伪随机数生成器调研报告

摘要

正文

- 随机数的定义
  - 真随机数
  - 伪随机数
- 伪随机数生成器的评价方法
  - 随机有效性
    - 普通PRNG有效性检验方法
    - 期望测试
    - 自相关测试
    - 离散傅里叶检测
    - 总结
    - CSPRNG检验方法
    - 基于密码学理论的安全性证明

基于密码学原语所设计的随机数生成器

基于数学难题的随机数生成器

- 基于统计学原理的安全性测试

线性复杂度测试

- 数列生成效率
  - 时间复杂度分析
  - 基准测试
- 常见的伪随机数算法评估
  - 普通随机数生成器算法
    - 平方取中法
    - 原始平方取中算法
    - Weyl序列平方取中算法
    - 线性同余法
    - 原始线性同余算法

Park-Miller算法
PCG算法
延迟斐波那契法
原始延迟斐波那契算法
改进型斐波那契法
线性反馈位移寄存器算法
原始线性反馈位移寄存器算法
Mersenne Twister算法
密码学安全的随机数生成器算法
Blum Blum Shub算法
Windows密码学安全伪随机数生成器
总结
主流语言/操作系统伪随机数生成器分析
伪随机生成器攻击尝试
基于数学理论的伪随机数攻击尝试
LCG攻击尝试
LCG参数乘数、增量、与模数已知
LCG参数增量未知
LCG参数增量与乘数未知
针对Java Random类的攻击尝试
伪随机数生成器的应用与展望
伪随机数生成器的密码学应用
伪随机数生成器的模拟应用
伪随机数生成器的程序开发应用
展望随机数生成器的发展
参考资料

姓名：黄政东

学号：12212230

摘要

现如今，由于对可靠随机序列需要，随机数生成器受到了广泛的应用于发展。本报告尝试对现有的随机数生成器进行调查研究，尝试阐述随机数概念的基本定义与类别，总结现有的伪随机数生成器评价方法并对其原理进行分析。同时，报告阐述了常见的伪随机数生成算法，并对常见的算法进行了理论分析与java代码实现，同时使用NIST随机数生成器检测工具对生成序列进行评估。随后，报告总结了现有主流语言与操作系统的随机数生成器实现，同时尝试使用数学理论对LCG算法及java Random类进行了攻击尝试。最后，报告总结了现有伪随机数的应用常见，并对伪随机数生成器的发展做出展望。本报告附件中包含了测试过程的原始数据、报告处理脚本与基于Java的部分伪随机数生成算法代码实现，并编写了统一的基准测试，该项目实现具备一定的可扩展性，开发者可以基于此项目较为方便地进行伪随机数生成器算法的开发与测试。

正文

随机数的定义

在计算机科学中，随机数往往被分为真随机数与伪随机数两大类，其两者之间存在较为明确的划分界限。综合传统的划分方式，本报告为各类随机数做出如下定义与阐述。

真随机数

真随机数（True-random number, TRN），是指通过完全随机的物理过程而产生的完全随机的数字序列[2]，该数字序列服从一定范围内的均匀分布，且每一项数的选取与数列中的其他项无关。真随机数的生成依赖于可靠且充分随机的物理过程，在物理世界中，大气噪声，放射性衰变，电路热噪声等物理过程可以在现有的物理学理论下视为可靠的完全随机物理过程，而与此同时，宏观世界中较为常见的如抛掷骰子，用户敲击键盘等事件，由于其难以被完全精确地控制，事件具备一定的随机性与不可预测性，因此可以在特定条件下被视为可靠的完全随机物理过程。

伪随机数

伪随机数（pseudo-random number, PRN），是指通过一系列特定数学算法所生成的，满足统计学意义上的随机性质的数字序列[2]。相较于真随机数，伪随机数的最大特点在于其数字序列具有较强的确定性，如对于采用种子生成法所生成的伪随机数序列，当算法的初始种子确定时，该算法所生成的随机数序列可被完全确定。现有的共识认为，由于数学算法所具备的确定性，仅依赖于数学算法的随机数生成

器只能生成伪随机数序列[4]，真随机数生成器的实现需要依赖于外部的物理过程。基于密码学安全理论，伪随机数又可分为强伪随机数与弱伪随机数， 三类随机数大致可通过以下性质进行划分：

类别	随机性	不可预测性	不可重现性
弱伪随机数	√	×	×
强伪随机数	√	√	×
真随机数	√	√	√

伪随机数生成器的评价方法

伪随机数生成器（pseudo-random number generator，PRNG），是一系列用于生成伪随机数的数学算法。现有的伪随机数生成器往往采用种子生成法：算法接收一个一段较短的，随机性良好的比特序列（种子）作为初值，在随后的迭代与运算过程中生成满足一定随机性质的比特序列（伪随机数）[1]。在伪随机数生成器的质量评估中，随机有效性与数列生成效率是两个重要的评价指标，可以作为衡量伪随机数生成器实用性的重要参考。

随机有效性

现有的划分体系将伪随机数生成器分为普通伪随机数生成器与密码学伪随机数生成器（Cryptographically secure pseudo-random number generator, CSPRNG）。依据德国联邦安全局所制订的BSI标准[3]，伪随机数生成器的安全性可以被分为如下四个等级：

- K1 – 产生的随机数序列彼此不同的概率应该是很高的
- K2 – 根据某些统计测试，无法分辨产生的序列和真随机序列。
- K3 – 给定任何子序列，任何攻击者都无法计算后续序列或者生成器的内部状态
- K4 – 给定生成器的内部状态，任何攻击者都无法计算之前的序列或者生成器之前的状态

上述等级中，满足K1等级的伪随机数生成器即可作为普通伪随机数生成器进行使用，而密码学安全级别的伪随机数生成器则需满足K3与K4标准，以提供更为安全可靠，具备对抗攻击性的随机数序列。同时，上述标准下的K1，K2标准往往可采用统计学方法进行验证，而K3，K4标准往往结合统计学与密码学理论工具进行验证或证明。

普通PRNG有效性检验方法

普通PRNG有效性检验主要围绕随机数序列是否满足统计学意义上的随机序列性质，因此主要采用统计学方法进行检验。统计学检验方法包含了多种统计测试工具，通过分析随机数生成器所生成的极大量数字

序列的统计学性质，统计学方法可以可度量地评价随机序列的均匀分布程度与各项独立程度。中国密码行业标准《随机性检测规范》[6]与美国国家标准与技术研究院 随机数生成测试方法[7]包含了用于测试生成器生成序列随机性的一系列测试方法，主要围绕序列频率，序列独立性，序列周期性等序列性质进行测试，本报告尝试阐述标准检测规范中常用的三种较为典型的随机性统计检测方法的检测过程并分析其背后的统计学原理。

期望测试

期望测试是最为经典的基于生成序列频率开展的随机性测试。该通过测试所生成的数字序列期望是否与相应的均匀分布期望相等以判断序列的随机性。以二进制数字序列为例，若为完全随机分布，序列中任意比特位的取值可视为离散型随机变量，其概率分布列如下所示

$$P(x = i) = p_i$$

x	0	1
p_i	0.5	0.5

依据离散型随机变量期望公式，0–1完全随机分布的期望如下所示

$$E(x) = \frac{1}{2}\sum_{j=0}^{1} p_i \cdot j = 0.5$$

在实际应用中，可采用数学方法计算算法所生成的数列期望，以判断其是否通过期望测试。但采用数学方法往往较为困难，因此可采用数理统计工具，通过抽样结果进行检验。

对于随机数生成器所生成的0–1序列，若该序列各项服从0–1完全随机分布，则可将该序列作为总体 x 的一个样本，其性质满足以下规律

$$seq: \{x_1 \cdots x_n\}, \forall i \in \{1 \ldots n\}, P(x_i = j) = p_{ij}$$

x_i	0	1
p_{ij}	0.5	0.5

由数理统计原理可知，当样本量充分大时，该随机序列的样本均值等于期望应等于总体期望[5]，其推导过程如下：

由辛钦大数定理，对于任意给定的实数ε>0，有

$$\lim_{n\to\infty} P\left\{\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| < \varepsilon\right\} = 1$$

因此可由期望公式得

$$for\ n \to \infty, E(\bar{x}) = E\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) = \frac{1}{n}\sum_{i=1}^{n} E(x_i) = \frac{1}{n} \times nE(x) = E(x)$$

因此，当样本量充分大时，可以通过判断所生成的二进制序列各项总体均值是否近似等于0.5以判断是否服从0–1完全随机分布。

由于随机性测试过程中，往往会生成较为庞大的随机序列，因此在实际应用可能会采用人工判断均值与期望的相近程度以决定是否存在非随机异常。参考假设检验的基本原理，本报告认为也可以采用二项检验的方式进行可量化的服从评估。二项检验可以较好的检验二元变量比例在一定置信水平下是否等于特定的假设值，其具体过程如下：

1. 确立原假设与备择假设

依据上述情况，我们设立如下的原假设（H_0）与备择假设（H_1）

$$H_0: E(\bar{x}) = E(x)$$
$$H_1: E(\bar{x}) \neq E(x)$$

2. 确定显著性水平α

显著性水平α代表了在原假设正确的情况下拒绝原假设的概率，常选取0.05或0.01作为取值。

3. 计算检验统计量p值

由于备择假设为不等于随机，因此，可采用双尾检验法[8]计算其检验统计量p，可推得计算公式如下

$$p\text{-value} = 2 \times \min\left(\sum_{k=0}^{x}\binom{n}{k}p_0^k(1-p_0)^{n-k}, \sum_{k=x}^{n}\binom{n}{k}p_0^k(1-p_0)^{n-k}\right)$$

4. 得出结论

将p值与显著性水平α进行比较，若p≤α，则推翻原假设，认为在该显著性水平下有足够证据证明随机数序列的期望与0–1完全随机分布期望不相等，反之则支持原假设，认定所生成的随机数序列通过了期望检验。

二项假设的优势在于其不依赖于正态分布近似，因此在选取样本量较小的条件下，仍可较为可靠的判断一定置信水平下的判断所生成的随机数序列是否满足0–1完全随机分布。

在现行的密码学规范中，往往采用单比特频数检测方法对随机数序列进行期望检测，单比特数检测法基于假设检验的原理进行测试，依赖于大样本量下的正态分布近似，其大致的检测流程与原理如下：

