Generator (using a block cipher), both of which will be introduced here. Furthermore, concepts behind the notion of entropy will be briefly touched on here as well.

2.1 Entropy

The key aspect to any usable cryptosystem is its security. One of the most influential people on this topic was Claude Shannon. In 1949, Shannon published a paper called "Communication Theory of Secrecy Systems", which is still considered a major contribution to the scientific study of cryptography [24]. There are various approaches to the study of a cryptosystem's security. This Chapter will focus on some of these basic approaches and will move into some of Shannon's major ideas.

2.1.1 Computational Security

Computational security deals with the notion of how much effort is required to break a cryptosystem. Let us define a cryptosystem to be computationally secure if the best algorithm for breaking it requires at least N operations, where N is a very large number. The problem arises in the fact that no known cryptosystem can be proven to be secure under this definition. In practice, cryptosystems are shown to be secure under certain types of attacks. For example, a certain cryptosystem may be shown to be secure against an exhaustive key search. In these cases, showing that a certain cryptosystem is secure against a specific type of attack says nothing about its security against other types of attacks. It is very specific to the attack.

2.1.2 Provable Security

Another approach to security is to reduce the security of a cryptosystem to primitives of such problems that are known to be difficult. For example, the security of a cryptosystem may be said to be secure if a given integer n cannot be factored (implying factoring is HARD). Cryptosystems deemed secure in this manner are called provably secure. This type of security is relative, as the security of the cryptosystem is proved relative to another problem and is not an absolute proof of security. This situation is similar to showing that a problem is NP-complete. It gives a relative estimation as to how hard a given problem is compared to other problems, but it does not show absolutely the computational difficulty of the problem.

2.1.3 Unconditional Security

This type of security for a given cryptosystem is based on the fact that the adversary has no bound on the amount of computations it is allowed to do. A cryptosystem is defined as unconditionally secure if it cannot be broken even with infinite computational resources.

2.1.4 Elementary Probability Theory

Since the computation time is infinite, the unconditional security of a cryptosystem cannot be studied from the point of view of computational complexity. An appropriate means of looking at the unconditional security of a cryptosystem is using probability theory. Only some of the main ideas of probability theory are needed to understand the concepts to be discussed later.

Definition 2.1 *A discrete random variable, say* **X**, *consists of a finite set* **X** *and a probability distribution* **D** *on* **X**. *The probability that the random variable* **X** *takes on the value x is denoted Pr[X=x] or Pr[x] if the random variable X is fixed. It must be satisfied that* $0 \leq Pr[x]$ *and for all* $x \in X$, *and*

$$\sum_{x \in X} Pr[x] = 1. \tag{2.1}$$

For example, consider the outcomes of tossing a coin to be a random variable. The set of outcomes can be defined as $\{heads, tails\}$. The probability distribution then for the set would be $Pr[heads] = Pr[tails] = 1/2$.

Suppose there is a random variable **X** defined on **X**, and $E \subseteq X$. The probability **X** takes on a value in the subset E is shown to be

$$Pr[x \in E] = \sum_{x \in E} Pr[x] \tag{2.2}$$

The subset E is called an event.

2.1.5 Perfect Secrecy

An in depth look at perfect secrecy needs a cryptosystem to be properly defined. Define a cryptosystem as a five-tuple (P,C,K,E,D) where the following conditions are satisfied:

1. P is a finite set of possible plaintexts
2. C is a finite set of possible ciphertexts
3. K, the keyspace, is a finite set of possible keys
4. For each $k \in K$, there is an encryption rule $e_k \in E$ and a corresponding decryption rule $d_k \in D$. Each $e_k : P \rightarrow C$ are functions such that $d_k(e_k(x)) = x$ for every $x \in P$.

In addition to this cryptosystem, for the case of perfect secrecy let us add the stipulation that a particular key $k \in K$ is used for only one encryption.

Let us suppose that the plaintext space P has a probability distribution. Therefore, the plaintext element defines a random variable X. Denote a priori probability that plaintext x occurs by Pr[X=x]. Also assume that the key k be chosen using some fixed probability distribution. This is not a common occurrence. Usually, the key is chosen at random to ensure all possible keys are equiprobable. The key therefore, will also define a random variable denoted k. Thus, the probability that key k is chosen from the keyspace is given by Pr[K=k]. Since the key is chosen before the message to be encrypted (plaintext) is known, it can be a safe assumption that the key and the plaintext are independent random variables.

The two given probability distributions for the plaintexts and keyspace induce a probability distribution on C. Thus, ciphertext elements can also be considered random variables which can be denoted as Y. The probability, Pr[Y=y], which represents the probability that y is the ciphertext transmitted, can be computed. For a $k \in K$, define

$$C(k) = e_k(x) : x \in P. \tag{2.3}$$

In other words, if k is the key, C(k) represents the set of all possible ciphertexts. Then, for every $y \in C$,

$$Pr[Y = y] = \sum_{k:y \in C(k)} Pr[K = k] Pr[X = d_k(y)] \tag{2.4}$$

For any $y \in C$ and $x \in P$, the conditional probability that given some plaintext x, that y is the ciphertext, $Pr\,[Y = y|X = x]$ can be computed.

$$Pr[Y = y|X = x] = \sum_{k:x \in d_k(y)} Pr[K = k] \tag{2.5}$$

Using Bayes' theorem, an equation for the conditional probability that given y is the ciphertext, that x is the plaintext can now be computed

$$Pr[X = x|Y = y] = \frac{Pr[X = x] \times \sum_{k:x \in d_k(y)} Pr[K = k]}{\sum_{k:y \in C(k)} Pr[K = k] Pr[x = d_k(y)]} \tag{2.6}$$

Definition 2.2 Perfect Secrecy. *A cryptosystem has perfect secrecy if* $Pr[x|y] = Pr[x]$ *for all* $x \in P$, $y \in C$. *In words, given that the observed ciphertext is y the probability that the plaintext is x is the same as that the probability of the plaintext is x with no information about the ciphertext.*

2.1.6 Entropy of a Cryptosystem

Perfect secrecy dealt with the main stipulation that each key is used for only one encryption. This is not always the case. As time progresses within a cryptosystem, it is not a secure quality that once a key is used it cannot be used again. If this was a reality, it would give an added advantage to the adversary as he will be able to rule out keys for further use as they are used.

Let us now consider the more general case where keys can be re-used to encrypt more and more plaintext. Given sufficient time, what kind of probabilities can a cryptanalyst be looking at to break such schemes using a ciphertext-only attack?

The main tool for this type of discussion dates back to 1948 where Claude Shannon introduced his concept of Entropy. Entropy can be defined as a mathematical measure of the amount of uncertainty that is present in a sample using the concepts of probability distribution.

The ideas behind entropy relate uncertainty to probability distributions. Consider a discrete random variable X which takes on values from the finite set X. As an example, take the set of coin tosses looked at earlier where the finite set is $(heads, tails)$. The probability distribution of the set, would be Pr[heads]=Pr[tails]=1/2. The coin tosses, if they were encoded using bits of information, could easily be encoded using 1 bit of information. A coin toss of heads could be encoded as a 1, and a coin toss of tails could be encoded as a 0. Therefore, the amount of information, or entropy of a coin toss is one bit. Furthermore, a string of *n* successive independent coin tosses would contain *n* bits of entropy, since the coin tosses could be encoded into a bit string of length *n*.

To further our understanding of entropy, consider a random variable X that takes on one of three possible values x_1, x_2, x_3 with the following probabilities:

$$\Pr[X=x_1]=1/2 \quad \Pr[X=x_2]=1/4 \quad \Pr[X=x_3]=1/4$$

The most efficient "encoding" is to have x_1 encode to 0, x_2 to 10, and x_3 as 11. Then, if looking at the encoding of X, the average number of bits in that encoding is

$$Pr[X = x_1] \times (1bit) + Pr[X = x_2] \times (2bits) + Pr[X = x_3] \times (2bits)$$

$$\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{4} \times 2 = \frac{3}{2}$$

This example suggests that an event that occurs with probability 2^{-n} could be easily coded with a bit string of length *n*. In a more general fashion, an outcome that occurs with probability *p* might be encoded by a bit string of length approximately $-log_2 p$. Taking a probability distribution on random variable X given notation $p_1, p_2, \ldots, p_n$, the measure of entropy, given notation H(X) is said to be the weighted average of the quantities $-log_2 p_i$.

Definition 2.3 Shannon Entropy. *Suppose X is a discrete random variable which takes on values from a finite set X. Then, the entropy of the random variable X is defined by the quantity*

$$H(X) = -\sum_{x \in X} Pr[x] log_2 Pr[x] \tag{2.7}$$

One thing to note in this definition is the fact that if Pr[x]=0, then $log_2 Pr\,[x]$ is undefined. From this, entropy can be redefined as the sum over all non-zero probabilities. Since the limit

$$lim_{y \to 0} x log_2 x = 0 \tag{2.8}$$

it is not a necessity that all $Pr\,[x] > 0$ for all x's. Another fact of note is that the logarithmic base of two is an arbitrary decision. Changing the base to any other base will change the entropy by a constant factor.

2.1.7 Types of Entropy Measurements

Entropy, as earlier stated, is a measure of information and randomness. There are three measures of entropy that are in widespread use: **Shannon entropy**, **Renyi entropy**, and **min entropy**. **Renyi entropy** is the least popular of the three but is still often seen in literature nonetheless [11].

The **Shannon entropy** H(X), named after its creator Claude Shannon, is the basic measure of entropy. Let S be a finite set with a probability distribution D on the set S. Then, the entropy of the distribution D is defined as

$$H(D) = -\sum_{x \in S} D(x) log_2 D(x) \tag{2.9}$$

The **Renyi entropy**, $H_{Ren}(D)$ of a distribution D is defined as

$$H_{Ren}(D) = -log_2 \sum_{x \in S} (D(x))^2 \tag{2.10}$$

The idea of Renyi entropy comes from the notion of how often collisions occur is a particular finite space when choosing values randomly according to a particular distribution.

The **min entropy** of a distribution D is defined as

$$H_{\infty}(D) = min\{-log_2 D(x) : x \in S\} = -log_2 max\{D(x) : x \in S\} \tag{2.11}$$

Min entropy can be thought of as measuring worst case of all possible probabilities in the distribution. It is not uncommon for a distribution to have fairly high Shannon entropy but have a small **min entropy** value [12].

2.2 Extractors

Extractors are algorithms that can transform a weak random distribution to a distribution that is close to uniform. The main idea behind extractors is the fact that although many sources in nature may not be completely random, they still have some randomness in them that can be used [7]. In other words, there are more weakly random sources available than there are sources that are truly random. So what can be done with such weak sources? Although they do not garner much use with their weak randomness in providing provably secure output, ideally something of this nature would be ideal:

To understand this diagram, it is important to define a weak source and the idea of something being close to random.

Figure 8: Structure of Extractors

Definition 2.4 Statistical Distance. *Let D_1 and D_2 be two distributions on a set S. Their statistical distance is*

$$\|D_1 - D_2\| = \frac{1}{2}\sum_{s \in S} |D_1(s) - D_2(s)| \tag{2.12}$$

If $\|D_1 - D_2\| \leq \epsilon$, *then* D_1 *is* $\epsilon -$ *close to* D_2.

This definition of closeness is as strong as can be hoped for. Specifically, if D_1 and D_2 are ϵ-close, then for any algorithm A,

$$\|Pr_{x \leftarrow_R D_1}[A(x) = 1] - Pr_{x \leftarrow_R D_2}[A(x) = 1]\| \leq \epsilon.$$

Over the years, different models have been used to convey the notion of weak sources. Von-Neumann considered sources to be weak if each bit from the source was independent, but biased. In other words, the probability of the bit being 1 is p, where p may not be $\frac{1}{2}$. Santha and Vazirani considered sources where each bit is biased and the bias of the bit depends on the bits that have come before it. The most general notion of weak sources came from the ideas of **min entropy**, and were put forward by Nisan and Zuckerman [20].

From the equation for **min entropy** (Page [23]), it can be seen that a distribution over $\{0,1\}^n$ that has **min-entropy** k can be called an (n,k)-source. Formally

Definition 2.5 (n,k)-source. *An (n,k)-source denotes some random variable X over* $\{0,1\}^n$ *with* $H_\infty(X) = k$.

