伪随机数生成器的原理及应用

谢妍

12412559

January, 2026

前言 在某次工概统的作业中，我编写了一个python程序，试图用蒙特卡洛方法来求解模型。令我震惊的是，一个理论期望不超过个位数的实验，我的程序竟然每次都会跑出同一个几十万的结果。百思不得其解时，出于偶然，我修改了随机数种子，实验结果回到了正常。这也正使我意识到，伪随机其实并不随机。

目录

1 PRNG

伪随机数生成器(PRNG, pseudo random number generator)是根据一个初始值，生成一个近似于随机数序列的算法。与之相对的是真随机随机数生成器(TRNG, true random number generator)，PRNG不依赖于外源而完全基于**算法**，而TRNG依赖于物理不可预测的随机源，如热噪声、放射性衰变、大气噪声。PRNG最致命的缺点无疑是其生成的序列“完全不随机”。冯·诺依曼曾说到：

> **Anyone who attempts to generate random numbers by deterministic means is, of course, living in sin.**

一个确定的算法，无论如何无法生成真正随机性。然而，我们在对安全性要求不高的实践中常常选用PRNG，是因为PRNG不需要经过复杂的硬件和信号处理，速度快、不受环境干扰，甚至是因为其可预测性。

各种随机的定义 正如之前所说，PRNG本质上是在用确定的算法做真随机的近似，也就是实现伪随机。要了解PRNG，我们要从基本定义开始。在计算机领域，伪随机通常指用确定的算法来达到统计上随机的方法。而统计随机，指序列没有可识别的特征或常规，并不具有客观的不可预测性。这并不是说一个有一定特征的序列就不符合统计随机。这里还有两个概念要区分：全局随机(global randomness)和局部随机(local randomness)。想象一个长度为100的序列，其1-10位为0，11-20位为1，21-30位为2，以此类推。统计学上，整个序列各数位的分布是完全均匀的，然而局部上完全不随机。因此，要检验一个PRNG的统计学性质，我们不仅要对其全局的数位分布进行统计，还要对其相邻位的相关性进行检查。

朴素PRNG实现 伪随机序列的生成常常基于数学公式或硬件设备。下面我们给出了四种较为常见的方法以及对于随机性的分析。

1.1 平方取中法

1946年，冯·诺依曼提出**平方取中法(Middle-square method)**，这是一个简单的PRNG，其伪代码如下。

Algorithm 1 Middle-Square Method

Require: digit width n, seed $x_0 \in \{0, 1, \ldots, 10^n - 1\}$, number of outputs T

Ensure: sequence $x_1, x_2, \ldots, x_T$

$B \leftarrow 10^n$ ▷ state space size

$H \leftarrow 10^{n/2}$ ▷ half-width base

for $t \leftarrow 0$ to $T - 1$ **do**

  $y \leftarrow x_t \times x_t$ ▷ square

  $x_{t+1} \leftarrow \left\lfloor \frac{y}{H} \right\rfloor \bmod B$ ▷ take middle n digits

end for

return $(x_1, x_2, \ldots, x_T)$

由于平方取中法本质上是一个确定性状态机，我们可以将它与图论联系起来。把每个状态看作一个顶点，每对 x与 $f(x)$ 以一条有向边表示。每个状态的下一状态完全取决于上一状态，每个点的出度都恰好为 **1**。

周期性 对于这样的图，我们可以证明：其每个连通分量都包含一个环加上若干指向环的入树。利用seed到环的距离与第一次访问已访问过的节点的距离，就可以计算出该环的长度。利用python进行平方取中法的模拟，以所有6位数作为种子，sample space = 1000000进行实验, 结果如下:

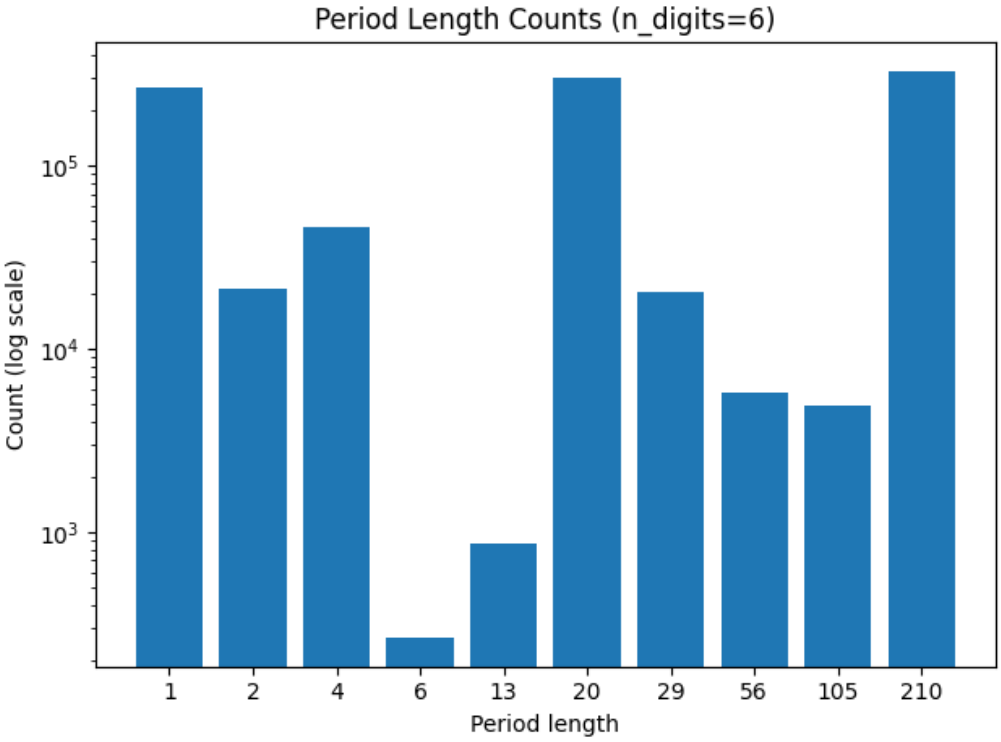

图 1: Period分布

$E[period] = 77.1$

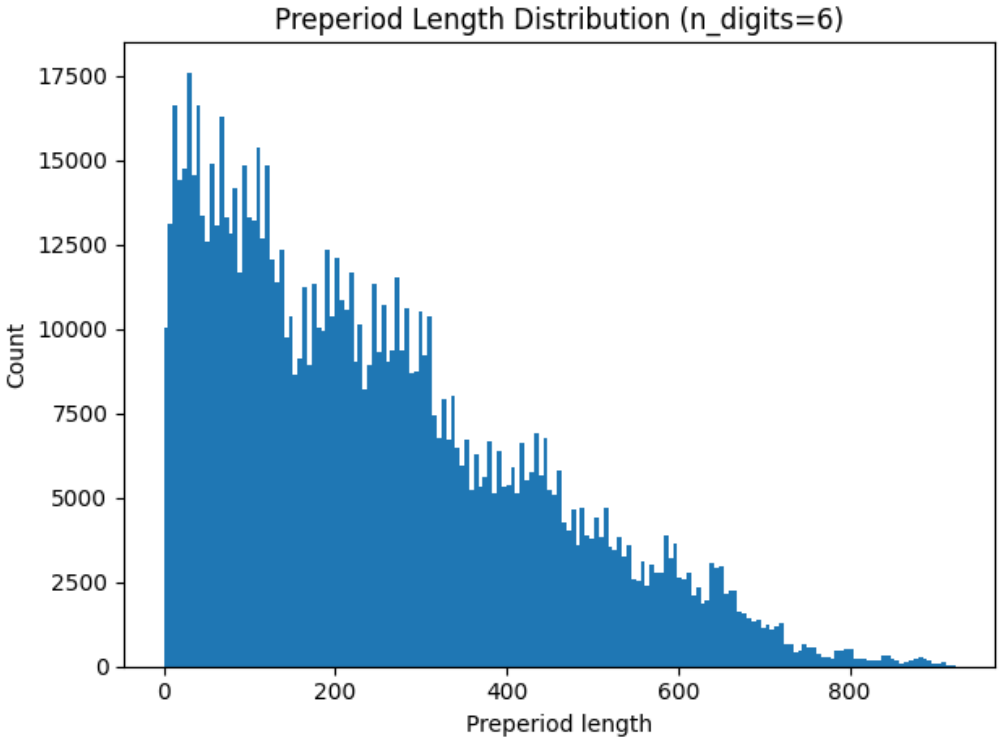

图 2: Preperiod分布

$E[preperiod] = 249.1$

环的数量：

表 1: Cycle length distribution (middle-square, $n = 6$)

Cycle length	# cycles
1	6
2	1
4	2
6	1
13	1
20	2
29	1
56	1
105	1
210	1

可以看到，平方取中法的周期几乎总是很短，对应平方取中法的图中，小环应该占比极大。而且取中间n位的操作会丢失所有高位和低位信息，大量不同的x会映射到同一x'，造成环上的入度较大。对于大多数种子(图的起点)，很快会收缩映射(many-to-one)到某个环。这两点造成了平方取中法短周期以及弱覆盖的性质。

数位分布　结果如图所示：

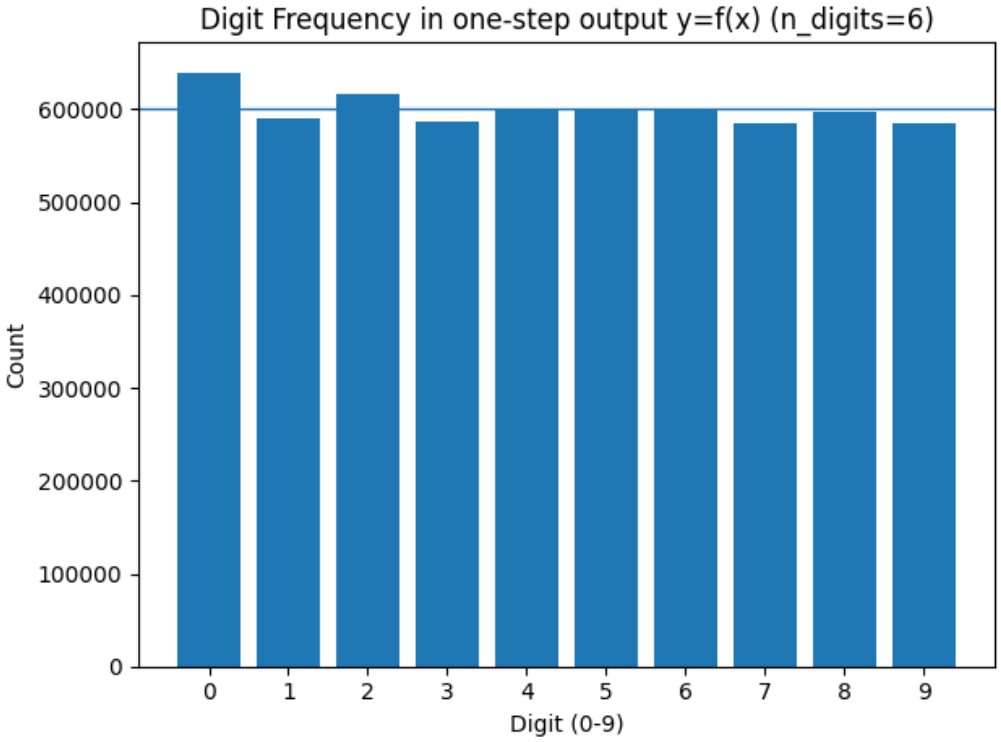

图 3: Digit frequency

就单个数位来说，除了0以外，其他分布都比较均匀。这一偏差来源于平方运算在十进制下的结构性限制(尤其是含 2、5 因子的输入会使x^2产生更多 0)，以及取中间位导致的信息丢失和 many-to-one 收缩，使输出更倾向出现较小数值与前导 0，从而抬高数字 0 的总体出现频率。

x-f(x)分布　结果如图所示：

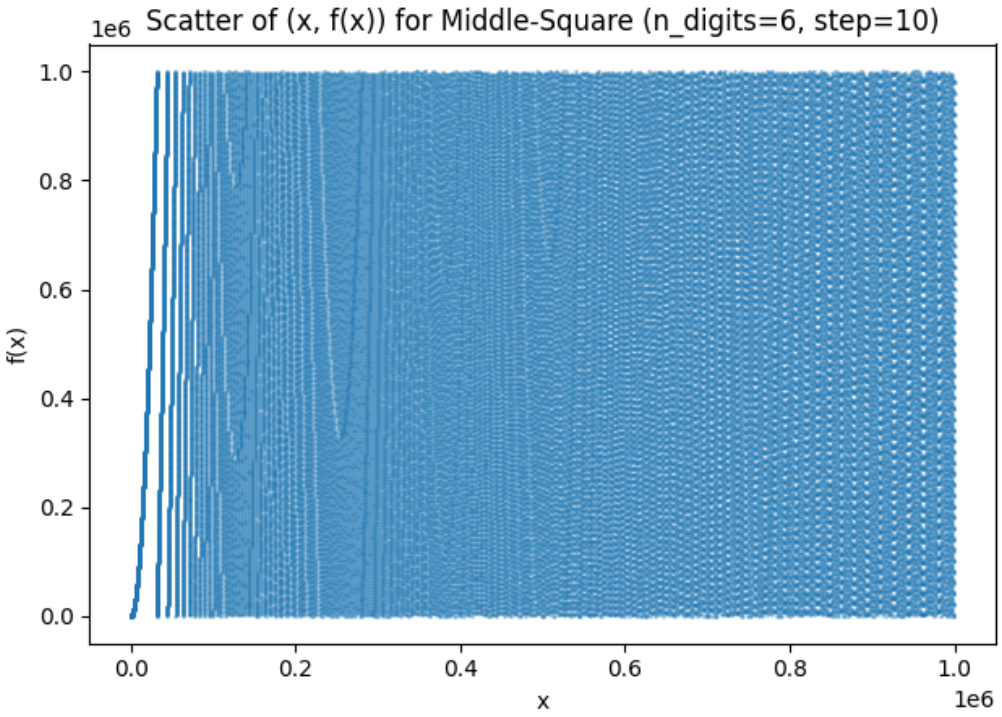

图 4: x-f(x)分布

在$x < 3 \cdot 10^6$时出现明显条纹状，x-f(x)相关性很强，随机性不高。

安全性 给定序列中任一数字，其后向可直接通过公式计算，前驱计算能在多项式时间内解决，攻击者可通过枚举或预计算反向映射快速求得前驱集合，从而实现倒推。取中间位造成的信息丢失使映射 many-to-one，虽使前驱不唯一，但结合多步观测可快速剪枝定位真实轨迹。因此平方取中法不具备实际安全性，不适用于要求安全性的场景。

