CRYPTOGRAPHICALLY SECURE RANDOM NUMBER GENERATORS

BRIAN STACK

Abstract. Random numbers and random primes play a role in many parts of cryptography. This makes finding these numbers of great importance. Herein we'll examine different techniques for doing just this, and see how they fit into the larger world of cryptography. This paper is primarily concerned with the mathematics of these systems, but to give the math a little motivation, we'll discuss engineering challenges inherent in creating real world PRNG systems.

Contents

1. Introduction

There is a story one of my professors tells that shows how most computer scientists feel about "random" numbers. The story is about a professor who goes to his colleague looking to get some random numbers in order to run a simulation. These were the days before widespread networking, and so he leaves a storage disk at the colleague's office. A short while later the disk is returned and the eager professor pops it into his machine anticipating a successful simulation with his new found entropy. However upon examining the contents of the disk, he finds to his dismay that it has been filled entirely with repeating 0's! How could this be?

It is notoriously difficult to define what randomness is in a rigorous way that accounts for all of the properties of the intuitive definition, which is simply that the sequence cannot be predicted. In section 3.5 we'll define randomness for our domain, but for now just work with it intuitively.

Generating random numbers is of great importance to many different algorithms, especially within cryptography. Unfortunately we cannot, in general, just ask the

Date: April 30, 2012.

This paper is for MATH408 Introduction to Cryptology with David Singer at Case Western Reserve University.

$$\begin{pmatrix} S_5 \\ S_4 \\ S_3 \\ S_2 \\ S_1 \end{pmatrix} = \begin{pmatrix} c_4 & c_3 & c_2 & c_1 & c_0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{pmatrix}$$

Figure 1. Matrix representation of LFSR

universe for a random sequence of bits[1], so we must come up with a way of making numbers that at least appear random. Since ancient times humans have created methods of finding random numbers such as rolling dice and flipping coins, but these are not sufficient for today's needs.

Therefore we'll concern ourselves mostly with pseudorandom number generators (PRNGs) herein. These are functions which have output that should be indistinguishable from a true random process [5]. In section 4 we'll discuss ways to test how good these generators are and find what makes certain PRNGs cryptographically secure. For now let's look at some historical and current PRNGs to see how they have evolved over time. A reader that is not very familiar with the concepts behind PRNGs may want to skip ahead to section 3 before coming back to section 1.1.

1.1. **Historical PRNGs.**

1.1.1. *Linear feedback shift register.* A rarity in this paper, an LFSR is a hardware PRNG, a diagram of an LFSR is in Figure 2. A matrix representation of an LFSR is shown in Figure 1. In this section, we'll build up from a polynomial to the matrix representation, then show how the matrix representation is related to the diagram. Ultimately, LFSRs can be represented by polynomials. As an example we'll work in

$$\mathbb{F}_{32} = \mathbb{Z}_2[x]/x^5 + x^2 + 1$$

Any element in that field is representable by a polynomial[2]

$$w = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \tag{1}$$

Where $a_i \in \mathbb{F}_2$ which can be represented by the row vector

$$[a_4\ a_3\ a_2\ a_1\ a_0]$$

Given this representation, how can we multiply w by an x? The solution relies on us being able to convert x^5 into elements of the field. So we can work as follows

$$\begin{aligned} xw &= a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 + a_0x \\ &= a_3x^4 + a_2x^3 + (a_1 + 1_4)x^2 + a_0x + a_4 \end{aligned}$$

[1]Such as radioactive decay or atmospheric noise. The former is a canonical example of a truly random natural process and the latter is used by random.org to generate their random numbers. In a standard personal computer, entropy is found by measuring drive seek timings, user interactions, network events, and uninitialized memory.

[2]Yes, this is directly from the notes I took in class.

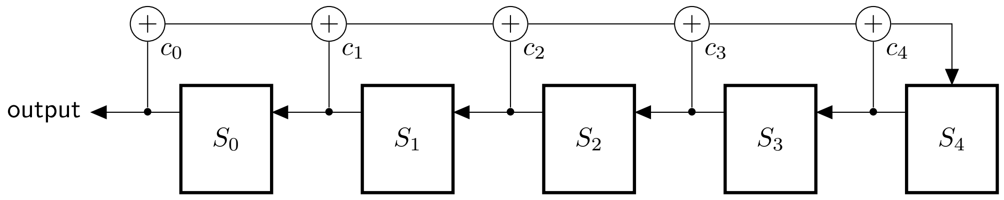

FIGURE 2. A 4-bit linear feedback shift register.

In a matrix and vector representation this is

$$[a_4\ a_3\ a_2\ a_1\ a_0]\begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

If we perform this operation repeatedly, we arrive at Figure 1. From there we can see that each state variable is simply a state in the machine described in Figure 2 and each tap off of the outputs is a weight c_i.

Linear Feedback Shift Registers have found uses in many places over the years, especially military radio communications, but unfortunately, are relatively easy to crack owing to their linear nature. The real benefit is in terms of speed because it can all be implemented in simple hardware. One way to make LFSR viable for use today is by combining the output in a non-linear fashion, making cryptanalysis much more difficult.