1. 将比特序列中的0,1分别转化为–1与1，即代入如下公式

$$seq: \{x_1 \cdots x_n\}, \forall i \in \{1 \ldots n\}, \mathrm{A}_i = 2x_i\text{-}1$$

2. 计算测试统计量S

由于随机序列中每个比特的权重被视为一致，因此可采用直接求和的方式，计算该随机序列的测试统计量S，其公式如下

$$S = \frac{\sum_{i=1}^{n}(2x_i - 1)}{\sqrt{n}}$$

若为严格地0–1随机分布，该顺序统计量S应当接近于0。随后，该测试方法通过正态分布近似进行进一步处理。

3. 计算P值并与显著性水平进行比较

在大样本量下，二项分布的分布曲线可近似为正态分布，因此，该测试方法采用标准正态分布中误差互补函数进行进一步处理，双尾测试下的p值可以通过如下公式求解

$$p = \mathrm{erfc}\left(\frac{|\mathrm{S}|}{\sqrt{2}}\right)$$

随后将p值与显著性水平α进行比较，若p≤α，则认为有足够证据表明该序列不满足0–1完全随机分布，反之则该随机数生成器通过单比特频数检测。

相较于本报告所提出的二项检验，单比特数检验依赖于正态分布近似，由于其避免了组合数的计算，计算效率可能更高，但该检测方法需要较大的随机数序列才能得到准确的生成结果。

自相关测试

自相关检验用于检测生成序列各项之间的独立性程度。该测试通过比较逻辑左移特定位数后的新序列与原序列间的相似程度而检验所生成序列内部的自相关程度。对于理想的随机序列，逻辑左移后得到的新序列应与原序列具备较小的相似程度。

为运用统计学方法可度量地确定原序列与新序列间的关联程度，自相关测试往往采用对两序列按位异或的操作形成待测序列，随后检测待测序列是否可以通过期望测试，其大致检测流程与原理如下：

1. 通过将原序列与左移序列按位异或生成待测序列，并求出序列各项总和A(d)，其公式如下

$$A(d) = \sum_{i=0}^{n-d-1} (x_i \oplus x_{i+d})$$

2. 由上述分析可知，若两序列中不存在明显的相关性，则异或操作后的新序列亦服从0–1完全随机分布，而由期望测试可知，当样本量相当大时，该二项分布可近似为正态分布，各项和A(d)代表二项分布中的成功次数，依据数理统计原理，可以推导出服从标准正态分布的统计量形式，其推导如下

   棣莫佛–拉普拉斯中心极限定理[5]讨论了二项分布下的随机变量序列的中心极限性质，其形式如下

$$\lim_{n\to\infty} P\left\{\frac{\eta_n - np}{\sqrt{np(1-p)}} \le x\right\} = \lim_{n\to\infty} P\left\{\frac{\sum_{i=1}^{n} X_i - np}{\sqrt{np(1-p)}} \le x\right\} = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt = \Phi(x)$$

   其中，$\{\eta_n\}$为服从参数为n,p的0–1二项分布的随机变量列，η_n代表随机变量列各项总和。

   由上述公式可构造服从标准正态分布的统计量V，其形式如下

$$V = \frac{A(d) - (n-d)p}{\sqrt{(n-d)p(1-p)}} = \frac{A(d) - \frac{(n-d)}{2}}{\sqrt{\frac{(n-d)}{4}}} = \frac{2\left(A(d) - \frac{n-d}{2}\right)}{\sqrt{n-d}}$$

3. 依据假设检验的方法对统计量V进行进一步处理，采用双尾检测形式下的误差概率计算公式求解其P值，其公式如下

$$p\text{-}value = \mathrm{erfc}\left(\frac{|V|}{\sqrt{2}}\right)$$

4. 将求解p值与所设定的显著性水平α进行比较，若p≤α，则表明序列内部存在较为显著的相关性，无法通过自相关性检验，反之则测试通过。

自相关测试通过逻辑左移异或与期望测试的方式，能够有效检验生成序列内部各项间的相关性程度，同时具有期望测试所具备的计算较为简便的特点。

离散傅里叶检测

离散傅里叶检验主要检测生成序列内部是否存在明显的周期性特征。离散傅里叶变换可以将离散的生成序列转换至频域，通过检测变换频谱图的尖峰高度判断是否存在明显的周期性特征。若该序列为完全随机序列，所得频谱图尖峰高度应低于某个门限值。该检测方法的大致流程与原理如下：

1. 将比特序列中的0,1分别转化为–1与1，得到新序列$\{A_n\}$，具体方程如下

$$seq: \{x_1 \cdots x_n\}, \forall i \in \{1 \ldots n\}, \mathrm{A}_i = 2x_i\text{-}1$$

2. 对新序列进行傅里叶变换得到复数序列 $\{f_0 \ldots f_{n-1}\}$，由于变换具有对称性，因此只需要检测前n/2项的尖峰高度是否低于门限值即可，故求解该序列前n/2项复数的模并记为新序列，其方程如下

$$\{F_{\frac{n}{2}}\} = \{mod_j\}, 其中\ \forall j \in \left[0,\ \frac{n}{2} - 1\right], mod_j\ =\ |f_j|,$$

3. 计算门限值T，并记录上述$\{F_{n/2}\}$序列中小于门限值T的项个数N_1

门限值T定义为：对完全随机序列进行上述处理，则超过该门限值的尖峰数量与总数比例不超过β。由傅里叶变换可知，由于其模数实数部分与虚数部分均服从正态分布N(0,n/2)[9]，则可定义服从于自由度为2的卡方分布的统计量Y，即有

$$Y = \left(\frac{C_i}{\sigma}\right)^2 + \left(\frac{S_i}{\sigma}\right)^2 \sim \chi^2(2), P(Y) = \frac{1}{2}e^{-\frac{Y}{2}}$$

定义统计量Z=Y/2，则由上述分布有

$$\int_{z_c}^{\infty} e^{-z} dz = \beta, T = \sqrt{nZ_c}$$

故可求得T的表达式如下

$$T = \sqrt{-n \ln \beta}$$

随机性测试标准中往往采用β=0.05为参数进行检验，在带入并求出门限值T后，统计$\{F_{n/2}\}$序列中小于门限值T的项个数，记为量N_1

4. 计算标准正态分布统计量d

在美国国家标准与技术研究院 随机数生成测试方法中，由中心极限定理与二项分布正太近似所得到的正态分布统计量d的公式如下

$$d = \frac{\mathrm{N}_1 - \mathrm{N}_0}{\sqrt{\beta(1-\beta)\frac{n}{4}}}$$

然而，后续的研究证明该统计量并不能精确的被近似看作标准正态分布， Pareschi等人对此进行了更为精确的修正[11]，修正后的统计量公式如下，该公式亦被新版中国密码学标准所采纳

$$d = \frac{\mathrm{N}_1 - \mathrm{N}_0}{\sqrt{\beta(1-\beta)\frac{n}{3.8}}}$$

5. 采用双尾检测形式下的误差概率计算公式求解P值，P值求解公式如下

$$p\text{-}value = erfc\left(\frac{|d|}{\sqrt{2}}\right)$$

随后与显著性水平α进行比较，若p≤α，则表明序列内部存在较为显著的周期性，无法通过离散傅里叶检验，反之则测试通过。

总结

普通PRNG有效性检验往往围绕完全随机数列的性质进行检验，在上述分析的三种有效性检验方法中，检验方法首先选取随机数列所具有的特殊性质（如0–1完全随机均匀分布），对伪随机数生成序列进行处理，并构造统计量以反应生成序列与完全随机序列的差异，随后采用抽样统计原理与二项分布正太近似等方式将该统计量转化为特定的标准分布，并求解其P值于显著性水平进行比较。

上述过程被广泛应用于各种随机性检验方法中，中国密码行业标准《随机性检测规范》中的多数检测方法均采用如上过程进行检验，该方法可以可量化地分析生成序列与完全随机序列的近似程度，并得出一定显著性水平下该伪随机数是否可近似视为完全随机序列的结论，可以有效评估伪随机数生成器的随机有效性。

CSPRNG检验方法

密码学安全的伪随机数生成器在普通伪随机数生成器标准下进一步要求生成器具备一定的抗攻击性，因此要求对其安全性进行进一步的检验。生成器安全性的检验可分为 基于密码学理论的安全性证明 与 基于统计学原理的安全性测试两类，本报告尝试对此进行阐述。

基于密码学理论的安全性证明

由于CSPRNG往往是基于密码学理论而构造出的具备可靠安全性的随机数生成器，因此，设计者往往运用理论工具对生成算法进行严格地安全性证明。基于密码学理论所设计并证明的伪随机数生成器可主要分为以下两类

基于密码学原语所设计的随机数生成器

该类伪随机数生成器往往采用基本密码学原语进行构建，如基于哈希函数的CSPRNG标准Hash_DRBG，基于块加密算法的CSPRNG标准CTR_DRBG，由于其基于成熟且有限条件下安全的密码学基础算法，开发者往往依赖密码学原语的安全性对伪随机数生成器安全性进行证明。

基于数学难题的随机数生成器

该类伪随机数算法往往基于被公认的困难数学难题而设计，如基于大素数分解难题的Blum Blum Shub算法，基于离散对数难题的Blum—Micali算法，该类算法由于所基于数学问题在现有理论或计算机技术下不存在快速的求解算法，因而可以在一定条件下保证该算法的安全性。

基于统计学原理的安全性测试

基于统计学原理的安全性测试往往参考已有的伪随机数生成算法进行测试，其中最为经典的便是基于LFSR的线性复杂度测试，本报告将尝试以此为例阐述其原理

线性复杂度测试

线性复杂度测试采用Berlekamp–Massey算法，通过计算生成序列各子序列的最短线性递推式，而线性递推式则描绘了线性反馈移位寄存器（LFSR）的行为，对于完全随机序列，其各子序列的LFSR长度应该充分长，因此，较短的LFSR表明该序列与弱随机序列或非随机序列相似，因此可以用于检验其安全性，其大致过程与原理如下