1.2 线性与置换同余生成器

线性同余生成器(LCG, Linear congruential generator)基于模运算，是最早的，也是最广为人知的伪随机数生成器之一。其定义式：

$$X_{n+1} \equiv aX_n + c \pmod{m}$$

一般称a为乘数(multiplier)，c为增量(increment)。

先考虑特例：$c = 0$

1.2.1 乘法LCG

又称multiplicative congruential generator(MCG), Lehmer RNG。因为$c = 0$，MCG的解满足：

$$X_{n+1} \equiv aX_n \pmod{m} \equiv a^{n+1}X_0 \pmod{m}$$

这是一个齐次线性变换。也就是说，解集

$$\mathcal{S}_0 = \{(D_n)_{n\geq 0} : D_{n+1} \equiv aD_n \pmod{m}\}$$

是一个 $\mathbb{Z}_m$-module。

注意到当$d = \gcd(a, m) \neq 1$时，序列会被压缩到d的倍数中，造成周期变短。

1.2.2 混合LCG

回到 $c \neq 0$ 的情况。

递推式不再是一个齐次线性变换，而是一个仿射(affine)。更具体地，设 $\mathcal{S}_c$ 为所有满足上述递推的序列集合：

$$\mathcal{S}_c = \{(X_n)_{n\geq 0} \in (\mathbb{Z}_m)^{\mathbb{N}} : \ X_{n+1} \equiv aX_n + c \pmod{m} \ \forall n\}.$$

$\mathcal{S}_c$ 一般不再是 $\mathbb{Z}_m$-模。

取任意一个特解序列 $(P_n) \in \mathcal{S}_c$，令差序列 $D_n = X_n - P_n$，则

$$D_{n+1} = X_{n+1} - P_{n+1} \equiv (aX_n + c) - (aP_n + c) \equiv aD_n \pmod{m}.$$

因此差序列满足齐次递推 $D_{n+1} \equiv aD_n \pmod{m}$。也就是说，$\mathcal{S}_c$ 是齐次解集S_0 的一个平移(affine coset)：

$$\mathcal{S}_c = (P_n) + \mathcal{S}_0.$$

1.2.3 PCG

置换同余生成器(PCG, Permuted congruential generator)在LCG的基础上增加了输出置换函数(output permutation function)，用于破坏其线性/仿射结构。输出置换函数常常是以下操作的组合：

RR (random rotation) 给定一个 2^b 位的输入，最高 $b-1$ 位用于循环移位量，对输入的下半部分位向右循环移位，丢弃最低的 $2^{b-1}+1-b$ 位。

RS (random shift) 从 $2b$ 位输入开始，前 $b-3$ 位用于移位量，该移位量应用于接下来的 $2^{b-1}+2^{b-3}-1$ 位，并将最终结果的下半部分输出，丢弃最低 $2^{b-1}-2^{b-3}-b+4$ 位。

XSH (xorshift) 相当于$x \wedge= x >> constant$。选择的常数为下一个操作没有丢弃的位的一半（向下取整）。

XSL(XOR) 对输入的上半部分与下半部分进行异或操作。

RXS (random xorshift) 类似于XSH，但使用一个随机的移位量。

M (multiply) 将输入乘以一个固定常数。

常见的置换输出函数有XSH-RR、XSH-RS、XSL-RR、RXS-M-XS和XSL-RR-RR。

其实，不止对输出进行优化可以改善统计性质，对输入进行优化也可以。如果我们把X_{n+1}写成X_n与X_{n-1}的线性组合，简单的线性关系将被打破。同时，我们也可以取X_{n-i}，其中i为线性随机数。这样虽然能够打破简单线性关系，但也只不过是线性状态的叠加，没有改变LCG易破解的本质。而同时，这种方法需要储存多个状态，不如PCG高效。

周期性 关于混合LCG的周期，我们有Hull–Dobell 定理：如果以下三个条件都成立：

$$\gcd(c, m) = 1$$

$$\forall p \text{ prime with } p \mid m,\ \ p \mid (a-1)$$

$$4 \mid m \ \Rightarrow\ 4 \mid (a-1)$$

那么该递推为满周期。

与平方取中法类似，LCG 诱导出一个图：每个节点 x 有唯一出边指向 $f(x)$。因此每个连通分量同样由一个有向环加上若干指向该环的入树构成。若 f 为双射，则每个连通分量退化为纯环（每点入度为 1），不存在入树与 preperiod。

对 LCG 而言，f 是双射当且仅当

$$\gcd(a, m) = 1.$$

此时所有种子都直接位于某个环上（preperiod 为 0）。

进一步地，在混合 LCG（$c \neq 0$）中，若满足 Hull–Dobell 定理的满周期条件，则整个状态空间 $\{0, 1, \ldots, m-1\}$ 构成唯一一个长度为 m 的大环。若不满足上述条件，则图会分裂为多个较短环，并在 $\gcd(a, m) \neq 1$ 时出现明显的入树结构，从而导致周期变短、覆盖性变弱。