1.1.2. *Middle-Square Method.* This method is attributed to John von Neumann, who first published a paper on it the early 1950's. As with all random number generators, we start with a seed. In this case, if we want to generate numbers of k digits, then we choose a seed of $k/2$ digits length. We'll square the seed and emit the result as the next number in our random sequence, and then take the middle k digits and use them as the next seed. The following is an example with $k = 6$.

$461,996$	Seed
$213,\mathbf{440},\mathbf{304},016$	Emit this as a random value
$440,304$	Becomes the next seed

This was a brave attempt at the creation of randomness, but in ultimately is a fairly weak method. The problem is that we enter small loops rather quickly where the same few numbers are generated over and over again [8].

1.1.3. *RANDU.*

...its very name RANDU is enough to bring dismay into the eyes
and stomachs of many computer scientists!
– Donald Knuth, *The Art of Computer Programming*

RANDU is a famous example of a failed random generator. It is a linear congruential PRNG, meaning it is defined by the relation

$$X_{n+1} \equiv (aX_n + c) \pmod{m} \tag{2}$$

Where a, c, and X_0 are all integers bounded between 0 and m inclusive. RANDU is an instance of an LCG where

$$\begin{aligned} a &= 65539 \\ c &= 0 \\ m &= 2^{31} \end{aligned}$$

These numbers were chosen because they were easy to implement and fast to run. These qualities were of more importance when RANDU was popular on the old IBM mainframes such as the 360 which even predated the computer to which Knuth dedicated the book being used for much of the research of this paper – the IBM650.

A visual inspection of the output of RANDU would not really be able to detect the flaw in the output. The best way to detect the flaw is described in section 4.3 in more detail, but the general idea is that there is far too much correlation between subsequent points.

1.2. **Modern PRNGs.**

1.2.1. *Blum Blum Shub.* This is a PRNG that actually happens to be cryptographically secure. The generator looks like

$$x_{n+1} = x_n^2 \pmod{m}$$

Where m is the product of two large primes. It is only really useful for security applications because the generation is actually very slow. This is especially true when compared to a system like LCGs which are generally terrible at security, but can generate lots of numbers *very* quickly. So long as integer factorization is difficult, this system is cryptographically secure.

1.2.2. *Mersenne Twister.* This is probably the most modern PRNG presented in this paper. Only first published in 1997, it is a very high quality generator. It is important to note, that although it has a long period before repeating itself, it is *not* cryptographically secure [9]. Interestingly, the algorithm itself is a riff on the LFSRs presented earlier.

The actual math behind the algorithm is rather complex[3], but after all of it, the actual code to implement a Mersenne Twister is only about 50 lines.

1.2.3. *Linear Congruential Generator.* This system was already described in section 1.1.3, however it is a good idea to note that many modern systems rely on LCGs for their randomness. Many programming language standard libraries such as Java, C, C++, and Forth all use LCGs because they are easy to implement and fast. If a programmer needs a more specific requirement, they are free to find another PRNG which will work for them.

2. Uses of random numbers

The uses of random numbers are many and extend beyond cryptography. In this section we'll look at some of these cases, both for general random numbers and for random prime numbers.

[3]This is actually one of the main complaints about the algorithm.

2.1. **Random numbers in general.** Non-cryptographically-secure random numbers have many applications. Much of this section is an expansion of a similar list from Knuth in the introduction to [8].

2.1.1. *Simulation.* This situation arises any time a computer program is written to model a real world event. Knuth uses the examples of nuclear physics and the arrivals of passengers at an airport as random events that a researcher may want to model. Once a statistical distribution is empirically found (or a sufficiently informed guess is made), a random number generator can be used to provide input to a simulation. Most programming languages provide a standard way to get pseudorandom numbers[4], and complex statistical software such as R offer PRNGs that can match different statistical distributions.

2.1.2. *Sampling.* This is a very appropriate use case considering that we'll be electing a President of the United States this year. Anyone who pays attention to the news will doubtlessly hear of survey upon survey predicting one or another candidate's success in primaries and general elections. In order for these surveys to be accurate, they must find a way to select a representative sample from the population. The easiest way to avoid bias is a process called random-digit dialing, where numbers are generated by an RNG, and *not* taken out of a phonebook at random. This avoids calling only those who have their phone numbers published [1].

2.1.3. *Numerical Analysis.* This is one of the more fun uses of random numbers. A famous example of this is the Monte-Carlo method that uses random numbers in order to form approximations. Monte-Carlo was originally developed by John von Neumann while working on the Manhattan project to simulate nuclear physics. A more simple use case is using it to compute π, the ratio of the circumference of a circle to its diameter[5].

The intuition is that inscribing a circle in a square should give us some way to find geometric facts about the circle, but then the question becomes, how do we find the ratio of surface areas *without* knowing π. The answer comes in the form of random numbers! Let's define a circle and a square shown in Figure 3 as

$$A = (2r)^2 = 4r^2$$
$$C = \pi r^2$$

Where A is the area of the square and C is the area of the circle. If we select many uniformly distributed points in the square, the ratio of points that are contained in the circle, to those that lie outside the circle is

$$\frac{\text{INSIDE}}{\text{TOTAL}} = \frac{C}{A} = \frac{\pi r^2}{4r^2}$$

So we see that

$$\pi = \frac{4 \times \text{INSIDE}}{\text{TOTAL}} \tag{3}$$

Using (3) and generating enough random coordinates we can arrive at remarkably accurate approximations. Table 1 shows how the approximation converges as the number of points increases.