Concerning **min-entropy** sources, if a given source has **min-entropy** *k*, then the probability that this source gives a particular string is at most 2^{-k}. More specifically, if a source has **min-entropy** *k*, then a sample of size $2^{k/2}$ from the source would be needed before any sort of repetition was seen. Such a source must also have a support that contains more than 2^k elements. Every (n,k) source is a convex combination of sources which are uniform distributions on sets of size 2^k [8].

The need for extractors is now well defined. The idea behind an extractor is to construct a function *EXT* that is both efficiently computable and deterministic that takes *n* bits of inputs from any source with **min-entropy** *k* and produces *m* bits that are statistically close to being uniformly random. Unfortunately, such a function DOES NOT EXIST!!

To see this, suppose *EXT* was an extractor that could extract just one bit from a source of **min-entropy** *n-1*. At least half the points (2^{n-1} points) in $\{0,1\}^n$ must get mapped to either 0 or 1 by the extractor. Therefore, if the input weak source is a flat distribution on these points, then the source has **min-entropy** *n-1*, but the output of the extractor is statistically far from being uniformly random.A flat distribution S for a given random variable X can be defined as a distribution where the probability that an element $\alpha \in S$ is equiprobable and for $\alpha \notin S$, the probability is zero.

Notice that there is a strong relation between the source picked as a counterexample to the extractor and the extractor itself. This leads to a relaxation to the problem where a function is picked randomly from a family of functions and that is used as an extractor.

Definition 2.6 Extraction Function. *A function EXT:*$\{0,1\}^n \times \{0,1\}^d \rightarrow \{0,1\}^m$ *is a (k,*ϵ*) extractor if for any (n,k) source X and for an Y chosen independently uniformly at random from* $\{0,1\}^d$,

$$\|EXT(X,Y) - U_m\| \leq \epsilon$$

where U_m *is the uniform distribution on m bits.*

2.2.1 Seeded Extractors

It is very easy to see that no single extractor function exists that can produce a close to uniform output on every input distribution having high **min-entropy**[20]. Previous work on the topic has dealt with this fact using one of two solutions. One solution is to have a short truly random seed as a secondary input to the extractor. The other solution looks at not using a seed, but using multiple sources of entropy to harness a close to truly random output from.

The seeded extractor approach allows the extractor to be probabilistic. In other words, in addition to the input to the extractor X, the extraction function can have an additional input Y which is uniformly distributed. Now, the obvious solution for a smart extractor would be to just output Y as it is guaranteed to be uniform. To avoid this, the added stipulation is that the input Y needs to be much shorter than the output length *m*. The additional input Y is called the seed of the extractor, and hence these types of extractors are called seeded extractors.

The natural application of extractors is to be able to simulate randomized algorithms even in settings where weak random sources are available. The line of research behind extractors has a long history starting with Von Neumann's research into taking a source with biased but identically distributed and independent bits and extracting them to a source with unbiased bits (Chapter 1.3.2). Santha and Vazirani considered looking at weaker sources of randomness and being able to use such sources to simulate certain randomized algorithms [22]. The modern definition of weak random sources was put forward by Zuckerman and was used to give a more concrete construction of extractors, although the actual term extractors was given later on by Nisan and Zuckerman [21].

Extractors are also used to yield expander graphs which were discovered by Wigderson and Zuckerman [27]. These findings can then link extractors to applications in super concentrators, sorting in rounds, and the routing of optical networks.

2.2.2 Seedless Extractors

There are many reasons to try and have seedless extractors. One obvious reason is the availability (or lack of) data that is distributed close to uniform (random) to offer as the seed. Random data is hard to come by in many computer environments and instances. Therefore, a need for seedless extractors would be opportune. Moreover, there is a polynomial overhead that goes along with seed enumeration that becomes too expensive especially for cryptographic settings. Along with the seed enumeration being expensive, cryptography, intrinsically prohibits such enumeration. Take for example trying to use a weak source to choose an encryption key through a seeded extractor. It will definitely be insecure to enumerate all secret keys produced using all seeds and then send the encryption of the message using all of these keys. More generally it seems as though seeded extractors cannot always be used in cryptography.

2.2.3 Seedless extractors for few independent sources

When extraction from one source of entropy is impossible without a seed, it would be a natural conjecture to try and look at what can be done if multiple sources of entropy are available that are of comparable quality (entropy-wise). Assuming that there is at least one source of entropy in nature is not that much weaker than assuming that there are several in a given environment.

As stated earlier, one of the first to consider such Multiple source extractors were Santha and Vazirani. In [22], they were able to show that O(log n) independent "semi-random" sources of length n and min entropy δn for every constant $\delta > 0$. Semi random constitutes a source that is weaker than a high-entropy source but still cannot be used to extract without a seed from only one such source.

In [6], Chor and Goldreich proved that for general min-entropy sources, if $\delta > \left(\frac{1}{2} + \epsilon\right) n$, then two sources are adequate. The Hadamard-Sylvester matrix

$$H : \{0,1\}^2 \times \{0,1\}^2 \rightarrow \{0,1\} \tag{2.13}$$

defined by

$$H(x,y) = \langle x,y \rangle \tag{2.14}$$

with the inner product in GF(2) is an extractor exactly of this type. The entropy is bounded by a value of n/2; there are exactly two sources of entropy n/2 on which H is constant. Vazirani was later able to show that a similar function can be used when the entropy comes from two independent sources with $\delta > \frac{n}{2}$, a linear number of bits that are close to the uniform distribution. Vazirani first showed his result for "semi-random" sources and later extended the result to general min-entropy sources [22].

2.3 PRGs

Let us begin by defining some basic concepts that will be needed later on, namely the definitions for a negligible function and a distribution ensemble.

Definition 2.7 Distribution Ensemble. *Suppose that for every n, Y_n is a distribution on $\{0,1\}^n$. Then we say that Y_n is a* ***distribution ensemble****.*

Definition 2.8 Negligible Function. *A function ν is negligible if for every k, there is an n_0 so that for all $n \geq n_0$, $\nu(n) \leq n^{-k}$.*

Pseudorandom Number Generators have a generic formulation that consists of three fundamental pieces. Namely, a stretch measure of the generators, the class of distinguishers that the generator is meant to fool, and resources which the generator is allowed to use [10].

Definition 2.9 Stretch Function. *A mandatory requirement when looking at pseudorandom generators is that using a deterministic algorithm that has the ability to stretch short strings, called seed, into longer strings called output sequences. Specifically, a pseudorandom generator G converts seeds of length k into l(k)-bit long output sequences, where* $l(k) > k$. *The function* $l : N \rightarrow N$ *is called the stretch function.*

The main reason for a pseudorandom generators stretch measure is necessity. One can ask, why not just use the k-bit seed as output? The answer lies in the necessity for information. The shortness of the k-bit seed is often not enough information that may be required from programs or protocols using the generator. An input seed may not be able to provide data at a rate that could be efficiently used by a querying program. So, the main purpose behind the generator is to provide pseudorandom data at a rate that is not obtrusive to what a querying protocol/program is trying to achieve. This leads into our next concept, that of computational indistinguishability. Since the seed file being used, is getting "stretched out" to produce the output, one must ensure that the output sequence is computationally indistinguishable from the input seed. In other words, an onlooking observer, whose "abilities to observe" are bounded by polynomial time, would not be able to tell the difference between the input seed file and the output sequence. This is a concept that was touched on earlier. The only requirement placed on the seed file is that it is close to random (Uniform).

Definition 2.10 Computational Indistinguishability. *Two distribution ensembles* X_n *and* Y_n *are computationally indistinguishable if for every distinguisher* **D** *(from the class of probabilistic polynomial time algorithms),* **D***'s distinguishing advantage is negligible as a function of* n, *where the distinguishing advantage is defined as*

$$|Pr[D(X_n) = 1] - Pr[D(Y_n) = 1]| \tag{2.15}$$

Where U_m stands for the Uniform Distribution over $\{0,1\}^n$ and the probability is taken over U_k (respectively $U_{l(k)}$) as well as over the coin tosses that D makes in the situation that D is probabilistic. The distinguisher ***D*** *can be thought of as trying to figure out whether the string outputted by the generator is random output (distributed as $G(U_k)$) or is a truly random string, distributed as $G(U_{l(k)})$).*

From what has been shown so far in this Chapter, a standard definition for what is expected from a Pseudorandom Generator can be created.

Definition 2.11 Pseudorandom Generator. *Suppose that for all n, $l(n) \geq n$. Let $G : \{0,1\}^n \rightarrow \{0,1\}^{l(n)}$ be a polynomial time function.* ***G*** *is a pseudorandom generator with stretch l(n)-n if the distribution ensembles $G(U_n)$ and $U_{l(n)}$ are computationally indistinguishable.*

2.4 Block Ciphers

A block cipher is an encryption function for fix-sized blocks. Current block ciphers typically use a block size of 128 bits (16 bytes) for AES (Advanced Encryrption Standard. However, other block cipher designs use other blcok sizes, sometimes much smaller. AES, for example, takes a 128-bit plaintext message and encrypt it resulting in a 128-bit ciphertext message. Obviously, the block cipher is reversible on the receiver's end using a decryption function that will take the 128-bit ciphertext message and decrypt it to the original plaintext message. For the sake of secrecy, the plaintext and ciphertext are always the same size, which is known as the block size of the block cipher[9]. With regards to this paper, we need only worry about encrypting the plaintext provided which will be touched later on.

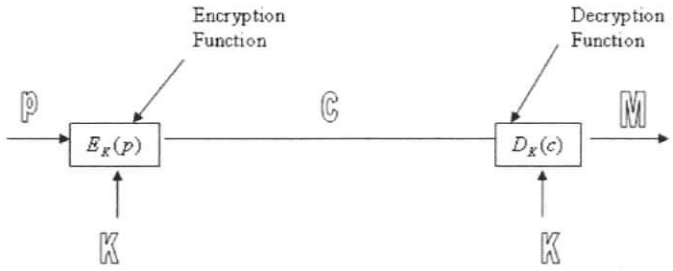

Figure 9: Block Cipher Design

To encrypt, and to decrypt for that matter, a secret key is needed. The secret key allows for the plaintext message to be hidden so that if the encrypted message is intercepted during transmission, it cannot be easily read. The secret key K is also a string of bits. Common key sizes are 128 and 256 bits for AES. We use the terminology $E_k(p)$ to denote the encryption of the plaintext and $D_k(c)$ to denote the decryption of the ciphertext.

It is useful to conceptualize a block cipher as a big table that is dependant on a key. For any fixed key, think of creating a lookup table that maps every plaintext to a unique ciphertext. Needless to say that this table could be quite big. Of course, it is not practical to build such a table in reality, but is a good conceptual model for the idea of a block cipher[9]. Since a block cipher is reversible, every entry in the conceptual table must be unique since each ciphertext when decrypted would need to map to a given plaintext, and the decryption function would not be able to do this if ciphertext was repeated. This big table will then contain every possible ciphertext for a given key exactly once, known as a permutation. A block cipher with a block size of k-bits specifies a permutation on k-bit values for each of the possible key values.

2.4.1 AES

AES stands for Advanced Encryption Standard. This standard was chosen through a proposal and committee process by NIST which saw 15 proposals submitted and from that 5 finalists emerge

[1]. Of the 5 finalists, Rijndael was chosen to be AES. Rijndael was the concatenation of the last names of the inventors of the system, Joan Daemen and Vincent Rijmen.

