

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний технічний університет України “Київський  
політехнічний інститут імені Ігоря Сікорського”

**КОМП’ЮТЕРНИЙ ПРАКТИКУМ З ДИСЦИПЛІНИ  
«МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ  
МЕХАНІЗМІВ» №2**

Реалізація алгоритмів генерації ключів гібридних криптосистем

Виконали:

Морозюк Анастсія

Гетьман Дмитро

Мітрофанова Еліна

Перевірила:

Селюх Поліна Валентинівна

Київ 2021

**Мета роботи:** Дослідження алгоритмів генерації псевдовипадкових послідовностей, тестування простоти чисел та генерації простих чисел з точки зору їх ефективності за часом та можливості використання для генерации ключів асиметричних криптосистем.

**Завдання:** Запропонувати схему генератора ПВЧ для інтелектуальної картки, токена чи смартфона. Розглянути особливості побудови генератора простих чисел в умовах обмеженої пам'яті та часу генерації.

## Теоретичні відомості

Генератори:

1. *Лінійний конгруентний генератор (генератор Лемера)* обчислює послідовність  $x_{n+1} = (a \cdot x_n + c) \bmod m$  для фіксованих значень  $a$ ,  $c$  та  $m$  і деякого початкового значення  $x_n$ . Доведено, що максимальний період такого генератора дорівнює  $m$  і досягається він за виконання таких трьох умов:

- числа  $m$  та  $c$  повинні бути взаємнопрості;
- число  $a - 1$  повинно ділитись на кожен простий дільник числа  $m$ ;
- якщо  $m$  ділиться на 4, то  $a - 1$  теж повинно ділитись на 4.

Значення  $x$  можуть повертатись безпосередньо або розглядатись як стани, з яких за допомогою деякого перетворення обчислюються байти вихідної послідовності.

**LehmerHigh** - модифікація лінійного конгруентного генератора, що генерує випадкові байти. Обчислюється послідовність невід'ємних 32-бітних чисел  $x_n$ ; значення параметрів дорівнюють  $m = 2^{32}$ ,  $a = 2^{16} + 1$ ,  $c = 119$ , початкове значення  $x_0$  обирається довільно, але не повинно дорівнювати нулю. LehmerHigh в якості  $n$ -того вихідного значення повертає старші 8 біт  $x_n$ .

2. *Генератор BBS (Блум-Блюма-Шуба)* побудовано на ідеях Блюма та Мікалі, однак для генерування псевдовипадкових бітів він використовує апарат теорії чисел. Доведено, що можливість вгадувати біти вихідної послідовності цього генератора еквівалентна можливості розв'язувати задачу факторизації.

Нехай  $p$  та  $q$  – різні великі прості числа виду  $4k + 3$  і  $n = pq$ . Початкове значення  $r_0 \geq 2$  обирається довільним чином. Вихідна послідовність  $x_1, x_2 \dots$  обчислюється за таким правилом:

$$\begin{aligned} r_i &= r_i^2 \bmod n, \\ x_i &= r_i \bmod 2, \end{aligned}$$

тобто  $x_i$  є останнім бітом числа  $r_i$ .

Байтова модифікація генератору BBS обчислює вихідну послідовність як  $x_i = r_i \bmod 256$ , тобто повертаються вісім молодших біт числа  $r_i$ .

3. Клас System.Security.Cryptography.RandomNumberGenerator.  
Представляє абстрактний клас, від якого походять усі реалізації криптографічних генераторів випадкових чисел.

## Конструктори

RandomNumberGenerator()	Ініціалізує новий екземпляр RandomNumberGenerator.
-------------------------	--

## Методи

Create()	Створює екземпляр реалізації за замовчуванням криптографічного генератора випадкових чисел, який можна використовувати для генерування випадкових даних.
Create(String)	Створює екземпляр вказаної реалізації криптографічного генератора випадкових чисел.
Dispose()	При перевизначенні в похідному класі звільняє всі ресурси, які використовуються поточним екземпляром класу RandomNumberGenerator.
Dispose(Boolean)	При перевизначенні в похідному класі звільняє некеровані ресурси, які використовуються RandomNumberGenerator, і за бажанням звільняє керовані ресурси.
Equals(Object)	Визначає, чи дорівнює вказаний об'єкт поточному об'єкту.
Fill(Span<Byte>)	Заповнює проміжок криптографічно сильними випадковими байтами.
GetBytes(Byte[])	При перевизначенні в похідному класі заповнює масив байтів криптографічно сильною випадковою послідовністю значень.
GetBytes(Byte[], Int32, Int32)	Заповнює вказаний масив байтів криптографічно міцною випадковою послідовністю значень.
GetBytes(Span<Byte>)	Заповнює проміжок криптографічно сильними випадковими байтами.
GetHashCode()	Служить хеш-функцією за замовчуванням
GetInt32(Int32)	Генерує випадкове ціле число від 0 (включно) до вказаної ексклюзивної верхньої межі за допомогою криптографічно надійного генератора випадкових чисел
GetInt32(Int32, Int32)	Генерує випадкове ціле число між вказаною нижньою межею, що включає в себе, і вказаною виключною верхньою