[4]For some strange reason even LaTeXwill generate random numbers, like this one: 716843448

[5]Really we should use τ, the ratio of circumference to radius, but that's a topic for another time

Number of Points	**Approximation of π**
10	3.2
100	3.24
1000	3.18
10000	3.1492
100000	3.1502
1000000	3.139552
10000000	3.1422364

TABLE 1. Convergence of approximations of π using Monte-Carlo methods.

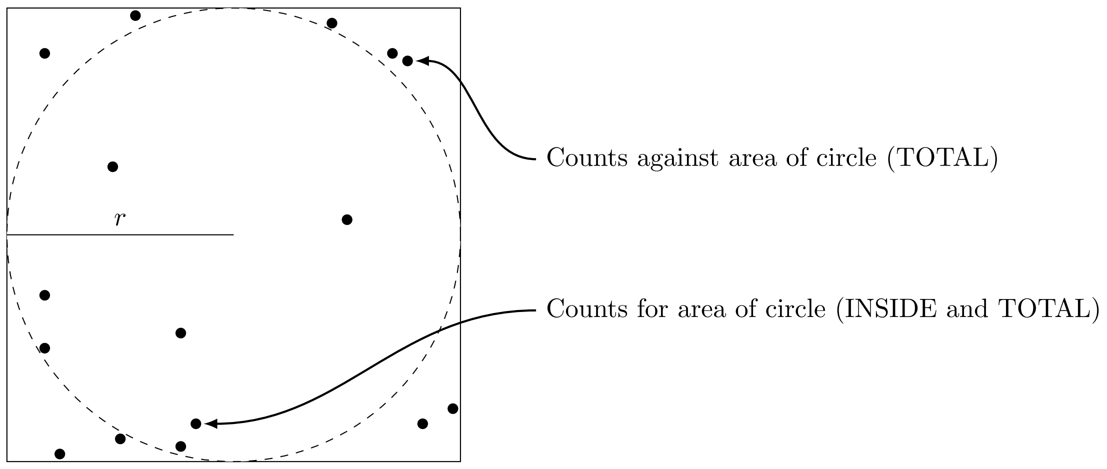
Counts against area of circle (TOTAL)
r
Counts for area of circle (INSIDE and TOTAL)

FIGURE 3. The circle inscribed in a square for the Monte-Carlo π approximation.

2.1.4. *Computer Programming.* This is Knuth's favorite use for random data. The idea here is to test computer programs with random input data. Any undergraduate computer science student will be intimately familiar with QUICKSORT, an asymptotically optimal comparison based sorting algorithm[6]. That undergraduate student will be able to tell you that the ordering of the input is of major importance to the number of steps the standard algorithm must complete in order to terminate. A randomly sorted input will be sorted in order in $O(n \log n)$ time, but a worst case yields $O(n^2)$ time.

This makes the need for randomized data apparent, but we in no way need true random data, just some numbers that can't be distinguished from one, making the testing of QUICKSORT implementations a ripe opportunity for pseudorandom number generators.

2.1.5. *Decision Making.* This also applies to QUICKSORT. An implementation called randomized QUICKSORT can give better chances of attaining that average case runtime of $O(n \log n)$, PRNGs are perfect for this purpose. The idea behind randomized QUICKSORT is that we can try to avoid the worst case conditions from the prior use case of random numbers. The worst case performance arises when

[6]Look to Appendix A for a brief overview of the algorithm.

the input array is already pre-sorted. A standard QUICKSORT splits the array on a value from a set point, for example the first value in the subarray.

2.1.6. *Recreation.* The most obvious use case for random numbers is in gambling. All kinds of electronic machines need random numbers to determine whether or not a player has won[7] the game. Clearly casinos want to avoid players being able to know what numbers will be coming up next, but generating that many true random numbers is infeasible. These machines use PRNGs for this purpose [4]. In that paper, the authors mention that CSPRNGs are recommended for gambling applications, and even analyze the use of chaotic systems such as the Lorenz attractor in 3 dimensions for producing cryptographically secure random numbers.

2.2. **Uses for cryptographically secure PRNG.**

The prior section discusses many uses for random numbers, but those problems in general don't require any sort of guarantee of cryptographic security. What follows are some applications of CSPRNGs. These generators will be discussed in an abstract sense in section 5.

2.2.1. *Nonces and Salts.* Nonces seem to have many and varied uses across all sorts of applications. This includes Bitcoin[8] and in general anything where a document is to be signed. Another example of this is making SSL secure connections unique [7].

Salts have been pushed to prominence recently by a number of database breaches at major organizations. Oftentimes these breaches will allow attackers to access important and secret data of the customers or members of those organizations. Even if this data is encrypted properly[9], it can sometimes be retrieved by a clever construction called a "rainbow table", salts can help prevent this sort of attack.