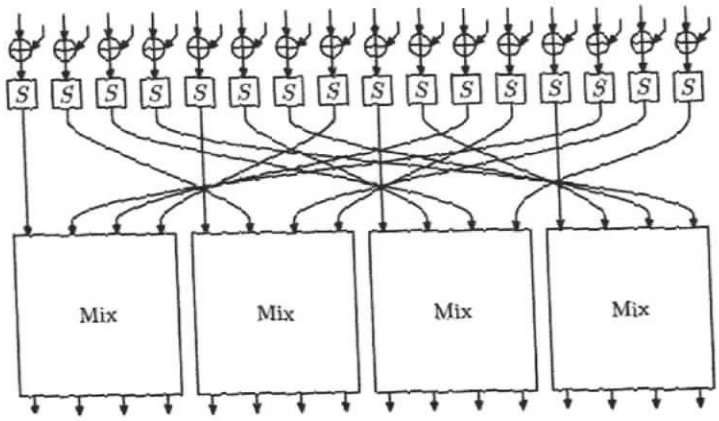

Figure 10: S boxes for AES

Figure 10 shows the structure of a single round of AES taking into account the subsequent rounds is very similar to this. The plaintext comes into the system as 16 bytes of data. The data is split into bytes, and each byte is XORed with the 16 bytes (128 bits) of the round key. Each of the 16 bytes outputted from the XORs is then used as an index into an S-box table that maps 8 bit (1 byte) inputs to 8-bit (1 byte) outputs. The S-boxes are all identical. The bytes are then rearranged in a specific order. Finally, the bytes are mixed in groups of four using a linear mixing function. The linear mixing function refers to the fact that each output bit of the XOR of several input bits [9].

This completes one round of AES. A full encryption consists of 10-14 rounds depending on the size of the key. There is a key schedule that generates the necessary round keys.

2.4.2 Block Cipher Modes of Operation

Block ciphers encrypt, as the name suggests, only fixed size blocks. If there is a need to encrypt a message that is not exactly one block large, some form of block cipher mode is used. Moreover,

block ciphers protect messages that have similar content, which will be discussed later on. A block cipher mode is another name for an encryption function that is built using a block cipher. For the Fortuna heuristic, the Generator makes use of an AES like block cipher encryption function in Counter (CTR) mode. The next Chapter will examine some different modes available for use with AES [9].

2.4.3 CBC

The cipher block chaining mode (CBC) is the most widely used block cipher mode. Added security that was not present in an older block chaining method known as ECB (electronic codebook) is added using an XOR of the previous plaintext block with the current ciphertext block. The standard notation and formulation of CBC is:

$$C_i = E(K, P_i \oplus C_{i-1}) \qquad for\ \ i = 1, \ldots k$$

where P_i represents block i of the plaintext message, C_i block i of the ciphertext message, and K is the key.

The Xoring procedure gives some added randomness to the plaintext blocks using the previous ciphertext blocks. This method avoids situations where identical plaintext blocks would be encrypted using the same key and encryption function giving an adversary extra information about the plaintext itself.

2.4.4 Fixed IV

The ciphertext blocks are determined by the function above. However, the function does not specify a value of the original ciphertext block C_0. The block is known as the *initialization vector* or *IV*. A fixed IV should be used for the same reason that previous ciphertext blocks are XORed with current plaintext blocks in the encryption function. Often time messages begin with similar

or identical blocks (i.e. how many sentences start with the word "The"). This information would be something an adversary would want to know.

2.4.5 Counter IV

An alternative idea would be used some kind of counter for the IV. Use IV = 0 for the first message, IV=1 for the second message and so on. Again, this method has its flaws. If the first blocks of the messages only differ slightly, then a simple IV could very well cancel out those differences yielding identical ciphertext blocks again. For example, the values 0 and 1 differ by exactly one bit. Consider the case where the first two messages to be encrypted differ by only one bit in the same location as the IV counter bit. This will lead to the first ciphertext block of the two messages to be identical. An adversary could quickly deduce information about the two messages, something a secure encryption scheme should try to avoid and not allow.

2.4.6 CTR

Counter mode, given the 3 letter abbreviation CTR, is a favorite of many who use block ciphers. Even though it has been around for some time, it never became very popular in literature and in application because it was not standardized as one of the official AES modes. It was very recently standardized by the NIST (National Institute of Standard Technology). CTR is a stream cipher mode. This means that the message itself is never used on input to the block cipher. Instead, the block cipher is used to create a pseudorandom stream of bytes called the key stream, and this stream is then XORed with the plaintext to generate the ciphertext.

CTR mode is defined as follows:

$$K_i := E(K, Nonce||i) \quad for \ \ i = 1, \ldots k \ C_i := P_i \oplus K_i$$

where P_i represents block i of the plaintext message, C_i block i of the ciphertext message, and K is the key.

Like any stream cipher, a unique nonce must be supplied of some form. Most systems build the nonce from a message number and some additional data to ensure its uniqueness. The Fortuna algorithm, which will be discussed later on, uses a counter value C and increments the counter for every block of output as the nonce.

Counter mode is a very simple method that can be used to generate a key stream. One of its main stipulations is that a key/nonce combination never be used twice. This is a common disadvantage for this type of block cipher mode. If a key/nonce combination is used more than once, information about the plaintext could be leaked[9].

2.5 Hash Functions

Hash functions are the most versatile of the cryptographic primitives. Hash functions can be used for a number of cryptographic activities including encryption, authentication, and even for randomness extraction to an extent.

The basics behind a hash function is to take an arbitrarily long input string and from it produce a fixed length output (usually shorter). They have many applications in cryptography. Any time a variable sized value needs to be mapped to a fixed size value, a hash function can be used. Hash functions can also be used as cryptographic pseudorandom generators to generate several keys from a single shared secret [9]. Even though hash functions are used in many cryptographic systems, less is known about them compared to what is known about block ciphers.

There are choices when looking for a hash function. Some of the more popular choices are something from the SHA family or MD5. One of the main advantages to choosing from the SHA family is that they were standardized by NIST and developed by the NSA [19].

Almost all hash functions are iterative hash functions. Iterative hash functions split the input into a sequence of fixed size blocks $m_1, m_2, \ldots m_k$. If the last block is not complete (not even bits to fill it), a padding rule is used to fill it. The message blocks are processed in order using a compression function and a fixed size intermediate state. The hashing process starts with a fixed value H_0. Preceding values are calculated using

$$H_i = h^{'}\left(H_{i-1}, m_i\right)$$

The final value H_k is the result of the hash function.

There are significant practical advantages to such an iterative design. Compared to a function that allows variable length input to it, such hash functions are easy to specify and implement. Secondly, since the input is broken up into fixed length blocks, the hash of the entire input can begin prior to having received the entire input. This a definite advantage where data becomes available in a stream, so the hash can be computed on the fly without even having to store the data.

2.5.1 SHA-1

The Secure Hash Algorithm as stated earlier, was designed by the NSA and standardized by the NIST. The first version was called SHA and is now referred to as SHA-0. There were some known weaknesses in this implementation that were resolved with the release of SHA-1[9].

SHA-1 is a 160-bit hash function which is loosely based on MD4. It has a lot of features in common with another current popular implementation known as MD5, however SHA-1 is noticeably slower in the range of two to three times.

SHA-1 has a 160-bit state which consists of 5 32-bit words (A,B,C,D,E).Similar to MD5, SHA-1 has 4 rounds consisting of elementary 32-bit operations. F is a nonlinear function that varies from round to round.

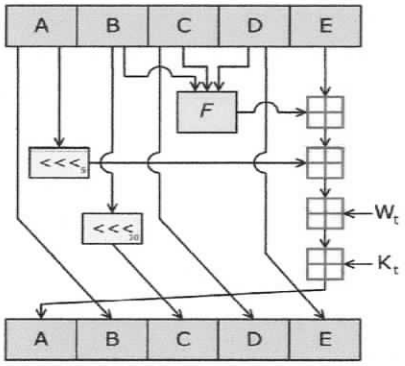

Figure 11: Design of SHA-1

2.5.2 SHA-256

Fairly recently, three new hash functions were introduced. SHA-256, SHA-384, and SHA-512 were created which corresponded to the bits of output respectively. They are designed to be used with the 128-, 196-, and 256- bit sizes of AES with a similar structure to SHA-1.

SHA-256 although similar in nature to SHA-1, is quite a bit slower. For significantly long messages (in the MB), computing the hash of such messages using SHA-256 takes about the equivalent amount of time to encrypting the message using AES. This is not a huge surprise because in all actuality computing the hash of a given message is more difficult a problem than encryption function.

3 Fortuna Heuristic and /dev/random

We now concentrate our focus on some pseudorandom generator models and implementations. Within the UNIX operating system, there are two random number generators implemented, namely dev/random and dev/urandom. The two applications differ from one another very slightly in implementation, however very significantly in impact. This will be discussed later on. Both applications are based on the Yarrow algorithm, named after the yarrow plant, which was created by Bruce Schneier, John Kelsey, and Niels Ferguson. Fortuna, named after the Roman goddess of chance, is an improvement on the original implementation of the yarrow algorithm that was put forth in 1999. This Chapter looks to describe the intricacies of the algorithm[9]. Moreover, initial research in the field of PRGs will be shown here. These include some statistical testing as to the quality of certain PRGs as well as tests that can be used to test PRG quality.

3.1 Fortuna

Fortuna's main advantage over the original Yarrow algorithm is its removal of entropy estimators[14]. There are three modules to the Fortuna algorithm, the **generator**,**accumulator**, and **seed file control**. The generator takes a fixed size seed and generates arbitrary amount of pseudorandom data. The accumulator has 2 responsibilities; it collects and pools entropy from various sources and it also occasionally reseeds the generator. The last part, the seed file control, ensures that the PRNG can generate random data even when the computer has just started up.

3.1.1 Generator

The first module of the algorithm is the generator that converts a fixed-size state to arbitrary long outputs. The internal state of the generator consists of a 256 bit block cipher and a 128 bit counter.

The generator can be summarized by viewing it as a block cipher in Counter mode. The counter mode generates a random stream of data, which will be the output of the generator.

When a user or application requests random data, the generator runs its necessary algorithms and generates pseudorandom data. Suppose an adversary learns the state of the generator upon the request for random data is completed. So that this chance of a breach not compromise the results the generator gave, after every request for random data, the generator creates an ACCESS of 256 bits which is used as the key for the block cipher. By doing this and forgetting about the old key, leaking of the old key at any given time will not give any info about the pseudorandom data created since the key is changed after every request.

When the key is redone at the end of every request, the counter remains intact. This avoids the problems of repeated key values. If the counter was to be reset every time for fixed sized requests, and a key value was to repeat, then the next key could be a repeat. It could in turn create a short cycle of key values, and in doing so provide a large advantage to an adversary. Even though this is still a very unlikely phenomena, by letting the counter run without resetting it at every request it is avoided altogether. Since the counter is 128 bits, a counter value will never be repeated as 2^{128} blocks is beyond the computational capabilities of most computers. Furthermore, a counter value of 0 indicates that the generator has not yet been keyed.

The generator is a useful module as part of the Fortuna algorithm. Its usefulness is not restricted to the Fortuna algorithm but could also be used as a separate module for any time pseudorandom data is required for a variety of applications.

3.1.2 Accumulator

The accumulator collects real random data from various sources and uses this information collected to reseed the generator.

3.1.3 Entropy Sources

The environment, as talked about earlier, contains many natural occurring sources of entropy. Any of these sources can be used to harness entropy just as long as one of the sources generates data that can remain unpredictable to an adversary. Unfortunately it is hard to guess how an adversary will act. So, the most secure approach is turn any source that can generate unpredictable data into an entropy source. Certain known sources include keystrokes, mouse movement, hard drive and printer responses [9]. It is not important if the adversary can mimic or predict any of these entropy sources, just as long as he is unable to predict all of them.

Each source is identified with a unique source number in the range of $[0 \ldots 255]$ (1 byte). The data in events used for entropy are a short sequence of bytes. A source should only include the data deemed unpredictable in each event. Data such as time stamps should not be included as it can be assumed that an adversary would have access to this information anyways.

The events of the various sources will be concatenated. To ensure that a string of information made up of the concatenation of all these different events is unique, the string needs to be parsable. Each event is encoded as a minimum of three bytes of information, if not more. The first byte contains the unique source number of the source the event is from. The second byte indicated how many bytes of data are to follow in this event. The subsequent bytes contain the data the source provided.

It is a safe assumption that an adversary will have knowledge of the output of some of the sources. To help combat this, the design of the PRNG should be to assume that some of the sources are under complete control of the adversary. In other words, all events generated and outputted from this source are under the control of an adversary. Moreover, an adversary can request output from the PRNG at any time.

