Pseudorandom Number Generators Research

12310302 王子旭

零、说在前面

对于本学期离散数学课程的内容中，对于PRNG的介绍并不多，同时也其在密码学中的应用也让我难以理解，因此，我选择了这个主题来更加深刻地了解这方面的知识，同时，我所在的课题组是做系统安全的，这对我未来的研究方向有一定的探索作用。

一、PRNG概述

1.PRNG基本定义回顾

在wiki百科（https://zh.wikipedia.org/wiki/伪随机数生成器）中有这样的介绍：

> **伪随机数生成器**（英语：pseudo random number generator，PRNG），又被称为**确定性随机比特生成器**（英语：deterministic random bit generator，DRBG），是一个生成数字序列的算法，其特性近似于随机数序列。伪随机数生成器生成的序列并不是真随机，因此它的每一个数完全由一个初始值决定，这个初始值被称为随机种子（seed种子有时使用接近于真随机的硬件随机数生成器生成）。尽管接近于真随机的序列可以通过硬件随机数生成器生成，但伪随机数生成器因为其生成速度和可再现的优势，在实践中显得尤为重要。

2.PRNG的重要性以及应用

那么PRNG到底有什么研究意义呢，我了解到PRNG有以下重要性以及应用：

> **1. 支撑现代计算的基础**
>
> PRNG 是计算机中生成随机数的核心工具，广泛应用于各种算法和系统中。它的存在弥补了计算机作为确定性系统无法生成真正随机数的缺陷。例如：
>
> - **模拟与建模**：在科学研究中，蒙特卡洛模拟广泛用于气候预测、金融风险评估等复杂系统的建模，而这些都依赖高质量的伪随机数。
> - **算法设计**：随机化算法（如快速排序、哈希表）在提高算法效率和稳定性方面起到关键作用。
>
> **2. 密码学中的核心角色**
>
> 在密码学中，随机数的安全性直接关系到整个系统的可靠性。PRNG 生成的密钥、初始化向量等必须具有高随机性和不可预测性。
>
> - **密钥生成**：高质量的随机数确保加密密钥的安全性，防止暴力破解。

> • **通信协议**：在 SSL/TLS 协议中，随机数用于建立安全会话，保护数据传输的机密性。
>
> **3. 数据分析与人工智能的驱动**
>
> PRNG 对于大规模数据分析和人工智能模型的训练至关重要。
>
> • **训练数据的随机抽样**：确保样本分布的多样性，提高 AI 模型的泛化能力。
>
> • **随机梯度下降（SGD）算法**：PRNG 在优化算法中用于随机选择数据点，加快收敛速度，提升模型性能。
>
> **4. 支持游戏与娱乐产业**
>
> 在游戏开发中，随机性是丰富游戏体验的重要元素。
>
> • **随机生成内容**：如地图生成、任务奖励等，依赖 PRNG 实现玩家体验的多样性和公平性。
>
> • **增强现实与虚拟现实**：在动态场景生成和互动设计中需要高效的随机数支持。
>
> **5. 测试与验证工具**
>
> PRNG 在测试和验证系统中起到关键作用。
>
> • **软件测试**：随机生成的输入可以有效覆盖各种场景，帮助发现潜在的漏洞和边界条件问题。
>
> • **硬件验证**：对处理器和网络设备的压力测试需要大量高质量的伪随机数据。
>
> **6. 创新与前沿技术的推动**
>
> 随着量子计算和混沌理论的发展，PRNG 的研究促进了新型随机生成器的出现，例如量子随机数生成器 (QRNG)。这些创新扩展了随机数的应用边界，为密码学和高精度模拟提供了更强大的支持。

确实在我们所学习的DSAA中，如果使用随机数来寻找快速排序的“标准”，我们的时间复杂度便会大幅降低，收获到一个更加优秀的算法。

😨**但回顾到这里，我的脑海中也冒出了一个新的问题：既然这样的随机数是能够通过seed来确定性地产生一个序列的，那它怎么在密码学中有深远的应用？又或者说，除了我们课上讲的想要根据加密结果推算出原来的密码非常困难，但是拥有密码能够轻易的算出加密结果这样通过算法复杂度来“卡”住别人，我们不能做到真正的随机吗？**

💡 **带着这样的问题，我先了解了更多的PRNG算法，直到我对于各种语言的API或者method有所研究，这个问题就迎刃而解了。**

二、不同方法的“随机性”和数学基础

1.线性同余生成器 (LCG)

1）介绍

A **linear congruential generator** (**LCG**) is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. The method represents one of the oldest and best-known pseudorandom number generator algorithms. The theory behind them is relatively easy to understand, and they are easily implemented and fast, especially on computer hardware which can provide modular arithmetic by storage-bit truncation.

> 应用自：https://en.wikipedia.org/wiki/Linear_congruential_generator

2）数学基础

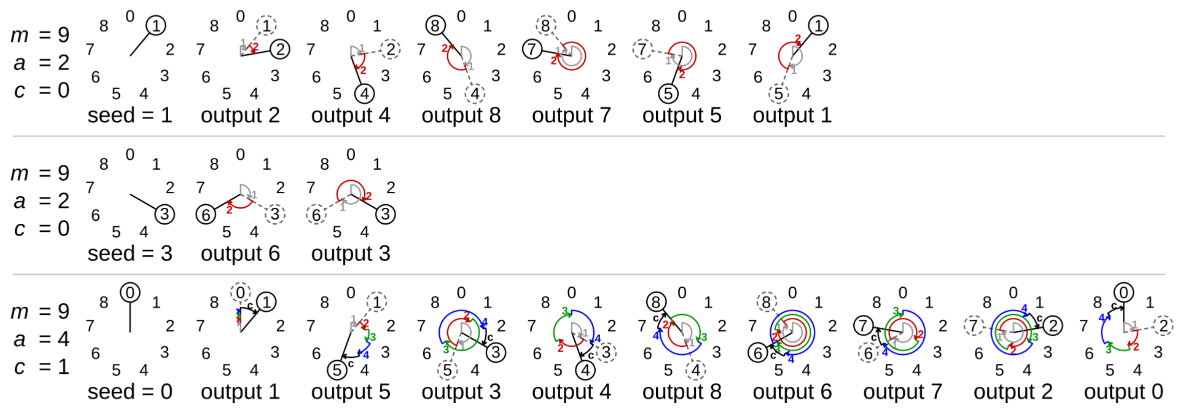
The generator is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \bmod m$$

这里的 X_0就是我们的seed

这样确实能够在一定程度上模拟出一系列1 ~ m的随机数序列，但是本质上还并不是随机的。

这里有一个简单的图表示例：



3）示例

这里有一个模拟LCG的简单的python代码：

```python
# Example parameters for LCG
def lcg(seed, a, c, m, n):
    values = []
    x = seed
    for _ in range(n):
        x = (a * x + c) % m
        values.append(x)
    return values

seed = 42
a = 1664525
c = 1013904223
m = 2**32
n = 10
print(lcg(seed, a, c, m, n))
```

2.梅森旋转法 (Mersenne Twister)

1）介绍

The Mersenne Twister was designed specifically to rectify most of the flaws found in older PRNGs.

The most commonly used version of the Mersenne Twister algorithm is based on the Mersenne prime. The standard implementation of that, MT19937, uses a 32-bit word length. There is another implementation (with five variants) that uses a 64-bit word length, MT19937-64; it generates a different sequence.