Rainbow tables are an advance upon techniques developed by Martin Hellman and later by Ron Rivest. Those two names are just about as big as they get in the world of cryptography, showing just how impressive rainbow tables really are. Rainbow tables were developed by Philippe Oechslin around 2003 in [10] around 2003. In that article, Oechslin gave the following measure of rainbow table's power:

> *Using 1.4GB of data (two CD-ROMs) we can crack 99.9% of all alphanumerical [Windows] passwords hashes (2^{37}) in 13.6 seconds.*

Rainbow tables exploit a time-memory trade-off, where much of the work is done ahead of time and stored for faster lookup during the actual cracking. This can allow many passwords to be cracked with a feasible amount of work. Ideally we could store the hashed values of all possible passwords withing a certain range. Unfortunately this would use up a prohibitive amount of space for any set of reasonably sized passwords. This led Rivest to propose the idea of distinguished points, an example of which would be "the first 10 bits of a key are zero". These distinguished points could be chained together in such a way that successive calculations on keys can be made until a distinguished point is found. The contribution of rainbow tables, and the reason that they have a performance speedup over Rivest's technique, is the avoidance of collisions in these calculations using a special data structure.

[7]Generally losing, of course.

[8]Read Tom Dooner's paper for a description of this.

[9]A naive way to implement passwords would be to store the plaintext of the password that is needed for authentication. This is not a good idea however, because this allows whoever has access to the database to read any password they want, and use it to improperly authenticate. A better method is to securely hash the password when it is created and each time it is entered, and compare the hashes of the password.

The Zappos Incident

In January 2012, Zappos.com, an online shoe marketplace, was attacked and its databases were accessed. This breach resulted in the accounts of 24,000,000 users being accessed. This included things like names, email addresses, and parts of credit card numbers, in addition to encrypted passwords. Zappos asked its users to change their passwords not only for their own site, but also for any other websites they may have used the same password for. Many in the web developer community felt that this indicated that Zappos had not been properly salting their hashed passwords. Simply doing this could have made all of their user's passwords safe, something that was not true for all but the most securely chosen passwords.

Similar attacks have occurred on other large databases. Much of the time, the passwords are not salted properly. It would be nice if we could blame Sony for not salting the passwords of their Playstation network users, except they didn't even hash the passwords to begin with! This resulted in 77 million user accounts being completely compromised when the network was hacked in early 2011.

So far as performance is concerned, Oechslin provides a few other metrics on top of the impressive time to crack the notoriously insecure LM hash[10] used in older versions of Windows. The results in [10] show that rainbow tables solve the exact same percentage of problems that Rivest's technique did (100%), in addition to being over 7 times faster.

The reason for this foray into rainbow tables, which the reader may have noticed are not random number generators, is to justify the need for adding salts to hashed passwords. Very simply put, salts are random data appended to human-generated passwords that prevent the use of lookup tables like those of the rainbow variety from being effective. The growth in size and randomness of the password makes it impractical to store all of the precomputed values necessary to create the rainbow chains. Salts are an excellent use case for randomness, and also do not in any way require true random numbers, making PRNGs perfect for this task.

2.3. **Uses for random primes.** Sometimes we need additional constraints on the random numbers we're generating. One case is that we require the numbers to be prime. This has obvious use cases in cryptography, and requires a very different technique of generation, discussed further in section 6.

2.3.1. *Key Generation.* Just a cursory check through the class text shows that RSA, ElGamal, and Diffie-Hellman all need random numbers in order to maintain secrecy [5]. Specifically, RSA and Diffie-Hellman need random primes, while Elgamal also needs uniform random numbers for its ephemeral key.

2.3.2. *Random Number Generation.* Oddly enough, we need large primes in order to use Blum-Blum-Shub, meaning that generating random primes is necessary for certain algorithms which generate uniform random numbers. Strange, huh?

[10]The primary problem with this system was that passwords of length greater than 7 characters were split into two 7 character halves. This made the problem of cracking them using techniques such as rainbow tables exponentially easier.

Class	Description	Example
character	Basic building block of alphabets	a,b,1,0
Σ	Alphabet: set containing all characters	$\{a, b\}$, $\{1, 0\}$
$w \in \Sigma^\star$	String in the language	$aaaabba, 011011001$
L	Language: set of strings	$a^\star b, (0 + 1)^\star$
$\mathcal{L}$	Language class contains languages	regular, context free

Table 2. Class hierarchy of languages.

3. Mathematical Background

In the following section we'll go over some of the background knowledge the reader will need to understand the rest of the paper. We'll spend most of our time defining randomness and explaining what these generators are, but first we'll give a quick overview of complexity theory. Complexity theory is important for understanding both our modern view of what randomness is and understanding how to evaluate the generators.

3.1. Theory of computation.

In order to prepare for the rest of this section, we'll have a quick introduction to computational complexity. We'll start from the simplest building blocks and build to Turing machines, which sit on the edge of the possible limit of computation. Each of the following subsections deals with a *model* of computation, and each one is more powerful than the one before it in terms of its expressive capabilities. The expressiveness of a language is, in essence, how complex of a string can be included in a language, something which we'll touch on more when discussing what randomness really is. Briefly, we'll define some terms in Table 2. The system in the table, in addition to the rest of this discussion of complexity are using the notation from [11]. The $\star$ operator means that the previous character is repeated 0 or more times, the $+$ is OR, and characters are concatenated by placing them next to each other. The role of languages, and the machines that will be introduced soon is to accept or reject strings from the language based on rules such as the ones in the table.

3.2. Deterministic finite automata.

An automata is a description of a way to accept and reject strings from a language. Each of the other machines we'll discuss here build from a DFA at their heart. Intuitively, a DFA is a collection of states, in addition to transitions between states. A simple DFA can be used to model a light in a room controlled by a switch.