3.1.4 Pools

The pools of random data are used to reseed the generator with. As stated earlier, an assumption can be made that the adversary can have control over a few sources. To overcome this, the pools need to be large enough so that the adversary can no longer enumerate efficiently the data stored in the pools for the events as they occur. Making the pools large enough barries the information the adversary might have known about information he was able to output to the pools from the sources he controlled (they become mixed with information from the other sources).

32 pools are used: $P_0, P_1, \ldots, P_{31}$. Each source distributes its random events over the pools in a cyclical fashion[9]. This ensures that a source's entropy is distributed evenly over all of the pools. As more and more random events happen for the different sources, the random data created from the source is appended to the pool's string of data.

The generator is reseeded every time the initial pool P_0 has enough data in it. Reseeds are categorized numerically by a number r that can take on values 1,2,3,. The reseed number determines which pools are to be used in the reseed. Pool P_i is included in the reseed if 2i is a divisor of r. Therefore, P_0 is used in every reseed, P_1 in every other reseed, P_2 every fourth reseed, and so on. After a pool is used in a reseed attempt, it is reset to the empty string.

3.1.5 Distribution of Events over Pools

As the events occur for the entropy sources, the event data needs to be distributed over the pools. The simplest solution would have the Accumulator itself be in charge of this. However, this can prove dangerous since an adversary could also make calls to this function. An adversary could make calls to this function every time data needed to be pooled, and hence try to influence which pool the data went to. If by some manner the adversary is able to send data from all the sources he does not control into P_0, then the multi pool system becomes ineffective as the randomness is reduced down to one pool. Single pool attack becomes a large possibility. Fortuna addresses

this problem by letting every event generator pass the pool number along with each event. For an adversary to have access over this information, he would have to have access to the memory of the program that generates the event. It can be safe to assume that if an adversary had that much access, other information would most likely be compromised as well.

3.1.6 Seed File Management

During proper operation, the sources of entropy are consistently delivering random event data to the accumulator which is used to reseed the generator. However, if the system was ever rebooted, the entire system would be put on hold in anticipation of the sources to generate random data so the generator could be reseeded. The solution to this is a seed file which is kept separately and is a file that is full of entropy. After a reboot, the PRNG reads from the seed file and uses its entropy to create a random seed. Once the seed file is used, it needs to be rewritten with new data.

One known problem is the time in between the generator being reseeded and the seed file being updated. During this time, an adversary could in fact compromise the state of generator if he knows the seed file. Here is the scenario, the adversary reboots the system and requests random data from the generator. The seed file is used to reseed the generator which the adversary intercepts and in turn learns the seed value. He then reboots the system before the program has a chance to update the seed file and in turn is able to gain the value that will be used to seed the generator. If a real request is then passed to the generator for random data, the adversary will be able to generate the same data on his own using the seed value from before the reboot. This is a major problem as the random data is often used to generate cryptographic keys.

The only known solution to this is to always ensure that the seed file remain secret. Having the seed file known to the adversary at any time could compromise the whole system. Proper procedures should be implemented to make sure the seed is always kept secret from the adversary.

3.2 /dev/random

This description of /dev/random is based on the version 2.6.10 of the Linux kernel which was released December 24, 2004. The pseudorandom generation of random numbers in Linux is done using the program random.c in the device directory of the kernel [26]. It actually has two separate implementations. /dev/random is the main implementation that claims to provide secure pseudorandom numbers when prompted to do so. One of its main features of note is that if the entropy sources from which it is drawing from run dry for any given reason, it ceases its output and waits for entropy to become available again. The other implementation, known as /dev/urandom is different from /dev/random in that it will continue to provide pseudorandom data even if the entropy pools have run dry of randomness. In this brief description of /dev/random one should be able to notice its similarities to the Fortuna Heuristic. It is believed that /dev/random is loosely based on Fortuna, however, in the literature on /dev/random there is no mention of this relation [26].

The generation of numbers by random.c is done by the use of three procedures that run asynchronously. The first procedure looks to gather entropy from events that occur inside the kernel. In the second procedure, the entropy collected using the first procedure is fed into a LFSR-like pool (Linear Feedback Shift Register), using a mixing function[11]. From here, when random bits are requested, the third procedure is invoked and random bits are generated. During all three of these procedures, the only non-linear cryptographic operation that is used is the SHA-1 hash function.

3.2.1 Pools and Counters

The entropy once collected from sources is kept in three entropy pools: *primary* (512 bytes), *secondary* (128 bytes) and *urandom* (128 bytes). As entropy sources add data to the primary pool, output from the primary pool is extracted and fed into the secondary and urandom pools using a SHA-1 hash function. SHA-1 hash functions have been shown to act as adequate extraction functions in situations like these [11]. The output of the Linux Random Number Generator (LRNG) is

then extracted from the secondary pool (for /dev/random) or the urandom pool (for /dev/random). In certain situations, if the primary entropy pool is full, system entropy might directly be added to the secondary pool.

Each pools has its own entropy estimation counter. This counter tries to keep an estimate to the amount of entropy currently present in the pool. It is decremented when entropy is extracted from the pool and is incremented when entropy is added to the pool. The counter is always decremented by the amount of entropy extracted form the pool[11]. With incrementation, the counter is incremented by the 'estimated' amount of entropy that is added to the pool. If bits are transferred from the primary pool to one of the other pools, the bits decremented from the primary pool are incremented in the other pools the bots are begin transferred to.

The entropy counter of the secondary pool plays a very key role and is the reason for the 'blocking' characteristic of /dev/random. Its task is to determine if there is enough entropy in the secondary pool to fulfill a request for random bits. If not, the LRNG tries to transfer bits from the primary pool to the secondary pool to try and meet the desired quota. If the primary pool is empty, the program blocks and waits for entropy to become available that can fulfill the request.

3.2.2 Entropy Harvester

Most computers do not have hardware based random number generators attached to their computers. To compensate for this, an operating system planning to implement a random number generator must use a software source to generate the entropy needed for such a device. The most common types of software based entropy sources are those that use timings of certain types, in this case, the inter-interrupt timings of various devices.

There are four specific entropy harvesting functions used in dev/random and dev/urandom:

1. addkeyboardrandomness( )

2. addmouserandomness( )

3. addinterruptrandomness( )

4. adddiskrandomness( )

These four functions will extract timing deltas from the appropriate interrupts or devices. Then, the respective function will use the generic addtimerrandomness() function to harvest the entropy and call the appropriate functions to add the entropy to one of the entropy pools used to store entropy.

Due to the asynchronous nature of the system, collected entropy can not just be added directly to the primary pool, but rather it is collected and batched, and at time intervals the batched data is added to the pools[9]. The default place to add the entropy collected and batched is to the primary pool, as discussed earlier. However, if this pool is full, entropy is added to the secondary pool. When the secondary pool is full, the process returns to the primary pool and so on. Entropy is never added to the urandom pool.

3.2.3 Generating Output

Random bits can actually be extracted from any one of the three pools. It is extracted from the urandom pool anytime /dev/urandom is called. It is extracted from the secondary pool when the function /dev/random is called and only when there is enough entropy in the pool to fulfill the request. The primary pool can also be used as output if the other two pools are empty when a request from random bits is made. The process of entropy extraction involves three steps:

1. updating the pool's contents

2. extracting random bits to be outputted

3. decrementing the entropy counter of the respective pool

The extracting of bits from the relevant pools is accomplished using a hash of the pool contents using SHA-1.

3.2.4 Maurer's Test

The pseudorandom generator /dev/random was part of the initial phase of research that looked into the implementation and statistical accuracy of many PRGs. A statistical test, known as Maurer's Universal test, was used as a basis to help conclude whether many of the known PRGs on the market today actually met some statistical testing so at least in a practical sense, one could have some conclusion on the output of such devices.

Maurer's test, created by Ueli M. Maurer in 1992, looks to encompass the previous defects discussed individually in the five known basic tests (Runs test, Poker test, Serial Test, Frequency Test, Autocorrelation Test) along with specifically looking at the nature of a given source to see if it is cryptographically usable [17]. There is one main advantage to Maurer's test over the previously discussed statistical tests. Rather than being uniquely created to detect a specific type of defect that can occur in a Random Bit Generator (RBG), Maurer's test is able to detect from anyone of a general class of defects that can be found in an ergodic stationary source that has finite memory. The defects it can identify include all those pointed out by the five basic tests. It is a common fact that if a sequence of bits can be compressed, it can be considered non-random. The purpose of the test is to determine whether a sequence of bits can be significantly compressed without any loss of information. If a sequence can be significantly compressed, it should be deemed non-random.

For information on the algorithm for Maurer's test, look to Appendix A. For the scope of this paper, all that needs to be known is that two threshold values for a given block size (L), t_1 and t_2 determine the range within which the output of Maurer's test must fall for the input sequence being tested to be deemed pseudorandom.

From the results in Table 2, it can be seen that /dev/random passes Maurer's test for block

Table 2: Mauer's Test Results on /dev/random

L	t_1	/dev/random	t_2
6	5.2077	5.2156	5.2277
7	6.1887	6.1953	6.2038
8	7.1781	7.1820	7.1892
9	8.1724	8.1758	8.1805
10	9.1694	9.1734	9.1753
11	10.1679	10.1715	10.1721
12	11.1673	11.1689	11.1703

sizes from L=6...12. These block sizes are within the recommended values for L [17]. It was at this point that our research hit a wall. The initial plan was that such a PRG, that is not suited for cryptographic use, would fail a statistical test such as Maurer's test and from these negative results, means or avenues could be explored to improve such a PRG. This did not occur. However, what was learned instead was that most of these statistical tests on their own do not offer much insight into PRGs as they are often very one-dimensional and incomplete. Many of these tests can be easily fooled with pre-determined input file sequences. A more concrete result would be attainable from strong theoretical conclusions accompanied by such tests, which the PRG /dev/random did not provide.

This initial research led to exploring other PRG's that were first and foremost theoretically shown to be secure. Of those looked at, a model created by Barak and Halevi showed the most promise. This model, was the inspiration for our model which will be described in the next Chapter.

4 Our Model

In this Chapter, a formal model and architecture for a pseudorandom number generator is presented. Such a PRNG, should be able to have resilience against an adversary that has control or knowledge of the output of two of the extractor's entropy sources. The properties that need to be met by the PRNG can be summarized by:

- **Resilience**. The output of the generator looks random to anyone that has no knowledge of the internal state. The state is set using the seed to the generator.

- **Forward Security**. Past output of the generator looks random even if the observer is able to learn the internal state at a later time.

- **Backward Security/Break-in Recovery**. Future output looks random to an observer even if they are able to learn the state of the generator at the current time. This hinges on the requirement that the generator is refreshed with data that has sufficient entropy.[2]

This by no means is a model that is diving into untouched territories in the field of pseudo random number generation. Many models have been suggested before. The uniqueness at the base of this model is the use of a multiple source extractor to extract entropy from entropy sources with a guaranteed min-entropy value and use the extracted entropy as a seed to the generator.

The model looks to separate the PRNG into two parts: the multiple source extractor and the generator itself.

4.1 Architecture

The architecture of our model, similar to [2], looks to separate the extraction process from the randomness generation process. The extraction process is done using an extractor, similar to the

designs discussed in 2.2. The extraction function converts a high-entropy input into a much shorter input that is close to the Uniform distribution. Taking into account the availability of entropy for different systems, some form of seedless multiple source extractor (Chapter 2.2.3) should be used. The details of the extractor will be discussed in Chapter 4.4.

The generator G can be chosen from a variety of choices that were discussed earlier (Chapter 2.4). In the design presented here, block cipher in Counter mode (AES in CTR mode), which was suggested in [3]. However, any standard cryptographic PRG, that takes as input a random seed and outputs a pseudorandom output of length twice that of the seed can be used. The output of the extraction function should equal the seed length of the PRG.

4.2 Extractor

The extraction process uses a deterministic, non-cryptographic, extraction function and uses independent entropy sources above a known min-entropy and is able to extract an output from these sources that has a distribution that is close to Uniform.

