# Pseudorandom Number Generation

Hongyi Shi

August 2024

## 1 Introduction

In random number generation, there are two broad kinds of random number generators (RNGs): pseudo-random number generators (PRNGs) and true random number generators (TRNGs). While both serve the purpose of generating random numbers, TRNGs are not used often due to, among other issues, being non-deterministic and slow. An example of TRNGs example that comes to mind is Cloudflare's use of a lava lamp wall and cameras to collect entropy: slow but "true".

In cryptography, RNGs are needed to create unattackable keys, the foundation of many cryptographic systems. In zero-knowledge proofs, RNGs are crucial for creating unpredictable challenges. In password salting, RNGs introduce randomness to hashed passwords, ensuring that identical passwords produce unique hash values, thus protecting against common attacks such as rainbow table lookups. Without reliable RNGs, many of our security mechanisms would be opened up against new methods of attack.

PRNGs, as the name suggests, generate numbers that are not truly random but rather deterministic sequences that appear random. But what are the specific characteristics that we look for in order to have a strong PRNG?

1. **Determinism**: A PRNG, when initialized with a specific seed, will always produce the same sequence of numbers. This is crucial for reproducibility in simulations, testing, and debugging, where consistent results are necessary to validate outcomes.

2. **Period Length**: PRNGs have a finite sequence length, known as the period, after which the sequence repeats. A long period is essential to avoid patterns or repetitions that could undermine the effectiveness of the random numbers, especially in applications requiring vast quantities of random values.

3. **Uniform Distribution**: The output of a PRNG should be uniformly distributed over the desired range, ensuring that each possible value has an equal chance of being selected. This uniformity is critical for fair simulations, statistical sampling, and modeling.

4. **Statistical Independence**: Successive numbers generated by a PRNG should be independent of each other, meaning that knowledge of one number does not provide any information about the next. This independence prevents predictability and ensures the integrity of the random sequence.

5. **Efficiency**: PRNGs are designed to be computationally efficient, producing random numbers quickly and with minimal resource usage. This efficiency is vital in real-time applications or scenarios requiring a large volume of random numbers.

In cryptographic contexts, standard PRNGs are insufficient due to their predictability. While PRNGs produce seemingly random sequences, their outputs can often be predicted given enough previous values. In contrast, Cryptographically Secure PRNGs (CSPRNGs) are specifically designed to resist such vulnerabilities. CSPRNGs possess additional security properties that make it computationally infeasible to predict future outputs or reverse-engineer the initial seed, even with knowledge of previous outputs.

# 2 Mersenne Twister

The Mersenne Twister is a PRNG that generates random numbers through a two-step process: value extraction from its internal state and subsequent tempering. It gains its name for its long period length equal to the Mersenne prime $2^{19937}$

## 2.1 Tempering Process

For a $w$-bit word length (typically $w = 32$ for MT19937), the algorithm extracts a value $x_k$ from the state. This raw value undergoes tempering to improve its statistical properties, defined by the following transformations:

$$
\begin{aligned}
y &\equiv x \oplus ((x \gg u)\&d) \\
y &\equiv y \oplus ((y \ll s)\&b) \\
y &\equiv y \oplus ((y \ll t)\&c) \\
z &\equiv y \oplus (y \gg l)
\end{aligned}
$$

where $\oplus$ denotes bitwise XOR, $\&$ is bitwise AND, $\gg$ and $\ll$ are right and left bitwise shifts, and $u$, $d$, $s$, $b$, $t$, $c$, and $l$ are carefully chosen constants. This process ensures a more uniform distribution of output numbers in the range $[0, 2^w - 1]$ which makes the Mersenne Twister so useful.

## 2.2 Twisting Process

The "twister" mechanism updates the internal state using a matrix linear recurrence over the finite binary field $\mathbb{F}_2$. The state update follows the recurrence relation:

$$x_{k+n} := x_{k+m} \oplus ((x_k^u | x_{k+1}^l)A) \tag{1}$$

where:

- $n$ is the degree of recurrence
- $m$ is the middle word
- $x_k^u$ represents the upper $w - r$ bits of $x_k$
- $x_{k+1}^l$ represents the lower $r$ bits of $x_{k+1}$
- $A$ is a twist transformation matrix

The matrix $A$ is defined in rational normal form:

$$xA = \begin{cases} x \gg 1 & \text{if } x_0 = 0 \\ (x \gg 1) \oplus a & \text{if } x_0 = 1 \end{cases} \tag{2}$$

where $x_0$ is the lowest order bit of $x$ and $a$ is a coefficient.

This twisting operation ensures unpredictable state evolution, contributing to the generator's exceptionally long period of $2^{19937} - 1$, which is a Mersenne prime.

## 2.3 Attacking the Mersenne Twister

The attack on the Mersenne Twister exploits the ability to reconstruct an Mersenne Twister's internal state from its outputs. This weakness stems from the reversible nature of the tempering function used in the final stage of number generation.

