**ОТЧЕТ**

**по лабораторной работе №1 «ГПСЧ»**

**по дисциплине «Криптографические методы защиты информации»**

**Работу выполнил: Коротаев Иван, группа БАС-19**

# **1. Постановка задачи**

1.1. Для линейного конгруэнтного ГПСЧ подободрать параметры таким образом, чтобы получилось равномерное распределение чисел в генерируемой последовательности. Показать равномерное распределение на графике.

1.2. Реализовать криптостойкий ГСПЧ (на выбор) для генерации целых чисел от 0 до 100. Проверить равномерное распределение. Проверить еще одно свойство теста на случайность (на выбор) NIST или Diehard.

# **2. Краткая теория**

**Подходы к построению генераторов случайных чисел:**

1. Используется природная неопределенность.

Например, шум, время, температура, излучение и т.д.

Недостатки:

* Сложно сгенерировать данную последовательность высокого качества
* Вычислительные затраты
* Дороговизна
* Невозможность воспроизвести последовательность

1. ГПСЧ, основанные на каких-либо вычислениях.

Пример: a0, a1 = f(a0), a2 = f(a1) и т.д.

**В наше время используют следующие виды ГПСЧ:**

* Линейный конгруэнтный ГПСЧ
* Линейный сдвиговый регистр с обратной связью

**Линейный конгруэнтный ГПСЧ**

**Дано X0**

**Xi+1 = a\*xi + b (mod N), Z / ZN**

**a, b, x0 – ключи генератора**

**Зная ключи - можно воспроизвести псевдослучайную последовательность**

**Так же этот генератор называют генератором смешанного типа**

**Мультипликативный генератор**

**Дано X0**

**Xi+1 = a\*xi (mod N), Z / ZN**

**N = 2b, b – число бит**

**a = 8T +- 3V, где T – целое число, а V – целое нечетное число**

**Период p = 2b-2 = N / 4**

**Период – это последовательность, которая повторяется через какое-либо число.**

**Например:**

**123 123 123**

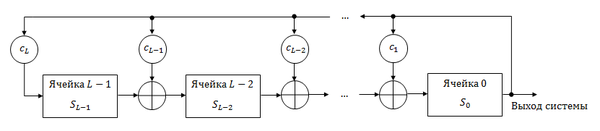
**Аддитивный генератор**

**Дано x0, x1**

**Xi+2 = xi+1 + xi (mod N)**

**P = 2b+1 – 2(b-1)**

**Линейный сдвиговый регистр с обратной связью**



**Вышеприведенные генераторы не удовлетворяют криптостойкости**

**Криптостойкие ГПСЧ – это такие ГПСЧ, где по представленной последовательности невозможно получить алгоритм.**

**Атаки:**

* **Атаки из прошлого – когда по набору чисел последовательности, мы модем определить число**
* **Атака из будущего – когда по набору чисел последовательности, мы можем получить предыдущее число**

**Как можно получить криптостойкое число? Нужно сгенерировать последовательность чисел и применить одностороннюю хеш-функцию.**

**Подходы к построению криптостойких ГПСЧ**

1. **Системно-теоретический**

* **Строится генератор**
* **Проверяются свойства**
* **Проверяется его криптостойкость**

1. **Сложностно-теоретический**

**Основывается на сложности какой-либо задачи**

**Примеры криптостойких ГПСЧ:**

**Генератор RSA**

Берем параметр N = pq, где p и q – простые числа и начальное значение последовательности x0< N. Каждый ее элемент вычисляется по формуле (1.4).

(1.4)

Результат генератора – наименьший значимый бит xi. Стойкость этого генератора соизмерима со стойкостью RSA. Если N достаточно большое, то генератор надежен.

**Генератор Blum-Blum-Shub (BBS)**

Основан на применение квадратичных вычетов по модулю n. Найдем два больших простых числа p и q, дающих при делении на 4 остаток 3. Произведение n = pq называется числом Блюма. Выберем x: НОД(n, x)=1.

Вычислим начальное значение генератора: x0=x2modn .