Definition 4.1 ***Multiple-Source Extractor.*** *A function* $f : (\{0,1\}^n)^c \rightarrow \{0,1\}^m$ *is a c-source (k,*ϵ*)-extractor if for every family of c independent (n,k)-sources* $X_1, \ldots, X_c$*, the output* $f(X_1, \ldots, X_c)$ *is* ϵ*-close to* U_m*, where* U_m *is the uniform distribution on m bits.*

In Chapter 2.2.3, the use of a Hadamard-Sylvester matrix was suggested as such a function. The Hadamard-Sylvester matrix is viable only for two independent sources at a time, however in our model, access to four independent sources is a viable option. To facilitate the use of all four independent sources, an iterated design that would combine two independent sources and then combine the output would make use of all available entropy sources. (Figure 14). Moreover, the use of an iterated design would in itself assure the security of the extractor even with an adversary in full control of one entropy source.

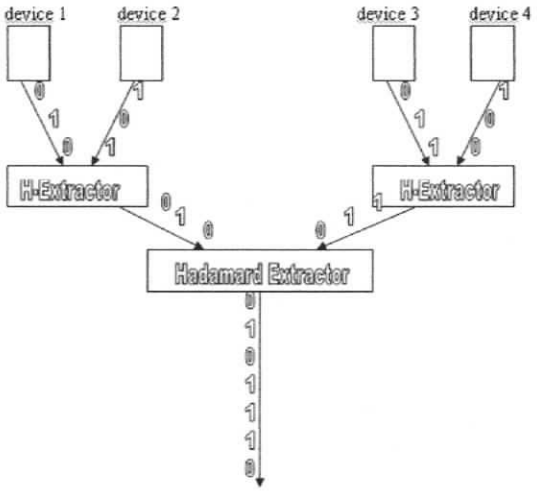

Figure 12: Architecture of Multiple Source Extractor

In [6], a formal proof is given to the properties of a Hadamard-Silvester extractor. The notion of a ϵ-robust random variable is introduced, which is defined as a variable (Z) assuming values in $\{0,1\}^m$ where for every $\alpha \in \{0,1\}^m$

$$(1-\epsilon) \bullet 2^{-m} \leq Pr\left[Z=\alpha\right] \leq (1+\epsilon) \bullet 2^{-m} \tag{4.1}$$

Hadamard-Silvester extractors are shown to be ϵ-robust in [6]. The basis of the proof are two lemmas in the paper. The first lemma, shows that for every function on $\{0,1\}^{2l} \mapsto \{0,1\}^m$, the probability distributions for all such functions (infinite) can be analyzed by only looking at the flat-distributions (finite) in the same space when dealing with the maximum and minimum probabilities. A flat distribution S for a given random variable X can be defined as a distribution where the probability that an element $\alpha \in S$ is equiprobable and for $\alpha \notin S$, the probability is zero.

So, since flat distributions can be examined for min/max probabilities, the proof goes on to state that for a t-dimensional Hadarmard-Silvester matrix, the sum of every submatrix $(r \times s)$ is

$\leq \sqrt{r \bullet s \bullet t}$. This fact of having small elements' sum for every submatrix within the Hadamard-Silvester matrix, leads to the construction being ϵ-robust.

4.3 Robust PRG

A robust pseudorandom generator looks to encompass the properties that are expected out of a secure PRG (Chapter 4.3) in a manner that is provably secure. It should be noted that the main security) parameter of the robust PRG is the state s. The main purpose of the state is to seed the Generator, but it is also essential to the security of the PRG itself. Since the generator will most likely act in a deterministic manner, knowledge of the state can compromise the output of the PRG. So, the state s becomes of the utmost importance not only as a close to truly random seed to the generator, but also as a security parameter to the entire PRG.

A robust pseudorandom generator consists of two functions:

$(r, s') \leftarrow next\,(s)$, where the seed to the generator acts as the state. The output of next is split into two m-bit blocks, the first m-bits is used as output to the generator and the second m-bits is used to refresh the internal state with the new state s'.

s' $\leftarrow$refresh(s,x), with x being a string of length at least m bits and s being the current state. Updates the current state using the data given by x.

The main security requirements for a robust pseudorandom generator should consist not only of secrecy of the state s, but computational indistinguishability of the output of the generator. In other words, if an adversary looks to compromise our system, he should not be able to learn any information about the state (close to truly random) by examining the output of the generator (output) within a finite amount of time (polynomial). The security requirements for a robust pseudorandom generator, first conceived in [2], can be formulated looking at probabilistic games being carried out by two players. One player is the system that attempts to implement the generator, whereas the other player is the adversary that wishes to attack the system. Using the concept of an ideal

world versus real world game, the 3 properties (Resilience, Backward Security, Forward Security) introduced in Chapter 4 are used to formulate the definition of security for the robust pseudorandom generator. The real world game looks to describe a real attacker that interacts with a robust generator. The ideal world game is meant to capture "'the most secure process one can possibly have"' [2].

From these two worlds, a construction can be deemed secure if an adversary can not distinguish between interacting with the generator in the real world and interacting with the secure process in the ideal world.

The real world game.

The adversary in the real world can be said to be an efficient procedure A that has a total of four interfaces to the generator, namely

1. **goodrefresh**(bit string)
2. **badrefresh** (bit string)
3. **setstate**(bit string)
4. **nextbits**().

The real world game is parameterized by a security parameter m that is the length of the seed file (state) of the generator, which in this case is 256 bits. The real world game begins with the player representing the system initializing the state of the generator to NULL, ($s \leftarrow 0^m$), and then the adversary interacts with the system using the above mentioned interfaces:

- **goodrefresh**($x \leftarrow f(X_1, \ldots, X_c)$) with the output of the extractor. The system sets $s' \leftarrow$ refresh(s',x) and $s^{'}$ is used to update the internal state (seed) to $s^{'}$.

- **badrefresh**(x) with a bit string x. The system sets $s' \leftarrow$ refresh(s',x) and s' is used to update the internal state (seed) to s'.

- **setstate**(s') with an m-bit string s'. The system returns the current state s to the adversary and changes it to s'.

- **nextbits()**. The system runs (r,s')$\leftarrow$next(s), and replaces the internal state s with s' and returns to the attacker the m-bit string r.

The game continues using these interfaces until the adversary decides to halt with some output 0,1, pending on which world he believes he is in. For a particular construction PRG =(next,refresh), let Pr[$A(m)^{R(PRG)} \leftarrow 1$] denote the probability that A outputs a "1"' after interacting with the real world game as above with the system that implements the generator PRG with parameter m. R(PRG) stands for the "'real world process"' from above.

The ideal world game.

The ideal-world game proceeds very similarly to the real world game. The main difference is the way the calls that A makes to its interfaces are handled. Whenever A thinks that it is learning something new from the system, typically from a **nextbits**() or **setstate**() call, the ideal process, not wanting to give any information about the system away, returns a random m-bit string to A independent of everything else.

The only exception to the above stated cases is when A already knows the internal state due to some previous **setstate** call. In this case, everything A sees until the next **goodrefresh** call should be in sync with the internal state it already knows. To combat this, the ideal process will maintain a flag called **compromised** that will be set on a **setstate** call and reset on a **goodrefresh** call. The system will act as a RPG when **compromised**=`true` and returns random strings when **compromised**=`false`.

The ideal world game is parameterized by the identical security parameter m as before. The game begins in a similar fashion as the real world game with the system player initializing $s \leftarrow 0^m$ and **compromised**←`true`. The adversary interacts with the system using the following interfaces:

- **goodrefresh**($x \leftarrow f(X_1, \ldots, X_c)$) with the output of the extractor. The system resets **compromised**←`false`.

- **badrefresh**(x) with a bit string x. If **compromised**=`true` then the system will set s' ←refresh(s,x) and updates the internal state (seed) to s'. If **compromised**=`false` nothing is done.

- **setstate**(s') with an m-bit string s'. **compromised**=`true` then the system returns to the attacker the current state s, and if **compromised**=`false` then the system chooses a new random string $s \leftarrow_R \{0,1\}^m$ and returns it to the adversary. The system in both cases sets **compromised**←`true` and sets the new internal state to s'.

- **nextbits()**. If **compromised**=`true`, the system runs (r,s')←next(s) and replaces the internal state s with s'. The m-bit string r is also returned to the attacker. If **compromised**=`false` then a new random string $s \leftarrow_R \{0,1\}^m$ is chosen and returned to the adversary.

The game continues using these interfaces until the adversary decides to halt with some output 0,1, pending on which world he believes he is in. For a particular construction PRG =(next,refresh), let Pr[$A(m)^{I(PRG)} \leftarrow 1$] denote the probability that A outputs a "1"" after interacting with the ideal world game as above with the system that implements the generator PRG with parameter m. I(PRG) stands for the "'ideal world process"' from above and note that the PRG is openly used to answer queries that are asked when the **compromised** flag is set to `true`.

A construction PRG is secure if no adversary can tell the difference between interacting with the system implementation the PRG in the real world as opposed to interacting with the ideal process. As a formal definition:

Definition 4.2 Robust PRG. *A PRG = (next, refresh) is a robust pseudorandom generator (with respect to an extracted seed file from a multiple source extractor) if for every probabilistic polynomial-time adversary algorithm A, the difference*

$$\left|Pr\left[A(m)^{R(PRG)} \rightarrow 1\right] - Pr\left[A(m)^{I(PRG)} \rightarrow 1\right]\right| \tag{4.2}$$

is negligible in the security parameter m.

4.4 Construction

After defining the model, a simple construction can now be shown to satisfy definition 4.2. Our model is broken into two components, one being the multiple-source extractor Ext:$\{0,1\}^{\geq 4\times m} \rightarrow \{0,1\}^m$ and a simple (non-robust) cryptographic PRG G:$\{0,1\}^m \rightarrow \{0,1\}^{2m}$ such that G(U_m) is computationally indistinguishable from U_{2m}. In other words, the PRG G() should be able to take an input seed file of m-bits, the security parameter, and expand this seed file to render an output of 2m bits which is computationally indistinguishable from the seed file. In this informal definition, U_i refers to the uniform distribution on $\{0,1\}^i$. Generators such as these can be built from a variety of cryptographic primitives that were discussed earlier in Chapters 2.4 and 2.5.

The notation (r, s')←G(s) that will be introduced below denotes that r is the first m bits of the output of the generator (2m bits) and s' is the last m bits. Also, denote G' as the function that on input $s \in \{0,1\}^m$ outputs only the first m bits of G(s). Our PRG operates as follows:

Operation of PRG. For a security parameter m, given a multiple source extractor Ext:$\{0,1\}^{\geq 4\times m} \rightarrow \{0,1\}^m$ and a cryptographic non-robust PRG G: $\{0,1\}^m \rightarrow \{0,1\}^{2m}$, from this, the robust PRG behaves as follows:

- It has a state (seed) $s \in \{0,1\}^m$

- It has a function called refresh(s,x) that returns $s^{'} \leftarrow G^{'}(s \oplus Ext(x))$.

- It has a function called next(s) that returns $(r, s^{'})$←G(s)

Theorem 4.3 . *Let m be a security parameter, let Ext:* $\{0,1\}^{\geq 4m} \rightarrow \{0,1\}^{m}$ *such that Ext is a multiple source extractor. Let G be a cryptographic PRG. Then, the construction above is a robust pseudorandom generator iff the input sources have min-entropy* $k > \frac{m}{2}$.

Proof: . Fix some efficient attacking algorithm A. Consider the following two experiments"'

1. **Experiment R** A interacts with the real system R(PRG)

2. **Experiment I** A interacts with the real system I(PRG)

The end goal is to show that A outputs "'1"' with the same probability in both experiments. On top of this, consider another experiment denoted **Experiment H** (Hybrid). The hybrid experiment is very similar to Experiment R, except that is uses a truly random m-bit string to refresh the state in a good refresh calls, instead of using output of Ext(x). In Experiment H the adversary A interacts with a process that works as follows:

- **goodrefresh**($x \leftarrow f(X_1, \ldots, X_c)$) with the output of the extractor. The system draws $d \leftarrow \{0,1\}^{m}$, sets $s^{'} \leftarrow G^{'}(s \ominus d)$, and updates the internal state (seed) to $s^{'}$.

