RANDOM NUMBER GENERATION

ASHWIN RAJAN

1. Introduction

Cryptography plays a major role almost everywhere online. However, the unrecognized backbone of cryptography is Random Number Generation. Accordingly, the power and security of a cryptographic system relies upon and is directly defined by the security of the random number generation system behind it. There are two main kinds of random number generation: truly random number generators (TRNGs) and pseudorandom number generators (PRNGs), which appear statistically random.

In cryptographic systems, PRNGs are used most often because TRNGs lack in two key ways: speed and reproduciblity. TRNGs are much slower than PRNGs mainly because they pull data from far-removed sources, like atmospheric noise. While data like this is much more secure than calculating sound around a computer (something that can be heavily influenced by computer fans) or measuring mouse movements, it takes a very long time to produce.[1] In fact, random.org takes approximately one second to create 1 random number, while an older PRNG takes one second to create 2 billion random numbers. Another issue with TRNGs is that they cannot provide reproducible results because they are truly random—results will never follow a predictable pattern, unlike with PRNGs. This poses many issues, one of which is with verification, as we will not be able to verify that the TRNG works and provides identical results with different software implementations. Overall, this is the main reason why we do not consider TRNGs for cryptographic purposes. Even within PRNGs, however, there is a constant trade off between security/unpredictability and efficiency; often times, the faster a PRNG works, the less secure it is.

Within PRNGs, there is a subsection of Cryptographically Secure Pseudorandom Number Generators (CSPRNGs). These satisfy the requirements for RNGs that are needed for cryptography. While they are sometimes slower than normal PRNGs, a certain baseline for security is necessary (and far more important than small changes in speed!). A universal way to measure and compare randomness and unpredictability is by using entropy. When entropy is too low, a PRNG risks high predictability, and thus becomes susceptible to attacks. Lastly, although previously many PRNGs were considered secure, over time, technology has greatly increased in its capacity to destroy these systems, and therefore, many are considered deprecated, or unusable for cryptographic purposes. For the rest of this paper, we will discuss less known PRNGs and CSPRNGs that vary greatly in the background mathematics and applications.

Date: November 2019.

[1] If interested in seeing how slow TRNGs are, check out random.org, which pulls data from atmospheric noise

2. Important factors of a PRNG

To evaluate the usefulness and the possible applications of a PRNG, there are many factors that one must look at, but the two most important are period length and efficiency.

2.1. **Period length.** Period length plays a key role in deciding whether a PRNG is secure because if it has a period of 2^{10}, for example, then it only takes 1024 numbers to repeat. This is far too few, and thus means that the predictability of a system will go down. Essentially, if an attacker knows one value, they know that value will continue to appear and can use that to his or her advantage. While it seems a small piece of information, only 1/1024, any information provided to an attacker can be used maliciously. The goal of a CSPRNG is to provide as little information as possible, which is hopefully none. However, it is also important to realize that, although a short period automatically rules out a PRNG, having a long period does not necessarily mean that a PRNG is cryptographically safe.

2.2. **Efficiency.** While period length is a major part of whether a random number generation system is bad, efficiency tests whether it is good. As mentioned earlier, there is a constant trade-off of efficiency and security, but once there is sufficient security, we must focus on effiency. Similarly to the example of random.org, we cannot rely on a program that produces a random number every second; to satisfy various cryptographic purposes, CSPRNGs must be able to generate far more numbers in a shorter span of time.

2.3. **Security.** The most obvious component of a random number generation is its level of security. For instance, does the random number generator have any major pitfalls; for example: does it only produce multiples of 2? Is it uniform across all numbers mod 623? These kind of questions, although seemingly arbitrary, can be the basis for an attacker's attempt at hacking a cryptosystem. If the attacker can find a vulnerability, then ey can take down an entire cryptosystem.

3. Random Number Generators

In this section, we will look through varying examples of PRNGs, the math behind them, and their advantages and disadvantages.

3.1. **Mersenne Twisters.**

Definition 3.1 (Mersenne Primes)**.** Mersenne primes are primes of the form $2^k - 1$, where k is a nonnegative integer.