Of course, we'll need 2 states, one for the light being on, and another for the light being off. The transitions between these states are the switch being flipped. The automata will look something like Figure 4.

One could imagine that we want to know whether or not the light is on after a series of switch flippings. Say we start in the off (q_0) state, and every 10 minutes, we check the state of the switch. Next to the "on" and "off" annotations in the figure are 1's and 0's with which we could encode our measurements of the switch's state. After a few hours, we may end up with an encoding that looks like 00001101100011.

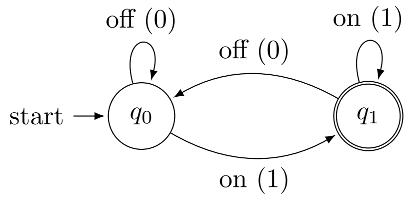

FIGURE 4. A deterministic finite automata modelling a light with a switch.

This would mean that as of the last measurement, the light is turned on. If we decide that we want the light to be on, we can call q_1 the accepting state, and say that any binary string which finishes with a 1 is accepted by the machine.

Formally, we define DFAs by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the alphabet, δ is transition function, q_0 is the initial state, and F is the *set* of accepting states. In the previous example, Q is the state of the light, Σ is the flipping of the switch that we've encoded in 1's and 0's, and δ is what happens when we see the switch is on and was formerly off.

DFAs can describe many strings, however they are not capable of describing all possible strings we may want to come up with. For instance, a string that repeats itself backwards is not expressible in this language class.

3.3. **Pushdown automata.** To give more power to our machines, we need to add some form of "memory". In pushdown automata, this is achieved through the use of a stack. A stack, as its name would imply, is basically like a pile of papers where only the top of the pile can be read from and written on. More papers can be added and removed, but at any given time, only the top piece of paper can be interacted with.

Formally, we define PDAs by a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$. Notice that this is identical to the DFA definition, except with the addition of Γ, which is the stack alphabet. This is the set of characters which can be written onto those pieces of paper, even though they may not be in the input being read.

Given this memory, it's fairly easy to come up with a way to accept a string that is made up of 2 parts, where the second part is the reverse of the first part. Being able to solve problems like this gives PDAs much more expressive power than simple DFAs. Still though, we can find a way to acquire more power. In the case of what we describe next, it is the most power possible!

3.4. **Turing machines.** The stack was a nice idea, but we're going to scrap it completely in favor of something more powerful. Rather than the stack, a Turing machine has an infinite tape. Just as before, we can read and write from this tape, but now we can move left and right over the tape, meaning that we aren't restricted to reading or writing from a single entry. This means that there is now a read-write head that follows the tape (Isn't this starting to feel a lot like a real computer?) Removing this restriction gives us a very powerful machine indeed; in fact, it is provably as powerful as any machine we can construct.

Again, we'll define the machine formally and again we'll add one more symbol to make a 7-tuple. The end result is $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. The symbols that are the same as before are identical, except we've removed the set of accepting nodes and replaced them with a single accept state, in addition we've added a reject state. In this case, we can define

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

Meaning that for every state and every tape symbol, we transition to a state, write a symbol, and move left or right. This is the machine we'll use in the next section to define Kolmogorov and more modern randomness.

3.5. **What is random?** There are three main formal frameworks in which to define "randomness". The first is a familiar statistic for anyone who studies cryptography - Shannon's entropy - specifically, the fact that anything random should have a uniform distribution. This is a fairly limiting view of randomness, and so we try again to come up with a definition.

This time we try an approach from Andrey Nikolaevich Kolmogorov, a Soviet mathematician from the 1900's. He saw randomness as a lack of structure [3] and therefore a measure of randomness was the length of a computer program (we can model this with the Turing machine from the prior subsection) that would be needed to generate an object with a certain structure. One can see the intuition here plainly: a simple repeating pattern can easily be generated by repeating the same commands over and over again in a loop, whereas something that is completely random would need each part of the output to be hard coded in to the program. The problem with this approach is that it is difficult to use this measure of randomness to construct random generators [3].

The definition we'll use in this paper and what seems to be the most commonly accepted measure is a small step beyond Kolmogorov's definition. It now views randomness as a function of the observer's abilities. Rather than measure randomness by the machine needed to construct it, we'll figure out how complex it is to figure out. In [3], the author describes a thought experiment to show why this measurement method is reasonable.

The scenario described in the experiment is that Alice is flipping a coin and Bob is observing. The goal is for Bob to decide, before the coin hits the ground, whether the coin will land on heads or tails. The only difference each time is how much auxiliary knowledge Bob knows (or can figure out) each time. The essence of the scenario is this

> Whether Bob has to guess the outcome before Alice flips, or while he can watch the coin flipping, he has the same 50-50 chance of making the correct guess.

This is important because even though he can examine every part of the coins flip, he can't know anything more than that it is random because his brain is not powerful enough to calculate all of the physics before the coin falls. The scenario ends with Bob having access to a computer which can watch the coin fast enough, and perform enough calculations in realtime, that he actually has a better chance of guessing the coin correctly.

