НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра математичних методів захисту інформації

Звіт

з комп’ютерного практикуму

«Реалізація алгоритмів генерації ключів гібридних криптосистем»

з кредитного модуля

«Сучасні алгебраїчні криптосистеми»

Варіант 1В

Виконала студентка

Групи ФІ-52мн

Остаповець Олеся

Київ 2025

1. Теоретичні відомості

Генерація випадкових послідовностей у засобах обчислювальної техніки здійснюється кількома підходами:

1. Апаратні генератори випадковості (TRNG).
   TRNG отримують випадкові біти з фізичних процесів, наприклад із шуму електронних компонентів. Перевага такого підходу — наявність "справжньої" ентропії, яка не є результатом детермінованих обчислень. Однак вихід TRNG може мати статистичні перекоси та залежати від якості апаратної реалізації, тому на практиці його часто застосовують як джерело первинної ентропії (seed), а також поєднують із постобробкою.
2. Системні генератори випадковості ОС (System RNG).
   Сучасні операційні системи реалізують системний генератор, який збирає ентропію з доступних джерел у тому числі апаратних, якщо вони присутні і надає прикладний інтерфейс для отримання криптографічно стійких випадкових байтів. Для "user endpoint terminal" це найтиповіший і рекомендований шлях отримання випадковості, оскільки відповідальність за збір ентропії та захист внутрішнього стану бере на себе ОС.
3. Псевдовипадкові генератори (PRNG).
   PRNG є детермінованими: маючи початкове значення seed, вони формують послідовність, яка статистично виглядає випадковою, але при відомому seed або внутрішньому стані може бути відтворена та передбачена. Такі генератори добре підходять для моделювання, тестів ігор або статистичних задач.
4. Криптографічно стійкі генератори (CSPRNG / DRBG).
   CSPRNG (або стандартизовані DRBG) будуються на криптографічних примітивах і мають властивість непередбачуваності: навіть знаючи частину виходу, атакувальник не може ефективно відновити наступні значення. Саме CSPRNG використовуються для генерації ключів і криптографічних параметрів. У Python практичним інтерфейсом для таких задач є модуль secrets, який спирається на системний CSPRNG.
5. Гібридні схеми.
   На практиці часто застосовується комбінований підхід: ентропія з TRNG або System RNG використовується для ініціалізації та періодичного оновлення стану, а швидку генерацію великих обсягів випадкових даних виконує CSPRNG. Це дозволяє поєднати переваги "справжньої" ентропії та високої продуктивності.

Групи алгоритмів тестування на простоту: детерміновані, імовірнісні, гіпотетичні.

Детермінований тест завжди дає правильну відповідь “просте/складене” без ймовірності помилки (за умови коректних обчислень). Класичний приклад – пробне ділення на всі прості до $\sqrt{n}$: простий у реалізації, але для 1024/2048/4096-бітних чисел практично непридатний через величезний час. Також існують теоретично важливі детерміновані алгоритми на кшталт AKS, який працює за поліноміальний час, але в практиці майже не використовується через великі константи та повільність у порівнянні з імовірнісними методами. Тому в реальних криптосистемах детерміновані методи застосовують або як дуже швидкий попередній фільтр для перевірки подільності на малі прості, або для невеликих чисел.

Імовірнісний тест простоти виконує k ітерацій. На кожній ітерації вибирається випадковий параметр основа і проводиться перевірка, яка для простих чисел завжди виконується. Якщо хоча б на одній ітерації перевірка не пройдена, то число точно складене. Якщо після k ітерацій тест не знайшов ознак складеності, число вважається ймовірно простим, а ймовірність помилки зменшується зі збільшенням k. Саме до цієї групи належать тести Ферма, Соловея–Штрассена, Міллера–Рабіна. На практиці найчастіше використовують Міллера–Рабіна, бо він швидкий і має хорошу теоретичну межу похибки.

Гіпотетичний тест простоти – це метод, у якого правильність або “детермінованість” (тобто гарантія без помилки) залежить від додаткових математичних припущень. Тобто алгоритм може працювати як звичайний тест, але він завжди даватиме правильну відповідь лише за умови істинності певної гіпотези. Якщо припущення не використовувати, то на практиці застосовують імовірнісну версію з контролем похибки. Прикладом є тест Міллера, у теоретичному варіанті він стає детермінованим за додаткових гіпотез, а його практична реалізація у вигляді тесту Міллера–Рабіна є імовірнісною і дозволяє зменшувати ймовірність помилки збільшенням кількості ітерацій k.

