МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

МАКСИМ'ЮК Анжеліна Іванівна

Алгоритми та програмний засіб для дослідження модулярного експоненціювання в асиметричних криптосистемах / Algorithms and Software for Studying Modular Exponentiation in Asymmetric Cryptosystems

спеціальність: 125 – Кібербезпека та захист інформації

освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконала студентка групи
КБм -21
А. І. Максим'юк

Науковий керівник
д.т.н., професор М.М.Касянчук

Кваліфікаційну роботу
Допущено до захисту:

«____» _____ ___2025 р.

Завідувач кафедри
_____ **В.В.Яцків**

ТЕРНОПІЛЬ – 2025

Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки
Освітній ступінь «магістр»
спеціальність: 125 – Кібербезпека та захист інформації
освітньо-професійна програма – Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ В. В. Яцків
«____» _____ 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
МАКСИМ'ЮК Анжеліні Іванівні
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Алгоритми та програмний засіб для дослідження модулярного експоненціювання в асиметричних криптосистемах / Algorithms and Software for Studying Modular Exponentiation in Asymmetric Cryptosystems
керівник роботи д.т.н., професор М.М. Касянчук
затверджені наказом по університету від 29 листопада 2024 року № 938.
2. Строк подання студентом закінченої кваліфікаційної роботи
5 грудня 2025 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:

- проаналізувати теоретичні основи асиметричних криптосистем, а саме RSA та Ель-Гамаля;
- дослідити методи модулярного експоненціювання, зокрема метод виділення квадрату, метод виділення кубу, векторно-модульний метод, систему залишкових класів та модифіковану форму системи залишкових класів;
- розробити алгоритмічне забезпечення криптосистем на основі оптимізованих методів експоненціювання;
- реалізувати програмний засіб на Python для дослідження та порівняння алгоритмів;
- оцінити продуктивність методів за експериментальними даними та виконати порівняльний аналіз.

5. Перелік графічного матеріалу у роботі:

- структурна схема асиметричного шифрування в криптосистемах;
- схеми процесу шифрування на основі криптоалгоритмів RSA та Ель-Гамаля;
- блок-схеми алгоритмів роботи криптосистем RSA та Ель-Гамаля;

- результати запуску розробленої програми для оптимізованих методів модулярного експоненціювання в криптосистемах RSA та Ель-Гамаля;
- графіки порівняння результатів модульного експоненціювання при різних розрядностях.

6. Консультанти розділів кваліфікаційної роботи

	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання: 29 листопада 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Теоретичні основи асиметричних криптосистем та модулярного експоненціювання	12.2024 р. – 03.2025 р.	
2	Реалізація модулярного експоненціювання в асиметричних криптосистемах	03.2025 р. – 06.2025р.	
3	Розробка програмного забезпечення для дослідження модулярного експоненціювання за допомогою Python	06.2025 р. – 11.2025 р.	

Студентка _____ Максим’юк А.І.
(підпис)

Керівник роботи _____ д.т.н., професор М.М.Касянчук
(підпис)

АНОТАЦІЯ

Максим’юк А.І. Алгоритми та програмний засіб для дослідження модулярного експоненціювання в асиметричних криптосистемах. – Рукопис.

Дослідження на здобуття освітнього ступеня «магістр» за спеціальністю 125 «Кібербезпека та захист інформації», освітньо-професійна програма «Кібербезпека». – Західноукраїнський національний університет, Тернопіль, 2025.

У роботі проведено аналіз методів модулярного експоненціювання, що застосовуються в асиметричних криптосистемах, розроблено програмний засіб, який забезпечує дослідження ефективності алгоритмів модулярного експоненціювання та демонструє їхню практичну придатність для підвищення швидкодії ключових операцій у криптосистемах RSA та Ель-Гамаля.

Ключові слова: КРИПТОГРАФІЯ, АСИМЕТРИЧНІ КРИПТОСИСТЕМИ, МОДУЛЯРНЕ ЕКСПОНЕНЦІЮВАННЯ, АЛГОРИТМИ, PYTHON

ABSTRACT

Maksymiuk A.I. Algorithms and Software for Studying Modular Exponentiation in Asymmetric Cryptosystems. – Manuscript.