环长可以通过preperiod与首次重复访问时间来计算。

LCG 的周期性质依赖于参数选择，在满足满周期条件时可达到极大周期并实现良好覆盖。

数位分布　结果如图所示：

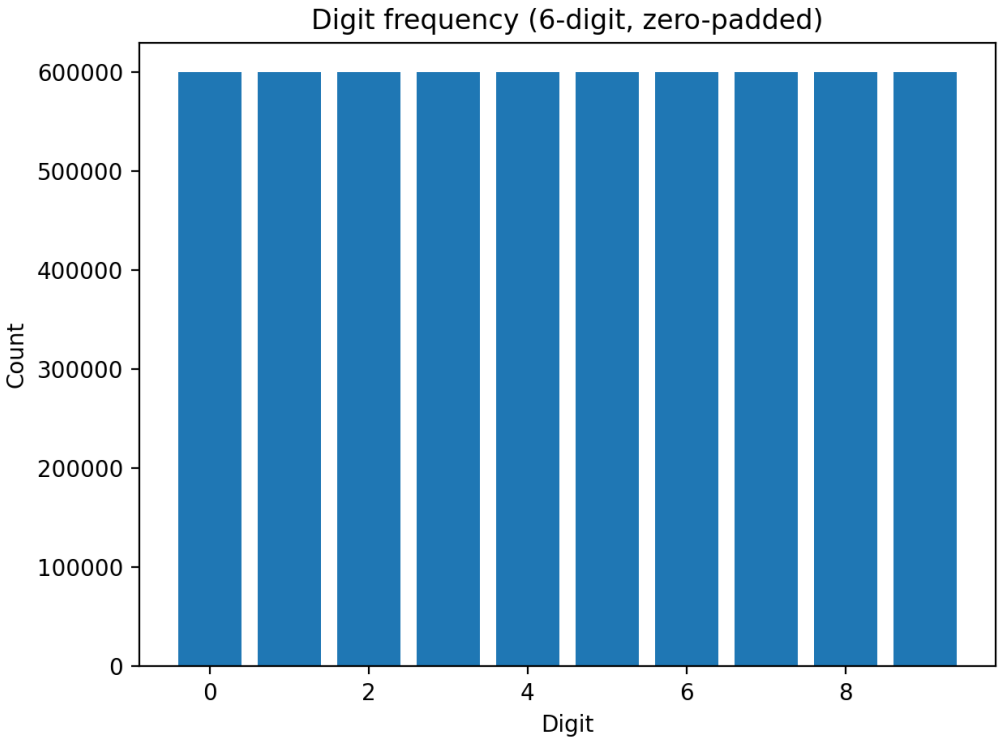

图 5: Digit frequency

可以看到相较于平方取中法，数位分布更平均，随机性更好。

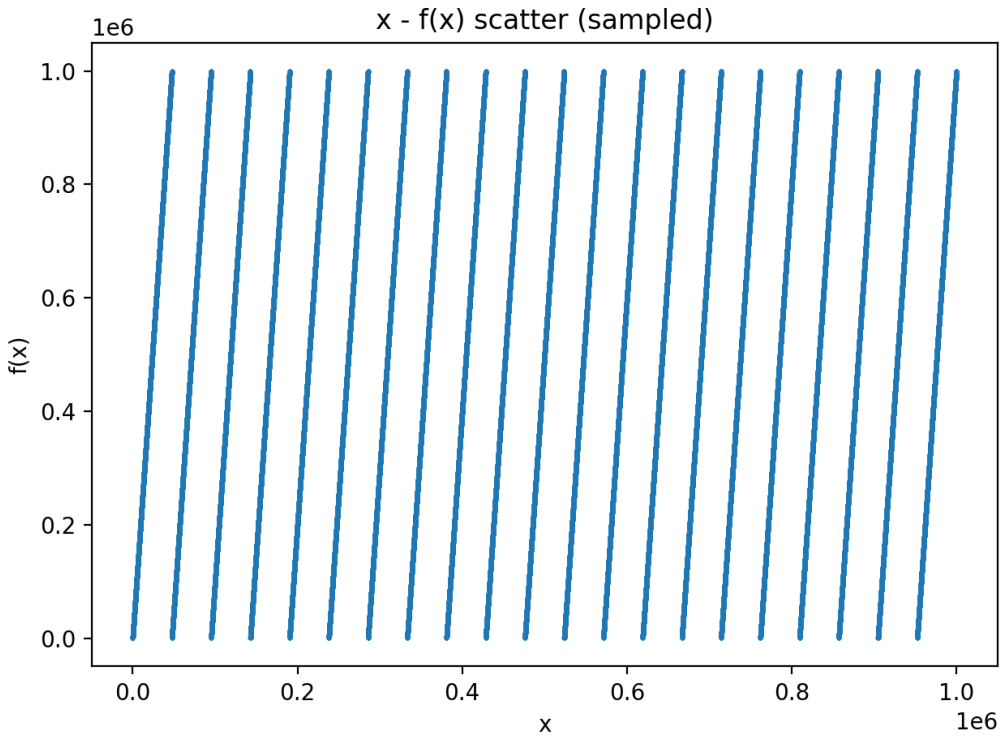

图 6: LCG x-f(x)分布

x-f(x)分布 出现了极强的条纹结构，这是由LCG自身的性质决定的。

LCG 公式：

$$f(x) = (ax + c) \bmod m.$$

将 mod$m$ 展开，可写为

$$f(x) = ax + c - km, \qquad k = \left\lfloor \frac{ax + c}{m} \right\rfloor.$$

因此，对每一个固定的整数 k，点集 $(x, f(x))$ 满足一条直线方程

$$y = ax + c - km,$$

其斜率恒为 a。当 $x \in [0, m)$ 时，$\frac{ax+c}{m}$ 的取值范围约为 $[0, a)$，从而

$$k \in \{0, 1, 2, \ldots, a - 1\}.$$

故散点图由大约 a 条线段组成。我们在实验中人为规定$a = 21, c = 1$, k 取 0 到 20，理论上会出现约 21 条线段，事实也如此。

前面已经提到，PCG可以破坏LCG的线性结构。我们选取置换函数XSH-RR，只需对LCG的输出再进行一次置换操作，结果如下：

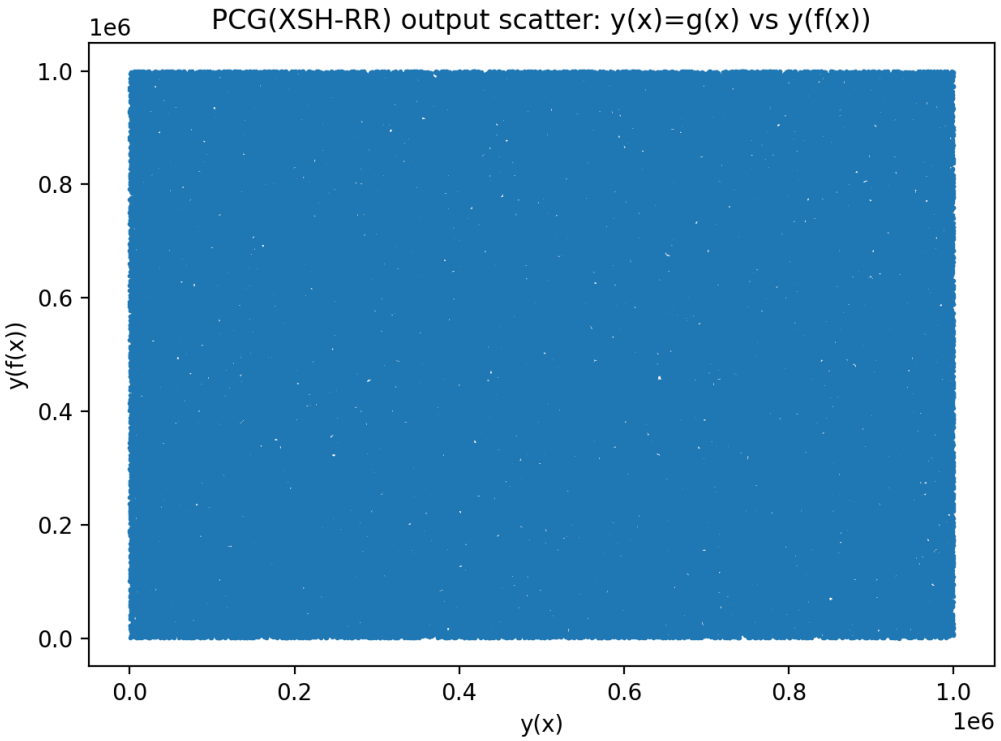

图 7: PCG x-f(x)分布

比特层面可见的线性结构已经被破坏了。PCG拥有更好的随机性，x-f(x)不再有简单线性关系。

dieharder测试 其实对于非密码学PRNG，已经有很成熟的测试程序，如Dieharder、TestU01。这里简单演示LCG的dieharder部分测试结果。

Test	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.99903408	WEAK
diehard_operm5	0	1000000	100	0.24289655	PASSED
diehard_rank_32x32	0	40000	100	0.44947024	PASSED
diehard_rank_6x8	0	100000	100	0.00000000	FAILED
diehard_bitstream	0	2097152	100	0.00000000	FAILED
diehard_opso	0	2097152	100	0.00000000	FAILED
diehard_oqso	0	2097152	100	0.00000000	FAILED
diehard_dna	0	2097152	100	0.00000000	FAILED
diehard_count_1s_str	0	256000	100	0.00000000	FAILED
diehard_count_1s_byt	0	256000	100	0.00000000	FAILED
diehard_parking_lot	0	12000	100	0.88193571	PASSED
diehard_2dsphere	2	8000	100	0.01224158	PASSED
diehard_3dsphere	3	4000	100	0.69657100	PASSED
diehard_squeeze	0	100000	100	0.75328745	PASSED
diehard_sums	0	100	100	0.77062470	PASSED
diehard_runs	0	100000	100	0.10869711	PASSED
diehard_runs	0	100000	100	0.89741629	PASSED
diehard_craps	0	200000	100	0.56377733	PASSED
diehard_craps	0	200000	100	0.78677959	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00000000	FAILED

表 2: Dieharder 测试结果

可以看到即使只展示了部分测试，还是非常全面的。缺点是测试需要时间比较长。

1.3 线性反馈移位寄存器

LFSR(Linear feedback shift register)是指给定前一状态的输出，将该输出的线性函数再用作输入的移位寄存器。 LFSR主要有Fibonacci 和 Galois两种实现，主要区别是Fibonacci使用多位XOR，而Galois使用只在移位

过程中使用分散的XOR。

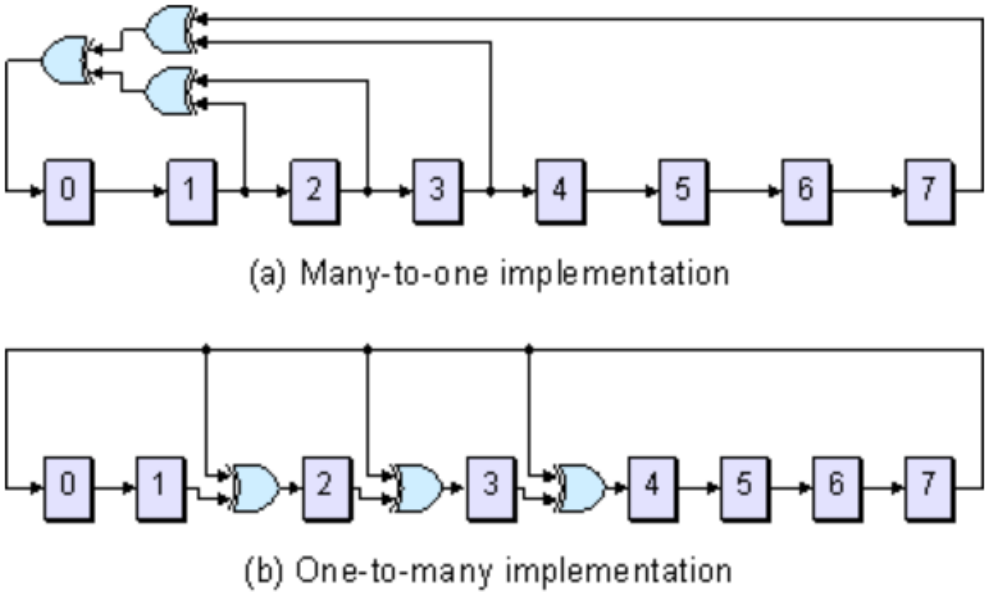

图 8: LFSR电路

具体步骤 规定寄存器长度为 n，时刻 t 的状态为

$$\mathbf{s}(t) = (s_{n-1}(t), s_{n-2}(t), \ldots, s_0(t)) \in \{0,1\}^n,$$

每拍右移一位移出s_0，并在最左端注入新bit。输出取移出位

$$y(t) = s_0(t).$$

全零态会出现锁死情况，因此要求初始种子 $\mathbf{s}(0) \neq 0$。

1.3.1 Fibonacci 型

给定 tap 集合 $T \subseteq \{0, 1, \ldots, n-1\}$（表示参与反馈异或的寄存器位下标）。每个时钟周期 $t \to t+1$ 的更新步骤如下：

1. **输出：** $y(t) = s_0(t)$。

2. **反馈计算：** 计算反馈位

$$f(t) = \bigoplus_{i \in T} s_i(t),$$

其中 $\oplus$ 表示 GF(2) 上的加法（XOR）。

3. **移位：** 对 $i = 0, 1, \ldots, n-2$，

$$s_i(t+1) = s_{i+1}(t).$$

4. **注入：** 将反馈位注入最高位

$$s_{n-1}(t+1) = f(t).$$

1.3.2 Galois 型

给定注入集合 $J \subseteq \{1, 2, \ldots, n-1\}$。每个时钟周期 $t \to t+1$ 的更新步骤如下：

1. **输出与反馈位：** 取

$$y(t) = s_0(t), \qquad b(t) = s_0(t).$$

2. **移位并注入：**

- 对最低位，右移一位：

$$s_0(t+1) = s_1(t).$$

- 对 $i = 1, 2, \ldots, n-2$，

$$s_i(t+1) = \begin{cases} s_{i+1}(t) \oplus b(t), & i \in J, \\ s_{i+1}(t), & i \notin J. \end{cases}$$

3. **写入最高位：**

$$s_{n-1}(t+1) = b(t).$$

T 与 J 选取恰当时，两者在周期与线性性质上等价。

LFSR通常适用于硬件开发，在此处不过多赘述。

1.4 梅森旋转算法

梅森旋转算法由松本真和西村拓士在1997年开发，其基本思想基于二元域 $GF(2)$ 上的线性递推，可以在相对可接受的计算/存储成本下，生成周期极长、在高维上表现良好的伪随机数序列，是R、Python、Matlab和C等语言的默认PRNG。

除了之前提到的评价PRNG的几个维度，还有一种方式被认为是PRNG最严格的评价指标：**k维v比特准确度 (k-distributed to v-bit accuracy)**。

k-distributed to v-bit accuracy（k 维、v 比特精度的均匀分布） 设 $\{X_i\}$ 是周期为 P 的 w-bit 伪随机序列，其中 $X_i \in \{0, 1, \ldots, 2^w - 1\}$ 且 $X_{i+P} = X_i$。定义截断函数 $t_v(\cdot)$ 为取 X_i 的前 v 个（最高有效）比特：

$$t_v(X_i) \;=\; \left\lfloor \frac{X_i}{2^{w-v}} \right\rfloor \in \{0, 1, \ldots, 2^v - 1\}.$$

对每个 i（通常按周期取模），构造长度为 k 的“v-bit 向量块”

$$B_i \;=\; \big(t_v(X_i),\, t_v(X_{i+1}),\, \ldots,\, t_v(X_{i+k-1})\big) \;\in\; \{0, 1, \ldots, 2^v - 1\}^k.$$

将 B_i 视为一个 kv-bit 向量（例如按顺序拼接 k 个 v-bit 截断值）。

若在所有 P 个块 $\{B_i\}_{i=0}^{P-1}$ 中，每一个可能的非零 kv-bit 向量都出现相同次数，则称 $\{X_i\}$ **是 k-distributed to v-bit accuracy**。形式化地，对任意 $u \in \{0,1\}^{kv} \setminus \{0^{kv}\}$，都有

$$\#\{\, i \in \{0, \ldots, P-1\} : B_i = u \,\} \;=\; \frac{P}{2^{kv} - 1}.$$

可以证明，梅森旋转算法具有k维v比特准确度。

我们在课上讲过，梅森质数是一类形如$2^n - 1$的素数。而MT19937就使用了$2^19937 - 1$这个梅森素数作为生成伪随机数的循环长度。下面我们以MT19937为例，介绍梅森旋转算法的具体实现。

梅森旋转算法实现(以MT19937为例) 两个重要概念：**Twist 和 Tempering**。

Twist　更新内部状态，决定周期与核心混合结构。

具体操作：

1. **拼接相邻 word 的高位与低位**

   用掩码把 $\texttt{mt}[i]$ 的高 $(w-r)$ 位与 $\texttt{mt}[i+1]$ 的低 r 位拼成 w-bit 的 x：

$$x \leftarrow (mt[i] \ \& \ upper\_mask) \ + \ (mt[i+1] \ \& \ lower\_mask).$$

   让“一个 word 的高位”和“下一个 word 的低位”发生耦合，避免每个 word 近似独立地演化，增强跨 word 的信息传播。

2. **线性反馈：右移并按最低位条件异或常数**

   计算

$$xA \leftarrow x \gg 1, \qquad \text{若 } (x \ \& \ 1) = 1 \text{ 则 } xA \leftarrow xA \oplus a.$$

   在 $\mathbb{F}_2$ 上用一个固定的 $w \times w$ 矩阵作用于 x，从而形成 LFSR 风格的反馈结构。它是 MT 达到最大周期与良好线性递推性质的关键之一。

3. **与远处索引做异或**

   将上面的结果与 $\texttt{mt}[i+m]$ 异或生成新状态：

$$\texttt{mt}[i] \leftarrow \texttt{mt}[i+m] \oplus xA.$$

   这里的 m使得更新依赖一个“远距离”的状态元素，增强状态混合的跨度，降低短期相关结构。

Tempering　改造输出，提升统计性质

当从状态数组取出一个 word $y = mt[index]$ 时， MT 并不直接输出它，而是做一串固定的位变换（移位、掩码、异或）：

$$\begin{aligned} y &\leftarrow y \oplus ((y \gg u) \ \& \ d), \\ y &\leftarrow y \oplus ((y \ll s) \ \& \ b), \\ y &\leftarrow y \oplus ((y \ll t) \ \& \ c), \\ y &\leftarrow y \oplus (y \gg l). \end{aligned}$$