Імовірнісні тести:

Тест Ферма

Алгоритм: Тест виконує kітерацій. На кожній ітерації обирається випадкова основа a($2 \le a \le n-2$) і обчислюється значення $a^{n-1} mod n$. Для простого n(за умови $\gcd(a,n) = 1$) має виконуватись рівність $a^{n-1} \equiv 1 (\mathrm{mod} n)$. Якщо для хоча б одного aотримано $a^{n-1} mod n \neq 1$, то n точно складене.
Теоретична похибка: для тесту Ферма немає універсальної межі похибки, оскільки існують числа Кармайкла — складені числа, які для багатьох основ

aповодяться “як прості” й можуть проходити тест. Тому збільшення kне гарантує такого ж надійного зменшення похибки, як у двох наступних тестів. Порівняння з похибкою ПЕОМ: за коректних обчислень на ПК головний ризик помилки тут — не “обчислювальна похибка”, а саме властивість тесту (існування чисел, які його обманюють).

Тест Соловея–Штрассена

Алгоритм: тест виконує k ітерацій. На кожній ітерації обирається випадкова основа a. Далі обчислюються дві величини:

1. $a^{(n-1)/2} mod n$
2. символ Якобі $\left(\frac{a}{n}\right)$, який приймає значення $-1,0,1$.
   Для простих n ці значення узгоджені між собою.

Якщо $\left(\frac{a}{n}\right) = 0$(тобто $\gcd(a,n) \neq 1$) або перевірка узгодженості не виконується, то n складене.

Теоретична похибка: для будь-якого складеного n частка “обманюючих” основ не перевищує $1/2$, тому:

$$P(\text{помилка}) \leq 2^{-k}.$$

Порівняння з похибкою ПЕОМ: оскільки в Python цілочисельні обчислення точні, обчислювальна похибка практично зводиться до апаратних збоїв, а контрольована похибка тесту, яка спадає як 2^{-k} є основним фактором невизначеності.

Тест Міллера–Рабіна

Алгоритм: нехай $n - 1 = 2^s \cdot d$, де d непарне. Тест виконує k ітерацій. На кожній ітерації обирається основа a, обчислюється $x = a^d mod n$, а далі послідовно виконується піднесення до квадрата x за модулем n(до $s - 1$разів). Для простого n отримані значення мають потрапляти в певні дозволені випадки (1 або $-1 mod n$). Якщо для обраної основи a перевірка не проходиться на жодному кроці – n складене.

Теоретична похибка: для будь-якого складеного n частка “обманюючих” основ не перевищує $1/4$, тому:

$$P(\text{помилка}) \leq 4^{-k}.$$

Порівняння з похибкою ПЕОМ: за умови коректної роботи ПЕОМ помилка обчислень у Python int практично відсутня, тому основний внесок у ризик

неправильного висновку дає лише ймовірність помилки самого тесту, яка швидко зменшується зі збільшенням k.

Тест	Теоретична ймовірність похибки(для складеного (n) після (k) ітерацій	Похибка ПЕОМ під час виконання обчислень
Ферма	Немає універсальної межі.	Практично 0
Соловей–Штрассен	$P \leq 2^{-k}$.	Практично 0
Міллер–Рабін	$P \leq 4^{-k}$.	Практично 0

У Python операції над цілими числами виконуються точно, без похибки округлення, тому похибка ПЕОМ у звичайних умовах не є визначальним фактором. Основним джерелом невизначеності є саме ймовірність помилки імовірнісного тесту простоти.

Алгоритми генерації простих чисел

Алгоритм генерації простих “Чебишова”:
Теорема Бертрана–Чебишова гарантує, що для будь-якого $n > 1$ існує хоча б одне просте p таке, що $n < p < 2n$. Тому можна взяти стартове число n і пройтись вперед по проміжку, поки не знайдемо просте.
Алгоритм:

1. Вибираємо випадкове непарне n потрібної бітової довжини, наприклад 2048 біт.
2. Перевіряємо n тестом простоти, зазвичай Міллера–Рабіна.
3. Якщо складене – беремо наступне непарне $n \leftarrow n + 2$ і повторюємо.