> 引用于：Mersenne Twister - Wikipedia

2）数学基础

首先这个算法有一个前置知识需要我们了解：

k-distribution

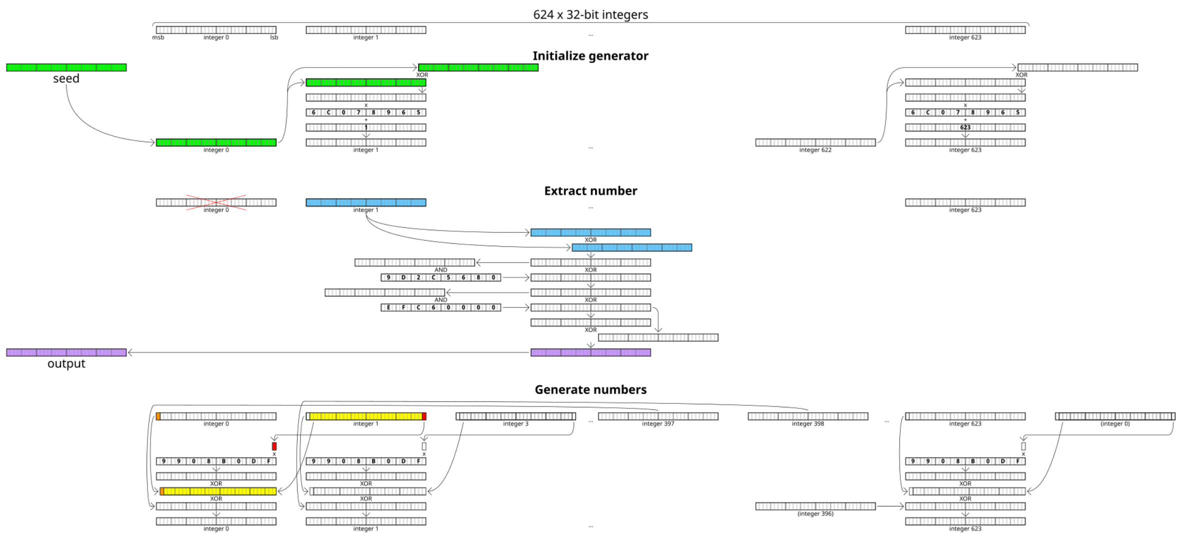
A pseudorandom sequence x_i of *w*-bit integers of period *P* is said to be *k-distributed* to *v*-bit accuracy if the following holds.

Let trunc$_v$(*x*) denote the number formed by the leading *v* bits of *x*, and consider *P* of the *k* *v*-bit vectors$(\mathrm{trunc}_v(x_i), \mathrm{trunc}_v(x_{i+1}), \ldots, \mathrm{trunc}_v(x_{i+k-1})) \quad (0 \le i < P)$.

Then each of the 2^{kv} possible combinations of bits occurs the same number of times in a period, except for the all-zero combination that occurs once less often.

这个算法的具体内容如下：

Algorithmic detail



Visualisation of generation of pseudo-random 32-bit integers using a Mersenne Twister. The 'Extract number' section shows an example where integer 0 has already been output and the index is at integer 1. 'Generate numbers' is run when all integers have been output.

For a *w*-bit word length, the Mersenne Twister generates integers in the range $[0, 2^w - 1]$.

The Mersenne Twister algorithm is based on a matrix linear recurrence over a finite binary field $\mathbf{F}_2$. The algorithm is a twisted generalised feedback shift register[4] (twisted GFSR, or TGFSR) of rational normal form (TGFSR(R)), with state bit reflection and tempering. The basic idea is to define a series x_i through a simple recurrence relation, and then output numbers of the form x_i^T, where *T* is an invertible $\mathbf{F}_2$-matrix called a tempering matrix.

The general algorithm is characterized by the following quantities:

- *w*: word size (in number of bits)
- *n*: degree of recurrence
- *m*: middle word, an offset used in the recurrence relation defining the series x, $1 \leq m < n$
- *r*: separation point of one word, or the number of bits of the lower bitmask, $0 \leq r \leq w - 1$
- *a*: coefficients of the rational normal form twist matrix
- *b*, *c*: TGFSR(R) tempering bitmasks
- *s*, *t*: TGFSR(R) tempering bit shifts
- *u*, *d*, *l*: additional Mersenne Twister tempering bit shifts/masks

with the restriction that $2^{nw-r} - 1$ is a Mersenne prime. This choice simplifies the primitivity test and k-distribution test that are needed in the parameter search.

The series x is defined as a series of *w*-bit quantities with the recurrence relation:

$$x_{k+n} := x_{k+m} \oplus \left(({x_k}^u \mid {x_{k+1}}^l) A \right) \qquad k = 0, 1, 2, \ldots$$

where $\mid$ denotes concatenation of bit vectors (with upper bits on the left), $\oplus$ the bitwise exclusive or (XOR), x_k^u means the upper *w* − *r* bits of x_k, and x_{k+1}^l means the lower *r* bits of x_{k+1}.

The subscripts may all be offset by -*n*

$$x_k := x_{k-(n-m)} \oplus \left(({x_{k-n}}^u \mid {x_{k-(n-1)}}^l) A \right) \qquad k = n, n+1, n+2, \ldots$$

where now the LHS, x_k, is the next generated value in the series in terms of values generated in the past, which are on the RHS.

The twist transformation *A* is defined in rational normal form as:

$$A = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots, a_0) \end{pmatrix}$$

with I_{w-1} as the $(w-1)(w-1)$ identity matrix. The rational normal form has the benefit that multiplication by *A* can be efficiently expressed as: (remember that here matrix multiplication is being done in $\mathbf{F}_2$, and therefore bitwise XOR takes the place of addition)

$$\boldsymbol{x}A = \begin{cases} \boldsymbol{x} \gg 1 & x_0 = 0 \\ (\boldsymbol{x} \gg 1) \oplus \boldsymbol{a} & x_0 = 1 \end{cases}$$

where x_0 is the lowest order bit of x.

As like TGFSR(R), the Mersenne Twister is cascaded with a tempering transform to compensate for the reduced dimensionality of equidistribution (because of the choice of *A* being in the rational normal form). Note that this is equivalent to using the matrix *A* where $A = T^{-1} * AT$ for *T* an invertible matrix, and therefore the analysis of characteristic polynomial mentioned below still holds.

As with *A*, we choose a tempering transform to be easily computable, and so do not actually construct *T* itself. This tempering is defined in the case of Mersenne Twister as

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

where x is the next value from the series, y is a temporary intermediate value, and z is the value returned from the algorithm, with $\ll$and $\gg$ as the bitwise left and right shifts, and $\&$ as the bitwise AND. The first and last transforms are added in order to improve lower-bit equidistribution. From the property of TGFSR, $s + t \geq \left\lfloor \frac{w}{2} \right\rfloor - 1$ is required to reach the upper bound of equidistribution for the upper bits.

The coefficients for MT19937 are:

$$\begin{aligned}(w, n, m, r) &= (32, 624, 397, 31) \\ a &= \mathrm{9908B0DF}_{16} \\ (u, d) &= (11, \mathrm{FFFFFFFF}_{16}) \\ (s, b) &= (7, \mathrm{9D2C5680}_{16}) \\ (t, c) &= (15, \mathrm{EFC60000}_{16}) \\ l &= 18\end{aligned}$$

Note that 32-bit implementations of the Mersenne Twister generally have *d* = $\mathrm{FFFFFFFF}_{16}$. As a result, the *d* is occasionally omitted from the algorithm description, since the bitwise and with *d* in that case has no effect.

The coefficients for MT19937-64 are:[5]

$$\begin{aligned}(w, n, m, r) &= (64, 312, 156, 31) \\ a &= \mathrm{B5026F5AA96619E9}_{16} \\ (u, d) &= (29, \mathrm{5555555555555555}_{16}) \\ (s, b) &= (17, \mathrm{71D67FFFEDA60000}_{16}) \\ (t, c) &= (37, \mathrm{FFF7EEE000000000}_{16}) \\ l &= 43\end{aligned}$$

Initialization

The state needed for a Mersenne Twister implementation is an array of *n* values of *w* bits each. To initialize the array, a *w*-bit seed value is used to supply x_0 through x_{n-1} by setting x_0 to the seed value and thereafter setting

$$x_i = f \times (x_{i-1} \oplus (x_{i-1} \gg (w - 2))) + i$$

for i from 1 to $n - 1$.

- The first value the algorithm then generates is based on x_n, not on x_0.
- The constant *f* forms another parameter to the generator, though not part of the algorithm proper.