Research for obtaining the education level «Master» in specialty 125 «Cybersecurity and Information Protection», educational and professional program «Cybersecurity». – West Ukrainian National University, Ternopil, 2025.

The study analyzes modular exponentiation methods used in asymmetric cryptosystems and develops a software tool that enables the evaluation of the efficiency of modular exponentiation algorithms and demonstrates their practical applicability for improving the performance of key operations in RSA and ElGamal cryptosystems.

Keywords: CRYPTOGRAPHY, ASYMMETRIC CRYPTOSYSTEMS, MODULAR EXPONENTIATION, ALGORITHMS, PYTHON

ЗМІСТ

ВСТУП

Актуальність роботи. Асиметричні криптосистеми є фундаментальною основою сучасної цифрової безпеки, оскільки забезпечують надійні механізми конфіденційності, цілісності та автентифікації даних у відкритих комунікаційних середовищах. Зростання обсягів інформації, поширення хмарних сервісів, розвиток електронного урядування, а також активне використання мережевих протоколів обміну даними зумовлюють потребу у високопродуктивних та стійких криптографічних алгоритмах [1-3].

У криптосистемах RSA та Ель-Гамаля ключовою операцією є модулярне експоненціювання [4]. Саме від швидкості виконання цієї операції залежить практична ефективність шифрування, генерації ключів та цифрового підпису. На сучасному етапі розвитку інформаційних технологій спостерігається тенденція до використання криптографічних ключів значної довжини (2048–4096 біт і більше), що суттєво збільшує навантаження на обчислювальні системи та вимагає впровадження оптимізованих алгоритмів експоненціювання [5, 6].

Попри значну кількість досліджень, питання вибору найбільш ефективного методу модульного експоненціювання залишається відкритим, оскільки різні алгоритми демонструють різну продуктивність залежно від розрядності чисел, структури модуля, архітектури процесора та специфіки конкретної криптосистеми. У практичних реалізаціях все більше значення набувають такі аспекти, як мінімізація проміжних переповнень, зменшення кількості операцій множення та оптимальна організація роботи з великими цілими числами. У зв’язку з цим особливої актуальності набуває аналіз та порівняння алгоритмів модулярного експоненціювання - бінарного, тріарного, векторно-модульного методу, а також способів на основі системи залишкових класів та модифікованої форми системи залишкових класів [7-9]. Дослідження їхніх властивостей дозволяє визначити найбільш ефективні стратегії для застосування у криптосистемах RSA та Ель-Гамаля, які є базовими інструментами сучасної криптографії.

Мета роботи. Метою даної роботи є розробка алгоритмів та програмних засобів для дослідження та порівняльного аналізу алгоритмів модулярного експоненціювання в асиметричних криптосистемах RSA та Ель-Гамаля, визначення їх ефективності за критеріями часової та обчислювальної складності, а також формування практичних рекомендацій щодо їх застосування.

Для досягнення даної мети ставились наступні **завдання**:

- дослідити основні принципи та класифікацію асиметричних криптосистем;
- проаналізувати сучасні криптосистеми RSA та Ель-Гамаля та визначити їх особливості;
- розглянути математичні основи модульного множення та експоненціювання у контексті криптографії;
- реалізувати алгоритмічні основи криптосистем RSA та Ель-Гамаля засобами програмування;
- розробити програмний засіб на мові Python для дослідження алгоритмів модульного експоненціювання;
- реалізувати та протестувати чотири методи модульного експоненціювання: векторно-модулярний метод, метод виділення квадрату, метод виділення кубу та метод прямого піднесення до степеня;
- провести порівняльний аналіз ефективності реалізованих методів за часовою та обчислювальною складністю.

Об’єкт дослідження. Процес модулярного експоненціювання для його застосування в асиметричних криптосистемах.

Предмет дослідження. Алгоритми модулярного експоненціювання, а саме метод виділення квадрату, виділення кубу, векторно-модулярний, використання системи залишкових класів та модифікованої форми систем залишкових класів тощо та їх програмна реалізація і порівняння ефективності.

Наукова новизна одержаних результатів.