алгоритм швидко знаходить ймовірно просте, бо за теоремою про розподіл простих ймовірність того, що число просте приблизно $\approx 1/\ln(n)$, тобто очікувано треба порядку $\ln(n)$перевірок, візьмемо як n для загального розрахунку складності
Складність: $O(k \cdot n4)$, складність одного запуску – $O(k \cdot n^3)$

Алгоритм Маурера
Маурер будує число так, щоб його простоту можна було довести. Для цього він використовує факт, що якщо відомий великий простий дільник q числа $n - 1$, то можна застосувати варіант критерію Поклінгтона і отримати сертифікат простоти.

Алгоритм:

1. Рекурсивно генеруємо менше просте q(приблизно половини потрібної довжини).
2. Підбираємо випадкове Rі формуємо кандидат:
$$n = 2Rq + 1$$
щоб n мав потрібну бітову довжину.
3. Підбираємо випадкову основу a і робимо перевірки, які за умовами критерію гарантують, що n просте.
4. Повертаємо n разом із доказом, який можна перевірити.

Складність: O(n4)

Порівняльний аналіз складності:

Алгоритм “Чебишова” (перебір кандидатів + тест простоти)

- Суть витрат: багаторазово перевіряємо різні кандидати на простоту
- Очікувана кількість кандидатів: $O(\ln\ N)$, а для $N \approx 2^n$це $O(n)$.
- Вартість однієї перевірки (Міллер–Рабін, k ітерацій): одна ітерація ≈ модульне піднесення до степеня $O(n^3)$, тому $O(k \cdot n^3)$.
- Разом:
$O(n) \cdot O(k \cdot n^3) = O(k \cdot n^4)$.

Алгоритм Маурера (просте з доказом)

- Суть витрат: будує кандидат спеціального виду і виконує кілька перевірок на основі модульних експоненціацій + рекурсивно генерує менші прості.
- Домінуюча операція: так само модульні піднесення до степеня $O(n^3)$.
- Очікувана кількість спроб до успіху: також порядку $O(n)$.
- Разом (порядок):
$O(n) \cdot O(n^3) = O(n^4)$.

Моделі клієнтів

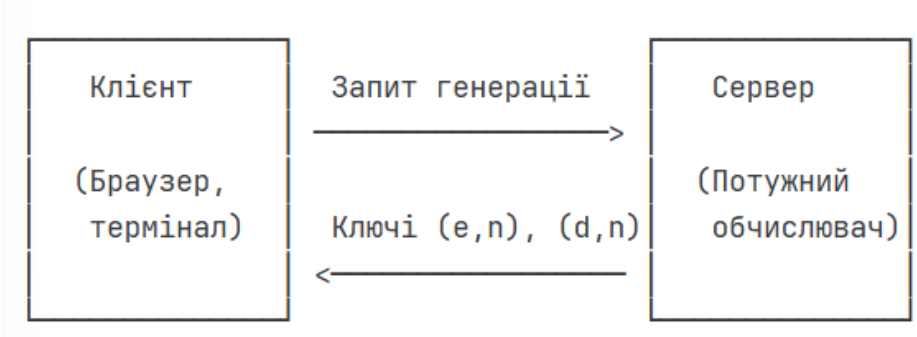
Тонкий клієнт

Концепція:

- Мінімальна обчислювальна потужність на клієнті
- Генерація ключів на віддаленому сервері

- Клієнт тільки використовує готові ключі

Архітектура:



Переваги:

- Мінімальні вимоги до апаратури клієнта
- Швидка робота навіть на слабких пристроях
- Централізоване управління (оновлення, аудит)
- Менші витрати на підтримку парку комп'ютерів

Недоліки:

- Залежність від мережі
- Ризик перехоплення ключів під час передачі
- Сервер знає всі приватні ключі
- Навантаження на сервер при багатьох клієнтах

Застосування:

- Корпоративні системи з високим рівнем довіри
- Термінали в банках
- Каси самообслуговування