Theorem 3.2. *Mersenne primes of the form 2^k-1 (with $k \geq 0$) must have a prime exponent k.*

Proof. For the sake of contradiction, let's assume that the exponent k is a composite number. Given an exponent $k = ab$, where a and b are both greater than 1, we have $m = 2^{ab} - 1$. However, in this case, $m = 2^{ab} - 1$ can always be factored: we have

$$\begin{aligned}
&(2^a - 1)(2^{a(b-1)} + 2^{a(b-2)} + \cdots + 2^a + 1) \\
&= 2^a(2^{a(b-1)} + 2^{a(b-2)} + \cdots + 2^a + 1) - 1(2^{a(b-1)} + 2^{a(b-2)} + \cdots + 2^a + 1) \\
&= (2^{ab} + 2^{a(b-1)} + \cdots + 2^{2a} + 2^a) - (2^{a(b-1)} + 2^{a(b-2)} + \cdots + 2^a + 1)
\end{aligned}$$

We can see that all the terms in this expression will cancel out except the edge terms 2^{ab} and -1, so this will evaluate to $2^{ab} - 1$ as desired.

Thus, whenever an integer is in the form of $2^m - 1$, where m has factors a and b, the integer will also be divisible by $2^a - 1$, which is a nontrivial divisor. Although we have reduced most of the non-prime cases, we are still missing 0 and 1. However, these cases are trivial, as we have $2^0 - 1 = 0$, and $2^1 - 1 = 1$. Neither of these results are prime, so therefore, we know that a prime of the form $2^k - 1$ must have a prime exponent k. ■

Definition 3.3 (k-dimensional equidistribution)**.** A k-dimensional equidistribution is a distribution in which all points on a k-dimensional space are possible to receive.

Definition 3.4 (Mersenne Twister)**.** The Mersenne Twister is a pseudo random number generator that relies on matrix linear recurrence over a binary field f_2.

Definition 3.5 (Simple Recurrence Relation)**.** A recurrence relation is an equation that recursively defines a sequence using previous terms in a function that determines future terms. More specifically, a simple recurrence relation determines n_{i+1} directly from n_i.

The Mersenne Twister uses a matrix linear recurrence in that it produces a series x_i using as a recurrence and then outputs numbers of the form $x_i * M$ where M is an invertible F_2 matrix. One way that the Mersenne Twister optimizes for speed is by using a matrix that simplifies calculations. In simplest terms, according to the Mersenne Twister's creators, the Mersenne Twister is an iteration of a fixed linear transformation on a fixed vector space.

3.1.1. *Advantages.* Although many other systems that work mathematically like the Mersenne Twister exist, there are multiple reasons why one should pick the Mersenne Twister.

(1) The extremely large period that is built into the Mersenne Twister makes it attractive. Although, as we stated earlier, a large period does not guarantee a successful PRNG, a small period is a major red flag.
(2) The Mersenne Twister passes many statistical tests for randmoness, including the stringent Diehard tests.
(3) A 623-equidistribution is very powerful because that means that any possible set of 623-tuples is possible to get. Thus, for example, even if a hacker finds 622 random numbers in a row, they still have no idea what the next number will be.
(4) Most implementations of the Mersenne Twister offer faster random number generation than other PRNGs, in part due to the efficient multiplication with the optimized matrix.

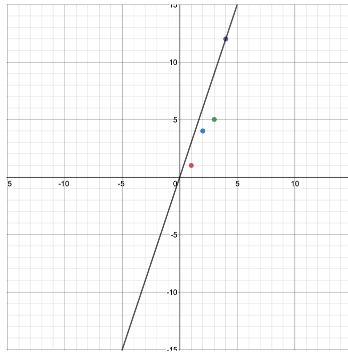
3.1.2. *Disadvantages.* However, there are also several issues with Mersenne Twisters.

(1) The most important disadvantage of the Mersenne Twister is it not being cryptographically secure in most variants. This is because of the k-distribution. After observing 624 iterations (in the MT19937 case specifically–this changes for different variants), one can predict all future iterations.
(2) The Mersenne Twister takes a long time to create create numbers that pass statistical randomness tests in specific cases, especially when the initial state has many zeroes.
(3) With multiple instances of the Mersenne Twister, and a similar initial state, both generators will produce similar sequences for a long time before eventually diverging.

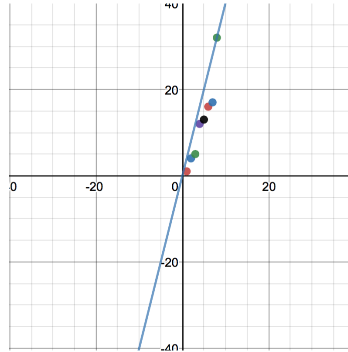
It's important to notice that all of these disadvantages are preventable; they can be easily avoided by handpicking initial states. Furthermore, there are variants of the Mersenne Twister–such as the CryptMT variant that uses the Mersenne Twister internally–that are cryptographically secure. For these reasons, the Mersenne Twister is one of the most used general-purpose PRNGs.