This thought experiment shows how randomness is only definable from the perspective of the observer. Given enough time and power, an omnipotent being should be able to see most things as non-random, but mere mortals are constrained by things like time and computational complexity, so we must inherently describe some things as random.

3.6. **Why are these Pseudorandom?**

Anyone who considers arithmetical methods of producing random
digits is, of course, in a state of sin.
– John von Neumann

This brings us to our PRNGs and why they are not considered fully random processes. This is simply because they are the result of an arithmetical process and therefore can't be random. The idea behind PRNGs is that they *stretch* the true random input. We can't possibly decide a random place to start from with a deterministic machine, and so we rely on true randomness to get us started. In [3] the author defines pseudorandomness formally on page 12. We reproduce it here for convenience.

Definition 1. A deterministic polynomial time algorithm G is called a pseudorandom generator if there exists a stretch function $\ell : \mathbb{N} \to \mathbb{N}$ (satisfying $\ell(k) > k \forall k$), such that for any probabilistic polynomial time algorithm D, any positive polynomial p, and all sufficiently large k it holds that

$$|Pr[D(G(U_k)) = 1] - Pr[D(U_{\ell(k)}) = 1]| < \frac{1}{p(k)}$$

where U_n denotes the uniform distribution over $\{0, 1\}^n$ and the probability is taken over U_k (resp., $U_{\ell(k)}$) as well as over the internal coin tosses of D.

This is just formalizing the idea that the stretch function will expand the number of bits of randomness that we are given by the true random input without making the resulting bitstring *too unrandom*.

4. Tests of Randomness

4.1. **Empirical vs. Theoretical Tests.** There are, in general, two ways to test a random number generator for its quality. Empirical tests are those which operate on sequences of random numbers generated by these RNGs. An obvious example of this is that a sequence that claims to be uniformly random, must in fact, have a uniform distribution. Another test of this type is the serial test discussed in section 4.2. We'll only cover a limited set of these tests, for a good overview of others, reference section 3.3.2 of [8] from Knuth[11].

Somewhat expectedly, the other class of tests is the class of theoretical tests. These tests are nice to have, and can provide insight into why certain processes work and others don't, however they can't replace the empirical tests. The test we discuss in section 4.3 is a mix of both theoretical and empirical.

4.2. **Serial Test.** This is a rather intuitive test. As an instructive example, let's limit ourselves to testing the output of random bitstring generators. Clearly with a uniform output it is desirable to have a roughly equal number of 1's and 0's. This can be likened to flipping a coin n times, where the expected outcome is $n/2$ heads and $n/2$ tails. However, for a random string, we can take this a step further and require that the substrings 00, 01, 10, and 11 are also roughly equal. If a string is random, we should be able to do this for longer substrings also. We can use a chi-squared test to give an actual numerical output to this test, allowing for a measurement of the quality of a generator.

[11] "Look to Knuth for more insight", may as well have been the title for this paper.

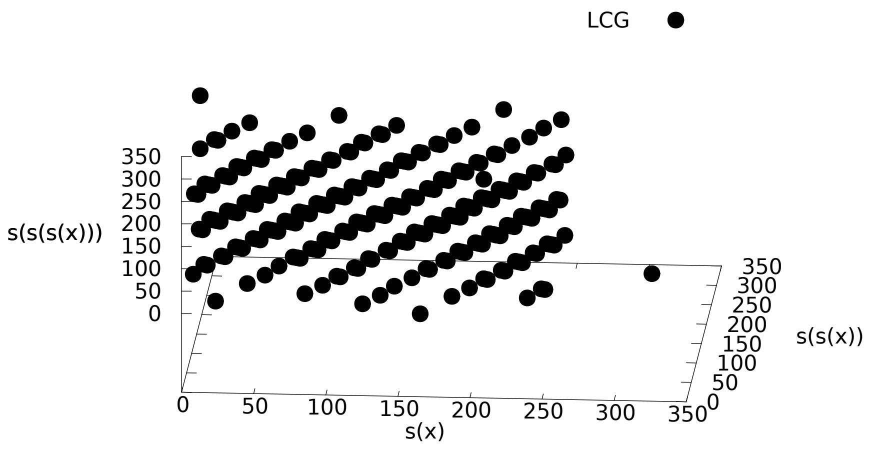

FIGURE 5. Example of spectral test for a PRNGs.

There are many other empirical tests, a group of which are included in the diehard suite (and the subsequent dieharder suite) of RNG testing software. Both consist of a number of different tests, each of which account for different issues that could arise with any given generator. According to the website[12] of the diehard suite, it consists of 17 separate tests. All of the tests return a p-value, which is a standard statistical value which helps to determine how likely the result is.

4.3. **Spectral Test.** According to Knuth in [8], this is one of the most powerful tests for examining the randomness of an RNG. Figure 5 shows the idea behind this method. The three axes ($s(x)$, $s(s(x))$, $s(s(s(x)))$) are three consecutive outputs from each respective PRNG. The linear congruential generator clearly is only landing on a few planes of the 3 dimensional space, whereas the Mersenne twister is more scattered about. The spectral test looks at the space between those planes and how the maximum distance between hyperplanes changes as more consecutive outputs ($s^k(x)$) are added and "plotted" in higher dimensions. True random data will have a constant distance between hyperplanes in all dimensions while poorly generated pseudorandom numbers will have a decreasing distance as we add more dimensions [8].