В псевдослучайной последовательности i-ым элементом является наименьший значимый бит xi, где xi = xi-22modn .

Особенностью данного генератора является то, что нет необходимости проходить через i −1 состояние для того, чтобы получить i-й бит. Если мы знаем p и q, то мы можем вычислить bi - младший значащий бит xi, которое находится по формуле (1.5).

(1.5)

Таким образом, есть возможность использовать криптографически стойкий ГПСЧ в качестве потоковой криптосистемы для файлов с произвольным доступом.

Для генератора BBS нельзя предсказать предыдущий или последующий бит, имея часть последовательности. Кроме того, можно использовать не только наименьший значащий бит, но и l битов xi. Предложенный генератор медленный, но может использоваться для высоконадежных приложений.

**Генератор Blum-Micali**

Стойкость алгоритма определяется трудностью решения задачи дискретного логарифма. Пусть g и p – простые числа. Ключ используется в качестве начального значения x0, последовательность формируется по формуле (1.6).

modp (1.6)

Выходом генератора является 1, если , и 0 в противном случае.

Если p достаточно велико, чтобы вычисление дискретного логарифма по модулю p стало физически невозможным, то этот генератор является безопасным.

**Генератор ANSI X9.17 (Тройной DES)**

Данный ГПСЧ используется во многих приложениях финансовой безопасности. В его основе лежит использование тройного DES. Генератор состоит из следующих частей:

* Вход: генератором управляют два псевдослучайных входа. Один является 64-битным представлением текущих даты и времени, другой - 64-битным исходным значением. Оно инициализируется произвольным значением и меняется в ходе генерации последовательности.
* Ключи: генератор использует три модуля тройного DES. Все три используют одну и ту же пару 56-битных ключей, которая держится в секрете и применяется только при генерации псевдослучайного числа.
* Выход: выход состоит из 64-битного псевдослучайного числа и 64-битного значения, которое будет использоваться в качестве начального значения при создании следующего числа.

Введем следующие обозначения:

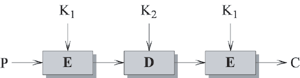
* DTi - дата и время начала i-ой стадии генерации.
* Vi - начальное значение для i-ой стадии генерации.
* Ri - псевдослучайное число, созданное на i-ой стадии генерации.
* K1 и K2 - ключи, используемые на каждой стадии.

Генерация чисел происходит следующим образом.

*,* (1.7)

*.* (1.8)

EDE обозначает 3 стадии шифрования тройного DES.



**Существует основные тесты на случайность:**

* NIST
* Diehard

# **3.Пошаговое решение**

1. Линейный конгруэнтный генератор.

**Xi+1 = a\*xi + b (mod N)**

Для того, чтобы у нас был хороший генератор, нам нужно, чтобы его период p был равен N – модулю

Существуют следующие правила подбора параметров, при которых период равен модулю.

**Правила:**

* Числа b и N должны быть взаимно простые
* Число a – 1 делится на все простые делители числа N
* Если N делится на 4, то и a-1 делится на 4
  1. Я подобрал следующие параметры:

a = 21

b = 31

N = 100

Проверим их по правилам:

1. Числа b и N являются взаимно простыми, так как b – простое +
2. Простые делители числа N – 2, 5; a – 1 = 20; 20 / 2 = 10; 20 / 5 = 4 +
3. N делится на 4 (100 / 4 = 25), и a – 1 = 20 тоже делится на 4 (20 / 4 = 5) +

* 1. Напишем программу, которая генерирует последовательности чисел
     1. Задаем параметры

1. # Параметры  
   a = 21  
   b = 31  
   n = 100  
   x0 = a  
   list\_of\_numbers = []  
   x = x0

1.2.2. Генерируем 100\*N псевдослучайных чисел согласно формуле **Xi+1 = a\*xi + b (mod N)**