- **badrefresh**(x) with a bit string x. The system sets $s^{'} \leftarrow refresh(s' \oplus Ext(x))$ and $s^{'}$ is used to update the internal state (seed) to $s^{'}$.

- **setstate**($s^{'}$) with an m-bit string $s^{'}$. The system returns the current state s to the adversary and changes it to $s^{'}$.

- **nextbits()**. The system runs (r,s')←next(s), and replaces the internal state s with s' and returns to the attacker the m-bit string r.

The proof will proceed as follows. First it will be shown that the view of A in **Experiment H** is statistically close to its view in **Experiment R**. On the other hand, for an adversary to be able to distinguish between **Experiment H** and **Experiment I**, the adversary would have to be able to break the underlying cryptographic PRG.

Let q_r be a (polynomial) upper bound on the number of goodrefresh() calls that A makes during their attack. Notice that the view of A is just a deterministic function with q_r calls to goodrefresh() of string of size m that are a result of the function Ext(). In **Experiment H** the distribution over these strings is a uniform distribution, since the string are chosen at random. In **Experiment R** each of these strings is set to Ext(x) where x comes from the entropy sources. But since the statistical distance between the output of the extraction function and the uniform distribution is bounded by 2^{-n}, from this it follows that the statistical distance between the view of A in the two worlds (Hybrid and Real) cannot exceed $q_r/2^n$.

To get into the second claim, it will be shown that distinguishing between **Experiment I** and **Experiment H** implies breaking the underlying cryptographic PRG. Let p_H stand for the probability that A outputs "'1"' in **Experiment H** and p_I represent the probability that A outputs "'1"' in **Experiment I**. We will show how, given any such A, there is a procedure B that can break the cryptographic protocol G with Advantage of at least $(p_H - p_I)/q$, where q is a polynomial upper bound on the total number of call made by A to ALL of its interfaces. Input to B comes in the form of 2 m-bit strings, r^* and s^*. Procedure B tries to determine if the pair (r^*, s^*) were chosen random and independently from $\{0,1\}^m$, or were set as $(r^*, s^*) \leftarrow G(s)$ for a randomly chosen $s \leftarrow_R \{0,1\}^m$.

Procedure B(r^*, s^*) uses the adversary A as a subroutine. Procedure B begins by choosing a random index $i^* \leftarrow_R \{1, 2 \ldots q\}$, and also setting $s \leftarrow 0^m$ and the flag **compromised**=`true`.

Procedure B then runs A, answering the first $i^* - 1$ calls of A with random bit strings, answering the i^*th call of A using input (r^*, s^*), and answering the $i^* + 1$ call and on using what is done in **Experiment H**. Specifically, the i^*th call of A is answered as follows:

- **goodrefresh**($x \leftarrow f(X_1, \ldots, X_c)$) with output from the multiple source extractor.
  - If $i < i^*$. B chooses at random $s' \leftarrow_R \{0,1\}^m$.
  - If $i = i^*$. B uses its input, setting $s' = s^*$.
  - If $i > i^*$. B chooses at random $d \leftarrow_R \{0,1\}^m$ and sets $s' \leftarrow G'(s \oplus d)$ where s is the current internal state.

  In either case, B updates the internal state to s' and sets **compromised**=`false`.

- **badrefresh**(x) with a bit string x.
  - If **compromised**=`true` or $i > i^*$. B sets $s' \leftarrow G'(s \oplus Ext(x))$.
  - If **compromised**=`false` and $i < i^*$. B chooses a random string $s' \leftarrow_R \{0,1\}^m$.
  - If **compromised**=`false` and $i = i^*$. B uses its input, $s' = s^*$.

  In either case, B updates the internal state to s' but does not set the flag **compromised**.

- **setstate**(). B returns the current state s to the adversary and changes it to s'. The flag **compromised** is set to `true`.

- **nextbits**().
  - If **compromised**=`true` or $i > i^*$. B sets $(r, s') \leftarrow G(s)$.
  - If **compromised**=`false` and $i < i^*$. B chooses random strings $r, s' \leftarrow_R \{0,1\}^m$.

– If **compromised**=`false` and $i = i^*$. B uses its input setting $r = r^*$ and $s^{'} = s^*$.

Either way, B updates the internal state s with $s^{'}$ and returns the m-bit string r to the adversary.

Procedure B continues in a similar fashion until A halts and outputs a bit, then B outputs the same but as well. Looking into the advantage procedure B has, consider q+1 experiments $H^{(i)}$, where $i = 0, 1, \ldots q$. In experiment H^i the first i calls of A to its interfaces are processed in a similar manner as B processes queries for the cases where $i < i^*$. The rest are processed the same way B processes cases where $i > i^*$. Let $p^{(i)}$ represent the probability that A outputs a one in experiment $H^{(i)}$.

It should be clear that $p^{(q)} = p_h$ (probability that A outputs a one in **Experiment H**), since B answers all the queries $i > i^*$ exactly the same as in **Experiment H**. The claim is that $p^{(0)} = p_0$. To see this, notice the flag **compromised** is set by B in an identical fashion to how it is set in **Experiment I**, and the queries in A are always answered as in the "'real world"' with PRG when **compromised**=`true` (both in the run of B and in experiment $H^{(0)}$). Moreover, all the queries $i > i^*$ when **compromised**=`false` are answered in B with random independent bit strings, just as they would be in experiment $H^{(0)}$. There is one tricky case however. The **setstate**() query is tricky, but notice that B returns the *current state* s, which is a random bit string that was never used by B in any other query, since $i < i^*$ and **compromised**=`true`.

When B chooses some i^* and its input is chosen at random ($r^*, s^* \leftarrow_R \{0,1\}^m$), then B outputs one with probability $p^{(i^*)}$. Moreover, when B chooses some i^* and its input is chosen as the output of G($(r^*, s^*) \leftarrow G(s)$ for $s \leftarrow_R \{0,1\}^m$), then B outputs with probability $p^{(i^*-1)}$. For the last case we note the following

- If the i^* call of A is **goodrefresh**() then B sets the internal state to $s^{'} \leftarrow G^{'}(s)$ where s is the randomness that was used to generate the input for B, whereas in the experiment $H^{(i^*-1)}$

the new state would be set to $s' \leftarrow G'(s \oplus d)$ with s as the prior state and d being a random choice drawn from $d \leftarrow_R \{0,1\}^m$, so the distribution of s' is the same in both.

- If the i^* call of a is **badrefresh**(x) when **compromised**=`false`, then again B would set the internal state (seed) to $s' \leftarrow G'(s)$ where s is the randomness that was used to generate the input for B. However, in experiment $H^{(i^*-1)}$, the new state would be set to $s' \leftarrow G'(s \oplus Ext(x))$ where s is the previous state. However, since **compromised**=`false` and all the calls upto $i^* - 1$ with **compromised**=`false` were processed with random strings, then in this case the previous state itself is uniform in $\{0,1\}^m$. It is also independent of all other things. So, again the distribution of s' is the same in both.

- If the i^* of a is **nextbits**() with **compromised**=`false`, then B sets $(r, s' \leftarrow G'(s))$ where s is the randomness that was used to generate the input for B. In experiment $H^{(i^*-1)}$ these values are set to $(r, s' \leftarrow G'(s))$ where s is the previous state. Again **compromised**=`false` and all the calls upto $i^* - 1$ were processed with random strings, then in this case the previous state itself is uniform in $\{0,1\}^m$. It is also independent of all other things. So, again the distribution of (r, s') is the same in both.

It follows from this detailed description that the advantage of B is at most $\frac{p_0 - p_h}{q}$. □

4.5 Overview

Figure 13 gives a summary of the motion of information from the entropy sources to the output of the generator. Four, independent, unbiased sources with min-entropy $\delta > \frac{n}{2}$ are necessary for input into the Multiple source extractor. Typical entropy sources in use in currently implemented PRG's were mentioned in Chapter 3.1.3. However, this is not an exclusive group. Any source that can be proven to have high min-entropy can be used as such a source.

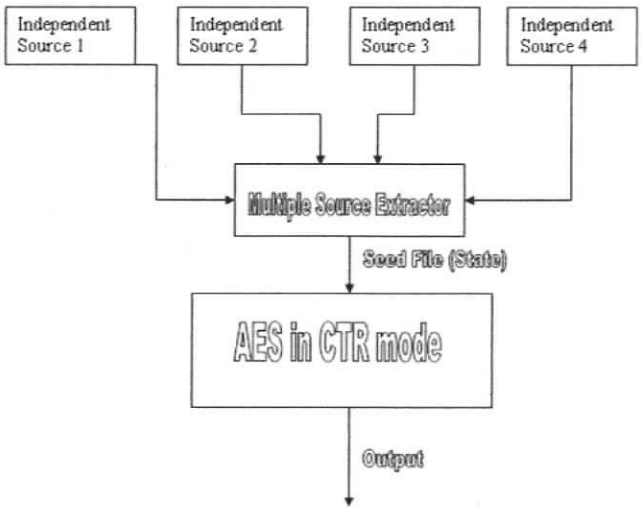

Figure 13: Overview of PRG

From there, the four sources flow entropy into the multiple source extractor. In Figure 13, it is just shown as a black box, however the details of its inner functions were given in detail in Chapter 4.2. The extractor implements an iterated Hadamard-Silvester Matrix (Figure 14). This design is very efficient but at the same time effective. In [22], it was shown that such a extractor when given sources with entropy source with min-entropy $\delta > \frac{n}{2}$, output a distribution that is close to uniform. This close to uniform output is essential for the generator, as the output will be used as the seed file and will draw its entropy for the overall output of the PRG from this seed file.

The final phase of the PRG feeds output from the extractor and uses it as a seed file to the generator. The generator is recommended in [2] and mentioned in Chapter 3.1.1 to be implemented in similar situations, so that is what is used here. However, similar to the extractor, this is not the only option. In Chapter 2.4 many different block cipher modes were introduced for AES. Moreover, other options exist to use as the PRNG. Any would suffice here. The main reason for using AES in CTR mode comes from its use in the Fortuna heuristic(Chapter 3.1). Although none of the literature shows any sort of proof of the ability of AES to provide pseudorandom numbers, its built in qualities described in [1] have lead to its use as a pseudorandom number generators.

One of the main qualities of AES is despite of countless and continued attempts at attacking its security [15], it has not succumb to many of the known cryptographic attacks on its security. This is a crucial fact, as in the case of the PRNG, if its security is compromised and the seed file becomes known, all values outputted by the PRNG could be easily replicated by the adversary until the next reseed. So, for any adversary running in polynomial time, AES can be considered secure and a viable option as a PRNG in such a setting.

5 Simulation and Test Results

The model outlined and described in Chapter 4 was tested using a simulation. All programming was done in the MATLAB programming environment due to the ease with which mathematical operations can be computed within the language. The simulation is by no means an attempt to create a robust pseudorandom generator. The main purpose behind running the simulation was to test the effectiveness of the Multiple source extractor, as well as the output of the Generator after being given a seed file from the extractor. Another point that needed to be addressed was the time efficiency of the multiple source extractor. The entire process of output generation lies directly on the production of a high entropy seed file. Hence, the performance of the multiple source extractor can be the bottleneck between an efficient PRG or an inefficient one.

5.1 Sources

The necessity for sources of entropy for the PRNG is crucial. Moreover, the stipulation that these sources not only have high min-entropy, but it also becomes pertinent that the sources contain independent, unbiased bits as well. Both the Fortuna heuristic and /dev/random look to use system information to generate the entropy required to run their PRNG. This type of high entropy information would be ideal for use in our PRNG as well. However, for use in a simulation, the use of the data used for programs such as /dev/random is hard to acquire.

One might assume that an OpenSource project such as Linux would have its source code available and that its security could be easily analyzed. However, /dev/random is neither well documented nor is there a clear description of the algorithm itself. Moreover, any changes to the program itself to help analyze it means making adjustments to the kernel itself, which leads to a new build and installation. This entire process can take a few hours to complete, and having to do this many times can lead to a very tedious process. On top of this, all calls for entropy are done

within the kernel itself, and extracting this information (thankfully so for those that use Linux for key generation), is a very difficult process to do without damaging the build itself.