- The value for *f* for MT19937 is 1812433253.
- The value for *f* for MT19937-64 is 6364136223846793005.[5]

3.算法示例

这里也有一个简单的C语言代码来模拟Mersenne Twister：

```
#include <stdint.h>
#define n 624
#define m 397
#define w 32
#define r 31
#define UMASK (0xffffffffUL << r)
#define LMASK (0xffffffffUL >> (w-r))
#define a 0x9908b0dfUL
#define u 11
#define s 7
#define t 15
#define l 18
#define b 0x9d2c5680UL
#define c 0xefc60000UL
#define f 1812433253UL
typedef struct
{
    uint32_t state_array[n];         // the array for the state vector
    int state_index;                 // index into state vector array, 0 <=
state_index <= n-1   always
} mt_state;
void initialize_state(mt_state* state, uint32_t seed)
{
    uint32_t* state_array = &(state->state_array[0]);
    state_array[0] = seed;                          // suggested initial seed =
19650218UL
    for (int i=1; i<n; i++)
    {
        seed = f * (seed ^ (seed >> (w-2))) + i;    // Knuth TAOCP Vol2. 3rd Ed.
P.106 for multiplier.
        state_array[i] = seed;
    }
    state->state_index = 0;
}
uint32_t random_uint32(mt_state* state)
{
    uint32_t* state_array = &(state->state_array[0]);
    int k = state->state_index;      // point to current state location
```

```
                                      // 0 <= state_index <= n-1   always
//  int k = k - n;                    // point to state n iterations before
//  if (k < 0) k += n;                // modulo n circular indexing
                                      // the previous 2 lines actually do nothing
                                      //  for illustration only

    int j = k - (n-1);                // point to state n-1 iterations before
    if (j < 0) j += n;                // modulo n circular indexing
    uint32_t x = (state_array[k] & UMASK) | (state_array[j] & LMASK);
    uint32_t xA = x >> 1;
    if (x & 0x00000001UL) xA ^= a;
    j = k - (n-m);                    // point to state n-m iterations before
    if (j < 0) j += n;                // modulo n circular indexing
    x = state_array[j] ^ xA;          // compute next value in the state
    state_array[k++] = x;             // update new state value
    if (k >= n) k = 0;                // modulo n circular indexing
    state->state_index = k;
    uint32_t y = x ^ (x >> u);        // tempering
             y = y ^ ((y << s) & b);
             y = y ^ ((y << t) & c);
    uint32_t z = y ^ (y >> l);
    return z;
}
```

4.Comparison with classical GFSR

In order to achieve the $2^{nw-r}-1$ theoretical upper limit of the period in a TGFSR, $\phi_B(t)$ must be a primitive polynomial, $\phi_B(t)$ being the characteristic polynomial of

$$B = \begin{pmatrix} 0 & I_w & \cdots & 0 & 0 \\ \vdots & & & & \\ I_w & \vdots & \ddots & \vdots & \vdots \\ \vdots & & & & \\ 0 & 0 & \cdots & I_w & 0 \\ 0 & 0 & \cdots & 0 & I_{w-r} \\ S & 0 & \cdots & 0 & 0 \end{pmatrix} \leftarrow m\text{-th row}$$

$$S = \begin{pmatrix} 0 & I_r \\ I_{w-r} & 0 \end{pmatrix} A$$

The twist transformation improves the classical GFSR with the following key properties:

- The period reaches the theoretical upper limit $2^{nw-r}-1$ (except if initialized with 0)

- Equidistribution in *n* dimensions (e.g. linear congruential generators can at best manage reasonable distribution in five dimensions)

3.Cryptographically secure pseudorandom number generator

1.介绍

> 在wiki百科（Cryptographically secure pseudorandom number generator - Wikipedia）中有这样的介绍：

A **cryptographically secure pseudorandom number generator** (**CSPRNG**) or **cryptographic pseudorandom number generator** (**CPRNG**) is a pseudorandom number generator (PRNG) with properties that make it suitable for use in cryptography. It is also referred to as a **cryptographic random number generator** (**CRNG**).

Most cryptographic applications require random numbers, for example:

- key generation
- initialization vectors
- nonces
- salts in certain signature schemes, including ECDSA and RSASSA-PSS
- token generation

The "quality" of the randomness required for these applications varies. For example, creating a nonce in some protocols needs only uniqueness. On the other hand, the generation of a master key requires a higher quality, such as more entropy. And in the case of one-time pads, the information-theoretic guarantee of perfect secrecy only holds if the key material comes from a true random source with high entropy, and thus any kind of pseudorandom number generator is insufficient.

Ideally, the generation of random numbers in CSPRNGs uses entropy obtained from a high-quality source, generally the operating system's randomness API. However, unexpected correlations have been found in several such ostensibly independent processes. From an information-theoretic point of view, the amount of randomness, the entropy that can be generated, is equal to the entropy provided by the system. But sometimes, in practical situations, numbers are needed with more randomness than the available entropy can provide. Also, the processes to extract randomness from a running system are slow in actual practice. In such instances, a CSPRNG can sometimes be used. A CSPRNG can "stretch" the available entropy over more bits.

2.数学基础

Requirements

The requirements of an ordinary PRNG are also satisfied by a cryptographically secure PRNG, but the reverse is not true. CSPRNG requirements fall into two groups:

1. They pass statistical randomness tests:

   - Every CSPRNG should satisfy the next-bit test. That is, given the first k bits of a random sequence, there is no polynomial-time algorithm that can predict the (k+1)th bit with probability of success non-negligibly better than 50%.[1] Andrew Yao proved in 1982 that a generator passing the next-bit test will pass all other polynomial-time statistical tests for randomness.[2]

2. They hold up well under serious attack, even when part of their initial or running state becomes available to an attacker: [3]

   - Every CSPRNG should withstand "state compromise extension attacks".[3]^:4^ In the event that part or all of its state has been revealed (or guessed correctly), it should be impossible to reconstruct the stream of random numbers prior to the revelation. Additionally, if there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the CSPRNG state.

For instance, if the PRNG under consideration produces output by computing bits of pi in sequence, starting from some unknown point in the binary expansion, it may well satisfy the next-bit test and thus be statistically random, as pi is conjectured to be a normal number. However, this algorithm is not cryptographically secure; an attacker who determines which bit of pi is currently in use (i.e. the state of the algorithm) will be able to calculate all preceding bits as well.

Most PRNGs are not suitable for use as CSPRNGs and will fail on both counts. First, while most PRNGs' outputs appear random to assorted statistical tests, they do not resist determined reverse engineering. Specialized statistical tests may be found specially tuned to such a PRNG that shows the random numbers not to be truly random. Second, for most PRNGs, when their state has been revealed, all past random numbers can be retrodicted, allowing an attacker to read all past messages, as well as future ones.

CSPRNGs are designed explicitly to resist this type of cryptanalysis.

Definitions

In the asymptotic setting, a family of deterministic polynomial time computable functions $G_k\colon \{0,1\}^k \to \{0,1\}^{p(k)}$ for some polynomial p, is a pseudorandom number generator (PRNG, or PRG in some references), if it stretches the length of its input ($p(k) > k$ for any k), and if its output is computationally indistinguishable from true randomness, i.e. for any probabilistic polynomial time algorithm A, which outputs 1 or 0 as a distinguisher,

$$\left| \Pr_{x \leftarrow \{0,1\}^k} [A(G(x)) = 1] - \Pr_{r \leftarrow \{0,1\}^{p(k)}} [A(r) = 1] \right| < \mu(k)$$

for some negligible function μ.[4] (The notation $x \leftarrow X$ means that x is chosen uniformly at random from the set X.)

There is an equivalent characterization: For any function family $G_k \colon \{0,1\}^k \to \{0,1\}^{p(k)}$, G is a PRNG if and only if the next output bit of G cannot be predicted by a polynomial time algorithm.[5]

A **forward-secure** PRNG with block length $t(k)$ is a PRNG $G_k \colon \{0,1\}^k \to \{0,1\}^k \times \{0,1\}^{t(k)}$, where the input string s_i with length k is the current state at period i, and the output (s_{i+1}, y_i) consists of the next state s_{i+1} and the pseudorandom output block y_i of period i, that withstands state compromise extensions in the following sense. If the initial state s_1 is chosen uniformly at random from $\{0,1\}^k$, then for any i, the sequence $(y_1, y_2, \dots, y_i, s_{i+1})$ must be computationally indistinguishable from $(r_1, r_2, \dots, r_i, s_{i+1})$, in which the r_i are chosen uniformly at random from $\{0,1\}^{t(k)}$.[6]

Any PRNG $G \colon \{0,1\}^k \to \{0,1\}^{p(k)}$ can be turned into a forward secure PRNG with block length $p(k) - k$ by splitting its output into the next state and the actual output. This is done by setting $G(s) = G_0(s) \| G_1(s)$, in which $|G_0(s)| = |s| = k$ and $|G_1(s)| = p(k) - k$; then G is a forward secure PRNG with G_0 as the next state and G_1 as the pseudorandom output block of the current period.

Entropy extraction

Main article: Randomness extractor

Santha and Vazirani proved that several bit streams with weak randomness can be combined to produce a higher-quality, quasi-random bit stream.[7] Even earlier, John von Neumann proved that a simple algorithm can remove a considerable amount of the bias in any bit stream,[8] which should be applied to each bit stream before using any variation of the Santha–Vazirani design.