Tempering通过移位把高位信息注入低位（或反向），再用异或把这些信息叠加，使得输出的每一位更像“多个状态位的组合”来修正可见的线性结构与低位问题，从而改善常用统计检验以及 k-distribution等指标。

下面是梅森旋转的具体步骤：

1. 状态表示 设字长 $w = 32$，状态数组长度 $n = 624$，并维护指针 $index$。内部状态为

$$\texttt{mt}[0..n-1] \text{（每个元素是一个 } w\text{-bit word）}, \qquad index \in [0, n].$$

对 MT19937，参数 $m = 397,\ r = 31$。

2. 初始化（seeding） 给定 32-bit 种子 s，用一个递推填满状态数组：

$$mt[0] \leftarrow s, \qquad mt[i] \leftarrow f \cdot \big(mt[i-1] \oplus (mt[i-1] \gg (w-2))\big) + i \quad (1 \le i \le n-1),$$

并令 $index \leftarrow n$，表示“状态已用尽，下一次取数前需要 twist”。

3. 扭转（Twist） 当 $index \ge n$ 时，执行 twist 生成下一批 n 个状态 word。定义掩码：

$$lower\_mask = (1 \ll r) - 1, \qquad upper\_mask = \big(\sim lower\_mask\big) \,\&\, ((1 \ll w) - 1).$$

对每个 $i = 0, 1, \ldots, n-1$，将相邻两个状态的“高位+低位”拼接成 x：

$$x \leftarrow (mt[i] \ \&\ upper\_mask) \ + \ (mt[(i+1) \bmod n] \ \&\ lower\_mask).$$

然后做一次线性反馈变换：

$$xA \leftarrow x \gg 1, \qquad \text{若 } (x \ \&\ 1) = 1 \text{ 则 } xA \leftarrow xA \oplus a.$$

最后更新状态：

$$mt[i] \leftarrow mt[(i+m) \bmod n] \oplus xA.$$

完成后令 $index \leftarrow 0$。

4. 取数（Extract）与回火（Tempering） 每次需要输出一个 32-bit 随机数时：

1. 若 $index \geq n$，先执行 **Twist**；
2. 取 $y \leftarrow mt[index]$，并令 $index \leftarrow index + 1$；
3. 对 y 做 tempering 以改善统计性质：

$$y \leftarrow y \oplus ((y \gg u)\ \&\ d),$$
$$y \leftarrow y \oplus ((y \ll s)\ \&\ b),$$
$$y \leftarrow y \oplus ((y \ll t)\ \&\ c),$$
$$y \leftarrow y \oplus (y \gg l),$$

4. 输出 y。

其伪代码如下：

Algorithm 2 MT19937 (32-bit) Pseudocode

Parameters: $w \leftarrow 32$, $n \leftarrow 624$, $m \leftarrow 397$, $r \leftarrow 31$, ...
State: $\texttt{mt}[0..n-1]$, $\texttt{index}$
procedure SeedMT(s)
    $\texttt{mt}[0] \leftarrow s$
    for $i \leftarrow 1$ to $n-1$ **do**
        $\texttt{mt}[i] \leftarrow \cdots$
    end for
    $\texttt{index} \leftarrow n$
end procedure
procedure ExtractNumber
    if $\texttt{index} \geq n$ **then**
        Twist
    end if
    $y \leftarrow \texttt{mt}[\texttt{index}]$; $\texttt{index} \leftarrow \texttt{index} + 1$
    Tempering: $y \leftarrow y \oplus (y \gg 11)$; ...
    return y
end procedure

由于真正 MT19937 的状态空间是 2^{19937}量级，无法像mid-square/LCG那样“全空间功能图”枚举 period/preperiod。我们人为定义一个在10^6状态上的确定性函数$f(x)$ MT-derived ，用于对比“输出混合/置换”效果与mid-square/LCG 的差异；它不等价于 MT 的真实内部状态周期。

数位分布　如图：

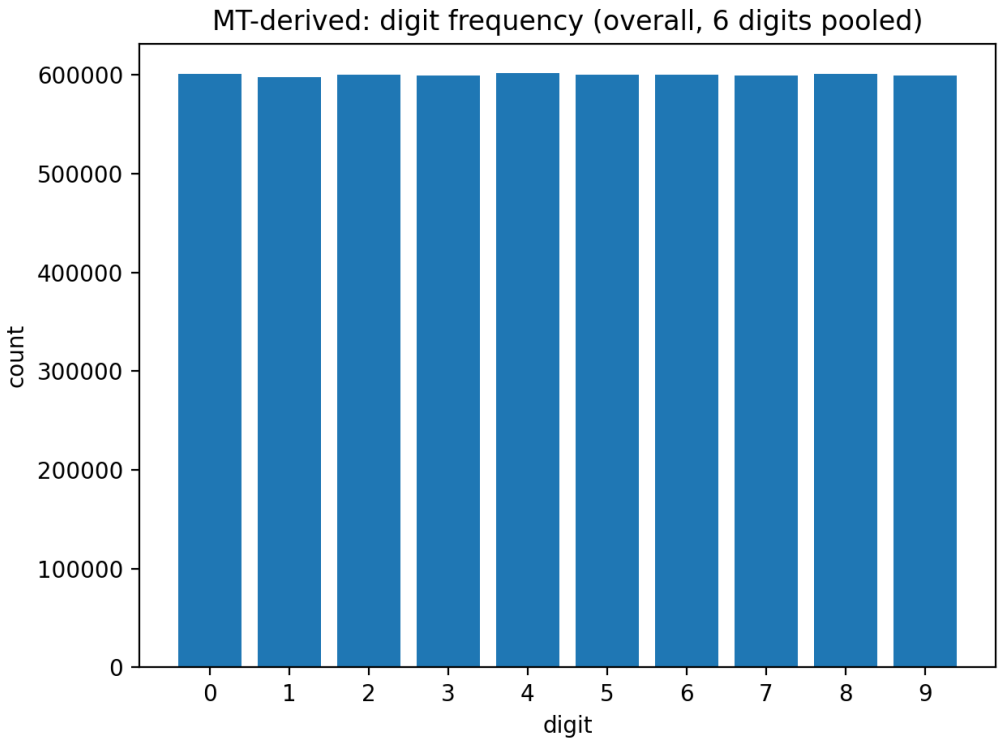

图 9: Digit frequency

分布均匀。

x-f(x)分布　如图：

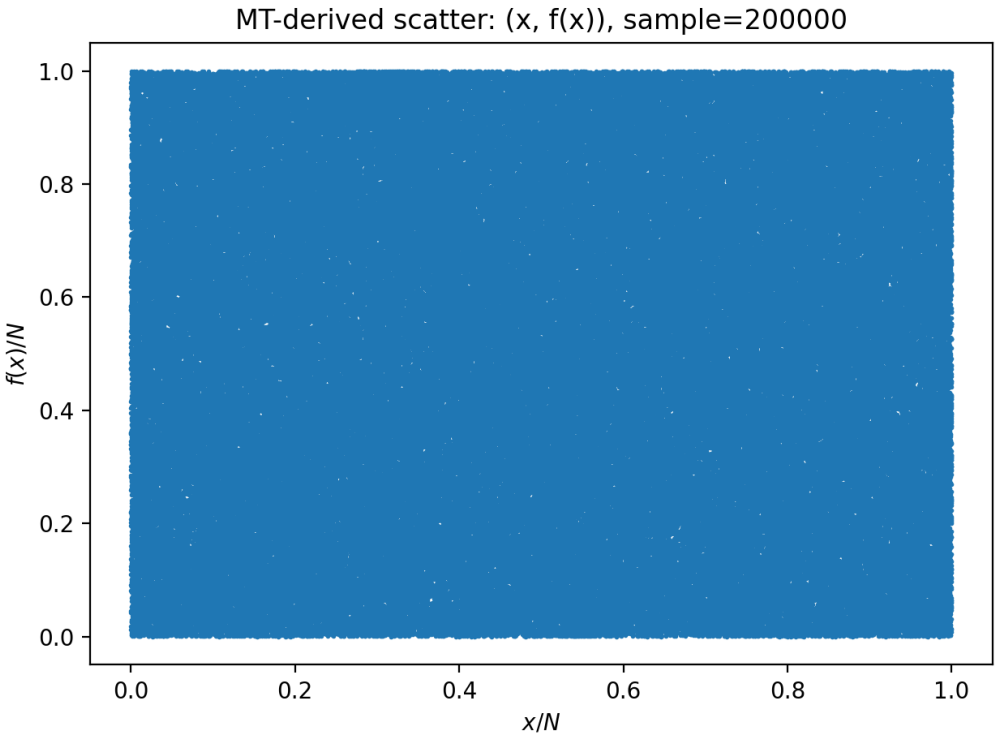

图 10: x-f(x)分布

分布均匀。

安全性 MT的核心仍然是核心是异或/移位的线性递推，而tempering的四种操作均可逆，这意味着，密码学意义上，MT是一个**不安全**的算法。

1.5 xoshiro系列

xoshiro 系列是 Blackman 与 Vigna 提出的现代非密码学 PRNG，一般被认为是 “XOR/shift/rotate” 家族的代表。其设计目标是：

- 在 64 位平台上尽可能**快**，适合数值模拟与 Monte Carlo；
- 状态非常小，缓存友好；
- 在 TestU01、PractRand 等统计测试中表现良好；

- 接口简单：给定种子后，每次调用返回一个 64 位伪随机数。

以 `xoshiro256**` 为例，其内部状态为

$$\mathbf{s} = (s_0, s_1, s_2, s_3) \in (\{0,1\}^{64})^4.$$

每轮输出与状态更新大致为：

1. **输出函数（star-star）**：先将 s_1 乘以一个固定奇数常量，再做一次循环移位，最后再乘以常数，得到输出

$$\text{out} = \big((s_1 \times C_1) \lll 7\big) \times C_2,$$

其中 $\lll$ 为 64 位左循环移位，C_1, C_2 是经验选取的奇数常数。这一“乘法 + 旋转” 的非线性输出层被称为 *scrambler*，用来打破底层线性结构。

2. **状态更新（xorshift+rotate）**：对 $\mathbf{s}$ 做一系列 XOR 和旋转：

$$\begin{aligned}
t &\leftarrow s_1 \ll 17,\\
s_2 &\leftarrow s_2 \oplus s_0,\\
s_3 &\leftarrow s_3 \oplus s_1,\\
s_1 &\leftarrow s_1 \oplus s_2,\\
s_0 &\leftarrow s_0 \oplus s_3,\\
s_2 &\leftarrow s_2 \oplus t,\\
s_3 &\leftarrow s_3 \lll 45.
\end{aligned}$$

可以看到，状态转移本质上仍是在 $GF(2)$ 上的线性变换（XOR 与移位），只是叠加了输出端的乘法与旋转以改善统计性质和位扩散效果。在实践中，xoshiro 系列具有高性能，适合替代传统的 LCG/MT 用于仿真和随机化算法，但密码学不安全。

2 CSPRNG

在进行仿真、蒙特卡洛与可复现实验时，我们应使用可设定种子的非密码学 PRNG，以保证实验可重复。而当涉及安全的随机量时，必须使用**CSPRNG**，其输出应当满足不可预测性。典型使用场景包括：

- **密钥材料**：对称密钥、私钥生成所需随机性等；
- **nonce/IV**：加密协议与模式中每次加密所需的随机量（唯一，不可预测）；
- **salt**：口令哈希盐（需要随机且唯一，以抵抗预计算攻击）；
- **token**：会话 ID、重置口令 token、CSRF token 等。

2.1 基本概念

密码学安全伪随机数生成器 (CSPRNG, Cryptographically Secure Pseudorandom Number Generator) 是一类通过确定性算法从一个短的、不可预测的种子扩展生成长随机序列的生成器，目标不仅要让生成序列在统计意义上满足分布，还要让破解者在多项式时间内无法预测序列。
我们之前提到过，一个确定的算法无法生成真正随机的序列。CSPRNG也并非完全随机，而是应该**不可重现**，即：

> 不能通过一部分给定的随机序列，而以显著大于$\frac{1}{2}$的概率在多项式时间内演算出序列的任何其他部分。

一个可用的CSPRNG必须要满足给定种子易于计算输出，而给定输出无法容易地计算种子。好的CSPRNG应该使输出在多项式时间内计算不可区分于真正随机。

在2006年美国国家标准与技术研究院发布的一份文件中，建议了四种 CSPRNG： Hash-DRBG、 HMAC-DRBG、 CTR-DRBG和 Dual-EC-DRBG。Dual-EC-DRBG因为历史原因，不再是标准实现。

2.2 标准演变

2.2.1 ANSI X9.17 与 ANSI X9.31

ANSI X9.17与 ANSI X9.31给出了一类经典的确定性伪随机生成机制，其共同特征是：用分组密码作为混合函数，维护一个秘密内部状态 V，并引入一个随迭代更新的时间/计数向量 DT。每轮迭代输出一个分组长度的随机块 R，并更新状态 V。