	межею за допомогою криптографічно надійного генератора випадкових чисел.
GetNonZeroBytes(Byte[])	При перевизначенні в похідному класі заповнює масив байтів криптографічно сильною випадковою послідовністю ненульових значень.
GetNonZeroBytes(Span<Byte>)	Заповнює діапазон байтів криптографічно міцною випадковою послідовністю ненульових значень.
GetType()	Отримує тип поточного екземпляра.
MemberwiseClone()	Створює неглибоку копію поточного об'єкта.
ToString()	Повертає рядок, який представляє поточний об'єкт.

## Тести перевірки на простоту

### Тест Ферма

Вхід: непарне число  $n$ .

Додатковий вхід:  $x$ , таке, що  $\gcd(x, n) = 1$ .

Перевіряємо виконання порівняння:  $x^{n-1} \equiv 1 \pmod{n}$ .

Вихід: якщо порівняння виконується, то « $n$  – складене», інакше – «не знаємо».

### Тест Соловея-Штрассена

Вхід: непарне число  $n$ .

Додатковий вхід:  $x$ , таке, що  $\gcd(x, n) = 1$ .

Перевіряємо виконання критерію Ойлера:  $\left(\frac{x}{n}\right) \equiv x^{(n-1)/2} \pmod{n}$ , де  $\left(\frac{x}{n}\right)$  – символ Ойлера.

Вихід: якщо критерій виконується, то « $n$  – просте», інакше – « $n$  – складене».

### Тест Міллера-Рабіна

Вхід: непарне число  $n$ :  $n - 1 = 2^s t$ , де  $t$  – непарне.

Додатковий вхід:  $x$ , таке, що  $\gcd(x, n) = 1$ .

1. Обчислюємо  $y_0 = x^t \bmod n$ . Якщо  $y_0 \equiv \pm 1 \pmod{n}$ , то вихід: « $n$  – просте», інакше йдемо до пункту 2.
2. Обчислюємо  $y_j = y_{j-1}^2 \bmod n$  поки не виявиться, що  $j = s - 1$ , або  $y_j \equiv \pm 1 \pmod{n}$  для деякого  $j < s - 1$ .
3. Якщо для деякого  $j < s - 1$  виконується  $y_j \equiv -1 \pmod{n}$ , то припиняємо роботу алгоритму з виходом: « $n$  – просте», інакше « $n$  – складене».

## Хід роботи

Було розроблено декілька генераторів псевдопипадкових послідовностей, та тестів перевірки чисел на простоту. Також було порівню роботу за часом розроблених функцій.

Було реалізовано генератори псевдоаипадкових послідовностей LehmerHigh та BBS (байтову модифікацію), також було генеровано послідовності за допомогою System.Security.Cryptography. RandomNumberGenerator. З отриманих послівностей було створено великі числа довжин 32, 64 та 128 байтів та фіксовано час роботи кожного генератора. Як видно з реультатів предсталених на рис.1.1, рис.1.2 та рис.1.3., в середньому RandomNumberGenerator працює швидше.

```
-----
Generator: LehmerHigh
Result: B0AA39D98938E8A8683817E7D7B7A7A7B7C7D7F8284888C90959AA0A6ACB3
Time: 00:00:00.0149762
-----
Generator: RandomNumberGenerator
Result: E7334B832D35C0489C0EDDF57969F8E78C8C079BD65D634F6F7B11599B8A6407
Time: 00:00:00.0055436
-----
Generator: BBS Bytes
Result: 2593B7353FF732F06148B0BDC08439E7A3828B0D8820977A0000000000001004
Time: 00:00:00.0005860
```

Рис.1.1. Довжина 32 байти.

```
-----
Generator: LehmerHigh
Result: C9229C159F18A24BE58F38E2AC663009E3CDA7919B959FA9B3DE08226C96D12B75C02A950F79F47F09943ED9843EF9B47F4A24FFEAD5C0CBC6D1ECF8134E79A
Time: 00:00:00.0001448
-----
Generator: RandomNumberGenerator
Result: E7445795F2DA60E0580642B828E75BD186482019360144A398A5AD8E1AEDBFB4E1CADB5C6B636D86B3BB359F8150C8849684B56687DDA76111AB179C826AF3D9
Time: 00:00:00.0000406
-----
Generator: BBS Bytes
Result: E6178D01AAD9021042D604DFE9F19630BED7E642807050286EC9D2FB0A9B110495F4076AEBADFC1CC02D4E7D91067A344E203B6BCD46AF44E41F4F4DA8000000
Time: 00:00:00.0001519
```