5. Cryptographically Secure PRNGs

In order for a PRNG to be cryptographically secure, it has to meet a few more requirements than a standard PRNG. These CSPRNGs are designed to defeat the ever-present cryptanalyst adversary. This means that they must be resistant to not only strong attacks at guessing their pattern, but also side channel attacks. This moves them far into the land of engineering - although much mathematical theory is used in deterring accurate guessing, side channel attacks can only be discouraged by good system design. In [7], Bruce Schneier gives a few ways that CSPRNGs are compromised, a few ways that these compromises are exploited, and describes

[12]`http://www.stat.fsu.edu/pub/diehard/cdrom/source/tests.txt`

the design philosophy of the Yarrow system he and his coauthors developed. We'll cover a few of them here. The biggest design concern they dealt with was how to keep the entropy pool that provides the seed to the generator from becoming known or controlled by an attacker.

5.1. **How CSPRNGs are Compromised.**

Schneier et al. list six ways this can happen. The first is through overestimating the amount of entropy contained in the seeds and using easily guessable starting values. This problem is made less severe in Yarrow by having very conservative estimations of entropy. Clearly this will make it hard to generate lots of randomness because the algorithm will run out of entropy quickly. Yarrow accounts for this by using 2 pools of entropy. The first with somewhat relaxed requirements that is used to quickly reseed the generator each time it feels that the stream is becoming predictable. The second is very conservatively managed for the reasons stated earlier in this paragraph, and is only used once in a while to completely reseed the operations.

Showing just how much of an engineering problem CSPRNGs are, the second reason cited in the paper is that seed files are mishandled. Seed files exist to give a pool of entropy to the algorithm immediately upon starting the system so that the user does not have to wait before useful randomness can be created. Unfortunately, this creates a whole host of difficulties on the part of the programmer who must implement the system. Files are moved around with reckless abandon by operating systems, writing them to different parts of disk and memory without taking into consideration the high security needs of a CSPRNG. This can be exploited by an attacker; so Yarrow is careful to simplify file management for the programmer. This is a consistent theme across Yarrow and it is the next way Yarrow avoids compromise: the system must be simple enough to actually implement.

In a more mathematical and somewhat obvious light, cryptanalytic attacks can compromise the CSPRNG. The system must be built with strong cryptography in mind so that the process can't be broken and the rest of the stream predicted.

Finally, these systems can be vulnerable to both side channel attacks and chosen-input attacks. Side channel attacks include things like timing attacks and other ways which information is leaked. The interesting part here is the chosen-input. This might be somewhat non-obvious because we're not encrypting anything here, but keep in mind that the entropy pool is mostly made up of functions applied to user actions. Yarrow defeats this by applying a cryptographic hash to all of the samples used for the entropy pools.

5.2. **How CSPRNGs are Exploited.**

The first and most terrible exploit possible once a seed is compromised is a permanent compromise. Apparently certain systems suffer from the property that once a seed is compromised, the system is essentially compromised forever. Yarrow avoids this with the constant reseeds mentioned earlier.

In addition, certain systems can be iteratively attacked once a single key has been compromised to find future ones, or even run backwards to find past results once a key is compromised. In addition, whatever compromise happens, if it occurs in such a way that high-value keys can be discovered - such as those for public key cryptography - the cost to the user is extremely high. Yarrow can call on any extra little bits of entropy it can find in order to gear up for actions such as creating keys for other cryptosystems. This is intended to be used rarely, but is an important feature nonetheless.

6. Secure *Prime* Generators

In general, the accepted method for generating primes is picking a range in which to find the primes and then trying again and again until a prime is found. Much of the time we cannot guarantee that a number we find is prime, only give a good probability of this being the case. Much of the time this involves trial division, which makes for a pseudo-polynomial algorithm which grows exponentially with the input. Clearly, we want to be able to do better than this.

In [6] the authors discuss methods of producing random primes efficiently. For their algorithm to work, we first need a PRNG and a primality test. The important benefit of the method in the paper is that it avoids trial division, which is a very costly part of the process. The algorithm will output a random prime number in an interval $[q_{min}, q_{max}]$. The idea behind the algorithm, and what makes it so clever is that it uses very smooth numbers to guarantee that prime candidates are coprime with many small integers, removing the need to do trial division. The algorithm can be improved to only emit safe and semi-safe primes if wanted. These improvements to generating random primes are necessary due to the disparity in computing power that exists in modern times.

7. Final Notes

This has been a long and winding journey through the very basics of random number generation. We've covered pseudorandom number generators, defining randomness, random prime generation, and how to test generators. Each of these is a deep topic worthy of a book in and of itself, but hopefully this has served as a worthy overview of each field.

References

[1] Bessette, Pitney, Brown, Langenegger, Garcia, Lewis, and Biles. *American Government and Politics*, pages 245–246. Wadsworth, Boston, Massachusetts, 2011.

[2] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*, pages 170–190. MIT Press, Cambridge, Massachusetts, 2nd edition, 2009.

[3] Oded Goldreich. *A Primer on Pseudorandom Generators*, volume 55 of *University Lecture Series*. American Mathematical Society, Providence, Rhode Island, 2010.

