Random Number Generation (RNG)

Author: Omkar Patel

Abstract. Random Number Generators (RNGs) are essential in various applications, including gaming, encryption, Monte Carlo simulations, sorting algorithms, and cryptocurrency. This article explores the various methods which computers use in order to produce random numbers, including pseudo-random number generation (PRNG) and true random number generation (TRNG). Due to the fact that computers operate in an established manner, they rely mostly on pseudo-random generation, a method in which an algorithmic process generates numbers, not truly random in the case where the state of the algorithm is known. True random number generation, on the other hand, utilizes unexpected physical events such as radioactive decay and atmospheric noise to generate (truly) random numbers. This article covers the advantages and disadvantages of these systems , as well as their real-world applications, concluding by highlighting situations in which a real RNG is required, emphasizing its role in improving system security. We will build a model involving an anode/cathode diode which acts as a basic TRNG.

Keywords: pseudo-random number generator (PRNG) · true random number generator (TRNG) · algorithm · unpredictable/predictable

1. Introduction

Random numbers are an essential part of everyday life, playing essential roles in fields such as probability theory, industrial testing, gambling, and computer simulations. Despite their ubiquity, generating truly random numbers is a challenging task, especially when using Pseudo-Random Number Generation (PRNG) approaches.

In the PRNG method, a system initializes a seed value which is processed through a series of mathematical functions and algorithms into a number within a set start and end range. However, this method is flawed, and in the following sections of this paper, we will look into the ways in which this system can be broken and exploited.

Unlike PRNG, True Random Number Generation (TRNG) methods adopt a different model. TRNG relies on an external source of randomness in order to establish the initial seed. This external source, which can include physical phenomena, such as atmospheric noise or radioactive decay, introduces an unpredictable element, enhancing the randomness of the generated numbers. Due to this, the same seed cannot be generated multiple times, creating a secure generating system.

Furthermore, understanding the limitations of PRNG is crucial, including its practical implications in various applications. For example, in the context of cryptographic systems, weakness in randomness of generated numbers can lead to vulnerabilities and compromise the security of the entire system.

In conclusion, as we move into the digital world with complex technologies, the need for secure random number generation becomes more crucial, requiring the exploration of alternative methods such as TRNG becomes essential.

2. Pseudo-Random Number Generators

Pseudo-Random Number Generators (PRNGs) are widely used in the field of computer science in order to generate sequences of numbers that demonstrate statistical properties of randomness. Most random number generators, including PRNGs, rely on seeds, initial values inputted into an algorithm that converts them into pseudo-random numbers through a series of mathematical operations. An important point to note about PRNGs is that they are predeterministic; when given the same seed, a PRNG will always return the same random number. We will discuss this flaw later on in this section.

2.1 Linear Congruential

A code example for the Linear Congruential PRNG algorithm can be seen either in this github or in the code snippet shown below.

```
import time

class PseudoRandom:
    def generate_seed(self) -> int:
        seed = int(time.time() * 1000)
        return seed

    def pseudo_random_generator(self, end_range) -> int:
        seed = self.generate_seed()
        random_number = ((seed * 6364136223846793005) % 2**64) % end_range
        #     Xnf     = ((Xn   *          a         ) % 2**64) % m
        return random_number

random_num = PseudoRandom().pseudo_random_generator(end_range=100)
print("Random number:", random_num)
```

First, the current time is extracted using the time module. This code then converts the time, for example, 1699926184.773952, into a larger seed by multiplying it by 1000, resulting in a value

such as 1699926184773.952. Using a linear congruential generator formula, a pseudo-random number is generated:

$$X_{nf} = ((X_n * a) \% 2^{64}) \% m$$

(Linear Congruential)

The number X_n, in this case the operating systems $time.time()$, is first converted into a seed by multiplying it by a constant a. We take the result modulo 2^{64} to ensure it's within a 64-bit range and then take the result modulo end_range to get a number between 1 and end_range.

2.2 Mersenne Twister

Another commonly used formula for number generation when given a seed is the Mersenne Twister Equation as seen below,

$$X_{n+1} = X_n \oplus ((X_n \gg w) \oplus 0x9908B0DF)$$

(Mersenne Twister)

Where X_n is the current state, w is the word size (typically 32 bits for MT19937), $\gg$ signifies a right bitwise shift, and XOR denotes bitwise XOR. The equation involves two bitwise XOR operations and a right shift, contributing to the unpredictability of the generated numbers. The constant *0x9908B0DF* serves as a parameter ensuring a sufficiently intricate update process. Through this iterative application of the equation, the Mersenne Twister achieves a balance between computational efficiency and statistical quality, making it a favored choice for various applications that require pseudorandom numbers.

