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 гененерації ключів асиметричних криптосистем.

Завдання: Запропонувати схему генератора ПВЧ для інтелектуальної картки, токена чи смартфону. Розглянути особливості побудови генератора простих чисел в умовах обмеженної 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 n x можуть повертатись безпосередньо або розглядатись як стани, з яких за допомогою деякого перетворення обчислюються байти вихідної послідовності.

   LehmerHigh - модифікація лінійного конгруентного генератора, що генеруює 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 (Блюм-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 генератору 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 … обчислюється за таким правилом:

   $$r_i = r_i^2 \bmod n ,$$

   $$x_i = r_i \bmod 2 ,$$

   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: $x^{n-1} \equiv 1 (mod\ n)$.

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: $(\frac{x}{n}) \equiv x^{(n-1)/2}\ (mod\ n)$, де $(\frac{x}{n})$ – 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 $y_0 = x^t mod\ n$. Якщо $y_0 \equiv \pm 1 (mod\ n)$, то вихід: «$n$ – просте», інакше йдемо до пункту 2.
2. Обчислюємо $y_j = y^2_{j-1}\ mod\ n$ поки не виявиться, що $j = s - 1$, або $y_j \equiv \pm 1 (mod\ n)$ для деякого $j < s - 1$.
3. Якщо для деякого $j < s - 1$ виконується $y_j \equiv -1 (mod\ n)$, 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 порівнно 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.

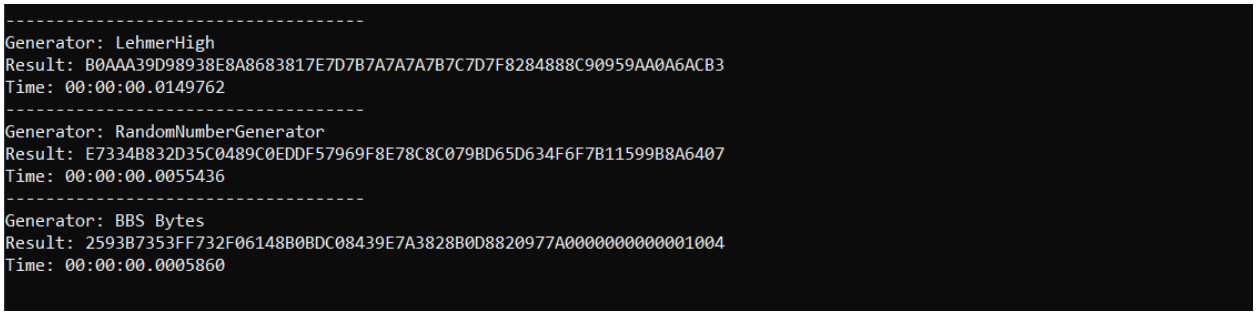

Рис.1.1. Довжина 32 байти.

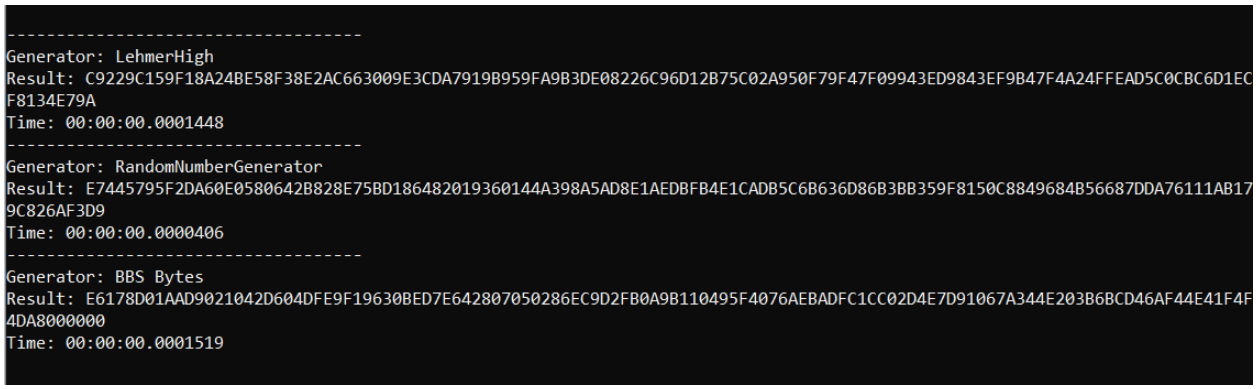

Рис1.2. Довжина 64 байти.