[4] C.M. Gonzlez, H.A. Larrondo, and O.A. Rosso. Statistical complexity measure of pseudorandom bit generators. *Physica A: Statistical Mechanics and its Applications*, 354(0):281 – 300, 2005.

[5] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer, New York, New York, 2000.

[6] Marc Joye and Pascal Paillier. Fast generation of prime numbers on portable device: An update. *Cryptographic Hardware and Embedded Systems*, 4249:160–173, 2006.

[7] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator.

[8] Donald E. Knuth. *The Art of Computer Programming*, volume 2, chapter 3. Addison-Wesley, Reading, Massachusetts, 1980.

[9] M. Matsumoto and T Nishimura. A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[10] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. 2003.

[11] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston Massachusetts, 2nd edition, 1997.

Appendix A. Quicksort Algorithm

A.1. **Background.** The sorting problem is a simple one to understand. Given an array of integers, rearrange the position of each integer in the array so that walking from the left of the array to the right we never find elements less than the ones we've already come across. This is non-decreasing order, and the complementary problem is to sort into non-increasing order, but any comparison algorithm which works for one will trivially work for the other.

Comparison based sorting algorithms have only one operation they can perform on elements in order to sort them. Somewhat unsurprisingly, this is comparing them. A comparison takes the form of

$$\begin{cases} -1 & a < b \\ 0 & a = b \\ 1 & a > b \end{cases}$$

where a and b are two integers passed into the function. This is a very general way to sort, and can accommodate many situations. It has been proven that comparison sort has an asymptotic limit of $\Omega(n \lg n)$, meaning that no algorithm can do better than this in the worst case. Reference section 3.1 for an explanation of this notation.

Quicksort was first devised by a British computer Scientist C.A.R. Hoare in 1960 while studying in Moscow as a visiting student. It actually has a worst case runtime of $O(n^2)$, but it is expected to run in $O(n \lg n)$ time according to [2]. The reason it is used is that it has very favorable constants hidden in the runtimes and the worst case can generally be avoided by using randomization.

Intuitively Quicksort actually works because if an array is split up all the way into into single element subarrays, the subarrays are trivially sorted. Then it just becomes a question of how to make sure these single element subarrays are in order when we split them. This is the job of the PARTITION method which puts all of the elements less than a certain value on the left of a pivot, and all of the equal or greater elements on the right. Understandably, once everything is said and done all of the elements are sorted in non-decreasing order.

For an in depth examination of Quicksort, reference [2]. In the next sections we'll only explore the parts that are important for understanding why random numbers are useful in Quicksort.

A.2. **Explanation.** The algorithm, as all divide-and-conquer algorithms do, has 3 steps:

Divide: Split the array into two parts, using PARTITION, such that the value of all of the elements are less than the value of an element we've chosen to split the array on, and the other subarray is comprised of elements greater than the split. Choosing this split value is where random numbers will come into use.

Conquer: Sort the two subarrays by recursively calling Quicksort on them. This accounts for a very small part of the total work done by the algorithm.

Combine: Once we've recursed all the way to the bottom, we need do no more work as the entire array is now sorted.

The key to Quicksort lies in the divide step. The difficult part to get right is picking which element to pivot on. The ideal situation is that we split each subarray exactly in half. We'll explain why in the next section. There are three basic choices for choosing this element:

Constant Element: This is method can actually work fairly well if we know what kind of input we'll be receiving. For instance, if we know that the input is already sorted, we can just pick the middle element and partition on it. Early versions of Quicksort picked the first element of each subarray rather than the middle one which led to worst case performance on already sorted arrays. Unfortunately, this worst case can happen for any sort of input we devise and this naive implementation does nothing about it.

Average of Elements: In an attempt to fix this, we can take $n > 1$ elements from the array and pick the median from among them in order to get a better chance at picking a favorable value. This does in fact give us a good chance of running in the expected case, but we can still improve upon this result.

Random Element: We'll show in the next section that with random input, Quicksort is proven to run in its expected time of $O(n \lg n)$. Rather than hoping for random input, if we randomize which element we pick to pivot on, we effectively randomize the data for ourselves. This will result in that optimal runtime we want.

A.3. **Analysis.** In order to trend towards our ideal case of a perfectly even split each time, we try to *not* have uneven splits each time, which is highly unlikely just by chance. Therefore we try to randomize in order to even out the number of good and bad splits. We can carry out an analysis by assuming that the input is random (something we guarantee with the randomized partitioning discussed above). We can describe the size of the problem by the recurrence

$$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n) \tag{4}$$

Where α is a number ranging between 0 and 1 exclusive. What this means is that at each step, the problem becomes two smaller problems, and at each step we need to do n work to divide the subproblems. The $\Theta(n)$ in (4) comes from the time needed for the partition to actually do its work, but importantly, α is what determines the final running time of the algorithm. If α is exactly $1/2$ then this algorithm will run in $O(n \lg n)$ time because at each level we do n work from the partition, and there are $\lg n$ levels due to splitting the problem in half each time.

Randomizing the split helps ensure that α is $1/2$ rather than some value closer to 1 or 0, which makes this analysis not work. The reason it breaks this analysis is that rather than bottoming out in $\lg n$ steps, the bottom won't be reached until n steps making the runtime $O(n^2)$, clearly not desirable.

Case Western Reserve University, Cleveland, OH 44106
E-mail address: `bis12@case.edu`