Товстий клієнт

Концепція:

- Повна обчислювальна потужність на клієнті
- Локальна генерація всіх ключів
- Автономна робота без сервера

Архітектура:

Переваги:

- Повна автономність (offline робота)
- Приватні ключі ніколи не передаються мережею
- Немає single point of failure
- Кастомізація під потреби користувача

Недоліки:

- Вимоги до апаратури (CPU, RAM)
- Тривалий час генерації (хвилини для RSA-4096)
- Складніше управління та оновлення
- Потенційна небезпека на недовірених пристроях

Застосування:

- Окремі користувачі (самозайняті, фрілансери)
- Високі вимоги до конфіденційності
- Системи без постійного інтернету

Практична частина

Реалізовано мінімальний набір функцій:

- генерація випадкових n-бітних непарних чисел (rand_odd);
- тести простоти: Ферма, Соловея–Штрассена з обчисленням символу Якобі, Міллера–Рабіна;
- генерація простого числа методом перебору кандидатів на основі Міллера–Рабіна (generate_prime_chebyshev).

Для розрядностей 1024/2048/4096 біт виконано вимірювання часу одного запуску кожного тесту для трьох значень k (5, 10, 20). Час визначався як середнє з кількох повторів. Окремо вимірювався час на числі, що є ймовірно простим, та на композиті тієї ж бітової довжини. Примітка: композит у бенчмарку сформовано як число, кратне 3 (для демонстрації швидкого відсіву), тому час для композитів може бути значно меншим за час на «важких» композитах (наприклад, добутках двох великих простих).

Результати:

Контрольні приклади (коректність тестів)

n	Очікувано	Ферма	Соловея–Штрассена	Міллера–Рабіна
2	просте	просте	просте	просте
3	просте	просте	просте	просте
5	просте	просте	просте	просте
17	просте	просте	просте	просте
19	просте	просте	просте	просте
21	складене	складене	складене	складене
221	складене	складене	складене	складене
561	складене	складене	складене	складене

Теоретичні межі похибки для різних k

k	Соловей–Штрассен $\leq 2^{-k}$	Міллер–Рабін $\leq 4^{-k}$	Ферма
5	3.125e-02	9.766e-04	немає універсальної межі
10	9.766e-04	9.537e-07	немає універсальної межі
20	9.537e-07	9.095e-13	немає універсальної межі

Час виконання тестів (мс)

1024-бітні числа

Тест	просте, k=5	просте, k=10	просте, k=20	складене, k=5	складене, k=10	складене, k=20
Ферма	18.752	38.587	78.531	0.001	0.001	0.001
Соловея–Штрассена	25.182	48.766	83.524	0.001	0.002	0.001
Міллера–Рабіна	20.205	40.980	85.398	0.001	0.002	0.001

2048-бітні числа

Тест	просте, k=5	просте, k=10	просте, k=20	складене, k=5	складене, k=10	складене, k=20
Ферма	128.802	262.487	518.495	0.002	0.002	0.002
Соловея–Штрассена	133.162	274.422	537.241	0.001	0.001	0.001
Міллера–Рабіна	129.840	259.329	506.521	0.002	0.002	0.003

4096-бітні числа

Тест	просте, k=5	просте, k=10	просте, k=20	складене, k=5	складене, k=10	складене, k=20
Ферма	914.147	1819.706	3594.262	0.008	0.002	0.003
Соловея–Штрассена	949.802	1867.154	3702.945	0.002	0.002	0.002
Міллера–Рабіна	916.675	1804.913	3632.555	0.002	0.002	0.003

Висновки

У роботі розглянуто підходи до генерації випадкових послідовностей та класи тестів простоти. Реалізовано і протестовано імовірнісні тести Ферма, Соловея–Штрасссена та Міллера–Рабіна, а також генерацію простих чисел методом перебору кандидатів. Експериментальні вимірювання показали, що час виконання тестів зростає зі збільшенням розрядності та кількості ітерацій k, а виявлення простих кандидатів потребує істотно більше часу, ніж відсів очевидних композитів. Для задач практичної криптографії найбільш уживаним є тест Міллера–Рабіна, оскільки він швидкий та має найкращу відому верхню межу похибки серед розглянутих тестів при заданому k.