### 2.3.1 Attack Introduction

The attack begins by collecting a sufficient number of consecutive outputs from the target Mersenne Twister generator. Typically, this requires 624 outputs, which corresponds to the size of the internal state array. Once these outputs are obtained, the attacker can use a process to "untemper" (reverse the tempering function) applied to each output. This reversal is possible because the tempering operations, while designed to improve the statistical properties of the output, are mathematically invertible. By applying the untemper function to each of the 624 collected outputs, the attacker can reconstruct the exact values in the internal state array of the Mersenne Twister. This reconstructed state is identical to the state of the original generator at the point just before it produced the first of the 624 observed outputs.

### 2.3.2 Example Untempering

Given the example tempering steps earlier, to untemper we simply need to reverse the operations. The specific constants for the Mersenne Twister are typically hardcoded into its implementation and are used if discovered.

Given the tempering operations:

$$
\begin{aligned}
y &\equiv x \oplus ((x \gg u)\&d) \\
y &\equiv y \oplus ((y \ll s)\&b) \\
y &\equiv y \oplus ((y \ll t)\&c) \\
z &\equiv y \oplus (y \gg l)
\end{aligned}
$$

The untempering steps can be represented as follows:

$$
\begin{aligned}
y &\equiv z \oplus (y \gg l) \\
y &\equiv y \oplus ((y \ll t)\&c) \\
y &\equiv y \oplus ((y \ll s)\&b) \\
x &\equiv y \oplus ((x \gg u)\&d)
\end{aligned}
$$

### 2.3.3 Attack Conclusion

With the internal state fully reconstructed, the attacker can create a clone of the original Mersenne Twister. This cloned generator will produce exactly the same sequence of random numbers as the original from that point forward. This is because the Mersenne Twister's next state is entirely determined by its current state, with no external inputs affecting its progression. Each time the generator produces a number, it updates its internal state in a deterministic manner. As long as the attacker's clone follows the same state update rules, it will remain in sync with the original generator, continuously predicting all future outputs.

# 3 Yarrow CSPRNG

Yarrow can be thought of as two separate parts. The first is an Advanced Encryption Standard (AES) based RNG that depends on a key, and the second is a process to update that key using truly random noise, similar to a TRNG. But because once the key is set, Yarrow is deterministic, and for it's extra security, it is considered a CSPRNG.

## 3.1 AES as RNG in Yarrow

At its core, Yarrow uses AES as its primary mechanism for producing random output. AES is a symmetric block cipher widely used for secure data encryption. It operates on fixed-size blocks of 128 bits, using keys that can be 128, 192, or 256 bits long. The algorithm processes data through a series of rounds, each involving four main operations:

### 3.1.1 SubBytes Step

Let $a_{i,j}$ represent the byte at position $(i, j)$ in the state matrix. The SubBytes transformation applies a non-linear substitution using an S-box:

$$b_{i,j} = S(a_{i,j})$$

where $S(a_{i,j})$ is defined as:

$$S(a_{i,j}) = A \cdot a_{i,j}^{-1} + C$$

Here, $a_{i,j}^{-1}$ is the multiplicative inverse of $a_{i,j}$ in Galois field GF($2^8$), $A$ is an invertible matrix, and $C$ is a constant vector. This step introduces non-linearity and ensures that the transformation is resistant to algebraic attacks.

### 3.1.2 ShiftRows Step

The ShiftRows operation involves a cyclic permutation of each row $i$ in the state matrix. Mathematically:

$$b_{i,j} = a_{i,(j+s_i) \mod n}$$

where $s_i$ denotes the shift amount for row $i$, and $n$ is the number of columns. For AES, $s_0 = 0$, $s_1 = 1$, $s_2 = 2$, and $s_3 = 3$.

### 3.1.3 MixColumns Step

Each column of the state matrix is treated as a vector in GF($2^8$) and multiplied by a fixed matrix $M$:

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = M \cdot \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix}$$

where $M$ is:

$$M = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Matrix multiplication in the Galois field GF($2^8$) involves both polynomial multiplication and addition (XOR), with multiplication carried out modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. This mixing ensures that the transformation has the desired cryptographic properties, such as resistance to differential and linear cryptanalysis, by spreading the influence of each byte throughout the entire state matrix.

### 3.1.4 AddRoundKey Step

The AddRoundKey transformation is performed as follows:

$$b_{i,j} = a_{i,j} \oplus k_{i,j}$$

where $k_{i,j}$ is the byte from the round subkey corresponding to $a_{i,j}$. This operation is simple bitwise XOR, ensuring that the transformation is easily reversible and key-dependent.

The number of rounds varies based on the key size: 10 rounds for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

## 3.2 Yarrow Implementation

Yarrow uses AES in a mode similar to counter-mode encryption. The PRNG maintains a secret key and a 128-bit counter as part of its internal state. To generate random numbers, Yarrow repeatedly encrypts the current counter value using the secret key with AES. The resulting ciphertext block is used as the random output.