A code example for the Mersenne Twister PRNG algorithm can be seen either in this github or in the code snippet shown below.

```
import time

class MersenneTwister:
    def __init__(self):
        self.state, self.index = [0] * 624, 0
        self.seed_mt(self.generate_seed())

    def generate_seed(self):
        temp, seed = int(str(time.time())[-1]), int(time.time() * 1000)
        seed *= temp if temp % 2 == 0 else 1 / temp
        return seed % 2**32

    def seed_mt(self, seed):
        self.state[0] = seed & 0xFFFFFFFF
        for i in range(1, 624):
            self.state[i] = (1812433253*(self.state[i-1]^(self.state[i-1]>>30))+i)
            Self.state[i] &= 0xFFFFFFFF

    def generate_mt(self):
        for i in range(624):
            y = (self.state[i] & 0x80000000) + (self.state[(i + 1) % 624] & 0x7FFFFFFF)
            self.state[i] = self.state[(i+397)%624]^(y>>1)^(0x9908B0DF if y%2!=0 else 0)

    def extract_random(self, end_range):
        if self.index == 0: self.generate_mt()
        y = self.state[self.index]
        y ^= (y >> 11) ^ ((y << 7) & 0x9D2C5680) ^ ((y << 15) & 0xEFC60000) ^ (y >> 18)
        self.index = (self.index + 1) % 624
        return y % end_range

    def main(self):
        self.seed_mt(self.generate_seed())
        print("Seed:", self.state[0])
        print("Random Number:", self.extract_random(100))

# Create an instance of MersenneTwister and run the main function
MersenneTwister().main()
```

The code shown above defines a custom random number generator using the Mersenne Twister algorithm. The *MersenneTwister* class includes functions to generate a seed based on the current time, seed the Mersenne Twister, generate random numbers, and extract random numbers.

The *generate_seed* function retrieves the current time, multiplies it by 1000, and returns the resulting value as the seed. The Mersenne Twister is seeded using the *seed_mt* function, and random numbers are generated using the *generate_mt* function. The *extract_random* function extracts a random number from the Mersenne Twister state. The *main* function initiates the process, seeds the Mersenne Twister with the generated seed, extracts a random number, and prints both the seed and the generated random number.

2.3 Flaws of PRNGs

The use of seeds in Pseudo-Random Number Generators (PRNGs) brings up an important constraint - determinism. If a PRNG is called with the same seed, it will always produce the same number due to the fact that a PRNG utilizes a set algorithm. This predictability is a major security problem, particularly in cryptographic systems where real randomness is required. For example, the linear congruential formula shown in the code snippet produces the same result when run on multiple instances at approximately the same time.

Recognizing this weakness makes it clear that more entropy sources are required to improve the unpredictability of PRNGs. While using system time as a seed is convenient, it falls short in cases requiring a higher level of randomization and security. As a result, this weakness requires a detailed assessment of the issues posed by PRNGs, as well as an investigation of the benefits provided by true random number generators (TRNGs) in improving these limits, which we shall accomplish in the following sections.

3. True Number Generators

True Random Number Generators (TRNGs), on the other hand, generate random numbers in a different manner than the pseudo-random generators. While PRNGs use deterministic algorithms

and seeds to generate purportedly random numbers, TRNGs use physical processes in order to obtain (true) randomness.

The randomness in a TRNG is generated by the inherent diversity in the physical processes involved, which eliminates the need for a seed or deterministic algorithm. This crucial distinction is that each TRNG output is independent of any previous state, making it nearly impossible to recreate the exact sequence of numbers created by a TRNG.

While PRNGs have uses when repeatability and efficiency are critical, TRNGs are particularly useful in situations requiring the highest level of unpredictability and security. As we go deeper into the area of True Random Number Generators, we will investigate the principles underlying their reliance on natural randomness, as well as its benefits and particular problems.

3.1 Entropy

The entropy of a random variable is the average level of information, surprise, or uncertainty inherent to the variable's possible outcomes. Given a discrete random variable *X*, which takes the values in the alphabet *X* and is distributed according to $X \rightarrow [0, 1]$:

$$H(X) := \quad - \sum_{x \in X} p(x)\, log \cdot p(x) = \exists[- logp(X)]$$

(Information Theory)

Where Σ denotes the sum over the variables possible values. The choice of base for log, the logarithm, varies for different applications. Base 2 gives the units of bits (or “shannons”), while the base *e* gives “natural units” *nat*, and base 10 gives units of “dits”, “bans”, or “hartleys”. An equivalent definition of entropy is the expected value of the self-information of a variable.