1. 将待测序列划分为N个长度为m的非重叠子序列，舍弃多余的比特数，N与m的关系如下

$$N = \left\lfloor \frac{n}{m} \right\rfloor$$

2. 使用Berlekamp–Massey算法计算每一段子序列的线性复杂度L_i, 其证明与求解过程大致如下[12]

线性递推式的定义为：对于给定的数列$\{a_n\}$与数列$\{r_m\}$,$\{r_m\}$为$\{a_n\}$的线性递推式当且仅当满足如下性质

$$\forall m+1 \le i \le n, a_k = \sum_{j=1}^{m} r_j \, a_{n-j} \; (m+1 \le n)$$

Berlekamp–Massey算法采用递归方法进行实现，假定先前过程已求得序列前i–1项的最短线性递推式为$R_c=\{r_m\}$，R_c的下标c为修改线性递推式的次数。考虑前i项序列的最短线性递推式：

$$设\Delta = \sum_{j=1}^{m} r_i \, a_{n-j} - a_j$$

若Δ=0，则该序列也是前i项序列$\{a_i\}$的最短线性递推数列

若Δ不为0，则对原R_c序列进行改造为R_{c+1}，修改过程如下

a. 初始情况：若c=0，则a_i为序列中第一个非零数，则将R_1置为含i个0的序列{0,0,...,0}即可

b. 递推情况：当c>0时，需构造零一线性递推式式R'，使得$R_{c+1}=R_c+R'$所得的R_{c+1}为$\{a_i\}$的最短线性递推数列，则$R'=\{r'_m\}$需满足如下性质：

$$\forall m' + 1 \le k \le i - 1, \sum_{j=1}^{m'} r'_j \, a_{k-j} = 0$$

选定历史版本R_{c-1}，记录其w为使R_{c-1}变得不合法的第一个位置，并设定mul的关系如下

$$mul = \frac{delta_i}{delta_w}$$

可构造序列如下，该序列包含i−w−1项0，可满足上述条件中对R'序列性质的要求

$$R' = \{0,0,\dots,0,mul,-mul \times R_{c-1}\}$$

因此，我们找到了前i项序列$\{a_i\}$的最短线性递推数列

遵循上述流程我们可以求得所给定序列的最短线性递推式，因此，可对每段子序列采用如上算法计算其最短线性递推式长度，并将其记为数列$\{L_m\}$

3. 计算理论完全随机序列的理论均值 μ与各项参数Ti，NIST标准所给定的公式如下：

Under an assumption of randomness, calculate the theoretical mean μ:

$$\mu = \frac{M}{2} + \frac{\left(9 + (-1)^{M+1}\right)}{36} - \frac{\left(M/3 + 2/9\right)}{2^M}.$$

For each substring, calculate a value of T_i, where $T_i = (-1)^M \bullet (L_i - \mu) + 2/9$.

4. 设置序列{v0...v6}，依据NIST标准，序列各项值定义如下

Record the T_i values in $v_0, \dots, v_6$ as follows:

If:	$T_i \le -2.5$	Increment v_o by one
	$-2.5 < T_i \le -1.5$	Increment v_1 by one
	$-1.5 < T_i \le -0.5$	Increment v_2 by one
	$-0.5 < T_i \le 0.5$	Increment v_3 by one
	$0.5 < T_i \le 1.5$	Increment v_4 by one
	$1.5 < T_i \le 2.5$	Increment v_5 by one
	$T_i > 2.5$	Increment v_6 by one

6. 计算统计量X^2(obs)，并求出其P值与显著性水平进行比较，NIST标准所提供的公式如下

Compute $\chi^2(obs) = \sum_{i=0}^{K} \frac{(v_i - N\pi_i)^2}{N\pi_i}$, where $\pi_0 = 0.010417$, $\pi_1 = 0.03125$, $\pi_2 = 0.125$, $\pi_3 = 0.5$, $\pi_4 = 0.25$, $\pi_5 = 0.0625$, $\pi_6 = 0.020833$ are the probabilities computed by the equations in Section 3.10.

Compute *P-value* = **igamc** $\left(\frac{K}{2}, \frac{\chi^2(obs)}{2}\right)$.

通过统计学方法，上述检测方法能够可量化的评估生成序列相较于LFSR生成序列是否具备一定的复杂程度，从而能够较为有效地检验随机数生成器的安全性。

数列生成效率

数列生成效率主要通过时间复杂度分析与基准测试两类方法进行开展，报告尝试对此进行阐述

时间复杂度分析

在密码学领域下的时间复杂度分析主要采用时间复杂度分析理论中的RAM模型进行分析，首先，时间复杂度分析将算法具体执行步骤抽象为理想模型下的计算机元指令操作并假定每个元指令的操作时间相等，随后通过数学分析使用渐进复杂度的表示形式表示算法运行时间与输入参数大小的增长关系，从而实现对算法时间消耗增长的数学描述，需要注意的是，在密码学领域下的输入参数大小往往以输入数值（组）的二进制形式长度进行计算。时间复杂度分析能够较好的反应算法耗时随输入增长的变化趋势，该趋势主要决定了算法的运行效率，因此该分析方法十分重要。但常见的时间复杂度分析往往忽略了低次项数与各项系数，因此可能难以对算法耗时做出精确的数学描述，在特定的条件下，该算法所忽略的这些系数可能会较为显著的影响算法的执行效率。

基准测试

基准测试主要通过设立统一的算法测试平台对各算法效率进行测试，通过尽可能保证测试环境的与输入参数的一致性，基准测试能够较为公平有效地对各算法的实际运行耗时进行对比，从而对算法的生成效率进行评估。然而与此同时，基准测试要求对特定算法进行基于特定平台的代码实现，部署公平有效地测试环境往往较为复杂。同时，基准测试的算法运行耗时往往存在一定误差，需要进行多次测试以降低误差的干扰。对于耗时较短的算法，由于其生成耗时结果受到误差影响可能较为严重，因此可能无法测得具有参考价值的运行耗时结果，需要扩增输入大小以提高耗时测测量精度。

常见的伪随机数算法评估

由前文分析可知，伪随机数生成器可依据其安全性划分为普通PRNG与CSPRNG两类生成器。随着密码学与数学理论的发展，现如今学界已提出数百种随机数生成算法，其中许多已被广泛应用。现有的数百

种随机数算法中，很大一部分算法是基于一些较早的，较为基础性的生成算法进一步改进而得到的。本报告尝试分析普通随机数生成器中四种经典的伪随机数生成算法及其衍生改进算法，以及三种经典的密码学安全的随机数生成器。同时，本报告将提供部分随机数生成器算法的java代码实现，并尝试使用NIST所提供的随机数生成器测试套件[13]对所实现的随机数生成器进行测试。

普通随机数生成器算法

平方取中法

原始平方取中算法

平方取中法是由冯·诺依曼于1949年所提出的一种用于生成为随机数序列的生成算法[14]。做为最早一批所提出的伪随机数算法，平方取中法较为简单，能够有效生成看似随机的伪随机数序列，但后续的研究表明其该算法的生成序列统计学随机性较差。平方取中法的算法过程如下：

1. 选取一个m位数x作为种子
2. 计算x^2，并在结果前补0使其成为一个2m位的数字
3. 选取中间的m位数作为新的种子，重复上述迭代过程

该算法主体的Java代码实现如下

```java
    @Override
    public long nextLong() {
        long next = seed * seed / TAIL;
        long res = 0;
        long term = 1;
        for (int i = 0; i < LENGTH; i++) {
            int v = (int) (next % 10);
            res += term * v;
            term *= 10;
            next /= 10;
        }
        seed = res;
        return res;
    }
```

选取种子x=1234，m=4，经测试可以发现，该算法能够生成随机序列，效果如下

```
times:100    val:5227
times:99     val:3215
times:98     val:3362
times:97     val:3030
times:96     val:1809
times:95     val:2724
times:94     val:4201
times:93     val:6484
times:92     val:422
times:91     val:1780
times:90     val:1684
times:89     val:8358
times:88     val:8561
```

同时测试观察到，自第56次生成随机数后，该算法陷入恒0循环，这提示该算法所生成的有效伪随机序列长度可能与初始种子长度有关，同时提示该算法可能在特定种子下陷入循环中，无法生成有效的伪随机序列。由于当前伪随机数较小时，平方取中法很可能取得全0序列从而陷入恒0循环中，这提示该算法所生成的伪随机数序列的随机数分布可能并不均匀。

```
times:46     val:5
times:45     val:0
times:44     val:0
times:43     val:0
times:42     val:0
times:41     val:0
times:40     val:0
times:39     val:0
times:38     val:0
times:37     val:0
```

以种子x=24，m=2为例，该算法所生成种子陷入了24–57循环，提示该伪随机数算法生成效果与初始种子具有较强的关系，同时提示该伪随机数算法周期可能较短。

```
times:100    val:57
times:99     val:24
times:98     val:57
times:97     val:24
times:96     val:57
times:95     val:24
times:94     val:57
times:93     val:24
times:92     val:57
times:91     val:24
times:90     val:57
times:89     val:24
times:88     val:57
```

使用NIST测试工具对该伪随机数算法所生产序列进行测试，选取初始种子位64位比特长度数字，测试序列长度为1000,000*10个比特位（后续实验采用相同的生成序列长度），测试结果如下

序号	测试项目	通过测试数/总测试数	平均P值
1	Frequency	0/10	0
2	BlockFrequency	0/10	0
3	CumulativeSums	0/20	0
4	Runs	0/10	0
5	LongestRun	0/10	0
6	Rank	9/10	0.122325
7	FFT	0/10	0
8	NonOverlappingTemplate	0/1480	0
9	OverlappingTemplate	0/10	0
10	Universal	0/10	0
11	ApproximateEntropy	0/10	0
12	RandomExcursions	ERR	/
13	RandomExcursionsVariant	ERR	/

14	Serial	0/20	0
15	LinearComplexity	9/10	0.739918

可以看到，即便是待测序列中并不存在全0循环，平方取中法所NIST检测后的数据结果仍仅通过了2项测试，这反应出该算法的随机性可能较差，该结论也同样被先前研究所证实。