Specifically, Yarrow uses AES over and over again until a random string of the desired length is achieved:

```
while(length > 0)
{
    n = MIN(length, AES_BLOCK_SIZE);
    yarrowGenerateBlock(context, buffer);
    osMemcpy(output, buffer, n);
    context->blockCount++;
    output += n;
    length -= n;
}

void yarrowGenerateBlock(YarrowContext *context, uint8_t *output)
{
    aesEncryptBlock(&context->cipherContext, context->counter, output);
    // Increment counter
    for(i = AES_BLOCK_SIZE - 1; i >= 0; i--)
    {
        if(++(context->counter[i]) != 0)
            break;
    }
}
```

After each encryption, the counter is incremented to ensure that each subsequent block of random data is generated from a unique input. This process effectively turns AES into a secure pseudorandom function, where the security of the generated numbers relies on the secrecy of the AES key and the large

space of possible counter values. Periodically, Yarrow updates its key using accumulated entropy, a process known as reseeding, which helps maintain long-term security and resist potential attacks. This use of AES allows Yarrow to quickly generate large amounts of cryptographically secure random data from a relatively small secret state, making it both efficient and secure for various applications requiring high-quality random numbers.

### 3.3 Entropy

While it's theoretically possible to calculate entropy precisely for a known probability distribution, real-world entropy sources often have complex, unknown distributions. This complexity necessitates the use of entropy estimation techniques, which approximate the amount of randomness in a given input.

Entropy, denoted as H, quantifies the average information content or uncertainty in a random variable X. For a discrete probability distribution, it's defined as:

$$H(X) = -\sum p(x) \log_2 p(x)$$

where $p(x)$ is the probability of outcome x.

In ideal random number generation, we seek maximum entropy, where all outcomes are equally likely. However, real-world entropy sources rarely conform to known probability distributions, making exact entropy calculation challenging. The Yarrow PRNG, leaves entropy estimation to the implementor of the algorithm. It is expected that the entropy estimation should be conservative in order to avoid false confidence in the amount of entropy. Examples of entropy collected include keystrokes, temperature, noise, and other system true random sources.

### 3.4 Slow Pool and Fast Pool

Yarrow utilizes a dual-pool system, consisting of a fast pool and a slow pool, to manage entropy and generate new keys. The fast pool is designed for quick incorporation of new entropy, allowing for rapid reseeding when any single source contributes enough entropy to reach a lower threshold. This ensures that the PRNG can quickly adapt to new randomness inputs. In contrast, the slow pool takes a more conservative approach, accumulating entropy over a longer period and requiring multiple sources to reach a higher threshold before triggering a reseed.

To update the key, the pool is hashed together using SHA-1. The specific process, however, differs depending on which pool triggers the reseed. For a fast pool reseed, the current key is combined with the hash of all inputs to the fast pool since the last reseed, effectively compressing the pool. This new value becomes the next key, quickly incorporating fresh entropy into the PRNG state. A slow pool reseed is more comprehensive: it combines the current key, the hash of all inputs to the fast pool, and the hash of all inputs to the slow pool.

# 4 Conclusion

Both RNG methods mentioned in this paper do not reflect the current standard for random number generation. Instead, they were chosen for their interesting generation methods. The Mersenne Twister, despite its long period and high-quality statistical properties, has vulnerabilities to state reconstruction attacks, leading to the development of its cryptographically secure variant, CryptMT. Yarrow, with its AES-based design and dual-pool system, represented a more security-focused approach from the outset. Yarrow was implemented in early versions of macOS and iOS but was later deprecated in favor of the Fortuna CSPRNG. The scrupulous reader may question the design choice where Yarrow uses SHA-1 to construct its keys from the pools. Although SHA-1 is indeed insecure as a hashing algorithm, nothing has yet been published regarding how that fact may be used to weaken Yarrow. Fortuna solves Yarrow's entropy estimation issues by doing away with it altogether, in favor of a 32-pool approach.

# References


[1] Murray, Bruce. *An Implementation of the Yarrow PRNG for FreeBSD*. 2002. Retrieved August 31, 2024. `https://www.usenix.org/legacy/events/bsdcon02/full_papers/murray/murray_html/`.

[2] citadel. *Fortuna—A Cryptographically Secure Pseudo Random Number Generator*. 2004. Retrieved August 31, 2024. `https://www.codeproject.com/Articles/6321/Fortuna-A-Cryptographically-Secure-Pseudo-Random-N`.

[3] Daemen, Joan. *The Rijndael Block Cipher*. Unpublished work.

[4] Dworkin, Morris J. *Advanced Encryption Standard (AES)*. 2023. NIST Federal Information Processing Standards Publication, No. 197-upd1. National Institute of Standards and Technology (U.S.). `https://doi.org/10.6028/NIST.FIPS.197-upd1`.

[5] Matsumoto, Makoto and Nishimura, Takuji. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*. ACM Transactions on Modeling and Computer Simulation, vol. 8, no. 1, 1998, pp. 3–30. `https://doi.org/10.1145/272991.272995`.