Another example of an entropy calculation that can be used in RNGs is the Shannon Entropy formula, If we choose n (in this case $n = 4$) numbers uniformly from $[1, z]$, we can calculate the entropy. Recall the definition of (Shannon) entropy:

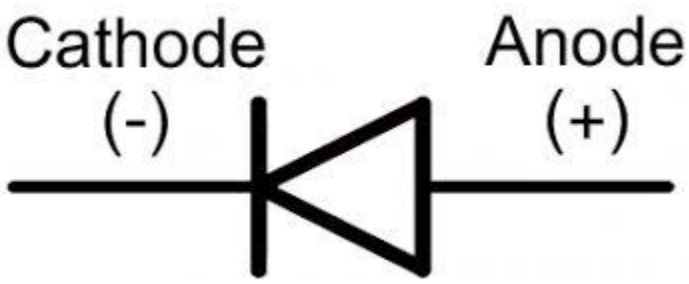
$$H(X) = \sum_{i=1}^{n} P(x_i)\, log_b\, P(x_i)$$

(Shannon Entropy)

Here X is the state (e.g. (1, 2, 3, 4)). In the case $z = 10$, the x_i values are the possible states, e.g. {(1, 1, 1, 1), (1, 1, 1, 2), …, (10, 10, 10, 10)}. $P(x_i)$ is the probability that the random number $X = x_i$. When the probability distribution is uniform, then the probabilities are all equal to $1/n$. Notice that this is a calculation not over the state itself, but over the probability distribution of all the possible states. In other words: The entropy is determined not on what the numbers are, but how they are chosen.

3.2 Diode Model

For this paper, we will be making a basic TRNG model using a diode. This type of TRNG takes advantage of the inherent unpredictability of electrical noise in the reverse-biased breakdown region of the diode.



The diode with its anode and cathode terminals is operated in reverse bias in this setup, entering a breakdown area where electronic noise occurs owing to random motion of charge carriers and thermal factors. An amplifier then amplifies the slight voltage fluctuations caused by the electrical noise. A comparator compares the amplified signal to a reference value, generating a binary output that reflects the stochastic changes in the diode's breakdown region. The resulting output is real randomness, and each generated sequence is inherently unique due to the physical

nature of the diode's activity. This hardware-based TRNG eliminates the need for a seed and provides a reliable solution for applications that require high degrees of unpredictability and cryptographic security.

4. Conclusion

Random Number Generators (RNGs) are essential in a wide range of applications, from gaming and encryption to simulation and cryptography. This article discussed the two main methods for generating random numbers: pseudo-random number generation (PRNG) and true random number generation (TRNG). TRNGs use physical processes to achieve real randomness, whereas PRNGs use deterministic algorithms and seeds.

Because of their repeatability and efficiency, pseudo-random number generators, such as Linear Congruential and Mersenne Twister, are commonly utilized. Their determinism, however, raises security concerns, particularly in cryptographic systems where true randomness is required. The shortcomings of PRNGs underscore the need for more unpredictable and secure techniques.

True Random Number Generators satisfy this need by harnessing physical processes' intrinsic unpredictability. To comprehend the unpredictability of TRNGs, the idea of entropy, which quantifies the quantity of information and uncertainty in a random variable, is investigated. The study presents a rudimentary TRNG model based on a diode, demonstrating the use of electrical noise in the reverse-biased breakdown area to generate real randomness. This hardware-based TRNG, which is free of deterministic algorithms and seeds, provides a dependable solution for applications that require maximum unpredictability and cryptographic security.

As technology progresses and the digital landscape changes, the need for safe random number generation is more pressing. Recognizing the limits of PRNGs and investigating alternatives, such as TRNGs, are critical steps in improving system security. The described diode-based TRNG model shows the possibility of using physical processes to generate really random numbers. Finally,

a thorough grasp of RNGs, their strengths and shortcomings, and real-world applications is critical for assuring the integrity and security of digital systems.

5. Works Cited

Chapter 3 Pseudo-Random Numbers Generators 3.1 Basics of Pseudo-Random Numbers Generators.

Jacak, Marcin M., et al. "Quantum Generators of Random Numbers." *Scientific Reports*, vol. 11, no. 1, 9 Aug. 2021, https://doi.org/10.1038/s41598-021-95388-7. Accessed 15 Sept. 2021.

"Polarity - Learn.sparkfun.com." *Learn.sparkfun.com*, learn.sparkfun.com/tutorials/polarity/diode-and-led-polarity.

Priyanka, et al. "Random Number Generators and Their Applications: A Review." *ResearchGate*, unknown, June 2019, www.researchgate.net/publication/335920952_Random_Number_Generators_and_their_Applications_A_Review.

"Pseudo Random Number Generator (PRNG) - GeeksforGeeks." *GeeksforGeeks*, 27 June 2017, www.geeksforgeeks.org/pseudo-random-number-generator-prng/.

"Pseudorandom Number Generator." *Wikipedia*, 22 Sept. 2021, en.wikipedia.org/wiki/Pseudorandom_number_generator.

Stipčević, Mario. "(PDF) True Random Number Generators." *ResearchGate*, 1 Nov. 2014, www.researchgate.net/publication/299824248_True_Random_Number_Generators.

Wikipedia Contributors. "Random Number Generation." *Wikipedia*, Wikimedia Foundation, 17 July 2019, en.wikipedia.org/wiki/Random_number_generation.