Weyl序列平方取中算法

Weyl序列平方取中算法是由Bernard Widynski所提出的改进型平方取中算法[15]。该算法借助Weyl序列改进了原始平方取中算法，显著延长了其生成的随机序列周期性与生成分布均匀性，同时避免了算法最终陷入全0循环。该算法的具体过程如下：

1. 初始化64位种子x，Weyl序列w与序列常数s，其中s为Weyl序列中一个二进制形式下0与1数量大致相等的奇数常数，而Weyl序列w被定义为一个周期为2^{64}的循环序列，其表达式如下

$$w_{i+1} = (w_i + \mathrm{s}) \bmod 2^{64}$$

2. 计算当前项随机数x_{i+1}的值，算法如下

$$x_{i+1} = middle(x_i^2 + w_{i+1})$$

其中，middle为平方取中法算法中进行补零并截取中间项的操作函数，重复该即可生成所需的随机序列。

上述过程主要部分的Java代码实现如下

```java
UnsignedLong seed,w,weylConst;//using Guava UnsignedLong.
//initialization
///...
@Override
    public long nextLong() {
        seed = seed.times(seed);
        w = w.plus(weylConst);
        seed = seed.plus(w);
        long xBits = seed.longValue();
        long high = (xBits >>> 32);
        long low = (xBits << 32);
        seed=UnsignedLong.fromLongBits(high | low);
        return high;
    }
```

选取种子x=0,设置Weyl序列w=0，weylConst=0xb5ad4eceda1ce2a9L，所生成的随机序列效果如下，可以看到算法正确地生成了随机序列

```
times:100    val:3048033998
times:99     val:3746490460
times:98     val:411637087
times:97     val:3336355023
times:96     val:285663429
times:95     val:1194354350
times:94     val:927646759
times:93     val:568977855
times:92     val:1922357842
times:91     val:556645914
times:90     val:2621741866
```

在论文中，Widynski详细证明了通过引入Weyl序列显著延长了生成序列的周期长度，同时避免生成序列陷入恒0循环。同时，Windynski证明了生成序列的分布均匀性，同时BigCrush工具的检测结果表明该算法通过了随机性检验。本报告尝试使用NIST伪随机数生成器测试工具对该算法生成序列随机性进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.534146
2	BlockFrequency	10\10	0.739918
3	CumulativeSums	20\20	0.72278
4	Runs	10\10	0.350485
5	LongestRun	10\10	0.066882
6	Rank	10\10	0.534146
7	FFT	10\10	0.739918
8	NonOverlappingTemplate	1467\1480	0.484722
9	OverlappingTemplate	10\10	0.350485
10	Universal	10\10	0.534146
11	ApproximateEntropy	9\10	0.350485
12	RandomExcursions	56\56	/

13	RandomExcursionsVariant	126\126	/
14	Serial	18\20	0.442316
15	LinearComplexity	10\10	0.911413

可以看到，生成序列通过了绝大多数的统计学随机性测试，表明该生成序列具备较好的统计学随机性，提示该生成器的生成质量较高，同时分析其算法过程可以发现，该算法相较于原始平方取中法仅增加了较为简单的Weyl序列计算加和过程，该算法的时间复杂度与原始算法相当，具备较高的生成效率。

线性同余法

线性同余法是由W. E. Thomson于1958年所提出的伪随机数生成算法，该算法历史悠久，算法过程简单，相较于平方取中法可以生成质量更高的随机数生成序列，因此得以被广泛应用于各种普通伪随机数需求场景中。同时由于该算法的经典性，许多伪随机数生成算法往往基于该算法原始形式进行改进与扩展，如Park–Miller generator，Permuted Congruential Generator等算法，本报告尝试对该算法原始形式及其改进形式进行阐述与测试

原始线性同余算法

原始线性同余法的算法过程大致如下：

1. 初始化伪随机数生成器种子X_0，生成器系数有：模数m，乘数a与增量c，其中若c为0，则该算法为Lehmer RNG算法。该LCG算法生成序列的最大周期为m，且多数时候无法达到m。为尽可能增大生成序列的周期长度，提高生成序列的随机性，通常采取如下的系数选择方法[17][18]：
    a. 选取m为尽可能大的数，往往选取素数或2的幂
    b. 若c不等于0，则在满足一定性质的条件下，该生成器输入任意种子均可以生成周期长度等于m的随机数序列，其性质如下
        i. m与c需互质
        ii. a–1可被m的任意质因子整除
        iii. 若m可被4整除，则a–1可被4整除
2. 定义生成序列{X_n}，，其生成序列的递归公式如下

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

上述算法主体部分的Java代码实现如下

```java
    @Override
    public long nextLong() {
        seed = (seed * a + c) & (mod - 1);
        return (int)seed >>> (48 - 32);
    }
```

使用NIST伪随机数生成器测试工具对该算法生成序列随机性进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.017912
2	BlockFrequency	10\10	0.911413
3	CumulativeSums	20\20	0.431122
4	Runs	10\10	0.739918
5	LongestRun	10\10	0.739918
6	Rank	10\10	0.911413
7	FFT	10\10	0.534146
8	NonOverlappingTemplate	1461\1480	0.508226
9	OverlappingTemplate	10\10	0.350485
10	Universal	10\10	0.213309
11	ApproximateEntropy	10\10	0.534146
12	RandomExcursions	48\48	/
13	RandomExcursionsVariant	108\108	/
14	Serial	20\20	0.373728
15	LinearComplexity	10\10	0.739918

出乎意料的是，原始线性同余法算法虽然较为简单，但仍基本通过了全部的统计学随机性测试，表明该算法具备较好的统计学随机性。

Park–Miller算法

Park–Miller算法即为Lehmer RNG的特殊实现，1988年Park and Miller的论文[19]建议将生成器参数设置为$m = 2^{31} - 1$，$a=7^5$，从而提供一个较好的生成序列。

依此调节LCG算法的参数，并使用NIST伪随机数生成器对该算法生成序列随机性进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.911413
2	BlockFrequency	10\10	0.534146
3	CumulativeSums	20\20	0.442316
4	Runs	10\10	0.350485
5	LongestRun	10\10	0.350485
6	Rank	10\10	0.534146
7	FFT	10\10	0.534146
8	NonOverlappingTemplate	1458\1480	0.507882
9	OverlappingTemplate	10\10	0.350485
10	Universal	10\10	0.122325
11	ApproximateEntropy	10\10	0.122325
12	RandomExcursions	48\48	/
13	RandomExcursionsVariant	108\108	/
14	Serial	20\20	0.739918
15	LinearComplexity	10\10	0.739918

可以看到，在正确地设置LCG参数的条件下，LCG类算法均具备较好的统计学随机性。

PCG算法

PCG算法是一种基于LCG算法的改进算法，由Dr. M.E. O'Neill于2014年提出[20]。该算法通过添加一个输出置换函数对输出结果的个各比特位进行打乱混合，从而避免了LCG算法低位周期较短的问题，其中

一种输出转置算法下的Java代码实现如下

```java
    private int rotate(int x, int r) {
        return x >>> r | x << (-r & 31);
    }
    @Override
    public long nextLong() {
        UnsignedLong x = state;
        int count = (int) (x.longValue() >>> 59);
        state = x.times(multiplier).plus(increment);
        long x2 = x.longValue();
        x2 ^= x2 >>> 18;
        x = UnsignedLong.fromLongBits(x2);
        int res = rotate((int) (x.longValue() >>> 27), count);
        return Integer.toUnsignedLong(res);
    }
```

使用NIST随机数生成器检测工具对该算法生成序列随机性进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.739918
2	BlockFrequency	10\10	0.911413
3	CumulativeSums	20\20	0.167817
4	Runs	10\10	0.122325
5	LongestRun	10\10	0.350485
6	Rank	10\10	0.534146
7	FFT	10\10	0.739918
8	NonOverlappingTemplate	1468\1480	0.540027
9	OverlappingTemplate	10\10	0.911413
10	Universal	10\10	0.739918
11	ApproximateEntropy	10\10	0.534146
12	RandomExcursions	48\48	/

13	RandomExcursionsVariant	108\108	/
14	Serial	20\20	0.534146
15	LinearComplexity	10\10	0.911413

可以看到，该算法基本通过了统计学随机性检验，具备较好的生成随机性。

延迟斐波那契法

延迟斐波那契算法（LFG）是一类基于斐波那契序列扩展形式的生成算法，其最早形式由G. J. Mitchell与D. P. Moore于1958年提出[17]。该类算法的通用递归表达式可表示如下

$$X_n \equiv X_{n-j} \star X_{n-k} \,(mod\ m), 0 < j < k$$

其中，星号表示某种二元关系运算符，如加和运算，乘积运算等。本报告尝试对原始的延迟斐波那契算法进行阐述

原始延迟斐波那契算法

G. J. Mitchell与D. P. Moore提出的原始算法的递归序列定义如下

$$X_n \equiv X_{n-24}+X_{n-55} \,(mod\ m), n \geq 55, m\ is\ even$$

原始算法的生成种子为序列{X_0...X_{54}}，m为一个充分大的偶数模数，通常选取m值为2的幂次方

通用LFG算法的Java实现如下，其中，LEN为生成序列数组长度，使用l，r表示当前生成项所依赖的前项

```java
//...initalize...
//        LEN = k;
//        LAG_1 = LEN - j;
//        LAG_2 = 0;
//        pointer = LEN - 1;
//...
@Override
    public long nextLong() {
        pointer = (pointer + 1) % LEN;
        int l = (LAG_1 + pointer) % LEN;
        int r = (LAG_2 + pointer) % LEN;
        return seeds[pointer] =
                operator.operate(seeds[l], seeds[r]) % mod;
    }
```

由于现有的LFG应用常借助LCG生成初始种子序列，本报告对此进行了实现，选取seed=10, $m=2^{32}$，生成结果如下，可以看到生成器正确生成了伪随机序列

```
times:100    val:1490709499
times:99     val:2824997865
times:98     val:16892579
times:97     val:1507602078
times:96     val:37632647
times:95     val:54525226
times:94     val:1562127304
times:93     val:1599759951
times:92     val:1654285177
times:91     val:3216412481
times:90     val:521205136
```