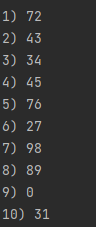
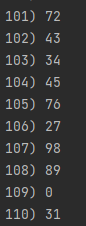
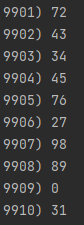
# Генерация псевдослучайных чисел  
for i in range(0, 100 \* n):  
 x = (a \* x + b) % n  
 list\_of\_numbers.append(x)

1.2.3. Отсортируем числа по их частотам и запишем в CSV файл

# Сортировка и подсчет частот  
counter\_numbers = Counter(list\_of\_numbers)  
counter\_numbers = collections.OrderedDict(sorted(counter\_numbers.items()))

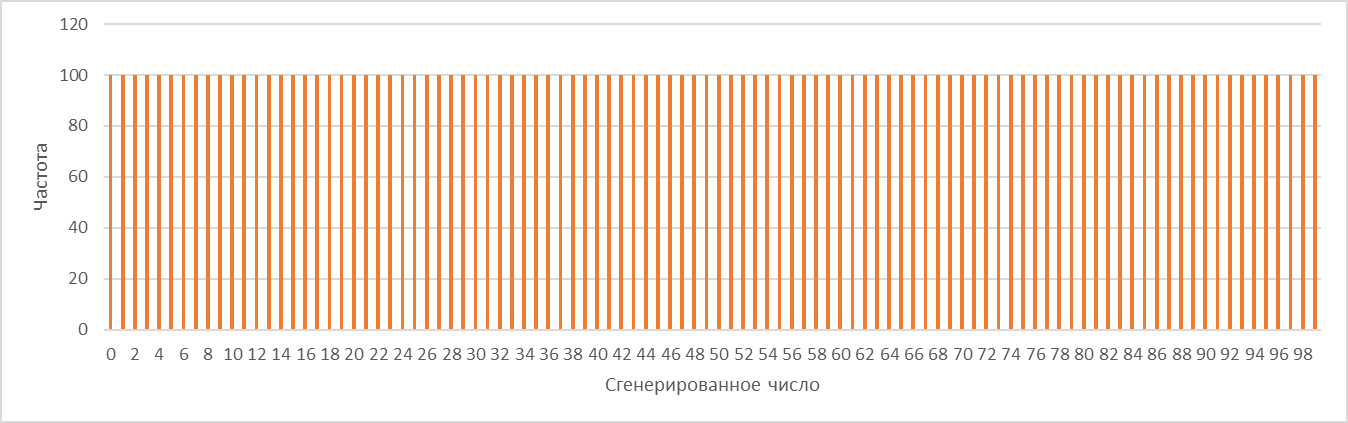
# Запись в файл  
with open('output\_file.csv', 'w', newline='') as output:  
 writer = csv.writer(output, delimiter=';')  
 for key, value in counter\_numbers.items():  
 writer.writerow([key, value])

1.2.4. Выведем все сгенерированные числа и найдем период



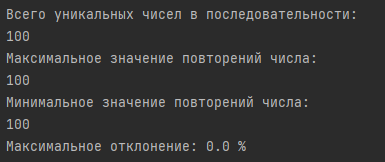
Мы видим, что последовательность повторяется через 100 или через модуль N. Это значит, что наши параметры подобраны правильно и период равен модулю ( p = N).

1.3. Покажем равномерное распределение на графике с помощью EXCEL.



На графике плохо видно, но все частоты равны 100. А это значит, что у нас получилось идеальное равномерное распределение.

1.4. С помощью программы посчитаем отклонение.



print('Отсортированный список (число, частота):\n', counter\_numbers, sep='')  
print('Всего уникальных чисел в последовательности: \n', len(counter\_numbers), sep='')  
  
max\_n, min\_n = max\_min\_number(counter\_numbers)  
print('Максимальное значение повторений числа: \n', max\_n, '\nМинимальное значение повторений числа: \n', min\_n, sep='')  
  
max\_dev = max(min\_n/100 - 1, max\_n/100 - 1)  
print('Максимальное отклонение:', max\_dev\*100, '%')