Designs

CSPRNG designs are divided into two classes:

1. Designs based on cryptographic primitives such as ciphers and cryptographic hashes
2. Designs based on mathematical problems thought to be hard

Designs based on cryptographic primitives

- A secure block cipher can be converted into a CSPRNG by running it in counter mode using, for example, a special construct that the NIST in SP 800-90A calls CTR_DRBG. CTR_DBRG typically uses Advanced Encryption Standard (AES).
  - AES-CTR_DRBG is often used as a random number generator in systems that use AES encryption.[9][10]
  - The NIST CTR_DRBG scheme erases the key *after* the requested randomness is output by running additional cycles. This is wasteful from a performance perspective, but does not immediately cause issues with forward secrecy. However, realizing the performance implications, the NIST recommends an "extended AES-CTR-DRBG interface" for its Post-Quantum Cryptography Project submissions. This interface allows multiple sets of randomness to be generated without intervening erasure, only erasing when the user explicitly signals the end of requests. As a result, the key could remain in memory for an extended time if the "extended interface" is misused. Newer "fast-key-erasure" RNGs erase the key with randomness as soon as randomness is requested.[11]
- A stream cipher can be converted into a CSPRNG. This has been done with RC4, ISAAC, and ChaCha20, to name a few.
- A cryptographically secure hash might also be a base of a good CSPRNG, using, for example, a construct that NIST calls Hash_DRBG.
- An HMAC primitive can be used as a base of a CSPRNG, for example, as part of the construct that NIST calls HMAC_DRBG.

Number-theoretic designs

- The Blum Blum Shub algorithm has a security proof based on the difficulty of the quadratic residuosity problem. Since the only known way to solve that problem is to factor the modulus, it is generally regarded that the difficulty of integer factorization provides a conditional security proof for the Blum Blum Shub algorithm. However the algorithm is very inefficient and therefore impractical unless extreme security is needed.
- The Blum–Micali algorithm has a security proof based on the difficulty of the discrete logarithm problem but is also very inefficient.
- Daniel Brown of Certicom wrote a 2006 security proof for Dual EC DRBG, based on the assumed hardness of the Decisional Diffie–Hellman assumption, the *x-logarithm problem*, and the *truncated point problem*. The 2006 proof explicitly assumes a lower *outlen* (amount of bits provided per iteration) than in the Dual_EC_DRBG standard, and that the *P* and *Q* in the Dual_EC_DRBG standard (which were revealed in 2013 to be probably backdoored by NSA) are replaced with non-backdoored values.

Practical schemes

"Practical" CSPRNG schemes not only include an CSPRNG algorithm, but also a way to initialize ("seed") it while keeping the seed secret. A number of such schemes have been defined, including:

- Implementations of /dev/random in Unix-like systems.
  - Yarrow, which attempts to evaluate the entropic quality of its seeding inputs, and uses SHA-1 and 3DES internally. Yarrow was used in macOS and other Apple OS' up until about December 2019, after which it switched to Fortuna.
  - Fortuna, the successor to Yarrow, which does not attempt to evaluate the entropic quality of its inputs; it uses SHA-256 and "any good block cipher". Fortuna is used in FreeBSD. Apple changed to Fortuna for most or all Apple OSs beginning around Dec. 2019.
  - The Linux kernel CSPRNG, which uses ChaCha20 to generate data,[12] and BLAKE2s to ingest entropy.[13]
- arc4random, a CSPRNG in Unix-like systems that seeds from /dev/random. It originally is based on RC4, but all main implementations now use ChaCha20.[14][15] [16]
- CryptGenRandom, part of Microsoft's CryptoAPI, offered on Windows. Different versions of Windows use different implementations.
- ANSI X9.17 standard (*Financial Institution Key Management (wholesale)*), which has been adopted as a FIPS standard as well. It takes as input a TDEA (keying option 2) key bundle *k* and (the initial value of) a 64-bit random seed *s*.[17] Each time a random number is required, it executes the following steps:
  1. Obtain the current date/time *D* to the maximum resolution possible.
  2. Compute a temporary value $t = TDEA_k(D)$.
  3. Compute the random value $x = TDEA_k(s \oplus t)$, where $\oplus$ denotes bitwise exclusive or.
  4. Update the seed $s = TDEA_k(x \oplus t)$.

Obviously, the technique is easily generalized to any block cipher; AES has been suggested.[18] If the key *k* is leaked, the entire X9.17 stream can be predicted; this weakness is cited as a reason for creating Yarrow.[19]

All these above-mentioned schemes, save for X9.17, also mix the state of a CSPRNG with an additional source of entropy. They are therefore not "pure" pseudorandom number generators, in the sense that the output is not completely determined by their initial state. This addition aims to prevent attacks even if the initial state is compromised.[a]

4.总结

前面LCG和Mersenne Twister这两个算法都较为容易理解，但是到了CSPRNG，其复杂性和于密码学的研究大大加深，我浅薄的数学基础让我无法完全理解其内容，但我明白了PRNG在密码学的基本相关场景中的应用，也明白了这三个算法之间的不同点和不同的应用场景：

1. **LCG**: 用于较为简单的随机数需求（如游戏中的非加密随机事件）。
2. **Mersenne Twister**: 用于科学计算、统计模拟等需要高质量随机数的场景。
3. **CSPRNG**: 用于密码学相关场景（如生成密钥、初始化向量、随机密码等），能够确保安全性和不可预测性。

到了这里，我就基本上能够明白我们能够使用什么方法来有效的在密码学中应用我们的PRNG了，但是之后在不同语言API中的研究，让我更加深刻理解到了PRNG应用的本质。

三、不同语言应用的方法

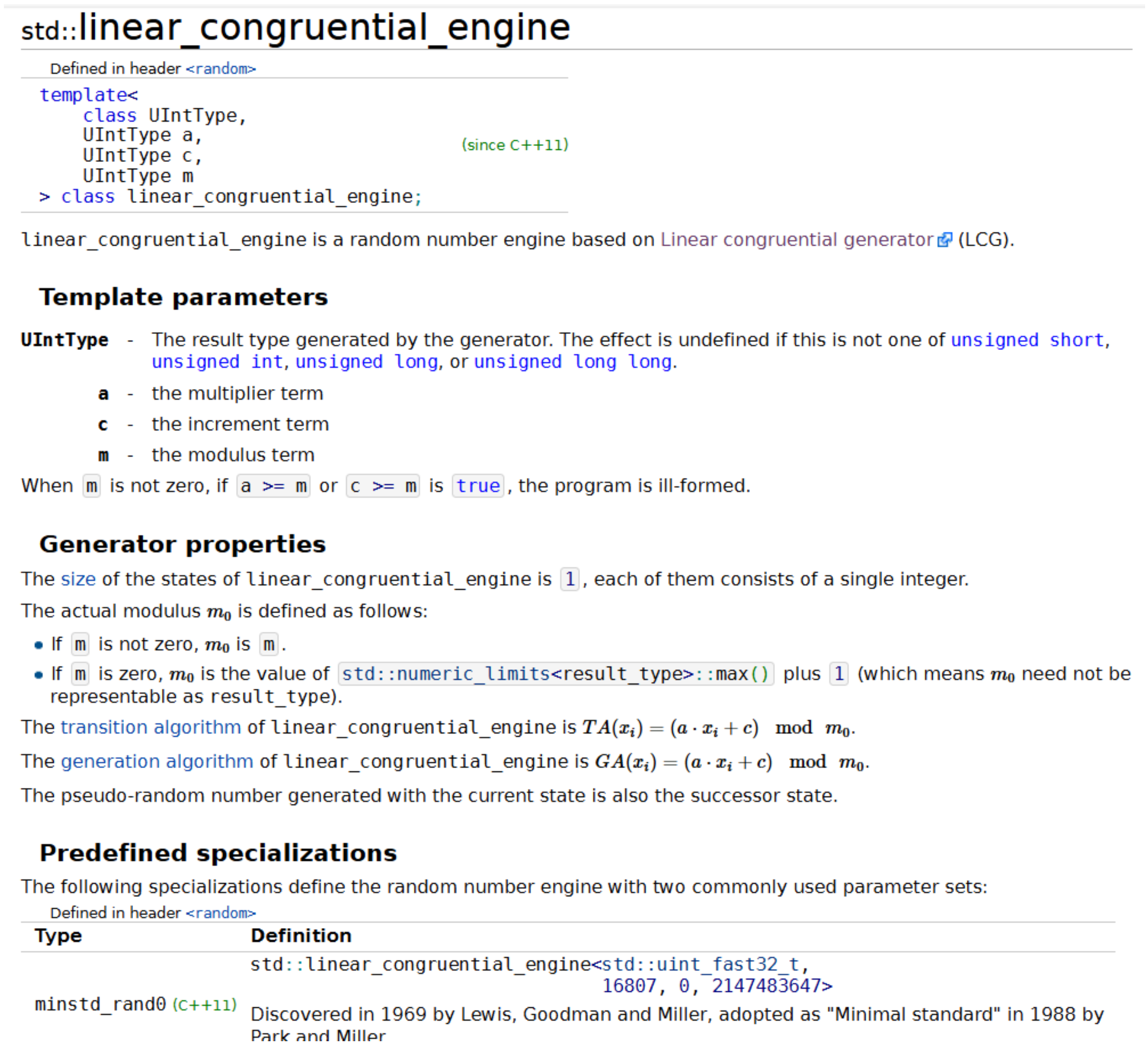
1.C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    // 初始化随机数发生器
    srand((unsigned int)time(NULL));
    // 生成并打印10个随机数
    for (int i = 0; i < 10; i++) {
        int random_number = rand();
        printf("%d\n", random_number);
    }
    return 0;
}
```

`rand()` 函数通常实现一个线性同余生成器（LCG）。`srand()` 函数用于设置随机数生成器的种子。

我在CppReference中找到了对于相关API的介绍[std::linear_congruential_engine - cppreference.com]

在C++11中就有了相关的算法引擎。



std::**linear_congruential_engine**

Defined in header `<random>`