2.2.2 历史演变

时间	事件与意义
1985	ANSI X9.17给出基于 DEA 多重加密（EDE）的伪随机 Key/IV 生成器。
1998	ANSI X9.31给出几乎同构的 RNG，并更强调 DT 在每次迭代更新。
2005	NIST 发布对 X9.31 A.2.4 的推荐扩展，将 AES 纳入该家族实现。
2011–2015	NIST 的算法迁移建议将 FIPS 186-2、X9.31、X9.62(1998)等legacy RNG标记为 deprecated（2011–2015），并在 2015 年后 disallowed。
2012/2015	NIST SP 800-90A形成现代 DRBG 体系，使用标准化的接口和明确的安全强度输入要求。

2.2.3 NIST SP 800-90A

NIST SP 800-90A将DRNG抽象为一套标准 API 与状态机：

1. **Instantiate** 从熵源获取 entropy input，并结合 nonce 与 personalization string，初始化内部状态。
2. **Generate** 在不引入新熵的前提下，确定性地产生所需随机比特，并更新内部状态。

3. **Reseed** 运行中再次注入熵以刷新内部状态，提高抗状态泄露与长期运行安全性。

4. **Uninstantiate** 擦除内部状态，避免残留。

把算法片段转换为可工程化落地的安全模块规范。

在构造层面，SP 800-90A Rev. 1 的主流 DRBG 族包括：

- **Hash_DRBG** 基于哈希函数；
- **HMAC_DRBG** 基于 HMAC；
- **CTR_DRBG** 基于分组密码 CTR 结构。

将在下一节详细介绍。
下面我们展示6种常见的CSPRNG。

2.3 6种CSPRNG

2.3.1 Hash DRBG

Hash-DRBG 将内部状态视为一个大整数/比特串（常用记号为 V），并配合一个常量/计数器（记为 C）与重播计数器（$reseed\_counter$）。每次生成输出时，先用哈希驱动的扩展函数从 V 派生所需比特，然后对 V 做一次确定性更新，以保证前向安全（forward security）与状态演化。

其实现将在下一节介绍。

2.3.2 HMAC DRBG

HMAC-DRBG（HMAC-based Deterministic Random Bit Generator）是一种基于消息认证码 HMAC的CSPRNG。它从高熵随机源获取一次或多次种子材料（seed material），在内部维护少量固定长度状态，并通过迭代计算 HMAC 来输出伪随机比特流。其安全性通常归约到所选哈希函数

在 HMAC 构造下的伪随机性假设，因此在工程实践中被广泛用于密钥、nonce、会话令牌等安全敏感场景。

HMAC-DRBG 的内部状态主要由两部分组成：密钥 K 与工作向量 V，二者长度均等于底层哈希函数的输出长度（例如 SHA-256 时为 256 bit）。初始化阶段将 K 设为全零、V 设为全一，然后把熵输入（以及可选的 nonce、个性化字符串）通过标准的 Update 过程混入状态，从而得到不可预测的初始 K, V。

在生成Generate阶段，生成器重复执行 $V \leftarrow \mathrm{HMAC}(K, V)$，并将得到的 V 依次拼接为输出，直到达到所需长度。生成结束后再次调用 Update 更新 K, V，以确保状态持续演化并降低状态恢复或预测未来输出的风险。当输出次数或输出总量达到阈值，或系统获取到新的熵输入时，可执行Reseed将新熵通过 Update 注入状态并重置计数器，以恢复更强的不可预测性。

2.3.3 CTR-DRBG

CTR-DRBG（Counter-mode Deterministic Random Bit Generator）属于Block Cipher DRNGs。它基于分组密码, 核心思想是维护一个内部密钥 K 与一个计数器/向量 V，每次生成随机数时对不断递增的 V 进行加密，得到的密文块按顺序拼接作为输出。

CTR-DRBG 的内部状态通常包括 K：密钥与 V：计数器。初始化时，从高熵源获取种子材料，通过派生函数压缩并扩展到所需长度，用以设置初始 K, V。在Generate阶段，生成器循环计算 $\mathrm{AES}_K(V)$，每输出一个分组就将 V 加一，以保证输入块不重复；生成结束后继续执行Update，从而使状态持续演化并降低基于输出恢复状态的风险。

2.3.4 Dual EC DRBG

Dual EC DRBG（Dual Elliptic Curve Deterministic Random Bit Generator）是一种基于椭圆曲线离散对数困难性构造的 DRBG。和其同样利用椭圆曲线的加密方法ECC，是RSA中，对公开密钥加密的主要算法之一。其基本做法是维护一个内部状态 s，在每轮中计算椭圆曲线点乘得到新状态与输出：先用固定点 P 计算 sP 并从其坐标导出下一状态，再用同一

个s计算 sQ 并截取其坐标的部分比特作为输出。它利用椭圆曲线点乘的单向性来实现不可预测的伪随机输出。

然而，有趣的是，P、Q两个是选取的固定点，但是大多数时候，我们并没有人为选取，而是保留了这两个点的默认值。如果设计参数的人在一开始让Q与P存在一个只有他知道的离散对数关系 $Q = dP$, 那么有：$sQ = s(dP) = d(sP)$，存在线性关系！对于对不知道 d 的人, 想要破解序列，需要计算椭圆曲线离散对数等NP问题。但是，如果d已知，得到一定数量序列中的元素, 就能在极低的时间成本内预测和重构序列。此外，该构造涉及多次椭圆曲线点乘, 性能较差，在实践中不如基于 AES 或 HMAC 的 DRBG 高效。

2.3.5 Blum Blum Shub

Blum Blum Shub的核心思想是在模 n 的乘法群上反复进行平方映射，并从状态中抽取少量bit作为输出。和RSA一样，BBS 的安全性建立在整数分解这个NP问题之上。

参数选择

- 选取两个大素数 p, q，满足 $p \equiv q \equiv 3 \pmod 4$（Blum primes）。
- 令模数 $n = pq$（Blum integer）。
- 选取种子 $s \in (\mathbb{Z}/n\mathbb{Z})^\times$，并设初始状态

$$x_0 = s^2 \bmod n,$$

使得 x_0 落在模 n 的二次剩余集合中。

状态更新与输出 BBS 以平方迭代作为状态转移：

$$x_{i+1} = x_i^2 \bmod n.$$

每轮从状态 x_i 中抽取输出bit。最常见且最稳妥的做法是输出LSB, 也可输出若干低位。

安全性 在攻击者不知道 p, q的前提下，从输出bit序列预测下一bit是NP问题，BBS是一个CSPRNG。

缺点 每步需要大整数模平方运算，性能显著慢于上述其他的 DRBG。

2.3.6 ChaCha

前文提到的 Hash-DRBG、HMAC-DRBG 和 CTR-DRBG 都属于 NIST SP 800-90A 推荐的 CSPRNG 构造。另一条重要路线是以流密码为核心的 CSPRNG，其中使用最广泛的是基于 ChaCha 的方案。

ChaCha 最初由 Bernstein 设计为一类 ARX（add–rotate–xor）流密码，其核心特点是：

- 内部仅使用 32 位加法（模 2^{32}）、异或与循环移位三种操作，易于在软件中实现且性能极高；
- 状态由 16 个 32 位字组成：固定常量、密钥、计数器与随机数（nonce）；
- 通过若干轮 “quarter round” 操作将状态强烈混合，从而生成伪随机的 512 bit 输出块。

ChaCha20 的一轮quarter round可抽象为对四个 32 位字 (a, b, c, d) 做如下 ARX 混合：

$$
\begin{aligned}
a &\leftarrow a + b, \quad d \leftarrow (d \oplus a) \lll 16, \\
c &\leftarrow c + d, \quad b \leftarrow (b \oplus c) \lll 12, \\
a &\leftarrow a + b, \quad d \leftarrow (d \oplus a) \lll 8, \\
c &\leftarrow c + d, \quad b \leftarrow (b \oplus c) \lll 7.
\end{aligned}
$$

一个完整的 ChaCha20 块函数对 16 个字反复执行 20 轮（10 个 double-round），最后将结果与初始状态按字相加并输出 512 bit 密钥流。

若忽略 “把密钥流与明文异或” 这一步，ChaCha本质上就是一个以 $(K, \text{nonce}, \text{counter})$ 为种子的 CSPRNG。 ChaCha 的设计初衷就是密码学安全，在现代操作系统和密码库（如 Linux 内核 CSPRNG、OpenBSD arc4random 新版、libsodium 的 randombytes钟被普遍采用。

2.4 Hash PRNG的简单实现

主要步骤

初始化 从系统熵源拿到 entropy，用哈希派生函数把这些材料压缩成内部状态 V，并由 V 再派生一个常量 C。

生成 每次需要输出时，反复计算 Hash(V) 并拼接起来得到输出；同时让 V 递增以保证每次哈希输入不同。

重播种 输出完成后用 Hash(V)、常量 C 和计数器等混合更新 V；生成次数或输出量达到阈值时从熵源再取新熵进行 reseed，恢复不可预测性。

伪代码 如下。

Algorithm 3 Hash-DRBG

Require: Hash function H, derivation function $\textsc{Hash\_df}(\cdot, \ell)$

function $\textsc{Instantiate}(entropy, nonce, personalization)$
  $seed \leftarrow entropy \,\|\, nonce \,\|\, personalization$
  $V \leftarrow \textsc{Hash\_df}(seed, seedlen)$
  $C \leftarrow \textsc{Hash\_df}(0x00 \,\|\, V, seedlen)$
  $reseed\_counter \leftarrow 1$
  return $(V, C, reseed\_counter)$
end function

function $\textsc{Generate}(V, C, reseed\_counter, n, additional)$
  if $additional \neq \emptyset$ **then**
    $V \leftarrow V + \textsc{Hash\_df}(additional, seedlen)$ ▷ mix in optional input
  end if
  $out \leftarrow \epsilon$
  while $|out| < n$ **do**

14: $out \leftarrow out \,\|\, H(V)$
15: $V \leftarrow V + 1$
16: **end while**
17: $out \leftarrow$ leftmost n bytes of out
18: $V \leftarrow V + H(0x03 \,\|\, V) + C + reseed\_counter$ ▷ state update
19: $reseed\_counter \leftarrow reseed\_counter + 1$
20: **return** $(out, V, C, reseed\_counter)$
21: **end function**

22: **function** RESEED$(V, entropy, additional)$
23: $seed \leftarrow entropy \,\|\, additional$
24: $V \leftarrow$ HASH_DF$(seed \,\|\, V, seedlen)$
25: $C \leftarrow$ HASH_DF$(0x00 \,\|\, V, seedlen)$
26: $reseed\_counter \leftarrow 1$
27: **return** $(V, C, reseed\_counter)$
28: **end function**

3 密码学标准中的伪随机数应用

此部分以 PKCS#1 与 PKCS#5 为例。

3.1 PKCS#1

PKCS#1 规定了 RSA 的加密与签名相关方案，其中 **OAEP**（Optimal Asymmetric Encryption Padding）用于加密随机化填充，**PSS**（Probabilistic Signature Scheme）用于签名随机化。两者的共同点是：同一明文/同一消息重复处理，输出也应呈现随机化特征，以满足语义安全/不可区分性或抵抗结构性攻击的需求。

- **OAEP**：需要随机种子（seed），再通过掩码生成函数（MGF，通常基于哈希）产生掩码，对明文编码进行随机化。
- **PSS**：需要随机盐（salt），盐被并入哈希结构，形成概率化签名，使同一消息每次签名结果不同。

安全要求：上述 seed/salt 必须来自不可预测的随机源（CSPRNG/DRBG/OS CSPRNG）。若随机性弱（可预测、重复或熵不足），则随机化目标被破坏，可能导致输出可被区分、可被关联或在某些实现情形下引发严重安全退化。

3.2 PKCS#5

PKCS#5 覆盖口令相关密码学机制，其中 PBKDF2（Password-Based Key Derivation Function 2）用于从口令派生密钥。PBKDF2 的核心随机量是 **salt**：

- **salt 的目标**：防止对常见口令进行预计算，并确保相同口令在不同条目下派生结果不同。
- **salt 的要求**：必须随机且唯一。salt 一般不要求保密，但必须不可预测/不可复用。

若 salt 复用或低熵，攻击者可显著降低离线字典攻击成本，并能通过相同派生结果推断口令复用关系。

标准与随机量对应表 如下。

标准	使用环节	随机量	要求
PKCS#1	RSA：OAEP（加密）/ PSS（签名）	OAEP：seed（MGF 掩码）；PSS：salt（哈希随机化）	高熵、不可预测、每次独立
PKCS#5	PBKDF2 口令派生	salt	随机且唯一