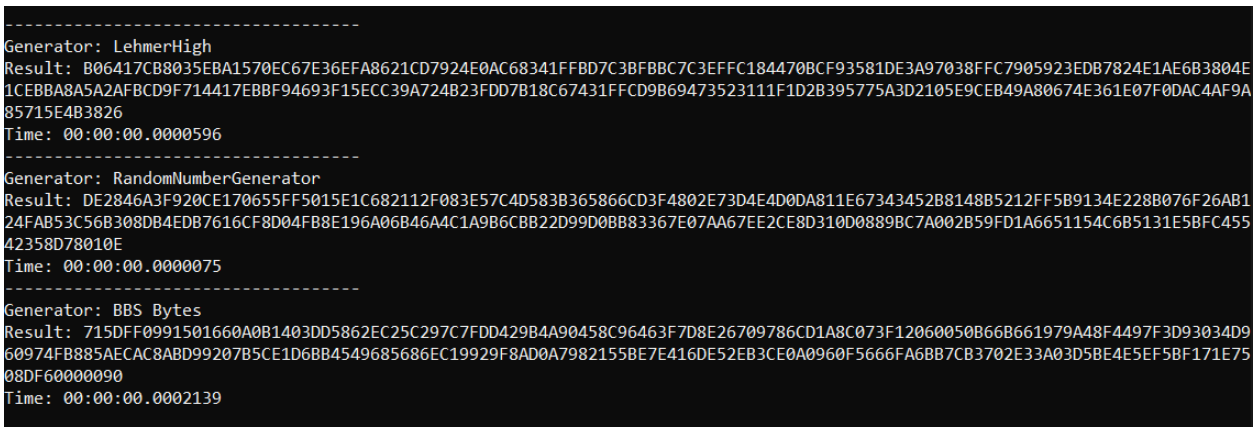

Рис.1.3. Довжина 28 байтів.

В ході роботи було також розроблено 3 тести перевірки чисел на простоту, а саме тест Міллера-Рабіна, тест Ферма та тест Соловея-Штрассена. Ці фукції 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 зпдпної дожнини поки воно не пройде тест Міллера-Рабіна на прстоту. Утворене число також гарантовано проходить всі інші тести на простоту рис.1.4.