```
template<
    class UIntType,
    UIntType a,
    UIntType c,
    UIntType m
> class linear_congruential_engine;
```
(since C++11)

`linear_congruential_engine` is a random number engine based on Linear congruential generator (LCG).

Template parameters

UIntType	-	The result type generated by the generator. The effect is undefined if this is not one of `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.
a	-	the multiplier term
c	-	the increment term
m	-	the modulus term

When `m` is not zero, if `a >= m` or `c >= m` is `true`, the program is ill-formed.

Generator properties

The size of the states of `linear_congruential_engine` is `1`, each of them consists of a single integer.

The actual modulus m_0 is defined as follows:

- If `m` is not zero, m_0 is `m`.
- If `m` is zero, m_0 is the value of `std::numeric_limits<result_type>::max()` plus `1` (which means m_0 need not be representable as `result_type`).

The transition algorithm of `linear_congruential_engine` is $TA(x_i) = (a \cdot x_i + c) \mod m_0$.

The generation algorithm of `linear_congruential_engine` is $GA(x_i) = (a \cdot x_i + c) \mod m_0$.

The pseudo-random number generated with the current state is also the successor state.

Predefined specializations

The following specializations define the random number engine with two commonly used parameter sets:

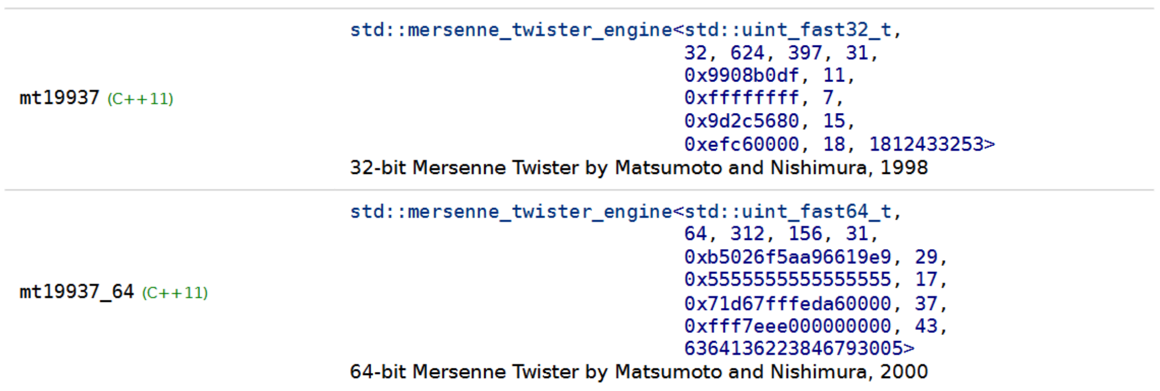
Defined in header `<random>`

Type	Definition
`minstd_rand0` (C++11)	`std::linear_congruential_engine<std::uint_fast32_t, 16807, 0, 2147483647>` Discovered in 1969 by Lewis, Goodman and Miller, adopted as "Minimal standard" in 1988 by Park and Miller

2.C++

```
#include <iostream>
#include <random>
int main() {
    // 创建一个Mersenne Twister随机数生成器
    std::mt19937 mt(std::random_device{}());
    // 生成并打印10个随机数
    for (int i = 0; i < 10; i++) {
        int random_number = mt();
        std::cout << random_number << std::endl;
    }
    return 0;
}
```

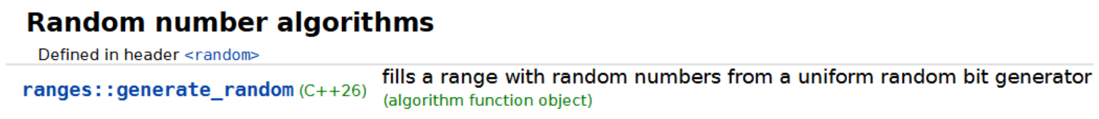
在C++11 引入了 `<random>` 头文件，其中包含了 `std::mt19937` 和 `std::mt19937_64` ，这是 Mersenne Twister 算法的实现。



mt19937 (C++11)	`std::mersenne_twister_engine<std::uint_fast32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253>` 32-bit Mersenne Twister by Matsumoto and Nishimura, 1998
mt19937_64 (C++11)	`std::mersenne_twister_engine<std::uint_fast64_t, 64, 312, 156, 31, 0xb5026f5aa96619e9, 29, 0x5555555555555555, 17, 0x71d67fffeda60000, 37, 0xfff7eee000000000, 43, 6364136223846793005>` 64-bit Mersenne Twister by Matsumoto and Nishimura, 2000

> Reference：[Pseudo-random number generation - cppreference.com]

在这个reference中，又更多对于Cpp中PRNG的API的描述，逐渐发展到C++26，渐渐引入了我们在工程概率统计（计算机另一门概统课程）中接触到的由均匀分布产生的随机数的算法，也直接有了常见分布的算法API：



Random number algorithms

Defined in header `<random>`

`ranges::generate_random` (C++26)	fills a range with random numbers from a uniform random bit generator (algorithm function object)

3.Java-java.util.Random

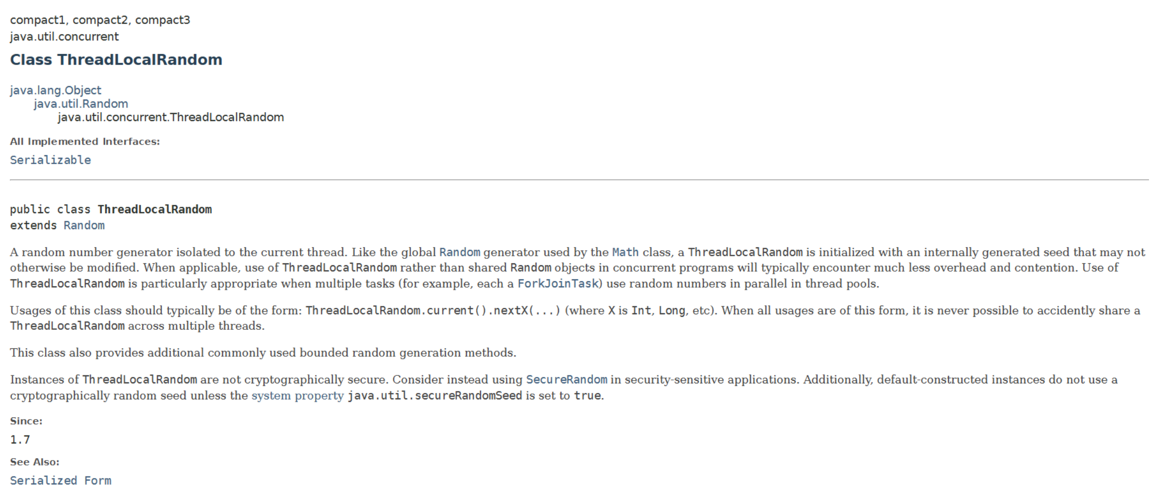
> Random Class:[Random (Java Platform SE 8)]

1) `java.util.ThreadLocalRandom`