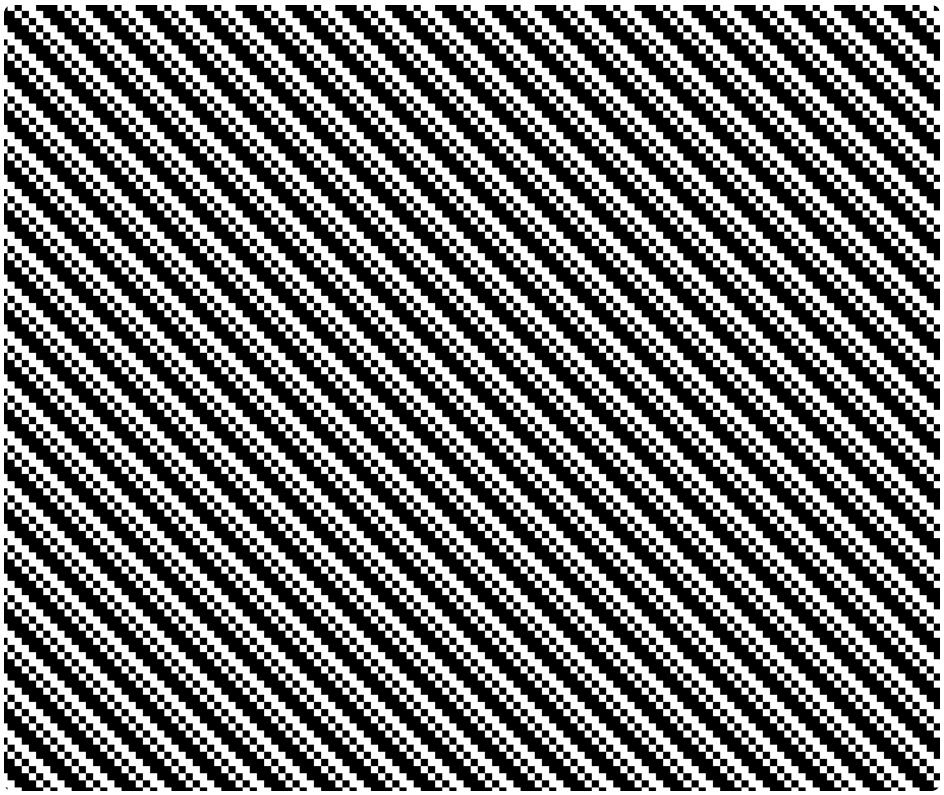
使用NIST随机性检测工具对所生成序列进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	0\10	0
2	BlockFrequency	0\10	0
3	CumulativeSums	0\20	0
4	Runs	0\10	0
5	LongestRun	10\10	0.534146

6	Rank	10\10	0.122325
7	FFT	10\10	0.350485
8	NonOverlappingTemplate	783\1480	0.143858
9	OverlappingTemplate	0\10	0
10	Universal	10\10	0.534146
11	ApproximateEntropy	0\10	0
12	RandomExcursions	/	/
13	RandomExcursionsVariant	/	/
14	Serial	10\20	0.0000995
15	LinearComplexity	10\10	0.534146

可以看到，该伪随机数生成器仅通过了部分随机性检验测试，提示该伪随机数生成器可能质量不佳。

通过将每次生成数字奇偶性转化成图像的黑/白像素，我们可以生成随机序列的奇偶性分布图，下图的LFG奇偶性分布图直观的展示了LFG生成序列内部具备较强的奇偶性关系，这提示该生成器的生成序列可能具备较强的内部规律，验证了统计学测试的结果。



改进型斐波那契法

基础的对原有LFG算法进行改进的操作往往围绕其通用递归式进行改进，如选用乘法，异或作为二元关系运算符，以Marsaglia所提出的改进算法为例，该算法使用乘法运算符进行迭代运算，其表达式如下[21]

$$X_n \equiv X_{n-24} \cdot X_{n-55} \, (mod\ m), n \geq 55, m = 4k, k \in N^+$$

其中，模数m为4的倍数，而种子序列{X_0...X_{54}}为奇数，且不全于模4下同余1。依此调节LFG算法的参数，并使用NIST伪随机数生成器对该算法生成序列随机性进行检验，测试结果如下：

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.122325
2	BlockFrequency	0\10	0
3	CumulativeSums	20\20	0.373728
4	Runs	10\10	0.350485
5	LongestRun	0\10	0
6	Rank	2\10	0
7	FFT	0\10	0
8	NonOverlappingTemplate	0\1480	0
9	OverlappingTemplate	0\10	0
10	Universal	0\10	0
11	ApproximateEntropy	0\10	0
12	RandomExcursions	3\12	/
13	RandomExcursionsVariant	72\72	/
14	Serial	0\20	0
15	LinearComplexity	10\10	0.534146

相较于原始算法，改进型算法相较于原始算法通过了不同的测试，在频率测试中表现较好，但同时存在一些未能通过的统计学测试，提示该算法可能不能很好的生成质量较高的伪随机序列。

线性反馈位移寄存器算法

线性反馈位移寄存器算法（LFSR）是一种基于移位寄存器与线性反馈函数的伪随机数生成算法，其主要创始人为Golomb, Solomon W. [22]，由于其生成序列较线性同余法具有更高的抗预测性，因而该算法及其改进形式受到了广泛应用。其改进与衍生形式包括了如Mersenne Twister算法等。本报告尝试对该算法原始形式及其衍生算法进行阐述与测试。

原始线性反馈位移寄存器算法

原始线性反馈寄存器的通用表达形式如下

$$X_{i+n} = F(X_i, X_{i+1}, \dots, X_{i+n-1})$$

其中，F为线性多元函数，其通用表达式为

$$X_{i+n} = \sum_{j=1}^{n} c_j X_{i+n-j}$$

本报告尝试基于此实现32位的线性反馈位移寄存器算法，选取线性反馈项为[32,22,2,1,0]，选取项系数为1，其代码实现如下。

```java
    @Override
    public long nextLong() {
        long bit = ((seed >>> 0) ^ (seed >>> 1) ^ (seed >>> 2) ^ (seed >>> 22) ^ (seed >>> 32)) & 1L;
        seed = (seed >>> 1) | (bit << 31);
        return seed;
    }
```

选取seed=10, 生成结果如下，可以看到生成器正确生成了伪随机数序列

```
times:100    val:2147483653
times:99     val:1073741826
times:98     val:2684354561
times:97     val:3489660928
times:96     val:1744830464
times:95     val:872415232
times:94     val:436207616
times:93     val:218103808
times:92     val:109051904
times:91     val:54525952
times:90     val:2174746624
```

使用NIST随机性检测工具对所生成序列进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.911413
2	BlockFrequency	10\10	0.350485
3	CumulativeSums	20\20	0.72278
4	Runs	10\10	0.911413
5	LongestRun	10\10	0.350485
6	Rank	0\10	0
7	FFT	10\10	0.350485
8	NonOverlappingTemplate	1462\1480	0.523586
9	OverlappingTemplate	10\10	0.213309
10	Universal	10\10	0.739918
11	ApproximateEntropy	9\10	0.739918
12	RandomExcursions	48\48	/
13	RandomExcursionsVariant	108\108	/
14	Serial	20\20	0.630949
15	LinearComplexity	0\10	0

可以看到，线性反馈位移寄存器算法生成了较好的随机数序列，通过了绝大多数的统计学测试，但未通过Rank测试于LinearComplexity测试。

Rank测试即为矩阵秩测试，该测试主要用于测试随机数序列内部的线性相关性，而LinearComplexity即为线性复杂度测试，该测试反映了各子序列的LFSR长度。由于该算法所使用的线性反馈寄存器位数较短，该算法并未通过上述两项测试，契合了前文中对该类测试的理论分析，表明线性反馈寄存器可能无法生成充分复杂的随机数算法，其序列内部具备一定的相关性。

Mersenne Twister算法

Mersenne Twister算法是由 Makoto Matsumoto与Takuji Nishimura所开发的算法[23]，该算法基于矩阵线性递归实现，可以生成相当长的序列生成周期（如MT19937算法的生成序列周期为$2^{19937}-1$），具备较强的随机质量，其算法大致过程如下：

1. 接收初始种子X0，随后通过递推式求出梅森旋转链作为初始参数，其递推公式如下，其中f与w为设定系数

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

2. 遍历旋转链，对该旋转链执行梅森旋转算法，该算法的递推公式如下，其中u与l为设定系数，A为旋转矩阵

$$x_{k+\mathrm{n}} = x_{k+m} \oplus \left(\left(x_k^u \mid x_{k+1}^l\right) \times A\right)$$

3. 对梅森旋转算法结果进行处理，求解得生成的随机数项，该过程的公式如下

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

本项目尝试使用java代码对该算法进行实现，其代码主体部分如下

```java
    private void initialize(){
        for (int i = 1; i < N; i++) {
            mt[i] = (F * (mt[i - 1] ^ (mt[i - 1] >> 30)) + i) & D;
        }
    }
    private void twist() {
        for (int i = 0; i < N; i++) {
            long x = (mt[i] & MASK_UPPER) + (mt[(i + 1) % N] & MASK_LOWER)
;
            long xA = (x >> 1) & D;
            if ((x & 1) != 0) {
                xA ^= A;
            }
            mt[i] = mt[(i + M) % N] ^ xA;
        }
        index = 0;
    }
    @Override
    public long nextLong() {
        if (index >= N) {
            twist();
        }
        long y = mt[index];
        y ^= (y >> U);
        y ^= (y << S) & B;
        y ^= (y << T) & C;
        y ^= (y >> L);
        index++;
        return y & D;
    }
```

使用NIST随机性检测工具对所生成序列进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.066882
2	BlockFrequency	10\10	0.350485
3	CumulativeSums	20\20	0.739918
4	Runs	10\10	0.739918
5	LongestRun	10\10	0.739918

6	Rank	10\10	0.534146
7	FFT	10\10	0.213309
8	NonOverlappingTemplate	1463\1480	0.469501
9	OverlappingTemplate	10\10	0.350485
10	Universal	10\10	0.213309
11	ApproximateEntropy	10\10	0.911413
12	RandomExcursions	40\40	/
13	RandomExcursionsVariant	88\90	/
14	Serial	20\20	0.19283
15	LinearComplexity	10\10	0.350485