表 3: PKCS#1 与 PKCS#5 中随机性的典型用途

4 不同编程语言中的随机数 API

不同语言的随机数接口通常分为两大类：

- **非密码学 PRNG API：**用于仿真、采样、随机化算法与游戏。特点是速度快、接口丰富（支持各种分布）、可设定种子以保证实验可复现。
- **CSPRNG：**用于密钥、nonce/IV、salt、会话标识符、重置口令 token 等安全敏感场景。其核心目标是不可预测性，通常由操作系统熵源支持。

4.1 各语言PRNG API

下面列出了几种主流语言的代表性随机数接口使用的算法，包括非密码学和密码学PRNG。

语言	PRNG	CSPRNG/DRBG
Python	MT19937	OS CSPRNG（SystemRandom / os.urandom）
C / C++	C: LCG及其变体； C++：实现相关	OS CSPRNG或密码库 DRBG/随机接口（如 OpenSSL）
Java (JDK)	java.util.Random: LCG；SplittableRandom: 生成器族	SecureRandom: Provider；常见为 DRBG 或 OS CSPRNG 封装
Go	math/rand: 版本相关，常见为 ChaCha8	crypto/rand: OS CSPRNG
JavaScript	Math.random(): 规范不指定	WebCrypto / Node crypto：通常依赖 OS CSPRNG
Rust	StdRng: 常见为 ChaCha	OsRng: OS CSPRNG
C# / .NET	System.Random: 版本相关	RandomNumberGenerator: OS CSPRNG

表 4: 不同语言的默认随机算法

4.1.1 C语言中的随机数

C 时代的 rand()接口有几个问题：

- **实现和质量完全依赖库实现**：很多实现只是一个 15 位左右的 LCG，周期短、相关性强。
- **可移植性差**：不同平台的 RAND_MAX、算法可能差异很大。
- **缺乏分布支持**：只能得到整数，需要用户自己做缩放、映射，容易写出有偏的代码。

因此，C++ 标准在 `<random>` 中引入了一套更系统、更可组合的设施。

4.2 C++ 中的随机数

C++11 及以后的版本在`<random>`头文件中提供了一套比较完整的随机数框架，和传统的 C 语言不同，核心思想是将随机数引擎和分布分开进行，生成固定区间内的伪随机序列，然后同个数学手段转化成各种分布。

- 随机数引擎（engine）：提供一个基础的、通常是均匀分布在某个整数区间上的伪随机序列。
- 分布（distribution）：把引擎给出的整数流经过变换，生成服从某个目标分布的随机变量。

引擎和分布可以经由用户自行组合。

4.2.1 engines

标准库中定义了多种伪随机引擎模板，典型的包括：

- std::linear_congruential_engine：线性同余生成器。
- std::mersenne_twister_engine：梅森旋转算法引擎。
- std::subtract_with_carry_engine：减带借生成器（lagged Fibonacci 家族）。

std::random_device std::random_device也放在<random> 中，但它的定位和前面这些“伪随机引擎”不同，而是一个熵源适配器。理想上，它的作用是提供非确定性随机数，用于给其它引擎播种。然而，硬件不允许时，它仍生成伪随机数。
检测方式是调用rd.entropy()函数:

- 如果返回一个大于 0 的值，就表示有真实熵源；
- 如果返回 0.0，则表示它只是伪随机模拟器。

4.2.2 distribution

C++ 标准库里的分布类负责把引擎提供的整数序列转换成成各种分布类型。

4.2.3 seeding

引擎的随机性完全由初始种子决定。C++ 提供了几种播种方式：

- 固定常数种子，序列完全可复现，常用于实验对比；
- 或用 std::random_device采集若干种子，再通过 std::seed_seq扩展到大状态。

4.2.4 线程安全

C++ 标准库的随机引擎线程不安全。如果多个线程试图访问与修改同一个引擎对象，就需要外部同步。更常见、性能也更好的做法是：

- 每个线程维护一个独立的引擎实例，用不同的种子初始化。
- 在主线程中用 random_device + seed_seq 生成一组互不相关的种子分配给各线程。

5 PRNG的评价指标

PRNG的评价取决于其用途，主要有统计与密码学两类。

5.1 统计仿真

统计仿真通常将输出视为近似独立同分布样本，测量重点在于边际分布是否正确以及序列相关性是否足够低。代表性指标包括：

- **数位分布**：分桶频数/直方图、十进制 digit frequency、卡方检验等，用于发现显著偏置。
- **相关性**：自相关（lag-k correlation）、serial test（相邻二元/多元组联合分布）等，用于发现周期结构或依赖关系。
- **几何结构**：绘制 (x_i, x_{i+1}) 或 (x_i, x_{i+t}) 散点，直观呈现线性同余类生成器在低维空间的条纹/格结构。

5.2 密码学

密码学应用（key/nonce/IV/salt/token）对随机性的核心要求是**不可预测性**，而不仅是统计学意义上的均匀。因此此类PRNG评价更接近安全分析与攻击视角验证：

- **不可预测性**：给定已观察输出，攻击者在多项式时间内既无法显著区分其与真随机序列，也无法以显著大于 $\frac{1}{2}$ 的概率预测下一bit输出。
- **抗状态恢复**：仅从输出无法恢复内部状态或内部参数。
- **回溯与前向安全**：即使某时刻状态泄露，也难以回推历史输出，且通过重播种可恢复未来安全性。
- **熵与实现安全**：审计播种方式与熵源质量，并可通过重复率/碰撞率等简单实验暴露风险。

因此，统计测试（如频数/卡方）在密码学中至多是必要非充分条件：通过统计测试并不意味着安全。

6 应用实验

分别针对不同非CSPRNG实现内部统计性质对比，与CSPRNG与非CSPRNG的对比，设计了两组实验：

6.1 抽奖公平性实验

6.1.1 实验设计

实验背景 在线抽奖、抽奖小程序和随机抽取获奖者都是典型的随机化应用场景。抽奖程序内部使用的随机数源应在参与者空间上近似均匀，且不存在可被利用的结构偏差。

现实中，不少系统直接使用简单 PRNG来驱动抽奖逻辑，例如：

$$winner = LCG\_next() \bmod N,$$

或直接基于语言自带的非密码学接口。若 PRNG 质量较差或使用方式错误，则在重复活动中会造成系统性的偏差或可预测性，使抽奖不公平。

实验模型与设置 我们搭建一个简化抽奖模型，对比

- LCG
- MT

两种PRNG在抽奖公平性上的表现。

参数配置为：

- 参与者总数： $n = 10000$；
- 每轮中奖人数： $k = 100$；
- 抽奖轮数： $rounds = 2000$，总中签次数为 $Rk = 2 \times 10^5$；
- 分桶数量： $buckets = 100$，每桶包含 $n/100 = 100$ 个参与者；
- 散点图采样点数： $scatter\_points = 80000$，用于绘制 (j_t, j_{t+1})。

度量指标 在理想情况下，每个参与者中签次数近似服从

$$\text{Binomial}\left(Rk,\ \frac{1}{n}\right),\quad \mathbb{E}[\text{wins}] = \frac{Rk}{n} = 20,\quad \text{Std}[\text{wins}] \approx \sqrt{20} \approx 4.47.$$

分桶后，每个桶的期望总中签次数为

$$\mathbb{E}[\text{bucket sum}] = \frac{Rk}{buckets} = 2000.$$

统计量与可视化 对于每一种随机源，我们统计如下指标：

1. **中奖次数分布：**记第 i 个参与者的中奖次数为 C_i。在理想均匀模型下，
$$C_i \sim \text{Binomial}\left(T, \frac{1}{N}\right),\qquad \mathbb{E}[C_i] = \frac{T}{N}.$$
我们记录 $C_{\min} = \min_i C_i$, $C_{\max} = \max_i C_i$ 以及标准差。

2. χ^2 **拟合优度检验：**将每个参与者视作一个桶，比较实测频数 C_i 与理论期望 T/N，
$$\chi^2 = \sum_{i=0}^{N-1} \frac{(C_i - T/N)^2}{T/N},$$
在样本足够大时，该统计量在理想情形下近似服从自由度为 $N-1$ 的 χ^2 分布。我们据此估计 p 值，用于判断实测分布是否显著偏离均匀。

3. **简单相关性检查：**可以进一步检查中奖序列 $\{w_t\}$ 的自相关（例如统计 $w_t = w_{t+1}$ 的次数、绘制 (w_t, w_{t+1}) 散点）以发现潜在周期或局部结构。

6.1.2 实验结果与分析

总体统计量 可以看到，LCG 与 MT 的平均中奖次数都精确落在 20，说明两者在一阶矩上没有偏差。然而 LCG 的标准差为12.98，约为理论值的三倍，而 MT 的标准差 4.41 与理论值非常接近。

RNG	均值	标准差	最小值	最大值
LCG	20.00	12.98	1	61
MT	20.00	4.41	6	38

表 5: 每人中奖次数的总体统计量（$n = 10000, k = 100, R = 2000$）

如果假定抽奖完全公平，则中奖次数大于 40 次的参与者大致对应于 4.5 个标准差以上的极端事件，理论期望人数远小于 1。图 **??** 中可以看到，在 LCG 情况下仍然出现了相当数量的 “高频中奖者”（40 次以上）和 “几乎从不中奖者”，这说明 LCG 在此抽奖方案下产生了显著的不公平。

单人中奖次数直方图 如图：

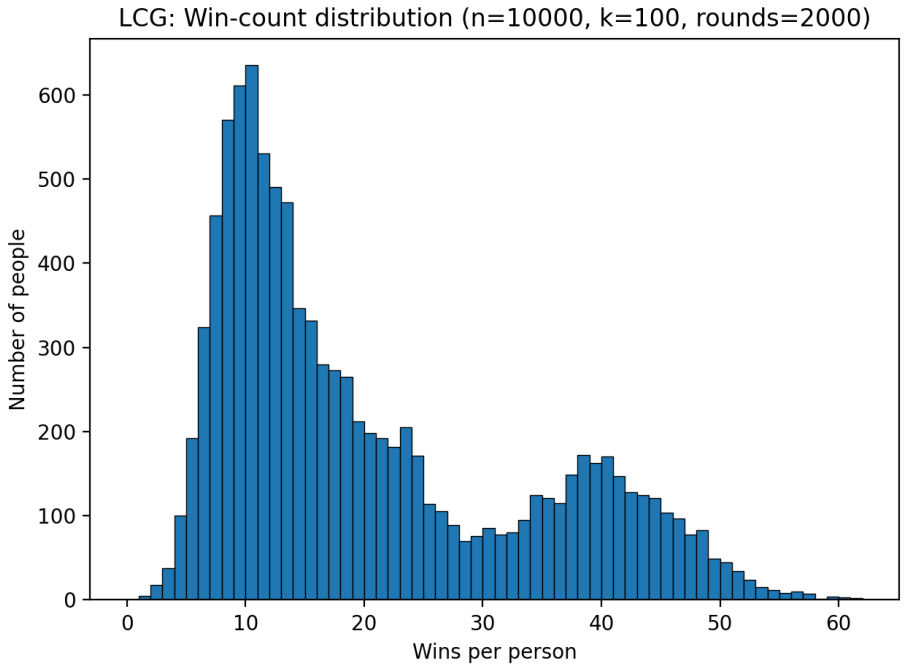

图 11: LCG

LCG 直方图呈现明显的双峰结构，一部分人在 10 次左右集中，另一部分在 35–45 次附近出现次峰，整体分布被严重拉宽，尾部也更厚。这

意味着参与者被分成了两个群体，与公平抽奖的单峰近似正态分布预期不符。

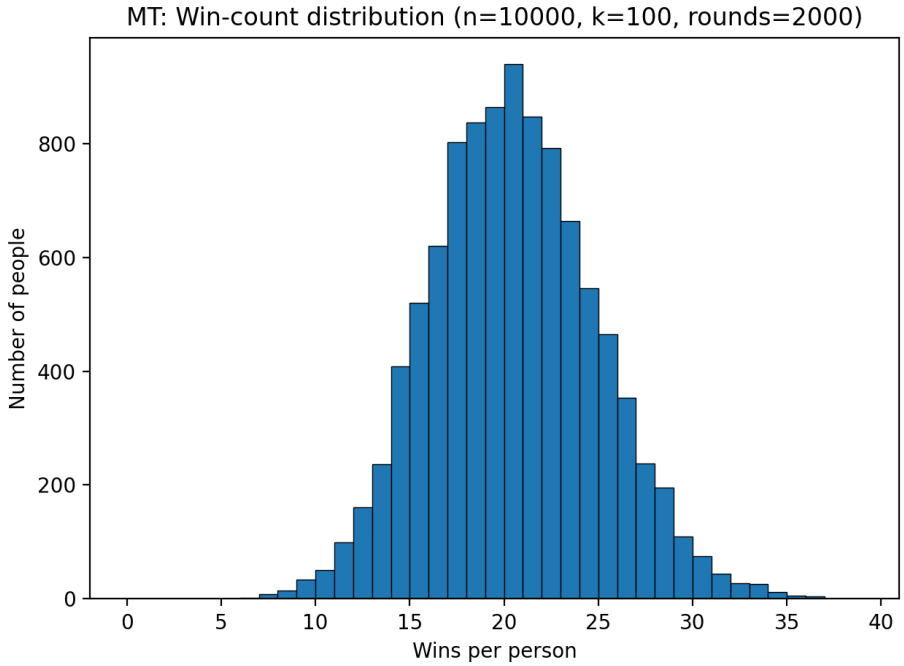

图 12: MT

而 **MT** 直方图近似一条对称的钟形曲线，峰值在 18–22 之间，左右衰减平滑，与 Binomial$(2000, 0.01)$ 或其正态近似的形状高度一致，未见异常峰或长尾。