```java
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        Random random = new Random();
        // 生成并打印10个随机数
        for (int i = 0; i < 10; i++) {
            int randomNumber = random.nextInt();
            System.out.println(randomNumber);
        }
    }
}
```

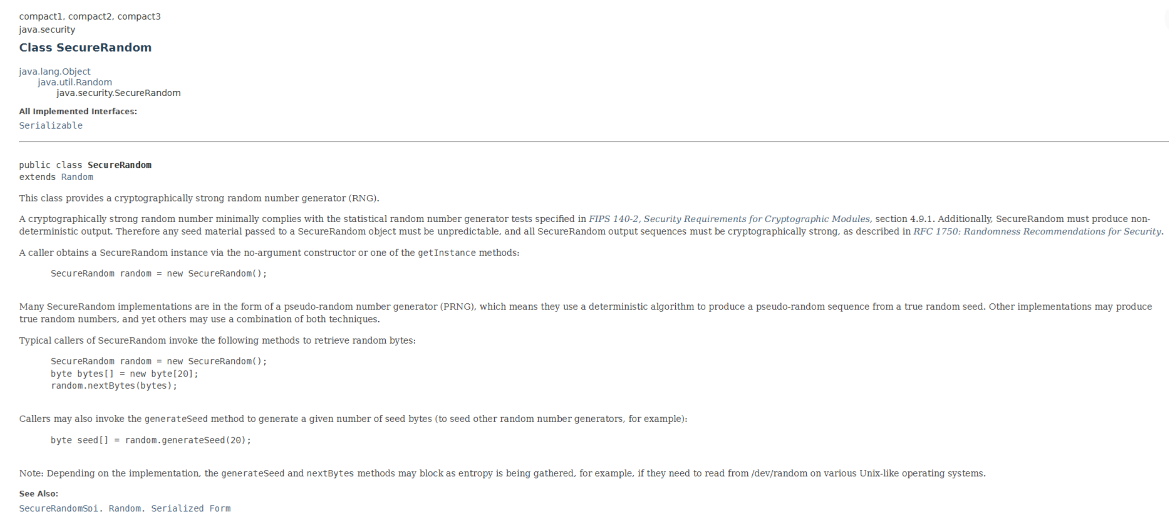
同时，我也在Oracle的官方文档中找到了对于Java ThreadLocalRandom Class的描述，有更多的应用方法，但其本质上还是使用的LCG算法来实现：

> [ThreadLocalRandom (Java Platform SE 8)]()
>
> 
>
> compact1, compact2, compact3
> java.util.concurrent
>
> **Class ThreadLocalRandom**
>
> java.lang.Object
>   java.util.Random
>   java.util.concurrent.ThreadLocalRandom
>
> **All Implemented Interfaces:**
> `Serializable`
>
> `public class ThreadLocalRandom`
> `extends Random`
>
> A random number generator isolated to the current thread. Like the global `Random` generator used by the `Math` class, a `ThreadLocalRandom` is initialized with an internally generated seed that may not otherwise be modified. When applicable, use of `ThreadLocalRandom` rather than shared `Random` objects in concurrent programs will typically encounter much less overhead and contention. Use of `ThreadLocalRandom` is particularly appropriate when multiple tasks (for example, each a `ForkJoinTask`) use random numbers in parallel in thread pools.
>
> Usages of this class should typically be of the form: `ThreadLocalRandom.current().nextX(...)` (where X is `Int`, `Long`, etc). When all usages are of this form, it is never possible to accidently share a `ThreadLocalRandom` across multiple threads.
>
> This class also provides additional commonly used bounded random generation methods.
>
> Instances of `ThreadLocalRandom` are not cryptographically secure. Consider instead using `SecureRandom` in security-sensitive applications. Additionally, default-constructed instances do not use a cryptographically random seed unless the system property `java.util.secureRandomSeed` is set to `true`.
>
> **Since:**
> 1.7
>
> **See Also:**
> `Serialized Form`

2) `java.security.SecureRandom`

```java
import java.security.SecureRandom;
public class Main {
    public static void main(String[] args) {
        SecureRandom secureRandom = new SecureRandom();
        // 生成并打印10个随机数
        for (int i = 0; i < 10; i++) {
            int randomNumber = secureRandom.nextInt();
            System.out.println(randomNumber);
        }
    }
}
```

> `java.security.SecureRandom` 类提供了一个加密安全的伪随机数生成器（CSPRNG）实现。
>
> 来自于：[SecureRandom (Java Platform SE 8)]()


compact1, compact2, compact3
java.security

Class SecureRandom

java.lang.Object
    java.util.Random
        java.security.SecureRandom

All Implemented Interfaces:
`Serializable`

```
public class SecureRandom
extends Random
```

This class provides a cryptographically strong random number generator (RNG).

A cryptographically strong random number minimally complies with the statistical random number generator tests specified in *FIPS 140-2, Security Requirements for Cryptographic Modules*, section 4.9.1. Additionally, SecureRandom must produce non-deterministic output. Therefore any seed material passed to a SecureRandom object must be unpredictable, and all SecureRandom output sequences must be cryptographically strong, as described in *RFC 1750: Randomness Recommendations for Security*.

A caller obtains a SecureRandom instance via the no-argument constructor or one of the `getInstance` methods:

```
SecureRandom random = new SecureRandom();
```

Many SecureRandom implementations are in the form of a pseudo-random number generator (PRNG), which means they use a deterministic algorithm to produce a pseudo-random sequence from a true random seed. Other implementations may produce true random numbers, and yet others may use a combination of both techniques.

Typical callers of SecureRandom invoke the following methods to retrieve random bytes:

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[20];
random.nextBytes(bytes);
```

Callers may also invoke the `generateSeed` method to generate a given number of seed bytes (to seed other random number generators, for example):

```
byte seed[] = random.generateSeed(20);
```

Note: Depending on the implementation, the `generateSeed` and `nextBytes` methods may block as entropy is being gathered, for example, if they need to read from /dev/random on various Unix-like operating systems.

See Also:
`SecureRandomSpi`, `Random`, `Serialized Form`

4.Javascript

The `Math.random()` function is not suitable for cryptographic purposes, as it is not truly random and can be predicted or manipulated by an attacker.

To generate a secure random number in JavaScript, you can use the Web Crypto API. The `Crypto.getRandomValues()` method lets you get cryptographically strong random values. Here is an example code that generates a secure random number:

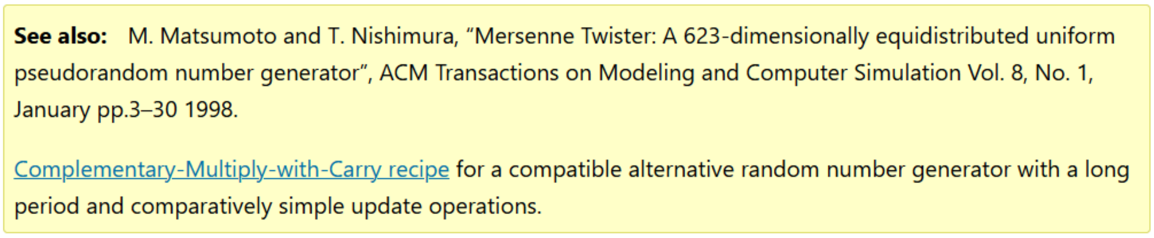
```
function generateRandomNumber() {
  const array = new Uint32Array(1); // Create a 32-bit unsigned integer array
  window.crypto.getRandomValues(array); // Fill the array with random values
  return array[0]; // Return the first element of the array as the random number
}
console.log(generateRandomNumber()); // Output a random number between 0 and
4294967295
```

5.Python

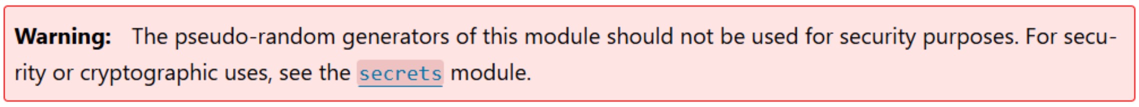
1) `random`

```
import random
# 生成并打印10个随机数
for _ in range(10):
    random_number = random.randint(0, 100)
    print(random_number)
```

Python 的 `random` 模块使用 Mersenne Twister 作为其核心生成器。


See also: M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”, ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

> 同样我也找到了相关的doc介绍地址：
>
> random — Generate pseudo-random numbers — Python 3.13.1 documentation
>
> 其中也说明了random是不能用于密码学加密的：
>
> 
> **Warning:** The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the `secrets` module.
>
> 我们应该使用下面的secrets