1. Вдосконалено підхід до оцінювання продуктивності алгоритмів модулярного експоненціювання шляхом реалізації та порівняння виділення квадрату, виділення кубу, векторно-модулярний, використання системи

залишкових класів та модифікованої форми систем залишкових класів що забезпечило можливість формування узагальнених критеріїв ефективності. реалізації програмного засобу на основі мови програмування Python;

2. На основі експериментальних результатів показано, що застосування оптимізованих методів модульного експоненціювання забезпечує помітне скорочення часу виконання операцій шифрування та розшифрування порівняно з базовими підходами.

3. **Практичне значення отриманих результатів.**

Розроблено програмний засіб для дослідження операцій модулярного експоненціювання в асиметричних криптосистемах RSA та Ель-Гамаля, що дозволило виконати експериментальний аналіз ефективності різних алгоритмів піднесення до степеня за великим модулем.

Публікації та апробація КР.

1. Максим’юк А., Касянчук М. Алгоритми та програмний засіб для дослідження модулярного експоненціювання в асиметричних криптосистемах. Безпека інформаційних технологій (ITSec-2025): Матеріали XIV Міжнар. наук.-техн. конф., 22–24 травня 2025. – Тернопіль (Україна), 2025. – С. 126 [10].

2. Максим’юк А., Касянчук М. Дослідження алгоритмів модулярного експоненціювання в асиметричних криптосистемах. Кібербезпека державних інституцій та подолання кризових станів : матеріали IV Міжнар. наук.-практ. конф. в 2 т. (Київ – Прага – Таллінн – Тернопіль), 18 листоп. 2025 р. / ІСЗЗІ КПІ ім. Ігоря Сікорського. Київ, 2025. Т. 1. С. 465–466 [11].

1 ТЕОРЕТИЧНІ ОСНОВИ АСИМЕТРИЧНИХ КРИПТОСИСТЕМ ТА МОДУЛЯРНОГО ЕКСПОНЕНЦІЮВАННЯ

1.1 Основні принципи та класифікація асиметричних криптосистем

Криптографічна система з відкритим ключем (або асиметрична криптосистема, асиметричне шифрування) – це система шифрування, при якій відкритий ключ $K1$ передається по відкритому (незахищеному) каналу зв'язку та використовується для шифрування повідомлень (рисунок 1.1). Для розшифрування повідомлень використовується секретний (або приватний) ключ $K2$. Наявність двох ключів – відкритого та закритого – й робить цю систему асиметричною. Відкритий ключ розсилається всім, хто бажає відправляти повідомлення адресату, а приватний ключ зберігається адресатом і не повинен нікому відправлятись. Навіть якщо знати відкритий ключ та все відправлене розшифроване повідомлення, неможливо знайти приватний ключ.

Рисунок 1.1 – Асиметричне шифрування

Серед характерних особливостей асиметричних криптосистем є:

1) відкритий ключ $K1$ та криптограма C, які можуть пересилатись незахищеними каналами зв’язку;

2) алгоритми шифрування та дешифрування є відкритими.

Без знання секретного ключа відновити вихідне повідомлення практично неможливо, оскільки обчислення секретного ключа з відкритого є математично складною задачею. Саме це забезпечує високий рівень безпеки таких систем навіть при передачі відкритого ключа незахищеними каналами зв’язку.

Крім того, асиметричні криптосистеми забезпечують підтримку таких важливих криптографічних властивостей, як цифровий підпис та аутентифікація – за допомогою приватного ключа підписують дані, а відкритим ключем інші можуть перевірити, що підпис справді належить тому, хто має приватний ключ [12, 13].

Перевагами асиметричних криптосистем є:

– відсутність необхідної попередньої передачі особистого ключа по надійному каналу (як у симетричних криптосистемах);

– наявність тільки одного секретного ключа, який зберігається тільки у одної із сторін;

– відсутність необхідності зміни ключа протягом досить довгого часу тощо.

Попри це, існують також і недоліки:

- більша довжина ключа, щоб досягнути криптографічну безпеку;

- значно вища обчислювальна складність порівняно з симетричними алгоритмами [14];

- хоча повідомлення шифруються надійно, самі сторони «засвічуються» фактом передачі, що може спричинити атаку;

- асиметричні системи вимагають значних обчислювальних ресурсів, тому на практиці їх використовують разом з іншими алгоритмами.