chi-square 检验　如图。

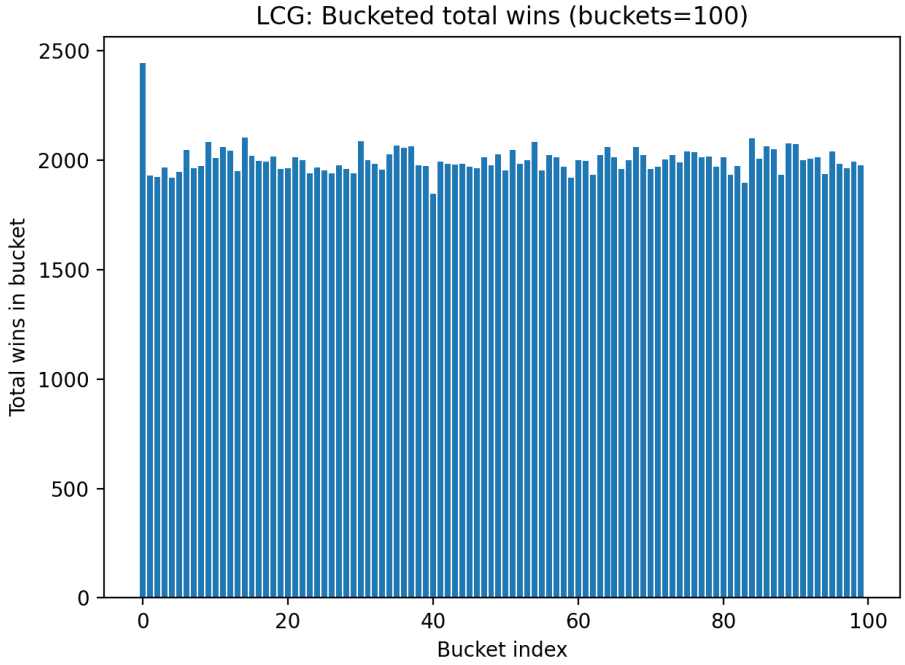

图 13: LCG：按编号分桶的中奖总次数

LCG 大多数桶在 1900–2100 之间波动，但第一个桶明显偏高），远远超出随机波动的合理范围。全局卡方统计量为 $\chi^2 \approx 210.0$，在自由度 99 的情形下远大于期望值 99，对应的 p-值极小，表明 “编号到中奖次数” 之间存在结构性偏差。

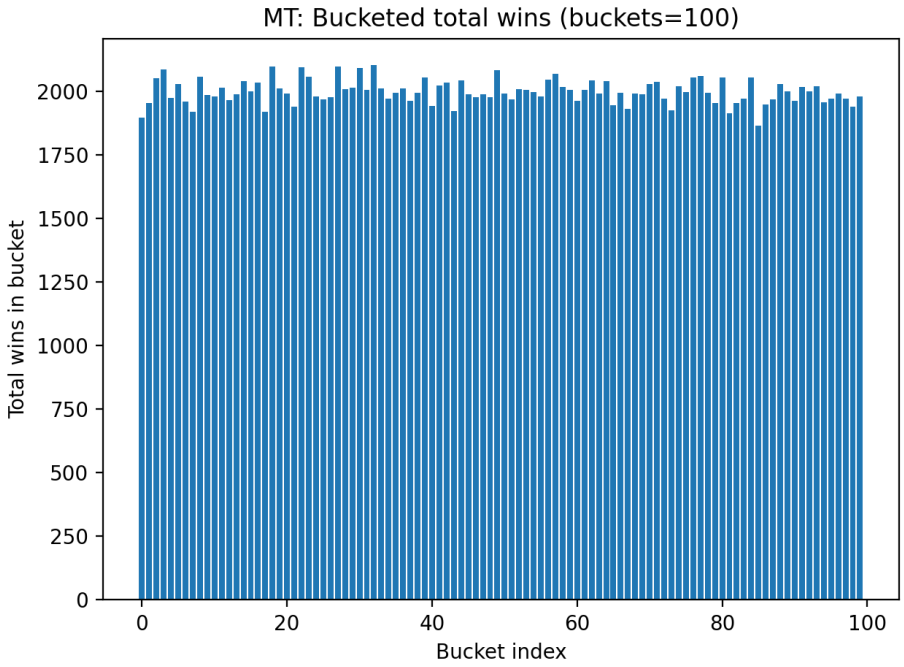

图 14: MT：按编号分桶的中奖总次数

MT 各桶的中奖总数集中在 1900–2100，没有任何一个桶呈现明显 “偏高” 或 “偏低” 的异常。对应的卡方统计量约为 $\chi^2 \approx 109.7$，略高于理论期望，但仍在合理波动范围内。

索引序列散点图　如图所示。

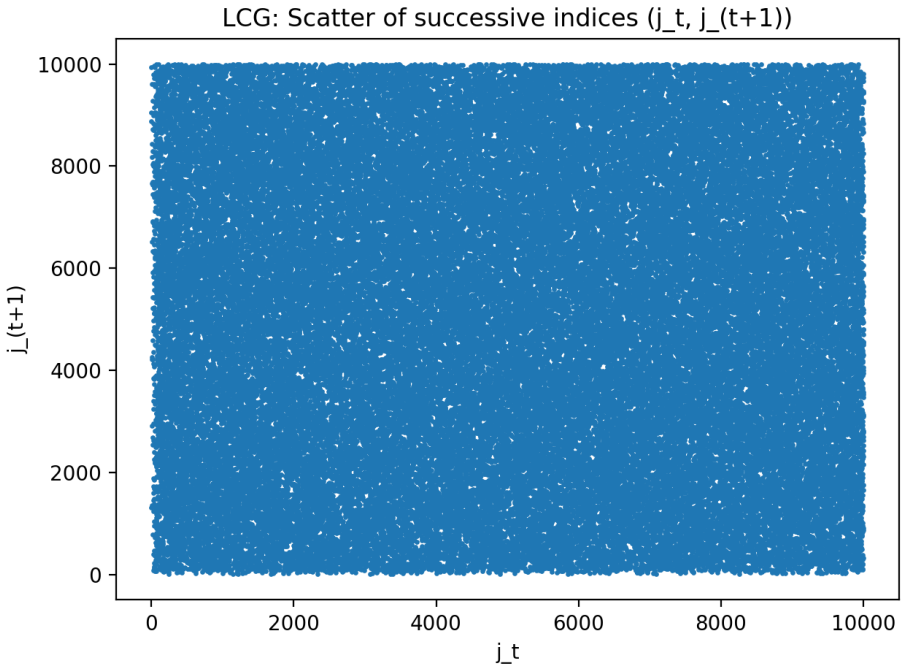
LCG: Scatter of successive indices (j_t, j_(t+1))
10000
8000
6000
4000
2000
0
j_(t+1)
0
2000
4000
6000
8000
10000
j_t

图 15: LCG散点图

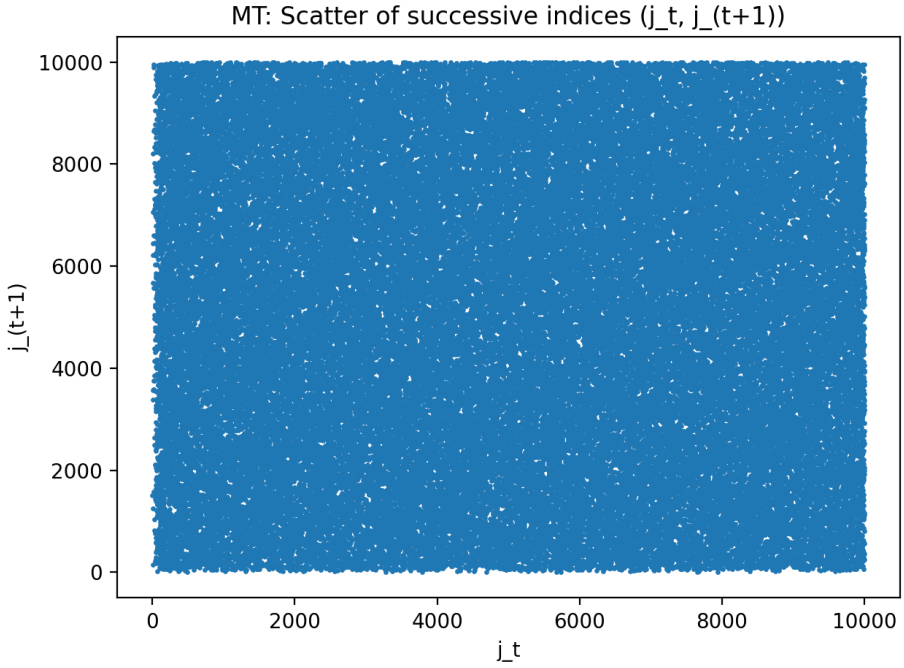

图 16: MT散点图

两种 RNG 的散点图都较为均匀，未出现明显条纹或空洞，表明 LCG 的问题主要体现在全流程组合之后的中奖次数分布上，而不是单次索引的局部相关性。

小结 在相同的抽奖规则与参数配置下， LCG 与 MT一阶均值上看似都“公平”，但 LCG 的中奖次数方差显著放大，中奖次数分布呈现双峰，桶级统计与卡方检验也暴露了对某些编号区间的系统性偏向。相比之下，MT 的统计特性与理论模型基本吻合，既未出现明显群体区分，也无明显编号区间偏置。因此，在本抽奖方案下，LCG 不应被视为公平的随机源，而 MT 至少在本实验规模和测试指标下表现为近似公平的选择。

6.2 Salt 生成实验

6.2.1 实验设计

背景 在Password-Based Key Derivation中，salt 指每条记录独立生成的一段随机值 S，与口令 P 一起输入到哈希/派生算法中。 salt 的主要作用是：即使不同用户使用相同口令，也能得到不同的存储结果，从而抑制彩虹表预计算与跨用户复用攻击。注意 salt 不要求保密（可与哈希结果一起存储），但必须满足高熵、不可预测、且每条记录唯一。因此，salt/token/nonce 等安全对象的生成应依赖 CSPRNG/DRBG 或操作系统安全随机源，而不能使用仅满足统计意义随机性的通用 PRNG。

实验目标 本实验以“生成 128 位 salt/token”为载体，对比三类随机源在低位分布与相邻相关性上的表现，并用图像化测量说明：为什么即便某些通用 PRNG 在宏观频数上看似合理，仍不适用于密码学用途；进一步从攻击者视角展示：错误播种（time-seed）与线性结构（LCG）会导致可预测性。

对比对象与实现方式 我们生成长度为 128 bit 的序列 $\{T_t\}$，分别采用：

- LCG：状态递推 $x_{t+1} = (ax_t + c) \bmod 2^{32}$。每个 token 由连续 4 个 32bit 输出拼接得到：$T = x_i \| x_{i+1} \| x_{i+2} \| x_{i+3}$。
- MT19937：同样以 4 个 32bit 输出拼接成 128bit token。
- OS CSPRNG：直接调用 secrets.token_bytes(16) 生成 16 字节随机串，并转为 128bit。

其中 LCG 与 MT 使用固定 seed ，OS CSPRNG 由系统熵源驱动。

测量指标 检查两个指标：

1. **低位边际分布**：统计 token 的低 4 bit频数分布。
2. **相邻相关性**：取每个 token 的低 16 bit 序列，绘制散点 (y_t, y_{t+1})。

此外，我们记录三类“辅助统计量”用于快速对齐实验规模与空间大小：完整 128 bit 的精确碰撞（理论上概率极低）、以及高位前缀碰撞（top 16/24/32 bit），用于解释生日悖论量级下的重复现象。

结果 本实验取样规模为 $N = 200000$。

指标	LCG	MT19937	OS CSPRNG
样本量 N	200000	200000	200000
128-bit 精确碰撞	0	0	0
Prefix-16 碰撞	137564	137522	137548
Prefix-24 碰撞	1144	1141	1165
Prefix-32 碰撞	0	4	6
Last-hex min	0	12265	12404
Last-hex max	50000	12683	12682