2) `secrets`

```python
import secrets
# 生成并打印10个安全随机数
for _ in range(10):
    random_number = secrets.randbelow(100)
    print(random_number)
```

说明：`secrets` 模块提供了对 CSPRNG 函数的访问，适用于管理密码和安全令牌等数据。

> doc：secrets — Generate secure random numbers for managing secrets — Python 3.13.1 documentation

6.R

```r
# 生成并打印10个随机数
for (i in 1:10) {
  random_number <- .Random.seed
  print(random_number)
}
```

R 的默认 PRNG 是 Mersenne Twister。

比如说R语言的随机数便可以用于蒙特卡洛（Monte Carlo）模拟，使用随机数估算 π 的值：

```
estimate_pi <- function(n) {
  count <- 0
  for (i in 1:n) {
    x <- runif(1, min = -1, max = 1)
    y <- runif(1, min = -1, max = 1)
    if (x^2 + y^2 <= 1) {
      count <- count + 1
    }
  }
  return(4 * count / n)
}

cat("Estimated π:", estimate_pi(10000))
```

7.C#-RNGCryptoServiceProvider

```
using System;
using System.Security.Cryptography;
public class MainClass {
    public static void Main() {
        RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
        // 生成并打印10个加密质量的随机数
        for (int i = 0; i < 10; i++) {
            byte[] randomNumber = new byte[4];
            rng.GetBytes(randomNumber);
            int value = BitConverter.ToInt32(randomNumber, 0);
            Console.WriteLine(value);
        }
    }
}
```

> [RandomNumberGenerator 类 (System.Security.Cryptography) | Microsoft Learn]
>
> [.NET 中的加密、签名和哈希算法概述 - .NET | Microsoft Learn]
>
> 在这个文档中我更加清晰地了解到了密码学中公钥加密和私钥加密的区别等

8.总结

同时，我在搜索相关资料的同时看到了这篇博客，让我大受启发，明白了我们究竟要怎么合理地理解并且应用PRNG

> [安全编码 101：如何使用随机函数 - SmartScanner --- Secure Coding 101: How to Use Random Function - SmartScanner]

实际上很多的PRNG都是并不“强”的（比如说LCG），很容易经过黑客的大量枚举或者简单的数学就能解密出来，那么我们应该改使用更加“强”的PRNG，就如上文中的 `java.security.SecureRandom`，Python secrets等等，使用简单的C的random或者其他简单PRNG来源的随机数生成器很可能具有以下风险：

> ### Random Function Vulnerabilities
>
> The seed and random number generation algorithm both can have weaknesses.
>
> ### Prediction
>
> You can say predictability is the main vulnerability of weak random number generators. PRNGs are inherently predictable as they are based on a mathematical formula. So if you know the seed and last random number, you can predict the next random number. This can lead to severe vulnerabilities.
>
> For example, consider a forget password functionality that words based on a random token. If the token can be predicted, an attacker can reset the password of any user.
>
> ### Collision
>
> Besides the predictability, Some random generators that have low quality produce duplicate values very often. This can increase the risk of collision.
>
> Consider an application that generates random session tokens for its users. The chances of producing duplicate session tokens are related to two factors:
>
> - Size of random space (length of session token)
> - Quality of the random selection

> Short-length tokens will have a higher chance of collisions. And bad random generator does not generate all possible values. This can lead to vulnerabilities where a user can see another user’s data.
>
> ### Seed Leakage or Manipulation
>
> Another vulnerability related to random number generators is choosing a weak seed. PRNG always generates the same sequence of numbers with the same initial seed value. If an attacker can find the used seed or manipulate it, he can easily generate same random numbers.
>
> In pratice, hackers use different methods to predict the next random number. Some methods may involve statistical analysis, while others may involve reverse-engineering the generator’s algorithm.

因此，在实际应用中，我们更应该去选择CSPRNG来增强我们的保密性、数据完整性等等。

四、PKCS中PRNG的应用

到了这里，我已经基本上明白了如何在密码学中应用PRNG，下面就是对在PKCS中PRNG应用的进一步了解，我搜索并且阅读了一下几篇blog或是文章，总结出了以下内容：

1.公钥密码体制安全性证明

公钥密码体制安全性证明关键技术及应用研究 - 百度学术

在第一篇文章中我了解到：

> 正如前面所述，公钥密码体制的安全性都是**基于求解某个数学难题**。密码学者将某 一个数学难题设计成**陷门单向函数**，然后**利用陷门单向函数构造有效的公钥密码算法**。 在研究了基础的数学难题之后，本节分析两种与本论文的研究密切相关，也是最常见的 公钥密码体制，及其算法的数学描述。

公钥加密本身是源于一个类似于NP问题，我们难以去解决这个问题，但是容易去验证这个问题。将别人解决没有密钥只有公钥来解决这个问题设定为相当难以解决的数学问题，同时将拥有密钥来解密作为容易解决的数学问题，我们就能够实现公钥的密码体制。

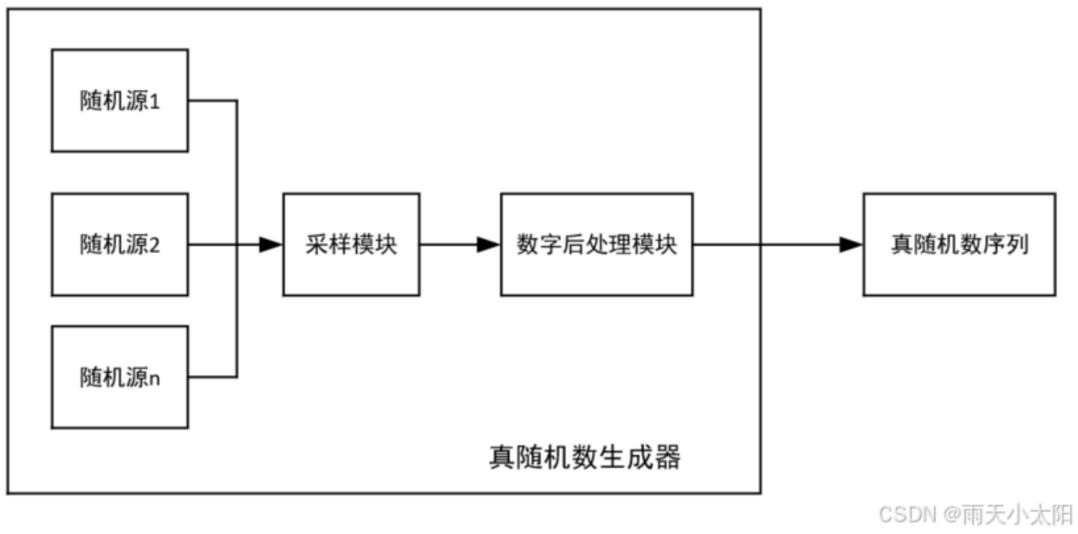
同时这篇文章中也有对于各种公钥密码体制的证明以及改善建议，如RSA，EIGamel、OAEP等等。

2.TRNG-真实随机数生成器

密码学工具6——随机数生成器(PRNG,TRNG设计最全版)_prng trng-CSDN博客

这篇文章中介绍了一种前文未出现过的算法：ANSI X9.17，其本质上也是一种不算太“强”的PRNG，同时我也了解到真实的随机数产生器（TRNG）是应用了我们在数字逻辑中学习的电路知识产生的：

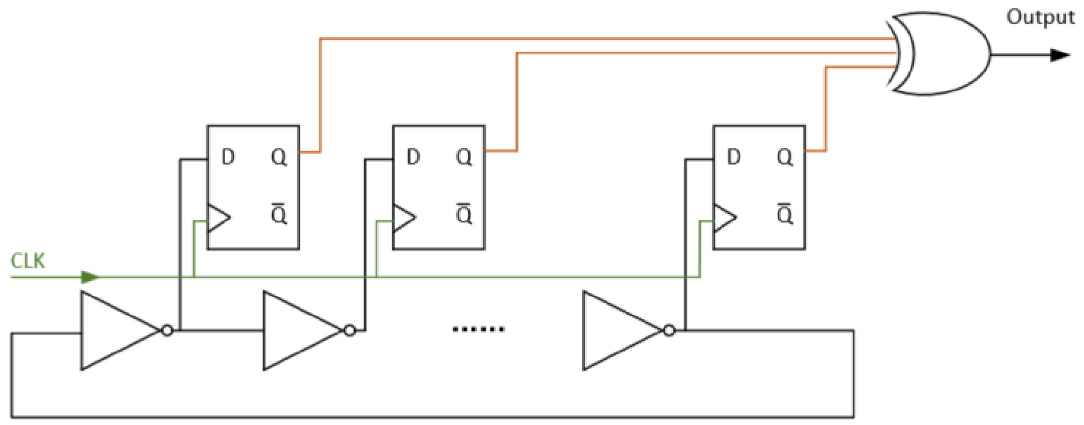
TRNG的整体框架：



整体模块

1）随机源设计

- **核心组件**：由奇数个反相器组成的环形震荡器，它没有稳态，只能在高低电平之间放大状态。
- **工作原理**：当环形震荡器中的某个反相器产生微小正跳变时，会在输出端产生自激振荡，形成周期性振荡信号。
- **采样方式**：使用D触发器作为采样开关，以低频时钟信号的上升沿对高频环形震荡器信号进行采样，利用相位噪声的随机性产生随机序列。