In light of this, an alternative means of generating high entropy sources was needed to perform the simulations. These sources needed to fulfill to requirements:

1. **Sources needed to be high min-entropy**. The Hadamard-Sylvester matrix (Equation (2.14)) has a min-entropy requirement that was originally $> (\frac{1}{2} - \epsilon)n$, but was lowered in [SanVaz] to $> \frac{n}{2}$

2. **Sources need to be independent**. The sources used in the extractor need to be independent of each other.

These two requirements were met using the **tcpdump** function available in the Linux environment. The tcpdump function prints out the headers of packets on a network interface. It was an avenue believed to provide the necessary min-entropy requirement stated above, and was proved to do so with our test results which will be discussed later on. Furthermore, collecting raw data from the network on different days and at different time intervals during the day was used as a heuristic method to achieve the independence necessary from the sources. Although it is not recommended without further analysis if the tcpdump function can indeed be used as an entropy source for a implemented PRNG, for the purposes of the simulation, **tcpdump** was able to provide the entropy sources necessary to implement and run the simulation.

For each run of the simulation, four independent sources were created and saved as *.bin* files. Files of size 123 MB were created to ensure the sample size requirements for entropy testing were met. Files were saved was `device1.bin`, `device2.bin`, `device3.bin`, and `device4.bin` for each run of the simulator.

5.2 Extractor Simulation

All simulations were run in the **MATLAB** environment. The extractor was simulated using the *.bin* files as input and creating the seed file used for the state of the block cipher as the output. The design of the extractor is very simple and efficient. Each pair of inputs are pooled for information until 8 bits of data is available. At this time, the inner product of these 8 bits is taken to produce 1 bit of output. From the two Hadamard extractors extracting from the 4 devices (2 sources per extractor), the output of these are again pooled until 8 bits are available. Again, the inner product of these 8 bits is taken to produce 1 bit of output that will be used directly in the seed file to the generator (AES in CTR mode). Once 256 bits of output necessary to seed the generator are extracted, regardless if the generator needs a reseed or not, the generator is reseeded and can await a request for output.

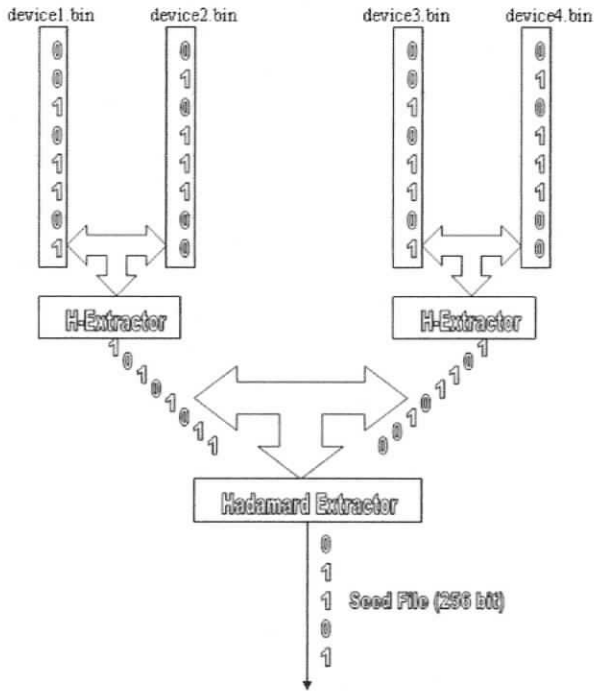

Figure 14: Extractor Simulation Flow Diagram

The choice of using the inner product of 8 bits was arbitrary, but seemed reasonable since the

seed file was a multiple of 8 and literature had suggested it. The downside to such a choice is that to generate 1-bit of output of the extractor, each source has to provide 64 bits of input. Therefore, for creating a 256-bit seed file, 16,384 bits of input are necessary from each source. This may seem expensive and it is, and may become a huge problem if sources, which produce data in real time, need to wait for system events to create entropy. If a source is unable to produce entropy, this type of design may bottleneck until data is available. It is suggested that entropy sources that are to be used with such a system be tested before hand, to try and determine what speed on average entropy becomes available from such sources. From this, a proper inner product bit size can be determined.

5.3 AES Simulation

AES in CTR mode was used as the generator for the simulation. The standard Rijnadael specification was used [1] with the adjustment made from the usually very popular 128-bit version to the 256 bit version used in this implementation. An existing implementation in MATLAB, created by Jorg Buchholz, was used as reference for this simulation. Buchholz version was 128 bit and was not in CTR mode, so modifications needed to be made to adjust for the seed file size (K) of 256 bits and the CTR.

The implementation did not require the decryption process, so it was abstained from being programmed. The program uses the seed file as the Key K and uses the counter as the plaintext P to encrypt. The counter is maintained as an 8 bit number, which is implemented each time the counter is used. So for a plaintext of length 256 bit, the counter will be incremented to represent numbers in the range of the starting block counter value and incremented by one for each corresponding block. The CTR is never reset, and only returns back to old values once it overflows.

Each seed file is permitted to encrypt four plaintext blocks of CTR values. This was also an arbitrary decision, however in a real implementation more plaintext can easily be run through the generator for a given key. The output of the generator is also saved to a *.bin* file which can then be

analyzed and tested if it is needed to.

5.4 PRNG Simulation

Data begins in the *.bin* files that were created using a dump from the function tcpdump within the Linux kernel. These four *.bin* files are fed into the Multiple source extractor as input. The iterated Hadamard-Silvester design of the extractor, take 8 bit blocks of input from the *.bin* files and runs them through the extractor similar to Figure 14. The extractor waits for 256 bits of output before using this output as the seed file to the Generator. The generator, which is AES running in CTR mode, uses this seed file generated by the extractor as the key to the encryption function of AES $C = E(K, P)$. The plaintext P is this case is the CTR values as described earlier. The output of the generator is just the ciphertext C which is also outputted to a *.bin* file for analysis purposes.

5.5 Test Results

The main purpose of the simulation was to create testable output. It is important that before such a PRNG is implemented, that its functionality and output be up to the standards and claims of the theoretical background behind them.

For the input *.bin* files, the output of the extractor, and the output of the generator, three measurements were taken. Namely, the min-entropy (Chapter 2.1.7), Shannon entropy (Chapter 2.1.7) and the statistical distance (Equation (2.12). The experiment was run 3 times in an attempt to ensure that the similar results could be repeated with some consistency. File sizes of the input files were large ($> 100MB$) to ensure a significantly sized output file (of both the extractor and AES) to ensure proper sample sizes for the statistical testing. The following chapter will summarize these results and look into the implications of them.

5.6 Run 1

All measurements for Runs 1,2, and 3 were taken with sample sizes of 20,000 blocks.

Table 3: Min-Entropy of All Data of Run 1

Block Size	device1.bin	device2.bin	device3.bin	device4.bin	Extractoroutput.bin
6	5.6646	5.585	5.5735	5.6646	5.9276
7	6.5064	6.62	6.4422	6.4421	6.8581
8	7.3808	7.3028	7.3413	7.4215	7.7991

Table 4: Statistical Distance from Uniform of All Data of Run 1

Block Size	device1.bin	device2.bin	device3.bin	device4.bin	Extractoroutput.bin
6	0.04445	0.04075	0.03895	0.0459	0.01055
7	0.0473	0.0560	0.6085	0.05865	0.01575
8	0.0773	0.0889	0.07975	0.7670	0.0218

5.7 Run 2

Table 5: Min-Entropy of All Data of Run 2

Block Size	device1.bin	device2.bin	device3.bin	device4.bin	Extractoroutput.bin
6	5.6956	5.672	5.5539	5.786	5.9334
7	6.6023	6.6121	6.3258	6.4438	6.8982
8	7.3912	7.3123	7.3395	7.4364	7.8012

Table 6: Statistical Distance from Uniform of All Data of Run 2

Block Size	device1.bin	device2.bin	device3.bin	device4.bin	Extractoroutput.bin
6	0.0512	0.04134	0.03923	0.0496	0.01121
7	0.0499	0.0572	0.6015	0.05932	0.01465
8	0.0873	0.0893	0.08065	0.07730	0.0192

5.8 Run 3

Table 7: Min-Entropy of All Data of Run 3

Block Size	device1.bin	device2.bin	device3.bin	device4.bin	Extractoroutput.bin
6	5.6723	5.6813	5.5639	5.792	5.9592
7	6.6122	6.6192	6.3238	6.4523	6.9011
8	7.4001	7.3236	7.3401	7.4293	7.8592

Table 8: Statistical Distance from Uniform of All Data of Run 3

Block Size	device1.bin	device2.bin	device3.bin	device4.bin	Extractoroutput.bin
6	0.0527	0.04254	0.03743	0.0501	0.01029
7	0.0501	0.0582	0.6114	0.0606	0.01592
8	0.0863	0.0896	0.08512	0.7840	0.0182

5.9 Input Sources

The input sources were created using the tcpdump function available within the Linux kernel (Chapter 5.1). For each experiment run, 4 input files were created using the tcpdump function.

In an attempt to ensure independence between the four inputs, which is a requirement of the Hadamard-Silverster Extractor. To accommodate this fact, files were saved on different days and at different times during the day and on different machines. With network traffic on large networks as unpredictable as it is, the assumption of independence between the sources is fairly safe.

The main stipulation on the sources of entropy comes directly from the Multiple Source Extractor itself. From Chapter 2.2.3, it was stated that there need to be exactly two sources of min-entropy n/2. Taking into account our design for the multiple source extractor, this would translate to all four input sources having to meet this requirement.

Table 9: Min-Entropy of Run 1

Block Size	device1.bin	device2.bin	device3.bin	device4.bin
6	5.6646	5.585	5.5735	5.6646
7	6.5064	6.62	6.4422	6.4421
8	7.3808	7.3028	7.3413	7.4215

Clearly, all four sources in the different runs meet the entropy requirements for a given block size. From here, this sources are passed through the Multiple source extractor and the output of the extractor itself is analyzed as well.

5.10 Output of the Extractor

The output of the extractor must fill the requirement that are theoretically guaranteed for a multiple source extractor of this kind (Hadamard-Silvester). In words, given input sources that have min-entropy $> \frac{n}{2}$, the output of the Extractor should be close to the Uniform Distribution. The best way to test for this, is to look at the statistical distance (Equation (2.12)) of the output compared to the uniform distribution of the same length. For example, for an n-bit string, there are 2^n possible

values that n-bit string can take on. In the uniform distribution, each of these bit strings should occur with probability $\frac{1}{2^n}$. Here, the goal is to see how far the output of the extractor is statically from the uniform distribution.

Table 10: Statistical Distance: Output of Extractor for Run 1

Block Size	output.bin
6	0.01055
7	0.01575
8	0.02180

Taking into account that the statistical distance is a running total for all n-bit values, the values shown in Table 9 are quite negligible, showing that the output of the extractor is quite close to the uniform distribution. The statistical distance from the uniform distribution of the output compared to the input sources, decreased on average by a factor of 4 (Appendix A). This is to be expected from the theoretical expectations of the Extractor. Of not also is the fact that the min-entropy of the output of the extractor increased considerably from the input sources, which was also expected (Appendix A).

5.11 Output of the Generator

The output of the generator takes the output of the extractor, uses it as a seed file to generate its own output. With the Rinjdael specification, it can be assumed that the generator, when given a seed file of high entropy, will produced output that is pseudorandom. The entropy of the output of the generator will not be as high as the input seed file, nor will its statistical distance be as close to the Uniform distribution. However, it will produce output that when analyzed by an adversary with limited resources, running in polynomial time say, will not be able to tell the difference between

this output and a random string (computationally indistinguishable).

6 Conclusion and Future Work

The use of pseudorandom generators is of major importance in the world of cryptography. All of the cryptographic protocols in commercial use today rely on the generation of random numbers as part of the protocol's security. The use of random numbers in generating passwords and encrypting data is critical to the security of information being transmitted on a daily basis. Of the many PRGs in use today, none have made use of newer developments in the world of extractors. Extractors are algorithms that can transform a weak random distribution to a distribution that is close to uniform. The algorithm in use need not be cryptographic at all, but the result needs to guarantee output that is close to random.