可以看到，MT算法所生成的随机数序列基本通过了统计学检验，有效改善了LFSR算法生成序列复杂度欠佳的问题，其生成序列具备较高的随机性质量。

密码学安全的随机数生成器算法

由前文分析可知，密码学安全的随机数生成器算法往往基于密码学理论所设计并证明，主要可分为基于密码学原语所设计的伪随机数生成器，基于数学难题所涉及的伪随机数生成器两类。同时，在实际应用中，如操作系统等平台往往借助外界物理设备参数的引入以实现近似于真随机数的随机数生成算法。本报告尝试分析两种CSPRNG算法，并使用NIST所提供的随机数生成器测试套件对所实现的随机数生成器进行测试。

Blum Blum Shub算法

Blum Blum Shub算法是 Lenore Blum, Manuel Blum与Michael Shub于1986年所提出的算法，该算法基于大因数分解这一数学难题所设计，在现有理论与技术下具备较高的密码学安全水平[24]，算法的大致过程如下：

1. 选取两个大素数p与q为种子，计算n=p*q 素数p与q应满足如下性质

$$p \equiv q \equiv 3\ (mod 4)$$

2. 接收随机数种子x_0，使用下列公式进行随机数生成迭代，初始种子必须满足与n互质，且大于1

$$x_i = x_{i-1}^2 mod\ n$$

3. 选取每项随机数的最低有效位，该最低有效位序列构成了生成器的生成序列

该算法主体部分的java代码实现如下，其中，大素数的选取使用了Java所提供的Miller–Rabin素性测试方法

```java
//BigInteger p,q,n,seed...
    public BBS() {
        super(-1);//unused
        p = bigPrime();
        q = bigPrime();
        n = p.multiply(q);
        seed = bigPrime();
    }
    public BigInteger bigPrime() {
        Random rnd = new Random();
        BigInteger res;
        do {
            res = BigInteger.probablePrime(bitLength, rnd);
        } while (res.mod(FOUR).equals(THREE) && res.isProbablePrime(certainty));
        return res;
    }

    @Override
    public boolean nextBit() {
        seed = seed.pow(2).mod(n);
        return seed.testBit(0);
    }
```

使用NIST随机性检测工具对所生成序列进行检验，测试结果如下

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	9\10	0.350485
2	BlockFrequency	10\10	0.534146
3	CumulativeSums	19\20	0.516869
4	Runs	9\10	0.534146
5	LongestRun	10\10	0.739918

6	Rank	10\10	0.122325
7	FFT	9\10	0.534146
8	NonOverlappingTemplate	1471\1480	0.501895
9	OverlappingTemplate	10\10	0.739918
10	Universal	10\10	0.213309
11	ApproximateEntropy	8\10	0.534146
12	RandomExcursions	64\64	/
13	RandomExcursionsVariant	144\144	/
14	Serial	20\20	0.825665
15	LinearComplexity	10\10	0.739918

可以看到，BBS算法所生成的随机数序列基本通过了统计学检验，其生成序列具备较高的统计学随机性。

Windows密码学安全伪随机数生成器

Windoews内部的RtlGenRandom函数提供了基于外部物理熵源的CSPRNG，依据所发布的算法摘要[25]，该算法采用SHA-1作为G函数，同时使用当前系统内部的变化参数作为熵源，如当前进程ID，高精度计数器，用户环境参数等，从而提高了其对抗攻击能力。查阅Java官方文档可知[26]，Java官方文档所提供的SecureRandom类在Windows操作系统环境下将使用RtlGenRandom函数生成随机序列，本报告基于此在windows环境下使用Java SecureRandom生成随机序列，从而实现对Windows所提供的随机变量生成器的测试，检测结果如下：

序号	测试项目	通过测试数\总测试数	平均P值
1	Frequency	10\10	0.213309
2	BlockFrequency	10\10	0.911413
3	CumulativeSums	20\20	0.562361
4	Runs	9\10	0.739918
5	LongestRun	10\10	0.534146

6	Rank	10\10	0.534146
7	FFT	10\10	0.911413
8	NonOverlappingTemplate	1467\1480	0.544747
9	OverlappingTemplate	10\10	0.534146
10	Universal	10\10	0.739918
11	ApproximateEntropy	10\10	0.122325
12	RandomExcursions	55\56	/
13	RandomExcursionsVariant	126\126	/
14	Serial	20\20	0.637032
15	LinearComplexity	10\10	0.534146

可以看到，Windows伪随机数发生器所创建的生成序列基本通过了统计学测试，具备较高的统计学随机性。

总结

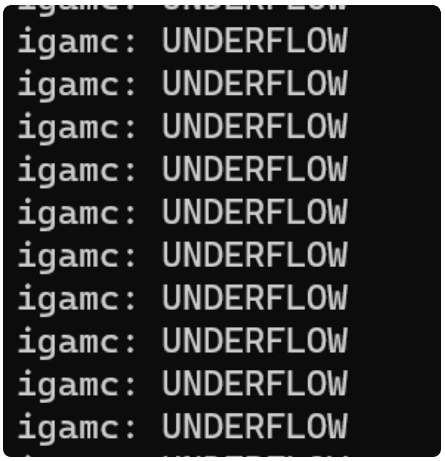
通过对经典的伪随机数生成器算法进行实验与测试可以发现，不同的PRNG生成算法所生成的随机序列具备较为明显的统计学差异，对于经典的生成算法，仅有LCG算法所生成的随机序列完整地通过了统计学测试，具备较好的统计学随机性，而原始LFSR算法除在线性复杂度上效果欠佳外，同样通过了大多数的统计学测试。对于原始平方取中法与LFG算法，由于其未能通过多数的统计学测试，提示该算法的生成质量可能较差。同时，对于由此类经典算法所衍生出的如MT算法、PCG算法、Weyl算法均完整地通过了统计学测试，同时对原有算法做出了进一步的改进，其生成器的质量较高。对于密码学安全级别的随机数生成器，所测试的BBS算法与WindowsCSPRNG接口均通过了随机性测试，算法同时具备较高的随机性与对抗攻击性。

综合上述实验结果，本报告尝试对PRNG算法的表现进行汇总评价，结果如下

PRNG算法	随机性检验结果	生成序列效率	密码学安全	算法实现难度
原始平方取中法	未通过	快	否	较低
Weyl序列平方取中法算法	通过	快	否	较低

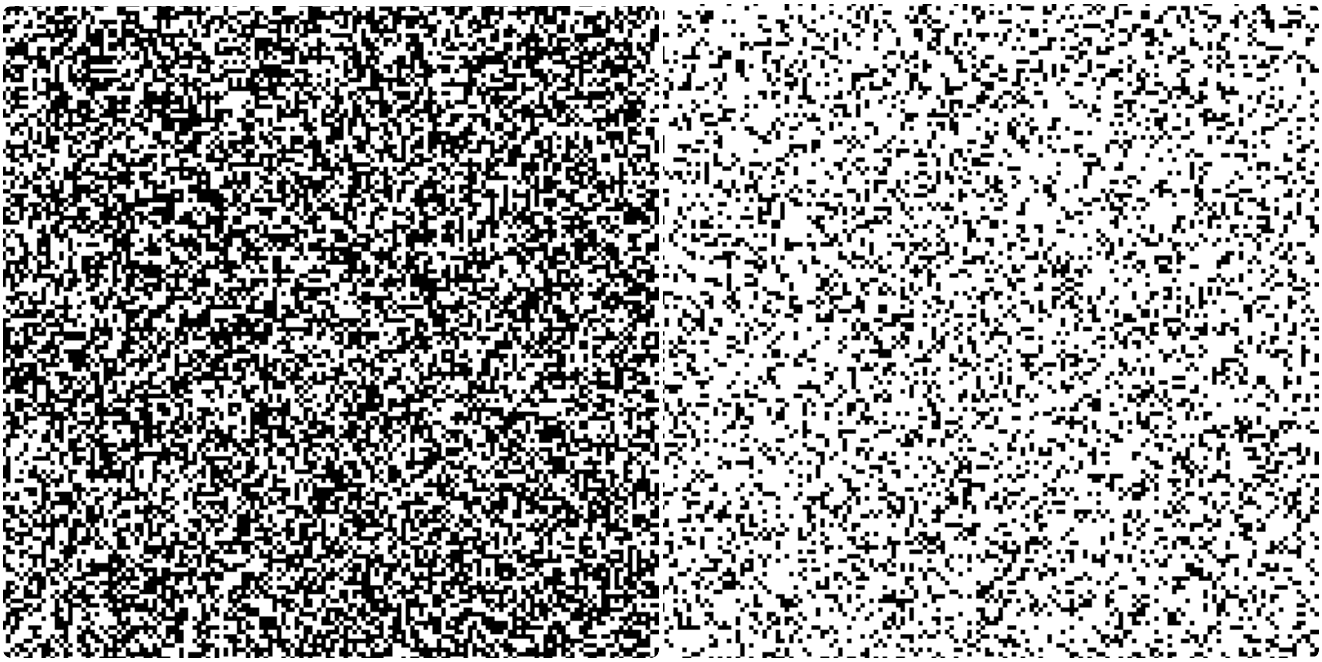
原始线性同余算法	通过	快	否	较低
Park–Miller算法	通过	快	否	较低
PCG算法	通过	快	否	中
原始延迟斐波那契算法	未通过	快	否	较低
改进型斐波那契算法	未通过	快	否	较低
原始线性反馈位移寄存器	基本通过	快	否	中
Mersenne Twister算法	通过	快	否	较高
Blum Blum Shub算法	通过	慢	是	中
RtlGenRandom (Windows CSPRNG)	通过	慢	是	/