Асиметричні криптографічні алгоритми зазвичай класифікують за типом математичної задачі, на якій ґрунтується їх стійкість, а також за основним призначенням – шифрування, обмін ключами або цифровий підпис. У таблиці 1.1 подано узагальнену класифікацію найбільш поширених асиметричних криптосистем, що використовуються на практиці.

Таблиця 1.1 – Класифікація асиметричних криптосистем

Тип асиметричної криптосистеми	Характеристика
RSA	Базується на складності факторизації великих чисел; використовується для шифрування, підпису та обміну ключами.
Ель-Гамаля (ElGamal)	Побудована на задачі дискретного логарифмування; застосовується для шифрування та цифрового підпису.
Diffie–Hellman (DH)	Протокол, що дозволяє двом сторонам узгодити спільний секрет без попереднього обміну ключами.
DSA	Алгоритм цифрового підпису на основі дискретних логарифмів, стандартизований державними установами США.
ECC	Використовує еліптичні криві, забезпечуючи рівень безпеки RSA при значно менших розмірах ключів.
Ed25519	Сучасний алгоритм цифрового підпису на еліптичних кривих, оптимізований для високої швидкодії та безпеки.
X25519	Популярний алгоритм обміну ключами на кривій Монгомері, застосовується у TLS, SSH та сучасних протоколах.

Криптографічні системи з відкритим ключем на сьогодні широко застосовуються в різних мережевих протоколах, зокрема, в протоколах TLS і його попереднику SSL (що лежать в основі HTTPS), в SSH. Також використовується в PGP, S / MIME. Асиметричні алгоритми використовуються для розповсюдження ключів швидших симетричних алгоритмів.

Отже, розуміння принципів побудови асиметричних криптосистем створює теоретичну основу для подальшого аналізу алгоритмів RSA та Ель-Гамаля й дослідження ефективності операцій модульного експоненціювання.

1.2 Огляд сучасних асиметричних криптосистем

Протягом останніх десятиліть асиметричні криптосистеми стали базою безпечного обміну даними в комп’ютерних мережах, забезпечуючи

конфіденційність, доступність та цілісність інформації. Їх також використовують для створення цифрових підписів і протоколів захисту, таких як TLS, SSH, PGP, S/MIME тощо [15-17].

Безпека таких систем ґрунтується на складних математичних задачах, які складно обчислити в оберненому напрямку, для прикладу, обчислення дискретного логарифма або факторизація великих чисел.

Однією із найпоширеніших і найвідоміших сучасних асиметричних криптосистем є RSA. Безпека RSA побудована на складності факторизації великого числа, яке є добутком двох великих простих чисел. Цей алгоритм забезпечує дуже високий рівень безпеки, однак вимагає значних обчислювальних ресурсів, особливо із зростанням довжини ключа.

Алгоритм ElGamal належить до іншого класу криптосистем. Його побудовано на задачі дискретного логарифму, і тому варто використовувати для шифрування/дешифрування невеликого обсягу даних. Алгоритм демонструє гомоморфні властивості, які дозволяють виконувати деякі обчислення над зашифрованими даними без їхнього розшифрування. Проте він поступається RSA у швидкодії при роботі з великими обсягами інформації.

Продовженням ідей дискретного логарифма стала криптографія на еліптичних кривих (ECC). Алгоритми цієї категорії, наприклад ECDSA та ECDH, забезпечують аналогічний рівень безпеки при значно коротших ключах. Завдяки цьому вони широко застосовуються в мобільних пристроях, смарт-картах і системах з обмеженими ресурсами [18]. Іншими варіантами є алгоритми на зразок Paillier, який має адитивні гомоморфні властивості, та Rabin, що подібний до RSA, але використовує квадратні залишки при шифруванні.

Незважаючи на високу надійність класичних асиметричних криптосистем, їхнім головним викликом стає поява квантових комп’ютерів. Завдяки алгоритму Шора вони здатні ефективно вирішувати задачі факторизації та дискретного логарифма. Це ставить під загрозу стійкість таких систем, як RSA, Ель-Гамаль чи ECC, до можливих атак.