表 6: Salt/Token 统计

prefix collisions prefix collisions 的量级主要由“样本量 vs 空间大小”决定，受生日悖论主导。以 prefix-16 为例，前缀空间为 $2^{16} = 65536$，而 $N = 200000$ 远大于该空间，因此必然出现大量重复；prefix-24 的空间为 2^{24}，在该样本量下出现千级重复同样属于正常量级。此处，prefix collisions 仅作为规模校验与对照。

6.2.2 实验结果与分析

如图。

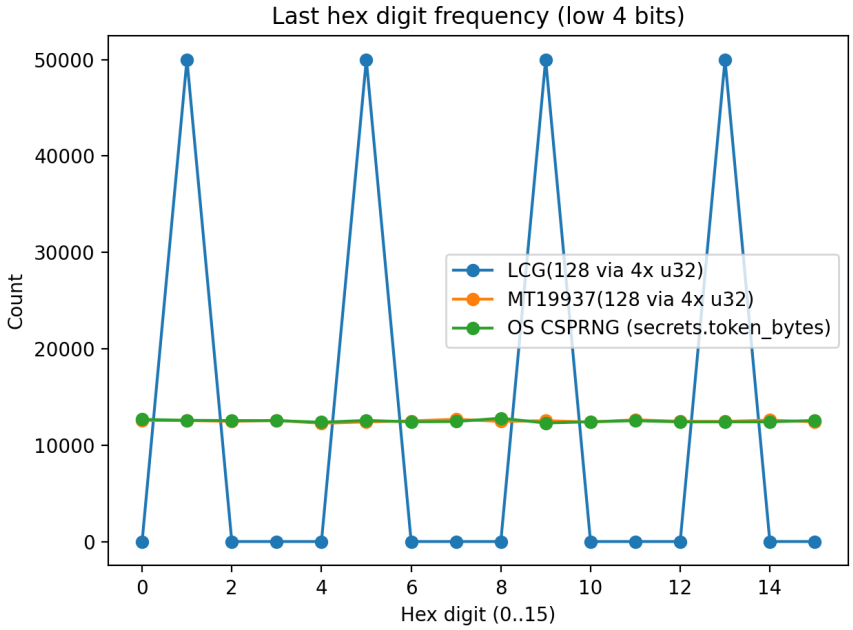

图 17: 低4bit频数分布对比

频数 如图，MT 与 OS CSPRNG 的最后一位十六进制数字频数接近均匀；而 LCG 的结果高度异常：仅有少数 digit 出现非零频数，呈现强烈的周期性与偏置，是 LCG 在模 2^k 下低位具有短周期结构的直接体现。尤其在本实现中，每个 token 由 4 个连续 32bit 输出拼接而成，因此 token 的最低 4 bit 等于“每 4 次输出中的固定位置”的最低 4 bit。对 LCG 而言，低位在步长为 4 的抽样下可能被锁定到特定同余类，从而导致低 4 bit 只能落在极少数取值上。这意味着攻击者可轻易区分该 token 是否由 LCG 生成，并可利用低位结构进一步推断内部状态，故 LCG 完全不适用于 salt/token。

散点 如图。

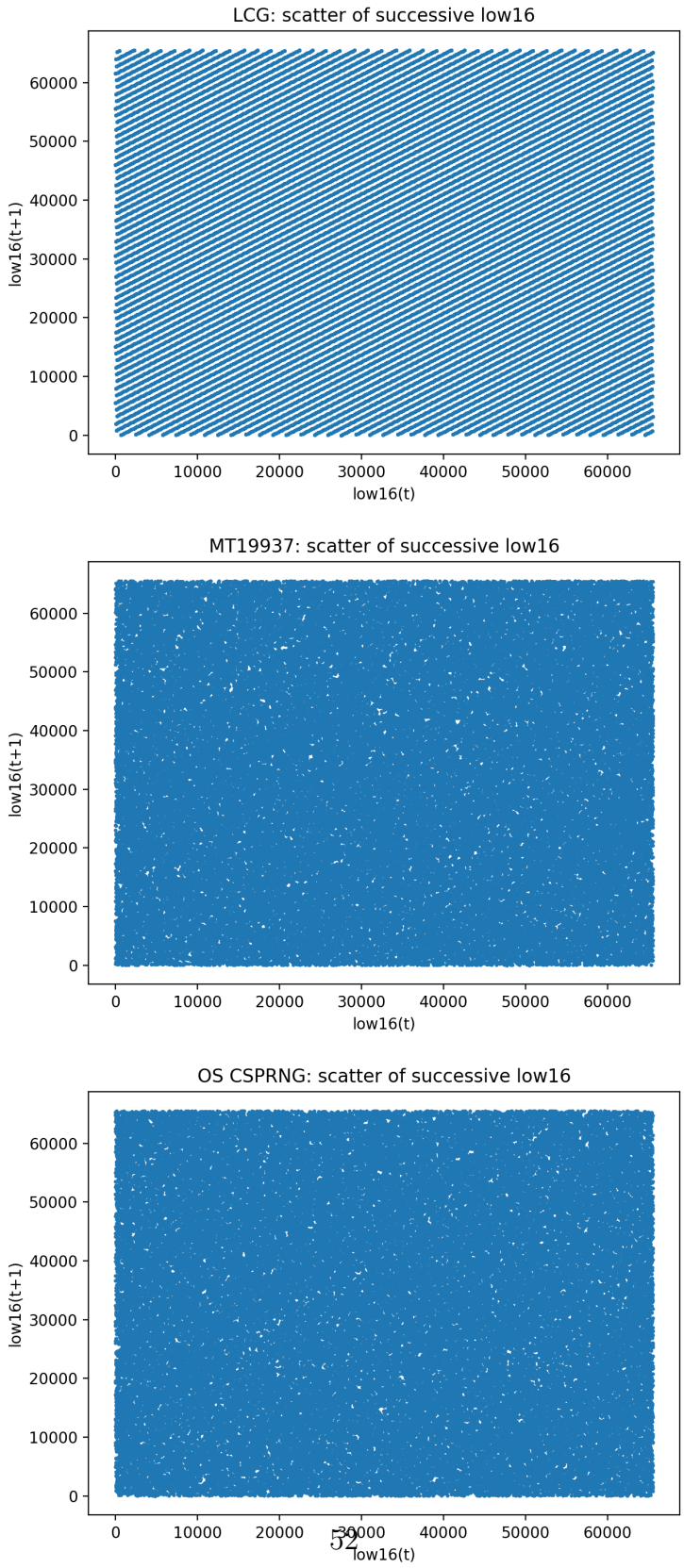

图 18: 相邻低 16 bit 散点对比

LCG 呈现清晰的晶格结构，说明相邻输出在低位存在强线性相关；而 MT19937 与 OS CSPRNG 的散点近似均匀铺满，未出现可见结构。这类条纹结构在密码学场景中意味着可预测性与可区分性风险。

攻击视角 仅凭统计图像已可判断 LCG 不适合安全对象，但密码学更关心“缺陷是否可被利用”。因此我们补充两个玩具攻击，用于说明：错误播种与线性结构会直接导向“可恢复/可预测”。

A. time-seeded导致 seed 可在小窗口内被恢复 在许多错误实现中，开发者用“当前时间戳”作为 PRNG 的 seed 生成 token/salt。时间戳熵极低，攻击者通常可从请求时间、日志、页面加载时间等估计 seed 落在很小窗口内，于是可以穷举匹配并恢复 seed。在本实验中，攻击者假设误差不超过 ± 30 秒，即成功恢复 seed：

- **LCG**：观察到首个 token 为 6c95b0aba5200e0ec4073e1523967870，在 ± 30 秒内恢复 $\hat{s} = 1768016764$。
- **MT19937**：观察到首个 token 为 bab948e82529d7dce3bb9b2f9c052a5f，同样在 ± 30 秒内恢复 $\hat{s} = 1768016764$。

由于 LCG 与 MT19937 都是确定性生成器，一旦 seed $\hat{s}$ 被恢复，攻击者即可复现实例并生成与受害者一致的后续输出序列，从而实现对后续 token/salt/nonce 的预测。这说明time-seeded 会把本应来自高熵随机源的安全对象，降维为几十次穷举即可解决的问题。

B. LCG 参数恢复：由三个连续输出恢复 a, c 并预测下一输出 LCG 满足 $x_{n+1} \equiv a x_n + c \pmod{2^{32}}$。若攻击者获得三个连续输出 x_0, x_1, x_2，则 $x_2 - x_1 \equiv a(x_1 - x_0) \pmod{2^{32}}$。当 $(x_1 - x_0)$ 在模 2^{32} 下可逆（等价于其为奇数）时，可解出 $a \equiv (x_2 - x_1) \cdot (x_1 - x_0)^{-1} \pmod{2^{32}}$，并进一步 $c \equiv x_1 - a x_0 \pmod{2^{32}}$。本实验的脚本输出给出一组成功恢复的实例：

$$x_0 = 1015568748, \quad x_1 = 1586005467, \quad x_2 = 2165703038.$$

恢复得到 $\hat{a} = 1664525$, $\hat{c} = 1013904223$，与真实参数完全一致；预测下一输出 $\hat{x}_3 = 3027450565$，与真实 $x_3 = 3027450565$ 一致。该结果表明：LCG

的线性结构不仅会在统计图中暴露，更能被攻击者直接利用实现恢复与预测。

在 PKCS 相关实践中，随机性不仅用于“看起来随机”，更关键是应满足不可预测性。salt/token/nonce/key 等安全对象，不应该使用统计用途的 PRNG（如 MT、PCG），而应使用 OS CSPRNG 或标准化 DRBG。

7 结语

7.1 报告总结

随机数生成是一个有趣而又充满细节的话题。本文首先从算法与数学结构出发，回顾了若干典型 PRNG 的设计思路与优缺点；随后，讨论了 CSPRNG 的安全性定义与标准演进，并通过 PKCS#1、PKCS#5 等规范展示了随机性在实际密码协议中的具体落地方式。最后通过两个实验，将不同PRNG与CSPRNG之间性能的区别直观化。

PRNG的选择依赖于具体场景。在数值仿真、Monte Carlo 和算法随机化中，应选择高速且统计性质良好的非密码学 PRNG（如 PCG、xoshiro 与 MT 系列），并合理设置种子以保证可重复性；而在密钥、nonce、salt、token 等安全敏感场景，则必须使用操作系统 CSPRNG 或符合 NIST SP 800-90A 等规范的 DRBG。

7.2 个人反思

总的来说，随机数是一个上手做之后才发觉其应用之广泛而复杂的话题。受能力与时间限制，本报告还有诸多不足，如未来得及对所有PRNG进行 dieharder 测试、完成详尽的攻击测试等。但在完成报告的过程中，我学会了怎么写latex和linux的基本操作。同时，由于对python的不熟悉，**部分python代码由ai辅助生成**。如今ai工具的强大能够极大程度减少对机械人力的需求，但真正的思考还是得自己来，否则只会成为Ctrl+C、Ctrl+V的受害者。

感谢阅读!

参考文献

[1] T. E. Hull and A. R. Dobell. Random Number Generators. *SIAM Review*, 4(3):230–254, 1962. doi:10.1137/1004061. `https://doi.org/10.1137/1004061`.

[2] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic Attacks on Pseudorandom Number Generators. In Serge Vaudenay (ed.), *Fast Software Encryption (FSE)*, pp. 168–188, 1998. Springer. doi:10.1007/3-540-69710-1_12.

[3] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.* Technical Report HMC-CS-2014-0905, Harvey Mudd College, 2014. `https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf`.

[4] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[5] NIST. *SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* 2015. `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf`.

[6] J. Jonsson and B. Kaliski. *RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2.* IETF, 2016. `https://www.rfc-editor.org/rfc/rfc8017`.

[7] S. Moriarty, B. Kaliski, A. Rusch, and M. Szydlo. *RFC 8018: PKCS #5: Password-Based Cryptography Specification Version 2.1.* IETF, 2017. `https://www.rfc-editor.org/rfc/rfc8018`.

[8] Python Software Foundation. *random — Generate pseudo-random numbers*. Python documentation. https://docs.python.org/3/library/random.html.

[9] Python Software Foundation. *secrets — Generate secure random numbers for managing secrets*. Python documentation. https://docs.python.org/3/library/secrets.html.

[10] cppreference.com. *<random> library and std::mt19937*. https://en.cppreference.com/w/cpp/header/random.

[11] Oracle. *Class java.util.Random*. Java SE documentation. https://docs.oracle.com/en/java/javase/.

[12] Oracle. *Class java.security.SecureRandom*. Java SE documentation. https://docs.oracle.com/en/java/javase/.

[13] The Go Authors. *Package crypto/rand* and *Package math/rand*. Go documentation. https://pkg.go.dev/crypto/rand https://pkg.go.dev/math/rand.

[14] Rust getrandom crate. *getrandom: Retrieve random data from system sources*. https://docs.rs/getrandom/.