# Максимальная и минимальная частота в последовательности  
def max\_min\_number(arr):  
 max\_num = - sys.maxsize  
 min\_num = sys.maxsize  
  
 for val in arr.values():  
 if val > max\_num:  
 max\_num = val  
 if val < min\_num:  
 min\_num = val  
 return max\_num, min\_num

**2. В качестве криптостойкого ГПСЧ я выбрал криптостойкий ГПСЧ, основанный на методе RSA.**

**Алгоритм:**

1. Генерируется 2 больших простых числа p, q.
2. Вычисляется N = p \* q
3. Вычисляется φ = (p-1)(q-1)
4. Выбирается случайное число e из диапазона 1 < e < φ, такое что (φ, e) = 1
5. Выбирается случайное стартовое число x0 из интервала [2, N-1]
6. Задается количество бит s , которое хотим сгенерировать
7. Для i от 0 до l(не включительно) выполняется:

Xi+1 = xie (mod N)

Берем наименее значимый бит xi и обозначаем его как zi

1. Выходная криптостойкая последовательность бит является z1, z2, … , zs

Безопасность RSA генератора опирается на сложность вскрытия RSA. Если N достаточно велико (разрядность 1024 бит), то генератор безопасен. В нашем случае я взял 256 бит, чтобы вычисление шли быстрее.

* 1. Реализация RSA для генерирования криптостойской последовательности бит.

1. # Алгоритм Криптографического ГПСЧ, основанного на RSA  
   def generate\_bits\_rsa(bits\_count):  
    # Генерация 2 256-битных чисел  
    p = Crypto.Util.number.getPrime(256, randfunc=Crypto.Random.get\_random\_bytes)  
    q = Crypto.Util.number.getPrime(256, randfunc=Crypto.Random.get\_random\_bytes)  
    # Расчет модуля  
    n = p \* q  
    # Расчет функции Эйлера  
    fi = (p - 1) \* (q - 1)  
    # Генерируем начальное значение  
    x0 = random.randint(2, n - 1)  
    x = x0  
    # Подбираем ключ e  
    e = random.randint(2, fi - 1)  
    while math.gcd(e, fi) != 1:  
    e = random.randint(2, fi - 1)  
    list\_of\_bits = []  
    # Генерируем биты  
    for i in range(0, bits\_count):  
    x = pow(x, e, n)  
    list\_of\_bits.append(x & 1)  
    return list\_of\_bits
   1. В нашей работе нужно создать генератор целых чисел на некотором интервале.

Это оказалась нетривиальная задача.

Например, пусть нам нужно сгенировать число в промежутке от 0 до 100 и у нас есть генератор, который равномерно генерирует битовую последовательность.

Мой алгоритм:

1. Генерирование 8 бит и перевод его в десятичное число num. Тут у нас будет вероятность выпадение числа от 0 до 127 равна 1 / 128.
2. Так как нам нужно получить число в промежутке от 0 до 99, то мы берем отстаток числа num от деления на 100 (num % 100).
3. Так как у нас может сгенеироваться число num от 100 до 127, то при взятии остатка оно перейдет от 0 до 27. То есть вероятность получить число от 0 до 27 будет в два раза больше, чем получить число от 28 до 99 и поэтому это распределение будет не равномерным.

Я нашел статью на сайте Microsoft как они попытались решить эту проблему:

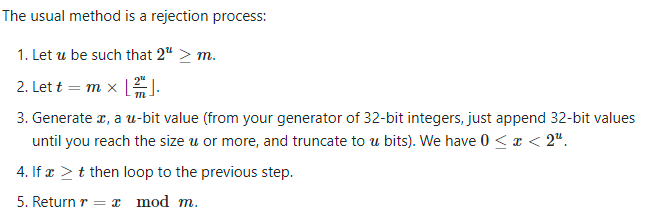
<https://learn.microsoft.com/en-us/archive/msdn-magazine/2007/september/net-matters-tales-from-the-cryptorandom>

Алгоритм:

Пусть m = Y – X, где Y – верхняя граница интервала, а X – нижняя. Сгенерированное число будет

X <= num < Y.