同时在测试过程中发现，本报告发现了，对于随机性较差的生成序列，NIST检测工具会提示如下的报错



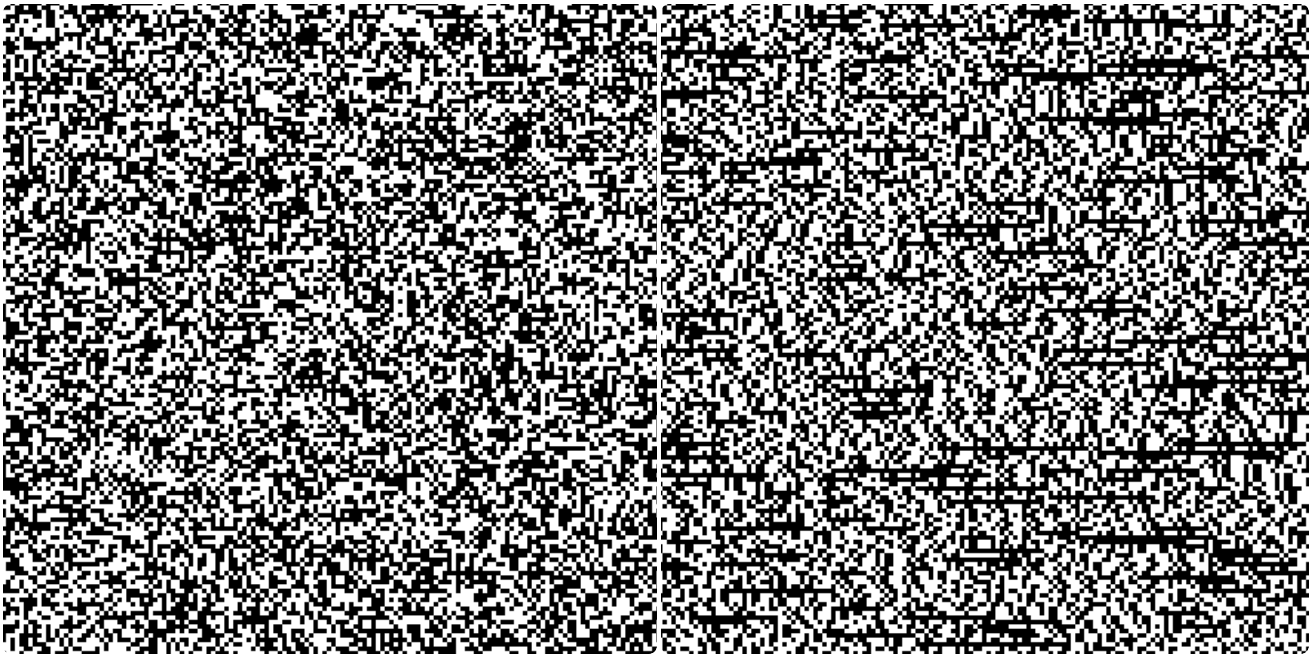
该报错提示随机序列的随机性较差，同时该条件下NIST检测工具将无法给出部分测试的测试结果。实际上，对于随机数生成器的生成序列，我们可以通过将该比特序列转化成图像的黑/白像素，从而可以在测试前直观的观察到生成序列的异常，从而重新评估生成器的实现质量。以上述实验数据为例，我们对比各算法生成序列的比特位图

1. PCG算法生成位图（左）vs 原始平方取中法生成位图（右）（相同缩放比例）



可以十分直观的看到，原始平方取中法生成的随机序列相当稀疏，这提示原始平方取中法的生成序列可能分布并不均匀，NIST检验测试结果最终验证了这一猜想

2. MT算法生成位图（左） vs 改进型斐波那契算法（右）（相同缩放比例）



可以直观地看到，尽管生成位图密度相近，但改进型斐波那契序列中出现了较多的横纹，这提示生成序列可能会以偏离完全随机情况概率下的频率生成恒1序列，提示该伪随机数算法可能存在缺陷，可能无法通过块频率测试，重叠模板测试等统计学测试，NIST的检测结果最终证实了这一猜想。

通过提前生成比特位图观察，可以在不进行统计学检测的条件下较为直观的发现伪随机数生成器生成序列的异常，从而判定伪随机数生成器随机质量可能较差，从而可以避免进行不必要的NIST测试。

主流语言/操作系统伪随机数生成器分析

由于随机数所具备的随机性质被广泛应用于计算机领域的开发实践中，主流的编程语言与操作系统提供了内置的随机数生成器用于生成所需的随机数序列。本报告尝试调研了并收集了十种主流语言所提供内置伪随机数生成器，其调研结果汇总如下

语言	普通PRNG接口（类/库）	实现算法	CSPRNG接口（类/库）	实现算法
C	rand()	LCG	/	/
Java	Random	LCG	java.security.SecureRandom	依赖于操作系统
C++	random	MT, LFG	random_device	依赖于操作系统
Python	random	MT	secrets.rand(依赖于操作系统)	依赖于操作系统
Swift	arc4random()	LCG	SecRandomCopyBytes	依赖于Mac操作系统
C#	Random	LCG	RandomNumberGenerator	依赖于操作系统
Go	math/rand	LCG	crypto/rand	依赖于操作系统
PHP	rand,mt_rand	LCG,MT	random_int()	依赖于操作系统
Rust	rand::rngs::smallRng	Xoshiro256++	rand::Rng::OsRng/StdRng	依赖于操作系统/ChaCha12
JS	Math.random	LCG	crypto.getRandomValue	依赖于操作系统

可以看到，多数主流语言提供了普通PRNG/CSPRNG两套不同的伪随机数生成器API，其中，普通PRNG主要使用LCG算法进行实现，而部分语言则使用LFSR类算法如MT算法进行实现，由前文实验可知，这两种算法所生成的伪随机数序列随机性较好，具备较高的生成质量。同时，由于LCG算法提出较早，其实现较为简单，因而受到了广泛的应用。同时，各语言所提供的CSPRNG主要调用了操作系统内置的随机数生成器，由前文分析可知，操作系统通过引入外部物理设备可以实现近似于真随机数生成器的生成效果，因而具有较强的对抗攻击能力，具备较好的密码学安全性。

同时，本报告调研了主流操作系统所提供的随机数生成器接口，调研结果汇总如下

语言	随机数生成器接口	实现算法

Windows	RtlGenRandom	SHA-1+熵源
Linux	linux/random	完全基于熵池
Mac OS	arc4random_uniform	Fortuna（依赖熵源）

可以看到，主流的操作系统的随机数生成接口均引入了外界物理设备作为熵源，由于熵源的不可预测性，这一实现能够有效提升生成算法的抗攻击性，从而提高算法的生成质量。

值得一提的是，对于早期的Windows2000操作系统，研究表明其内置的未公开算法的随机数生成器的具备较大缺陷[26]，可以通过单次窃取状态位攻击随机数生成器。进一步的研究表明，尽管Windows2000操作系统所使用的操作系统的随机数生成算法引入了熵源，但出于效率原因很少借助该熵源进行种子刷新。这提示通过引入外置熵源提高CSPRNG算法的安全性的操作效果依赖于熵源生成函数的定义与伪随机数生成器的内部实现，不当的算法实现很可能无法保证伪随机数生成器的安全质量。

伪随机生成器攻击尝试

基于数学理论的伪随机数攻击尝试

LCG攻击尝试

基于前文所描述的LCG理论我们可以发现，若已知LCG生成器的部分输出随机数，则有可能通过数论方式对LCG算法的后续生成序列进行预测。本报告对此进行探究，并依据理论分析实现对Java Random库的生成攻击，依据所已知的参数信息，可分为以下几种情况进行预测。

LCG参数乘数、增量、与模数已知

若LCG参数乘数a，增量c与模数m均已知，则只需要知道当前伪随机数生成值X_0，即可直接带入带入线性同余方程便可预测后续的伪随机数序列

$$X_1 = (aX_0 + c)\ mod\ m$$

已知全部参数的情况实际上十分常见，绝大多数的编程语言与开源代码中均公开了其伪随机数生成器的实现方式与具体算法参数。同时，由于LCG算法需选取恰当的参数才能实现对于多数种子均生成较为良好的随机数序列，因此，LCG算法的推荐参数组往往较少且被公开，可以依此对未知参数的LCG生成器带入参数组进行猜测尝试

LCG参数增量未知

若给定LCG算法参数增量未知，则通过已知连续的两项随机数值即可预测后续的生成序列，其算法过程如下

1. 由于c小于m，故由原始伪随机数序列算法移项可的c的求解公式，公式如下

$$c = (aX_i - X_{i+1})\ mod\ m$$

2. 由上步骤获得了LCG算法的全部参数，同上文分析方法即可预测后续的生成序列

LCG参数增量与乘数未知

若给定LCG算法参数增量与乘数均未知，则通过已知连续的三项随机数值即可预测后续的生成序列，数学分析如下

对于三项连续随机数值X_i，X_{i+1}，X_{i+2}有如下关系

$$X_{i+1} = (aX_i + c)\ mod\ m$$
$$X_{i+2} = (aX_{i+1} + c)\ mod\ m$$

联立得

$$X_{i+2} - X_{i+1} = a(X_{i+1} - X_i)\ mod\ m$$

故由于a为小于m的系数，移项可得a的求解公式

$$a = \frac{X_{i+2} - X_{i+1}}{X_{i+1} - X_i} mod\ m$$

后该问题转化为LCG参数增量未知条件下的求解问题，后续过程参见上文

针对Java Random类的攻击尝试

Java Random类属于上述分析中所有参数均已知的情形，通过查阅标准库源码可以得到相应系数与生成器具体实现方式。值得注意的是，在标准源码实现中，Java的一系列获取随机值的接口是通过截取当前48比特位随机数项的部分位数而获得的，如nextInt方法通过舍弃低16位而实现，因此，通过nextInt方法所获取到的值并非完整的随机数生成项。本报告尝试对此进行数学分析，并使用Java实现攻击算法

设所截断位数位低 t 位，随机数种子序列位$\{X_n\}$，每次调用nextInt所获得的随机数生成序列为$\{S_n\}$，输出时被截断舍去的序列位$\{U_n\}$，则三序列各项具有如下关系

$$\begin{cases} D = 2^t \\ X_i = S_i D + U_i\ ,0 \le U_i < D \end{cases}$$

对于生成序列第i项与第i+1项，二者具有如下关系

$$X_{i+1} = (aX_i + c)\ mod\ m$$

带入有 $$(a(S_iD + U_i) + c)\ mod\ m = S_{i+1}D + U_{i+1}$$

整理得 $$((aS_i - S_{i+1})D + c)\ mod\ m = U_{i+1} - aU_i$$

上式中，等式左侧可依据参数与所提供的连续两项随机数生成序列求出具体数值，则只需求解符合该等式的未知数元组（U_i,U_{i+1}），该元组解需满足上述中对U的取值限制关系