3.2. **Fortuna.** Fortuna is a family of CSPRNGs, and leaves some choices open to the implementors. It was created by Bruce Schneier and Niels Ferguson in 2003.

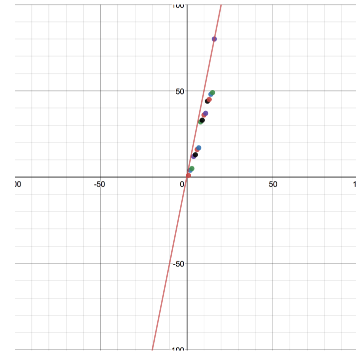
3.2.1. *Entropy accumulator.* The most unique aspect of Fortuna is its entropy accumulation process. While most PRNGs use linear entropy accumulation, Fortuna pulls in a nonlinear fashion from multiple sources of entropy. The first reason why Fortuna does this is because it allows for more reliable randomness; Fortuna can pull entropy from mouse movements, computer noise, and fan speed all together. This means that even if one of the forms of randomness becomes predictable or hijacked, then the other ones will continue to protect the unpredictability of Fortuna. The key part of the entropy accumulation is that, for the nth reseeding, pool k is used only if n is a multiple of 2^k. This means that pools with a higher index are pulled from less. More exactly, pool k has entropy pulled from it with probability 2^{-k}. The second reason that this type of entropy accumulation is better than linear accumulation is because it allows pools with higher indices to gather greater amounts of entropy over time before it gets pulled. However, when we graph entropy accumulation linearly in comparison to this nonlinear entropy accumulation, we find something contradictory. As



(a) 4 total reseedings



(b) 8 total reseedings



(c) 16 total reseedings

you can see, for any finite integer[2] number of reseedings, a linear entropy accumulation has a greater amount of entropy from 0 reseedings to n reseedings (where the amount of entropy is equal). While this initially implies that a linear entropy accumulation is better, this neglects the fact that we do not originally know how long Fortuna will be running for. If one uses a linear entropy accumulator, and overestimates the amount of reseedings/entropy necessary, then ey will not have optimized entropy accumulation. By the end, the nonlinear accumulation will have a greater amount of entropy. Conversely, if one underestimates the amount of reseedings/entropy necessary, then ey will not have the necessary amount of entropy, and at a certain point, they will no longer be able to pull new entropy, leaving the random number generation easily hackable and predictable.

3.2.2. *Generator.* The generator of Fortuna is not particularly special. In fact, it is just based on a block cipher. The choice is left to the implementor, but the creators of Fortuna supplied three recommended options: Serpent, AES, or Twofish. There are a few challenges, however, in a simpler implementation. In using a 128-bit block cipher, there would statistically be one repeated identical block for every 2^{64} blocks produced. However, there are no repeated

[2] Technically, the finite integer must be a power of 2, but that can be rounded up or down to.

blocks in the first 2^{128} blocks produced by a 128-bit cipher in counter mode. To prevent the statistical deviation, the key is changed at most every 2^{16} blocks.

3.2.3. *Notes.* Fortuna was based upon the Yarrow PRNG, and is very similar. They differ in two ways, both in the entropy accumulator. Firstly, Yarrow uses only two pools of entropy, while Fortuna uses 32. Next, Yarrow requires entropy estimation for each of the pools. This is not necessary in Fortuna because it is guaranteed that as long as there is one pool still running, there will be enough entropy. In other words, the only way for someone to hijack Fortuna is for them to have control over every source of entropy.

3.2.4. *Advantages.*

(1) Firstly, and most importantly, Fortuna is a CSPRNG. This allows its usage to be far wider than other PRNGs.
(2) Fortuna's entropy accumulation allows enough entropy to accumulate in one of the lesser used pools that it can eventually guarantee security (and continue increasing it).
(3) Fortuna's entropy accumulation also allows for greater security. Many PRNGs have only one pool of entropy, which means that if it is hijacked (or potentially just a poorly chosen source of entropy), then the whole PRNG will be ruined. However, Fortuna is resistant to entropy source attacks, as it has so many different sources.

3.2.5. *Disadvantages.*

(1) There really is only one main disadvantage of Fortuna. The problem is that Fortuna has a few restrictive prerequisites (for instance: entropy of the inputs must be constant). Fortunately, this can be fixed, and the paper, "How to Eat Your Entropy and Have it Too — Optimal Recovery Strategies for Compromised RNGs," recommends possible changes to Fortuna to fix this problem.