Алгоритм:



Идея метода заключается в том, что 2u диапазон разбит на поддиапазон размера t (кратного m) и другой другой поддиапазон меньше 2. Если u-битное значение x попадает в первый диапазон, тогда берем его по модулю m, что дает равномерное значение, поскольку t ялвяется кратным m. В противном случае нужно повторить генерацию битов.

Я не до конца понял алгоритм, но попытался программно повторить это.

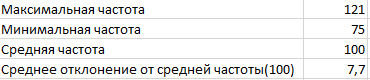
Я в своей программе генерировал не 7 бит (от 0 до 127), а 32 бита(от 0 до 4294967295).

# Функция генерации целого числа в заданном промежутке  
def random\_int(min\_value, max\_value):  
  
 # Количество генерируемых бит  
 COUNT\_OF\_BITS = 32  
  
 # Верхняя граница 32 битного числа  
 MAX\_UINT32 = 4294967295  
  
 if min\_value > max\_value:  
 raise 'Неверно заданы границы'  
 if min\_value == max\_value:  
 return min\_value  
  
 # Разница между нижней и верхней границей промежутка  
 diff = max\_value - min\_value  
  
 while True:  
 # Генерирование бит  
 rand\_bits = generate\_bits\_rsa(COUNT\_OF\_BITS)  
 # Перевод из массива битов в целое число  
 rand = from\_bitarray\_to\_number(rand\_bits)  
 max\_val = 1 + MAX\_UINT32  
 remainder = max\_val % diff  
 if rand < max\_val - remainder:  
 return min\_value + (rand % diff)

# Перевод из двоичного массива в число  
def from\_bitarray\_to\_number(bitarr):  
 template\_str = ''  
 for i in bitarr:  
 template\_str += str(i)  
 num = int(template\_str, 2)  
 return num

# Функция генерации последовательности случайных чисел  
def random\_sequence():  
 # Цикл, в котором получаем 10000 случайных значений на промежутке  
 list\_of\_nums = []  
 for j in range(0, 10000):  
 list\_of\_nums.append(random\_int(0, 100))  
  
 # Подсчет частоты сгенерированных чисел  
 counter\_nums = Counter(list\_of\_nums)  
 counter\_nums = collections.OrderedDict(sorted(counter\_nums.items()))  
  
 # Запись в файл  
 with open('output\_random\_sequence.csv', 'w', newline='') as output:  
 writer = csv.writer(output, delimiter=';')  
 for key, value in counter\_nums.items():  
 writer.writerow([key, value])

Я сделал цикл, который получает 10000 случайных значений в промежутке от 0 до 99 и записывает их частоты в CSV файл.



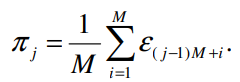
**Приблизительно получилось раавномерное распределение.**

* 1. **Блочный тест на частоту (NIST)**

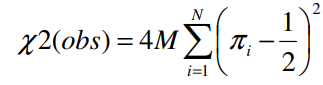
Суть теста заключается в определении доли 1-ц в блоке, состоящем из M бит и сравнении его с ожидаемым значением M/2.

Алгоритм теста:

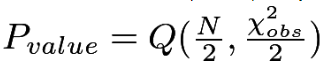
1. Битовый поток разбивается на N блоков по M бит каждый, остаток отбрасывается.
2. Далее для каждого блока j = 1, 2, …, N вычисляется доля πj 1-ц в нем:

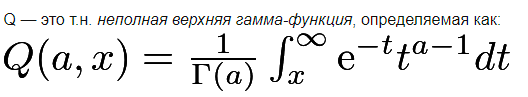


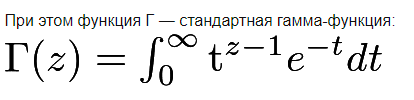
1. Вычисляется статистика χ2 по формуле:



1. После этого вычисляется P\_value по формуле:





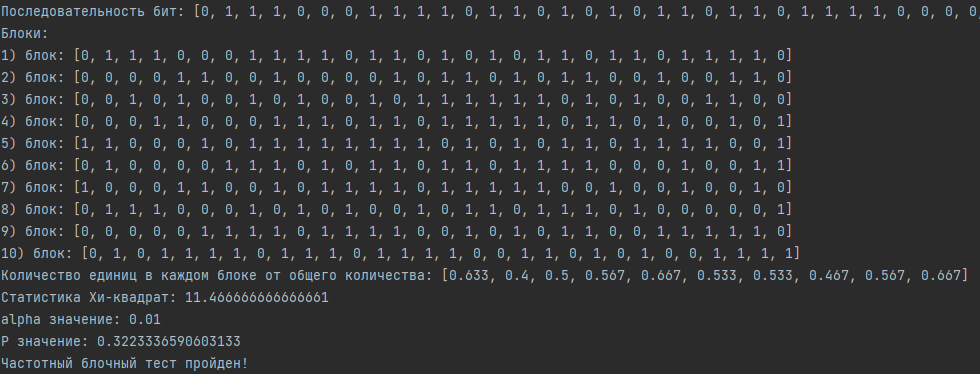


1. Если значение P\_value > α, то числа не случайные.

**Последовательность считается случайной, если P-значение > 0.01. Рекомендуется анализировать последовательности длиной не менее 100 бит, а также должны выполняться соотношения M >= 20, M > 0.01n и N < 100.**

* + 1. Программная реализация

# Блочный тест на частоту  
def block\_frequency():  
 # Размер блока  
 size\_of\_block = 30  
  
 # Количество генерируемых битов  
 size\_of\_bits = 300  
  
 # альфа значение  
 alpha = 0.01  
  
 # Генерация бит  
 list\_of\_bits = generate\_bits\_rsa(size\_of\_bits)  
 print('Последовательность бит: ', list\_of\_bits, sep='')  
 list\_of\_blocks = []  
  
 # Количество блоков  
 count\_of\_blocks = len(list\_of\_bits) // size\_of\_block  
  
 # Разделение на блоки  
 print('Блоки: ')  
 for i in range(0, count\_of\_blocks):  
 list\_of\_blocks.append(list\_of\_bits[30 \* i:30 \* (i + 1)])  
 print(i + 1, ') блок: ', list\_of\_blocks[i], sep='')  
 list\_of\_propotrion\_ones = []  
  
 # Подсчет единиц в каждом блоке  
 for i in list\_of\_blocks:  
 count\_of\_ones = i.count(1)  
 list\_of\_propotrion\_ones.append(count\_of\_ones / size\_of\_block)  
 print('Количество единиц в каждом блоке от общего количества: ',  
 [round(item, 3) for item in list\_of\_propotrion\_ones], sep='')  
  
 # Статистика Хи-квадрат  
 chi2 = 0  
 for i in list\_of\_propotrion\_ones:  
 chi2 += pow((i - 0.5), 2)  
 chi2 \*= 4 \* size\_of\_block  
 print('Статистика Хи-квадрат: ', chi2, sep='')  
  
 print('alpha значение: ', alpha, sep='')  
   
 # Вычисление P\_value  
 p\_value = sc.gammaincc(count\_of\_blocks / 2, chi2 / 2)  
 print('P значение: ', p\_value, sep='')  
  
 # Проверка  
 print('Частотный блочный тест пройден!' if p\_value > alpha else 'Частотный блочный тест провален!')

****

**Частотный блочный тест был успешно пройден.**

# **4. Анализ и вывод**

Мы построили линейный конгруэнтный ГПСЧ и подобрали к нему параметры. Мы сделали последовательность из 10000 сгенерируемых элементов и нашли период генератора. Затем, мы построили график распределения частот и показали, что генератор имеет равномерное распределение.

Во второй части лабораторной работы, мы реализовали криптостойкий ГСПЧ, основанный на методе RSA для генерирования последовательности бит. Мы столкнулись с проблемой генерации целых чисел в заданном промежутке, но попытались решить ее. Так же мы показали, что генерируемая последовательность имеет равномерное распределение. Последней нашей задачей мы проверили генерируемую последовательность на случайность одним из тестов NIST.