基于该公式可以设计对于nextInt方法的攻击算法，将上述公式整理得如下表达式

$$((aS_i - S_{i+1})D + c + aU_i)\ mod\ m = U_{i+1}$$

遍历所有U_i的可能取值直至U_{i+1}符合取值限制，此时，我们找到了未知数元组（U_i,U_{i+1}）的解，从而成功复原了两项随机数种子项的完整数值。由于对于nextInt方法，其所需遍历的可能取值次数不超过2^{16}，因此该算法能够较快求得生成器内部的种子数值。

上述算法的主体部分java代码思想大致如下

```java
    private JavaRandomAttacker(long s0, long s1, int bitWidth) {
        D = 1L << LEN - bitWidth;
        long dif = (((a * s0 - s1) % m) * (D % m) + c) % m;
        long u1 = guessPair(dif, D);
        seed = s1 * D + u1;
    }

    private static long guessPair(long dif, long max) {
        for (int i = 0; i < max; i++) {
            long res = (dif + a * i) % m;
            if (res < max) return res;
        }
        throw new RuntimeException("err!");
    }
```

对上述攻击算法效果进行实验，实验结果如下，可以看到，上述攻击器正确预测了随机数生成器的后续序列

```
assume: 1107254586   real:1107254586
assume: 1773446580   real:1773446580
assume: 254270492    real:254270492
assume: -1408064384  real:-1408064384
assume: 1048475594   real:1048475594
assume: 1581279777   real:1581279777
assume: -778209333   real:-778209333
assume: 1532292428   real:1532292428
assume: 1591762646   real:1591762646
assume: -1492345098  real:-1492345098
```

对上述攻击算法进行100000次检验，检验其是否能够正确预测后续所生成序列前100项，可以看到上述算法成功通过了完整的攻击实验

```java
public void fullRandTest() {
    int TEST_TIMES = 100000;
    int PREDICT_LEN = 100;
    for (int i = 0; i < TEST_TIMES; i++) {
        Random random = new Random();
        int s0 = random.nextInt(), s1 = random.nextInt();
        JavaRandomAttacker attacker = JavaRandomAttacker.rangeInt(s0, s1);
        for (int j = 0; j < PREDICT_LEN; j++) {
            int a = attacker.nextInt();
            int b = random.nextInt();
            assert a==b;
        }
    }
    System.out.println("testPass!");
}
```

```
Tests passed: 1 of 1 test – 2 sec 700 ms
2 sec 700 ms   "C:\Program Files\Java\jdk-17.0.4.1\bin\java.exe" ...
2 sec 700 ms   testPass!
```

可以看到，仅仅通过获得两项连续的随机数种子的片段（如所获取的nextInt值），我们便可推断后续的全部生成序列，更为糟糕的是，在Java Random类库的API实现中，nextLong与nextDouble的实现正是由两个连续的随机数种子片段所拼接而成，这意味着只需已知两个接口的任意一个输出便可预测后续全部的生成序列，这提示Java Random类所产生的随机数不应用于任何已知随机值可能破坏程序安全性与有效性的场景。

伪随机数生成器的应用与展望

由于各领域对随机数生成的迫切需要，自伪随机数生成器诞生以来，其便被广泛应用于各种领域。报告综合相关资料尝试对此进行简要地阐述，并展望伪随机数生成器的未来发展

伪随机数生成器的密码学应用

伪随机数生成器最为重要的应用之一便是密码学场景。作为密码学领域的基础性工具，大多数密码原语的安全性都依赖于高质量的不可预测的随机数，伪随机数生成器在该领域具备广泛的应用，其中主要包括[27]：

1. 加密算法：伪随机数生成器被应用于加密算法的实现，如El Gamal等加密算法使用随机数提高的算法的安全性，部分加密算法则依赖于密码学安全的伪随机数生成器以保障算法的可靠性与安全性。同时，伪随机数生成器也被应用于密钥派生与数字签名之中。
2. 随机填充：伪随机数生成器同样被应用于对序列的随机填充常见，如密码加盐或信息混淆中，往往采用需要伪随机数生成器生成随机序列进行填充操作。

伪随机数生成器的模拟应用

伪随机数生成器被广泛应用于计算机模拟过程中，主要包括：

1. 蒙特卡洛类算法：蒙特卡洛类算法依赖于随机数而实现，需要提供能够生成具有较好统计学随机性质随机序列的生成器，这类算法被广泛应用于工程开发与实践中
2. 仿真类算法：在仿真领域，如交通流量模拟、人口统计模拟或生态系统模拟，PRNGs用于生成符合特定分布的数据，以模拟现实世界中的随机事件。
3. 抽样统计操作：抽样统计操作通过从一个大数据集中抽取代表性样本为了进行统计分析，随机性较好的PRNG可以较为有效的选取样本点，从而确保样本具备统计学需求的随心。

伪随机数生成器的程序开发应用

随机数生成器被广泛应用于实际领域的软件开发中，如游戏领域常使用的随机洗牌，验证码生成，抽奖等功能的实现中。与认为CSPRNG仅适用于密码学领域的误区不同的是，程序开发领域往往也需要使用具备对抗攻击性的CSPRNG，如上述所提到的场景中，一旦业务过程暴露了一定的生成器参数（如往往不可避免地提供给用户的随机结果），该业务的随机序列就有可能被预测，从而带来严重的安全隐患。同时对于一些实现不佳的项目，服务器内的随机数生成器可能仅仅在如服务器重启等特定时刻才进行初始种子重置，因而生成周期较短的，对抗攻击性较弱的伪随机数生成器可能会更容易受到攻击，也更容易造成广泛且长期的负面风险，因此，程序开发领域也需要对所使用的随机场景进行评估，综合效率考虑是否使用CSPRNG级别的密码生成器，避免伪随机数生成器受到攻击造成的损失。

展望随机数生成器的发展

在计算机科学领域高速发展的背景下，随机数算法的发展已日渐成熟，逐渐研究并提出了一系列具备可靠随机性与安全性的随机数生成算法，在未来，随机数生成器将被进一步广泛使用，对现有的随机数生成器的改进与创新可能围绕提供更加高效的PRNG（尤其是CSPRNG）算法，进一步引入物理过程（如量子过程）提高其随机性与安全性，同时，开发者可能将进一步采取措施，避免不佳实践造成PRNG生成可预测的随机序列，以及采用对抗随机数后门攻击的密码算法[28]以避免后门攻击。

参考资料

[1]Salil Vadhan, Pseudorandomness
https://people.seas.harvard.edu/~salil/pseudorandomness/introduction.pdf

[2]Zhouhui Lian,Intelligent Optimization Methods

https://www.icst.pku.edu.cn/zlian/docs/20181023154331943266.pdf

[3] Schindler, Werner. Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators (PDF). Anwendungshinweise und Interpretationen (AIS). Bundesamt für Sicherheit in der Informationstechnik: 5—11. 2 December 1999 [19 August 2013].

[4] WolfSSL 2021.7, True Random vs. Pseudorandom Number Generation

https://www.wolfssl.com/true-random-vs-pseudorandom-number-generation/

[5] 李贤平（复旦大学）．概率论基础（第三版）．北京．高等教育出版社．2010

[6] 国家密码管理局, GM/T 0005-2021 随机性检测规范

[7] Andrew Rukhin (NIST), Juan Soto (NIST), James Nechvatal (NIST), Miles Smid (NIST), Elaine Barker (NIST), Stefan Leigh (NIST), Mark Levenson (NIST), Mark Vangel (NIST), David Banks (NIST), N. Heckert (NIST), James Dray (NIST), San Vo (NIST), Lawrence Bassham (NIST), A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, National Institute of Standards and Technology(NIST)

[8] Binomial Test, UTEXAS

https://sites.utexas.edu/sos/guided/inferential/categorical/univariate/binomial/

[9] Song-Ju Kim, Ken Umeno, Corrections of the NIST Statistical Test Suite for Randomness, 2004

[10] Pareschi, F., Rovatti, R., & Setti, G, On Statistical Tests for Randomness included in the NIST SP800-22 test suite and based on the Binomial Distribution

[11] F. Pareschi, R. Rovatti, and G. Setti, Second-level NIST randomness tests for improving test reliability.

[12] J. Massey, Shift-register synthesis and BCH decoding, IEEE Transactions on Information Theory

[13] NIST SP 800-22: Download Documentation and Software

[14] John von Neumann, Various Techniques Used in Connection With Random Digits

[15] Bernard Widynski, Middle-Square Weyl Sequence RNG

[16] Thhomson, W.E. A Modified Congruence Method of Generating Pseudo-random Number

[17] Donald Knuth, The Art of Computer Programming

[18] Marsaglia, George, Random Numbers Fall Mainly in the Planes

[19] Park, Stephen K. Miller, Keith W. , Random Number Generators: Good Ones Are Hard To Find

[20] O'Neill, Melissa E., PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.

[21] George Marsaglia, Comp.Sci. and Statistics: Symposium on the Interface

[22] Golomb, Solomon W., Shift register sequences. Laguna Hills, Calif.: Aegean Park Press.

[23] Matsumoto, M.; Nishimura, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator

[24]Blum, L.; Blum, M.; Shub, M. (1986). A Simple Unpredictable Pseudo-Random Number Generator

[25]RtlGenRandom函数, Microsoft Document

https://learn.microsoft.com/zh-cn/windows/win32/api/ntsecapi/nf-ntsecapi-rtlgenrandom?redirectedfrom=MSDN

[26]Dorrendorf, Leo; Zvi Gutterman; Benny Pinkas. Cryptanalysis of the Random Number Generator of the Windows Operating System

[27] Andrea Rock, Salzburg, Marz,Pseudorandom Number Generators for Cryptographic Applications

[28] 康步荣, 张磊, 张蕊, 孟欣宇, 陈桐. 抗随机数后门攻击的密码算法