3.3. **Middle Squared Method.** The Middle Squared Method is a severely deprecated and very primitive example of a PRNG. It takes a $2n$ digit number, and squares the middle n digits. This yields either $2n$ or $2n-1$ digits. However, for this process to work, an even number of digits is necessary. Thus, to compensate in the case of $2n-1$ digits, a leading zero is added.

3.3.1. *Examples.* Using a random number generator, we have three four digit numbers as examples: 2796, 9158, and 4684.

(1) 2796, 6241, 0576, 3249, 0576,... The sequence then repeats with 0576 and 3249 indefinitely.
(2) 9158, 0225, 0484, 2304, 0900, 8100, 0100, 0100,... The sequence then produces 0100 indefinitely.
(3) 4684, 4624, 3844, 7056, 0025, 0004, 0000, 0000,... Once again, the sequence produces 0000 indefinitely.

We can clearly see that the middle-squared method does not work, even for non-cryptographic purposes.

3.3.2. *Disadvntages.*

(1) The middle-squared method converges to 0, so even if it does not fall into a cycle of repeating numbers, it still will eventually converge to 0.
(2) As described above, a sequence can often fall into a repeating cycle.

3.4. **Blum-Micali algorithm.** The Blum-Micali algorithm is a CSPRNG that depends on the difficulty of the discrete logarithm problem for its security. Given an odd prime p, a primitive root g (when taken modulo p), and seed x_0, we have $x_{i+1} = g^{x_i} \bmod p$. The *ith* output of the Blum-Micali algorithm is 1 if $x_i < (p-1)/2$. Otherwise, the output is 0.

3.4.1. *Pitfalls.* The most common problems with the implementation of the Blum-Micali algorithm is with the prime numbers chosen. They have to be prime numbeers chosen specifically so that they are resistant to algorithms that can solve the discrete logarithm problem on certain primes.

(1) Most generally, one cannot use a small prime. With smaller primes, it is very easy to solve the discrete logarithm problem.
(2) The Pohlig-Hellman algorithm makes the discrete logarithm problem very easy to solve for certain primes. If $p-1$ has (relatively) small prime factors (is k-smooth for a small k), then an attacker can use the Pohlig-Hellman algorithm to greatly reduce the time complexity of solving the discrete logarithm problem. Although receiving a number with all small prime factors is rare[3], it is important to consider this, as the Blum-Micali algorithm is used for cryptographic purposes.

3.4.2. *Advantages.*

(1) As you can probably guess, the main advantage of the Blum-Micali algorithm is it being a CSPRNG. This greatly increases its usability.
(2) Another key factor of the Blum-Micali algorithm is its dependence on the discrete logarithm problem. The discrete logarithm problem is computationally intractible, and for general cases, can only be whittled down to time complexity of $O(\sqrt{n})$. While much better than the guess and check trivial method which yields $O(n)$, this is still far from polynomial time. Thus, the Blum-Micali algorithm (and many others!) are still safe for the time being.

3.4.3. *Disadvantages.*

(1) The main disadvantage of the Blum-Micali algorithm is the one discussed earlier. The prime number has to be picked carefully. Although the chance of getting a p-smooth $p-1$ is unlikely, it must still be accounted for as a possibility.
(2) An arguably bigger problem with the Blum-Micali theorem is its dependence on the discrete logarithm problem. Although also listed as an advantage, that is only for the time being. Even in the present, there is (minimal) progress towards using quantum computing and Shor's algorithm to essentially break the discrete logarithm problem. However, depending on one's opinion, this disadvantage can be disregarded, as it is a problem with most secure or commonly used PRNGs.

[3] In fact, the expected value for the largest prime factor of a number n is close to $n/log(n)$

4. PRNGs in Cryptography

Cryptography relies on CSPRNGs to create unpredictability. However, the last disadvantage we discussed highlights a larger problem within Cryptography and how it must change in the following years. With quantum computing on the rise and major advances in it coming yearly, we can expect a large upheaval of previously secure cryptosystems having to be replaced. However, this can be avoided, for the most part, by focusing on the random number generators themselves. Firstly, we must understand how CSPRNGs are used in Cryptography. When using a cryptosystem, a random seed is necessary. As cryptosystems rely on (often recursive) mathematical algorithms to create future numbers. However, for the initial seed, we need to use a random number generator; if we use the same number again and again, it will (due to the mathematical nature of the cryptosystem) create the same sequence. Clearly, random number generators are extremely important to cryptography, but also exist in many different forms, some useful for cryptographic purposes, and some, not.