У зв’язку з цим у сучасній криптографії активно розвивається новий напрям – постквантова криптографія (Post-Quantum Cryptography). Головна мета

– створення алгоритмів, здатних протистояти квантовим обчислювальним методам. Серед найбільш перспективних напрямів відзначають:

- решіткові криптосистеми (lattice-based), зокрема NTRU та Kyber, які ґрунтуються на задачах типу Learning With Errors і показують стійкість до квантових атак;

- хеш-підписи, як SPHINCS+, які забезпечують автентичність без залучення складних числових структур;

- мультивимірні алгоритми (multivariate), зокрема Rainbow, що використовують системи багаточленів над кінцевими полями.

Таким чином, сучасна асиметрична криптографія включає великий спектр методів – від класичних алгоритмів, перевірених десятиліттями, до новаторських постквантових рішень, які лише починають находити прикладне використання. Всі вони мають спільну мету: забезпечити безпеч

1.2.1. Криптосистема RSA

Даний алгоритм у 1978 році запропонували Р.Райвест, А.Шамір і А.Адлеман. Його назва виникла з перших трьох літер прізвищ авторів (**R**ivest, **S**hamir, **A**dleman). Криптосистема стала першим шифром, розробленим на принципах асиметричного шифрування. А зараз є однією із найпоширеніших асиметричних систем шифрування, що чудово забезпечує конфіденційність, доступність та цілісніть даних (рисунок 1.2).

Математична стійкість криптосистеми RSA ґрунтується на обчислювальній складності проблеми факторизації цілих чисел. Зокрема, розкладання дуже великого натурального числа, яке є добутком двох простих множників, є задачею, що вважається практично нерозв'язною. Ця складність унеможливлює визначення таємного ключа на основі загальнодоступного відкритого ключа. Схема криптоалгоритму RSA представлена на рисунку 1.3.

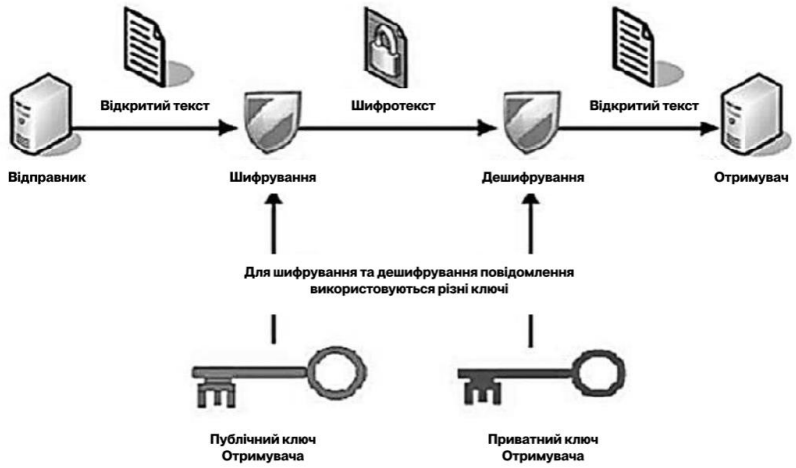

Рисунок 1.2 – Криптоалгоритм RSA

Рисунок 1.3 – Схема криптоалгоритму RSA

Нехай користувач А хоче передати повідомлення користувачу В. У криптосистемі одержувач, тобто користувач В, повинен сформувати ключі.

Процедура шифрування та розшифрування в даній криптосистемі така:

1) вибір двох простих довільних числа P і Q;

2) обчисення значення модуля $N = P*Q$ (якщо довжини P і Q складають не менше 100 десяткових знаків, то розкласти модуль N на множники за реальний час не є можливим);

3) обчислення функції Ейлера за формулою $\varphi(N) = (P-1)(Q-1) = P * Q - P - Q + 1$, та випадковим чином вибирає значення відкритого ключа K так, щоб виконувалися умови $1 < K \leq \varphi(N)$ та НСД$\big(K, \varphi(N)\big) = 1$;

4) обчислення значення середнього ключа k, використовуючи розширений алгоритм Евкліда $k \equiv K^{-1}(mod\, \varphi(N))$;

5) формування ключів: користувач В відправляє користувачу А пару чисел N та K по незахищеному каналу.