Рис1.2. Довжина 64 байти.

```

-----
Generator: LehmerHigh
Result: B06417CB8035EBA1570EC67E36EFA8621CD7924E0AC68341FFBD7C3BFBB7C3EFFC184470BCF93581DE3A97038FFC7905923EDB7824E1AE6B3804E
1CEBBA8A5A2AFBCD9F714417EBBF94693F15ECC39A724823FDD7B18C67431FFCD9B69473523111F1D2B395775A3D2105E9CEB49A80674E361E07F0DAC4AF9A
85715E4B3826
Time: 00:00:00.0000596
-----
Generator: RandomNumberGenerator
Result: DE2846A3F920CE170655FF5015E1C682112F083E57C4D583B365866CD3F4802E73D4E4D0DA811E67343452B8148B5212FF5B9134E228B076F26AB1
24FAB53C56B308DB4EDB7616CF8D04FB8E196A06B46A4C1A9B6CBB22D99D0BB83367E07AA67EE2CE8D310D0889BC7A002B59FD1A6651154C6B5131E5BFC455
42358D78010E
Time: 00:00:00.000075
-----
Generator: BBS Bytes
Result: 715DFF0991501660A0B1403DD5862EC25C297C7FDD429B4A90458C96463F7D8E26709786CD1A8C073F12060050B66B661979A48F4497F3D93034D9
60974FB885AECAC8ABD99207B5CE1D6BB4549685686EC19929F8AD0A7982155BE7E416DE52EB3CE0A0960F5666FA6BB7CB3702E33A03D5BE4E5EF5BF171E75
08DF6000090
Time: 00:00:00.0002139

```

Рис.1.3. Довжина 28 байтів.

В ході роботи було також розроблено 3 тести перевірки чисел на простоту, а саме тест Міллера-Рабіна, тест Ферма та тест Соловея-Штрассена. Ці функції повертають TRUE якщо число просте та FALSE якщо число складене. Було реалізовано функцію генерації простих чисел заданої довжини GeneratePrime. Вона працює наступним чином: генерує число згідної довжини поки воно не пройде тест Міллера-Рабіна на простоту. Утворене число також гарантовано проходить всі інші тести на простоту рис.1.4.

```

Prime number: 21482712318851957774351401682694065347732761656352158772606002783191325951837
Test results:
-----
Test: Fermat
Result: True
Time: 00:00:00.0015745
-----
Test: Solovay-Strassen
Result: True
Time: 00:00:00.0033347
-----
Test: Miller-Rabin
Result: True
Time: 00:00:00.0135303

```

Рис.1.4. Перевірка простого числа на проходження тестів на простоту

Вибір тесту Міллера-Рабіна пояснюється його надійністю в порівнянні з іншими тестами. Оскільки тести Ферма та Соловея-Штрассена в деяких випадках можуть давати результат “просте” коли число насправді складене. Така поведінка може відбуватись при перевірці чисел Кармайкла. Результати такої ситуації на прикладі числа 6601 (рис.1.5., рис.1.6, рис.1.7):



```
Prime number: 6601
Test results:
-----
Test: Fermat
Result: False
Time: 00:00:00.0035138
-----
Test: Solovay-Strassen
Result: True
Time: 00:00:00.0019063
-----
Test: Miller-Rabin
Result: False
Time: 00:00:00.0015996
```

Рис.1.5. Перевірка на простоту числа Кармайкла

```
Prime number: 6601
Test results:
-----
Test: Fermat
Result: True
Time: 00:00:00.0018416
-----
Test: Solovay-Strassen
Result: True
Time: 00:00:00.0041381
-----
Test: Miller-Rabin
Result: False
Time: 00:00:00.0015423
```

Рис.1.5. Перевірка на простоту числа Кармайкла

```

Prime number: 6601
Test results:
-----
Test: Fermat
Result: True
Time: 00:00:00.0018356
-----
Test: Solovay-Strassen
Result: False
Time: 00:00:00.0017241
-----
Test: Miller-Rabin
Result: False
Time: 00:00:00.0013134

```

Рис.1.5. Перевірка на простоту числа Кармайкла

Як видно при перевірці числа Кармайкла лише тест Міллера-Рабіна дає стабільно однаковий результат, при чому правильний, саме тому краще застосовувати його при генерації простих чисел.

Номер запуску	Міллер-Рабін	Ферма	Соловей-Штрассен
1	FALSE	FALSE	TRUE
2	FALSE	TRUE	TRUE
3	FALSE	TRUE	FALSE

## **Висновки**

В результаті виконання роботи ми дослідили різні алгоритми генерації псевдовипадкових послідовностей, тестування простоти чисел та генерації простих чисел з точки зору їх ефективності за часом та можливості використання для гененерації ключів асиметричних криптосистем. Також ми поглибили наші знання у використанні C# BigInteger.