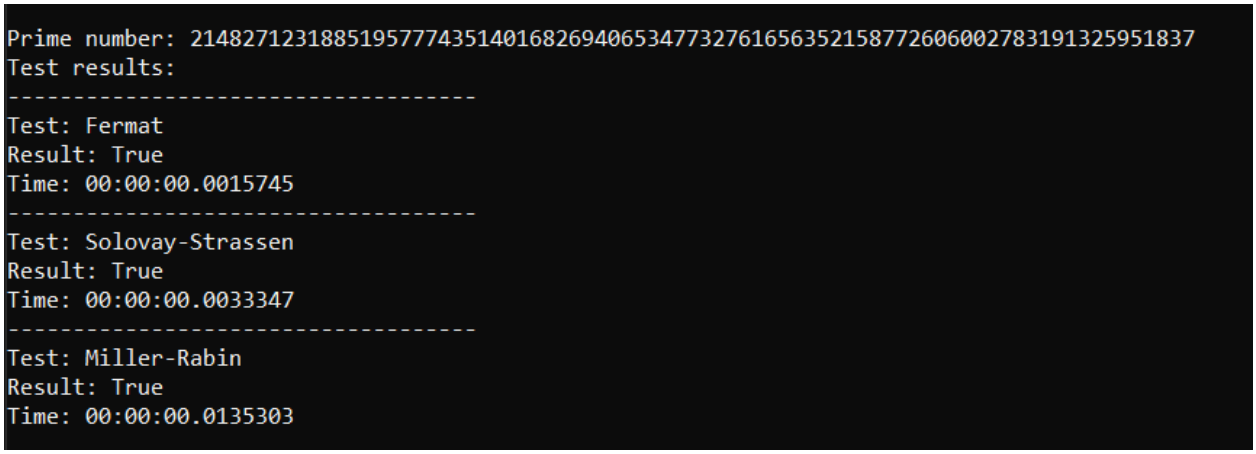

Рис.1.4. Перевірка прсотого числа на проходження тестів на простоту

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

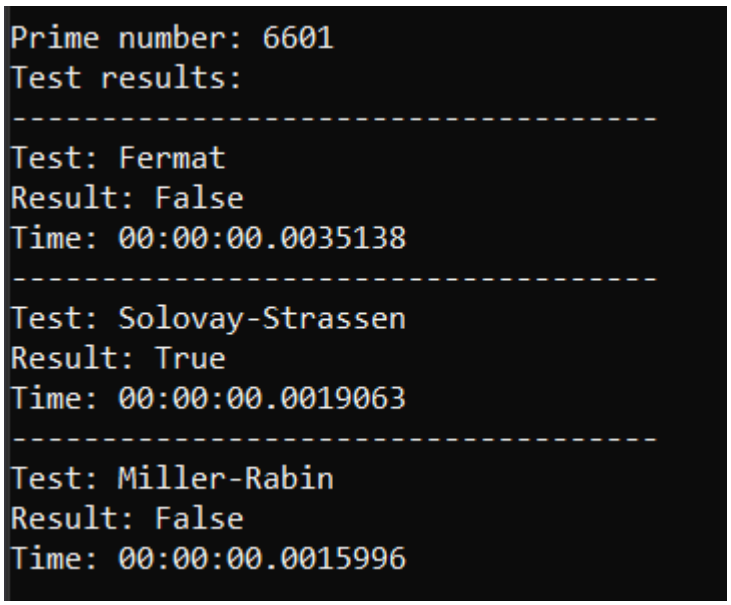

Рис.1.5. Перевірка на простотут числа Кармайкла

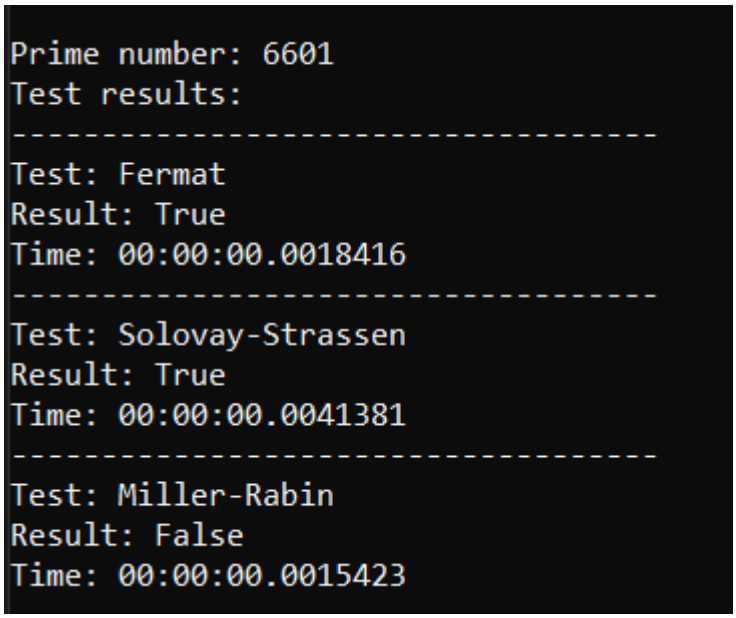

Рис.1.5. Перевірка на простотут числа Кармайкла

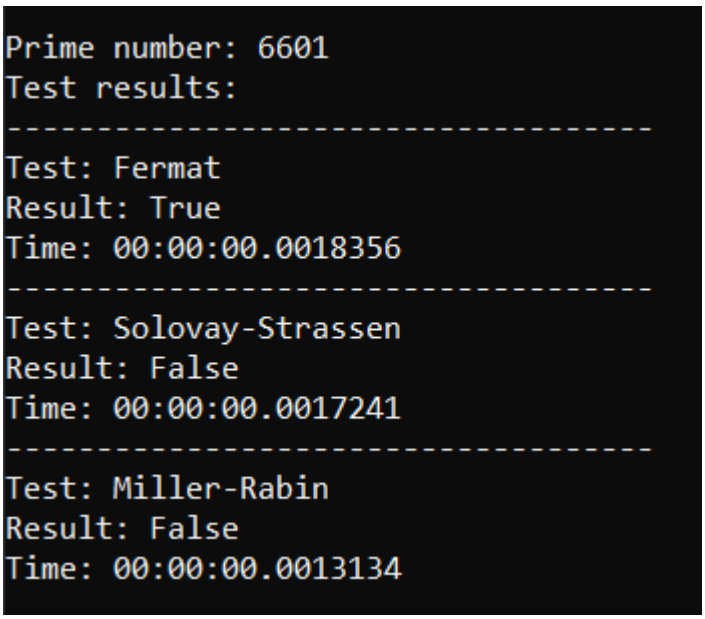

Рис.1.5. Перевірка на простотут числа Кармайкла

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.