Older PRGs such as /dev/random and the Fortuna heuristic look to use currently available entropy sources within a certain system to use to create the necessary randomness. Using a similar foundation, our model looks to use entropy sources within a given system, an extractor to "'extract"' the randomness, and a block cipher (AES) to generate the pseudorandom output.

The future of such a project is promising. With the availability of the open source Linux kernel and its entropy sources, the next step would be the actual implementation beyond the scope of a simulation of this model in the kernel itself. This would be a very intricate undertaking, however could prove to be a more effective PRG than the currently existing /dev/random. Any system with the availability of some entropy sources, can be used to program such a model for pseudorandom output.

The future work in extractors will also lead to a better model. The currently implemented Hadamard-Silvester extractor is simple but very time efficient, however somewhat data inefficient. Future work in this area may lead to a better data efficient extractor. With our model created the way it is, pieces of the model can be cut out and replaced with something more efficient or promising when it comes along.

References

[1] National Institute of Standards and Technology (NIST). *Advanced Encryption Standard.* Available on: `http://csrc.nist.gov/CryptoToolkit/aes/`.

[2] B. Barak, and H. Shai, *A model and architecture for pseudo-random generation with applications to /dev/random.*, ACM Conference on Computer and Communications Security, 2005, 203-212.

[3] B. Barak, R. Impagliazzo, and A. Wigderson, *Extracting Randomness Using Few Independent Sources*, FOCS, 384-393, 2005.

[4] B. Barak, G. Kindler, et. al., *Simulating independence: new constructions of condensers, ramsey graphs, dispersers, and extractors.*, STOC 2005, 1-10.

[5] J. Buchholz, *MATLAB Implementation of the Advanced Encryption Standard.*, http://bucholz.hs-bremen.de, December 2001.

[6] B. Chor and O. Goldreich, *Unbiased Bits From Sources of Weak Randomness and Probabilistic Communication Complexity*, May 1987.

[7] Y. Dodis, et al. *Randomness Extraction and Key Derivation Using the CBC, Cascade and HMAC Modes.*, CRYPTO 2004, 494-510.

[8] Y. Dodis, et. al. , *On the (Im)possibility of Cryptography with Imperfect Randomness.*, FOCS, 196-205, 2004.

[9] N.Ferguson and B. Schneier, *Practical Cryptography*. John Wiley and Sonds, 2003.

[10] O. Goldreich, On the Foundations of Modern Cryptography, CRYPTO, 1997, 46-74.

[11] Gutterman, Zvi et al., *Analysis of the Linux Random Number Generator*, March 2006.

[12] S. Hardy, *The e2tandom Entropy Harvester and PRNG for Linux*, Tsumego Foundation Research, 2004.

[13] Johan Hstad, Russell Impagliazzo, Leonid A. Levin, Michael Luby, *A Pseudorandom Generator from any One-way Function.* SIAM J. Comput. 28(4): 1364-1396 (1999).

[14] J. Kelsey, B. Schneier, and N. Ferguson. *Yarrow-160: Notes of the design and analysis of the yarrow cryptographic pseudorandom number generator.* In Selected Areas in Cryptography, LNCS vol. 1758, 13-33, Springer, 1999.

[15] J. Kelsey, B. Schneier, D. Wagner and Chris Hall, *Cryptanalytic attacks on pseudorandom number generators,* Fast Software Encryption, LNCS 1372, S. Vaudenay, Ed., Springer-Verlag, 1998, pp. 168-188.

[16] M. Luby, *Pseudorandomness and Cryptographic Applications*, Princeton Computer Science Notes, Princeton University Press, 70-105, 1996.

[17] U. Maurer, *A Universal Statistical Test for Random Bit Generators*, Journal of Cryptology, Vol. 5, No. 2, 1992, pp. 89-105.

[18] A. Menezes, et. al., *Handbook of Applied Cryptography*, CRC Press, 1997.

[19] A. Rukhin, et al. *A STATISTICAL TEST SUITE FOR RANDOM AND PSEUDORANDOM NUMBER GENERATORS FOR CRYPTOGRAPHIC APPLICATIIONS*, NIST Special Publication 800-22, 2004, 45-50.

[20] N. Nisan and T. Ta-Shma, *Extracting Randomness: A Survey and new constructions.* Journal of Computer and System Sciences, 58(1):148-179, 1999.

[21] N. Nisan and D. Zuckerman. *Randomness is Linear in Space*, J. Comput. Syst. Sci., Vol. 52(1), pp. 43-52, 1996.

[22] M. Santha and U. V. Vazirani, *Generating quasi-random sequences from semi-random sources*, In J. of Computer and System Sciences, 63:612–626, 1986.

[23] R. Shaltiel, *Recent Developments in explicit constructions of extractors*, Weizmann Inistitue of Science, 2003.

[24] C.E. Shannon. *Communication theory of secrecy systems*. Bell System Technical Journal, 28(4), October 1949.

[25] Luca Trevisan, *Constructions of Near Optimal Extractors Using Pseudo-Random Generators*, Electronic Colloquium on Computational Complexity, Report No. 55 (1998).

[26] T. Ts'o. `random.c` - Linux kernel random number generator. `htpp://www.kernel.org`.

[27] A. Wigderson, and D. Zuckerman, *Expanders that beat the eigenvalue bound: explicit construction and applications*, Combinatorica, 19 (1999) 125–138.

APPENDIX

The test itself can be defined by three positive integer valued parameters L, Q, and K. The total length of the test sequence s_N is N=(Q+K)L, where Q is the number of initialization blocks and K is the number of test blocks. Choosing of Q and K will be discussed later on.

The first step to perform the test is to specify the block size L, and split the output of the generator (test sequence) into non-overlapping blocks of length L. After the blocks are partitioned, the entire segment of blocks N should be partitioned into two segments: the initialization segment (Q L-bit blocks) and the test segment (K L-bit blocks). If there are bits remaining at the end of the sequence that have not been partitioned or do not complete an L-bit block, they can be ignored.

To initialize the test, the first Q L-bit blocks are examined. A table is created for each possible L-bit value. The L-bit value is used as an index to the table. For example, if L=5, there are $2^5 = 32$ possible L-bit value $(0 \ldots 31)$, hence a table with 32 columns would be created. The block number of the last occurrence of each L-bit block is updated in the table as each of the $(1 \ldots Q)$ blocks are examined.

Next, each of the remaining K blocks in the test sequence are examined to determine the number of blocks since the last occurrence of the same L-bit block. In the table that was created in the previous step, replace the value of the previous occurrence of the L-bit block with the current one. Finally, add the calculated distance between the re-occurrences of the L-bit block to an log2 accumulating sum off all the differences detected in the K blocks.

The final step involves computing the test statistic and ensuring that it lies between the threshold values for a given value of L. The test statistics is given by

$$f_n = \frac{1}{K} \sum_{i=Q+1}^{Q+K} log_2 A_n (s_N) = \frac{1}{K} \sum_{i=Q+1}^{Q+K} log_2 i - T_j \qquad \text{(A-1)}$$

where T_j is the table entry that corresponds to a re-occurrence of the i^{th} L-bit block given in decimal representation. In other words, take the accumulating sum from the previous step and divide the sum by the total number of K L-bit blocks examined.

Now that the test statistics has been calculated, the distribution of the test statistic for a truly random sequence needs to be known in order to determine whether the test statistics for a given test sequence be accepted or rejected. For a given random sequence of bits RN, the mean and variance of a single term $log_2 A_n (R_N)$ of the sum defining $f_N (R_N)$ can be computed as $Q \rightarrow \infty$. Because the expected value of the average of several random variables is equivalent to the average of the expected values,

$$E[f_N (R_N)] = E[log_2 A_n (R_N)] \tag{A-2}$$

The variance of the sum of statistically independent variables is equal to the sum of the variances. However, in this case, the quantities defined by $A_n (R_N)$ are not completely independent. As a consequence of this dependency, the quantity of the variance of $f_n (R_N)$ is a little smaller than expected. Let c(L,K) denote the amount by which the standard deviation of $f_n (R_N)$ is reduced compared to what it would have been if $A_n (R_N)$ were completely independent. Then,

$$Var[f_n (R_N)] = c(L, K)^2 \bullet \frac{Var[log_2 A_n (R_N)]}{K} \tag{A-3}$$

If we choose $L \geq 3$, $c\left(L, 2^L\right)$ becomes very close to 0.8. . Similarly, taking K¿¿2^L gives 0.5, 0.6, and 0.65 for L=4, L=8, and L=12 respectively. Using elaborate simulations, it has been found that for $K \geq 2^L$,

$$c(L,K) \approx 0.7 - \frac{0.8}{L} + \frac{(4+32/L)\,K^{-3/L}}{15} \qquad \text{(A-4)}$$

makes for a good approximation equation for c(L,K).

To implement the test, Maurer recommends one chooses the parameter L somewhere in the range of $6 \leq L \leq 16$. Q should be chosen with a lower threshold of $Q \geq 10 \bullet 2^L$ and K with a lower threshold of $K \geq 1000 \bullet 2^L$. The reason for the high block total for Q is so that with high probability it can be guaranteed that every L-bit pattern occurs at least once in the initialization sequence.

Once an L value is chosen, the $E[f_N(R_N)]$ and $Var[log_2 A_n(R_N)]$ are given in the following table for $6 \leq L \leq 16$.

Table 11: Variance and Expected Values for Mauer's Test

L	$E[f_N(R_N)]$	$Var[log_2 A_n(R_N)]$	L	$E[f_N(R_N)]$	$Var[log_2 A_n(R_N)]$
1	0.7326495	0.690	9	8.1764248	3.311
2	1.5374383	1.338	10	9.1723243	3.356
3	2.4016068	1.901	11	10.170032	3.384
4	3.3112247	2.358	12	11.168765	3.401
5	4.2534266	2.705	13	12.168070	3.410
6	5.2177052	2.954	14	13.167693	3.416
7	6.1962507	3.125	15	14.167488	3.419
8	7.1836656	3.238	16	15.167379	3.421

Once the value f_n is calculated for the sequence being tested, it is compared to threshold values

t_1 and t_2. A test sequence should be rejected iff either

$$f_n(s_N) < t_1 \quad or \quad f_n(s_N) > t_2$$

The thresholds t_1 and t_2 adhere to the following equations

$$t_1 = E[f_n(R_N)] - y\sigma \quad and \quad t_2 = E[f_n(R_N)] + y\sigma$$

where the standard deviation σ is given by

$$\sigma = c(L,K)\sqrt{Var[log_2 A_n(R_N)]/K} \tag{A-5}$$

The value y corresponds to the number of standard deviations $f_n(s_N)$ is allowed to be away from the mean value. This value must be chosen such that $N(-y) = \frac{\rho}{2}$. $N(x)$ is the integral of the normal density function and is defined as

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\pi} e^{-\xi^2/2d\xi} \tag{A-6}$$

A table of $N(x)$ can be found in any statistical text book. For example, to obtain a rejection rate of α=0.01 or 0.05, the corresponding y values would be y=2.58 and y=3.422 respectively.

6.1 Choosing the Input Sizes

The test requires a long sequence of bits ($n \geq (Q+K)L$). The sequence is divided into two segments of L-bit blocks, the first being the initialization sequence of Q blocks and the latter being the test sequence of K blocks. According to Maurer, the block size L should be chosen so that

$6 \leq L \leq 16$. As stated earlier, the initialization segment Q should be chosen so that $Q \geq 10 \bullet 2^L$ the test segment of K blocks should be so that $K \geq 1000 \bullet 2^L$. A summary of given input sizes for different L values is given below.

Table 12: Input Sizes for Maurer's Test

L	$n \geq$	Total Blocks	Q	min K
6	387840	64640	640	64000
7	904960	129280	1280	128000
8	2068480	258560	2560	256000
9	4654080	517120	5120	512000
10	10342400	1034240	10240	1024000
11	22753280	2068480	20480	2048000
12	49643520	4136960	40960	4096000
13	107560960	8273920	81920	8192000
14	231669760	16547840	163840	16384000
15	496435200	33095680	327680	32768000
16	1059061760	66191360	655360	65536000