随机源设计

2）数字后处理设计

- **主要模块**：线性反馈移位寄存器（LFSR），用于扰乱随机数序列。
- **工作原理**：
  - 移位寄存器由若干个寄存器组成，每个寄存器存放1bit数据。
  - 每个时钟周期，移位寄存器向右移动一位，左侧空出的位由反馈函数计算结果填充。
  - 反馈函数通常是对寄存器中某些位进行异或操作。
- **特征多项式**：定义了哪些位在反馈函数中参与异或。
- $$f(x) = c_n x^n + c_{n-1} + \ldots + c_2 x^2 + c_1 x + 1$$

  数字后信号处理特征多项式
- **注意事项**：LFSR的值不能全为零，否则状态将不会改变。

3）输出

- 经LSFR数字处理后的随机数列，就是TRNG模块输出的随机数序列。

3.公钥加密应用

在Micrsoft的.NET文档（[.NET 中的加密、签名和哈希算法概述 - .NET | Microsoft Learn]）中我看到了以下内容：

这里有一个通俗易懂的例子能够很好地理解公钥加密是怎么做的：

> 公钥加密使用必须对从未经授权的用户保密的私钥和可以公开给任何人的公钥。 从数学上来讲，公钥和私钥是相互链接的；**使用公钥加密的数据只能用私钥解密，而使用私钥签名的数据只能使用公钥进行验证。 公钥可供任何人使用；它用加密要发到送私钥所有者的数据。 公钥加密算法也称为非对称算法，因为加密数据需要一个密钥，而解密数据需要另一个密钥。** 基本加密规则禁止密钥重复使用，并且每个通信会话的两个密钥应该是独有的。 但是，在实践中，非对称密钥的生存期通常很长。
>
> 双方（Alice 和 Bob）可能按如下所示使用公钥加密：首先，Alice 生成公钥/私钥对。 如果 Bob 想要向 Alice 发送一条已加密的消息，他会向她索要公钥。 Alice 通过非安全网络向 Bob 发送她的公钥，然后 Bob 使用此密钥对消息进行加密。 Bob 将加密的消息发送给 Alice，然后她将使用自己的私钥对其进行解密。 如果 Bob 通过非安全通道收到 Alice 的密钥（例如，公用网络），则 Bob 容易受到中间人攻击。 因此，Bob 必须与 Alice 确认他具有其公钥的正确副本。
>
> 在 Alice 公钥的传输期间，未经授权的代理可能会截获此密钥。 此外，同一代理可能会截获来自 Bob 的加密消息。 不过，此代理不能使用公钥解密此消息。 该消息只能用 Alice 的私钥进行解密，而该私钥没有进行传输。 Alice 不使用她的私钥加密给 Bob 的回复消息，因为任何具有公钥的人都可以解密此消息。 如果 Alice 想要将消息回复给 Bob，她会向 Bob 索要他的公钥，并使用此公钥加密消息。 然后，Bob 将使用自己的关联私钥解密消息。
>
> 在此方案中，Alice 和 Bob 使用**公钥（非对称）加密来传输密钥（对称）**并使用密钥加密对双方会话的其余部分进行加密。

同时我们也能够看到对于公钥加密的应用

> .NET 提供以下实现公钥算法的类：
>
> - RSA
> - ECDsa
> - ECDiffieHellman
> - DSA
>
> RSA 允许加密和签名，但 DSA 仅可用于签名。 DSA 不如 RSA 安全，我们建议使用 RSA。 Diffie-Hellman 只能用于密钥生成。 一般情况下，公钥算法的使用比私钥算法的更受限制。

对于这几个算法，我了解到了PRNG在其中的应用如下：

RSA

- **用途**：RSA算法可以用于加密和数字签名。
- **PRNG的应用**：
  - **密钥生成**：在创建RSA密钥对时，PRNG用于生成大质数，这些质数是构成私钥和公钥的基础。

> - **填充方案**：在加密过程中，某些填充方案（如PKCS#1）需要随机数来生成填充数据，以防止诸如选择明文攻击等攻击手段。

ECDsa

- **用途**：ECDsa算法用于数字签名。
- **PRNG的应用**：

> - **签名生成**：在生成签名时，PRNG用于生成一个随机的nonce（一个只使用一次的数值），这个nonce与私钥和消息哈希结合来生成签名。nonce的随机性对于签名的安全性至关重要。

ECDiffieHellman

- **用途**：ECDiffieHellman算法用于密钥交换，它允许双方在不安全的通道上安全地交换密钥。
- **PRNG的应用**：

> - **私钥生成**：在初始化ECDiffieHellman密钥交换时，PRNG用于生成私钥，私钥随后用于生成公钥。
> - **密钥派生**：在密钥交换过程中，可能会使用PRNG来生成额外的随机数据，以增强派生密钥的安全性。

DSA

- **用途**：DSA算法仅用于数字签名。
- **PRNG的应用**：

> - **签名生成**：与ECDsa类似，DSA在生成签名时也需要一个随机或伪随机的nonce。这个nonce与私钥和消息哈希结合来生成签名。

4.总结

通过阅读更多的文献资料，我总结出以下对于PRNG在PKCS中的应用：

1）公钥密码体制的安全性证明

公钥密码体制自Diffle和Hellman提出以来，已被广泛应用于加密、签名、密钥协商等领域。随着其应用的增加，公钥密码体制的安全性成为了研究的热点。研究者们希望通过**形式化证明的方法来证实公钥密码体制的安全性**，并在此基础上**指导密码体制的设计和应用**。这种“安全性证明”或“可证明安全”的概念应运而生。

2）伪随机数生成器在公钥密码体制中的应用

伪随机数生成器在公钥密码体制中扮演着重要角色，尤其是在密钥生成、初始化向量生成等方面：

1. **密钥生成**：在公钥密码体制中，密钥的随机性对于保证加密的安全性至关重要。PRNG提供了必要的随机性，以确保密钥不易被预测。
2. **初始化向量**：在加密过程中，初始化向量（IV）的使用可以增强加密算法的安全性。PRNG用于生成IV，以防止重放攻击和其他类型的攻击。
3. **安全性和效率的平衡**：公钥密码体制需要在安全性和计算效率之间找到平衡。PRNG的应用有助于在不牺牲太多效率的情况下提高安全性。

3）研究成果和趋势

研究者针对公钥密码体制的安全性证明进行了深入的研究，提出了一系列优化和改进措施。例如，针对RSA和ElGamal等著名公钥密码体制的改进，提出了形式化的安全性证明，这些证明显示改进方案的安全性等价于解决它们所包含的陷门单向函数。

4）结论

伪随机数生成器在公钥密码体制中的应用是确保其安全性的关键因素之一。随着密码学研究的深入，PRNG的应用和优化将继续是提高公钥密码体制安全性的重要研究方向。

五、总结

通过这次project，我**认识到了更多PRNG算法**，更加清楚了**PRNG在不同计算机语言中的应用以及他们之间的区别**，更加深了我**对于PRNG在密码学特别是公钥加密中的应用**。简单的PRNG并不能合理地对于我们的信息进行加密，只有通过**公钥密码安全性证明**等等才能保证我们公钥密码体制的**正确性和安全性**，更加合理地应用在我们信息的传输中。对于现在大数据时代下的我们，我经常能够感受到自己的身份信息、自己这一段时间的喜好被人知晓，更有一种只要他人想要我的信息，我便“**裸露**”地出现在大众面前的感觉，因此，我们更应该重视这些加密过程，探索出更加合理、有效的加密机制，不仅仅是保护好我们的**个人隐私**，也是对于我们**社会**、我们**国家的保护**有着重要的作用。

六、Reference

本文中大部分引用都在具体文段有所展现，可以直接点击链接查看。