Якщо користувач А хоче передати користувачу В повідомлення М, то він має виконати певні кроки. Для початку він за допомогою узгодженого протоколу перетворення, відомого як доповняльні схеми або таблиця перетворення, перетворює вихідний відкритий текст на блоки, кожен з яких можна представити в виді числа $M_i = 0,1,2, \dots, N-1$. Опісля він обчислює зашифрований текст C_i, використовуючи відкритий ключ користувача В - K, за допомогою рівнянняг:

$$C_i = M_i^k mod\ N \tag{1.1}$$

Це може бути зроблено досить швидко, навіть у випадку, коли текст перевищує 500-бітну розрядність.

Користувач В розшифровує прийняту криптограму з використанням секретного ключа за формулою:.

Розшифрування відбувається наступним чином:

$$M_i = C_i^k mod\ N \tag{1.2}$$

Відповідно при виконанні даної операції відбувається відновлення вихідного повідомлення:

$$(M_i^K)^k \equiv M_i^{Kk} mod\ N \tag{1.3}$$

З умови

$$Kk \equiv 1(mod\ \varphi(N)) \tag{1.4}$$

випливає твердження, що $Kk \equiv k_0\varphi(N) + 1$ для деякого цілого k_0, отже:

$$M_i^{Kk} \equiv M_i^{k_0\varphi(N)+1} mod\ N \tag{1.5}$$

Згідно з теоремою Ейлера:

$$M_i^{\varphi(N)} \equiv 1\ mod\ N \tag{1.6}$$

тому

$$M_i^{k_0\varphi(N)+1} \equiv M_i\ mod\ N \tag{1.7}$$

Для шифрування необхідно знати пару чисел *K, N,* а для дешифрування – *k, N*. Перша пара - відкритий ключ, друга - закритий. Знаючи відкритий ключ, можна обчислити значення закритого ключа. Необхідною проміжною дією цього перетворення є знаходження множників *P* і *Q*, для чого потрібно розкласти *N* на множники - ця процедура займає дуже багато часу.

Криптостійкість алгоритму ґрунтується на складності розкладання великих чисел на прості множники, а також на обчислювальній складності споріднених задач теорії чисел. Для ефективної реалізації алгоритму RSA часто застосовують Китайську теорему про залишки, яка дозволяє значно пришвидшити процес розшифрування.

1.2.2. Криптосистема Ель-Гамаля

Схема шифрування Ель-Гамаля була запропонована в 1985 році Тахером Ель-Гамалем і може використовуватися як в режимі шифрування даних, так і в

режимі електронного цифрового підпису. Надійність алгоритму базується на складності обчислення дискретних логарифмів.

Даний алгоритм відноситься до класу ймовірнісних алгоритмів криптографічного захисту інформації. Особливість алгоритмів даного класу полягає у тому, що повідомленню *M* відповідає не один криптотекст *C*, а деяка родина криптотекстів C_M , причому кожен член $C \in C_M$ цієї родини вибирається з певною ймовірністю. Іншими словами, для кожного повідомлення *X* результат роботи алгоритму $E(X)$ є випадковою величиною, розподіленою на множині Cx . Щоб дешифрування було однозначним для будь-якої пари різних повідомлень C_1 та C_2 відповідні їм множини криптотекстів C_{X_1} та C_{X_2} не повинні перетинатися. Більше того, має бути ефективний алгоритм дешифрування, який використовує таємний ключ і за будь-яким $C \in Cx$ визначає *X*.

Таку криптосистему вважають надійною, якщо для будь-якої пари повідомлень X_1 та X_2 однакової довжини *l*, випадкові величини $E(X_1)$ та $E(X_2)$ неможливо відрізнити одну від одної ніяким ймовірнісним поліноміальним алгоритмом із ймовірністю, вищою за 1/*l*. Тобто лише з незначною ймовірністю за двома криптотекстами можна визначити, відповідають вони одному й тому ж повідомленню чи різним. Такий рівень надійності, в принципі, недосяжний для детермінованих систем, у випадку яких пара різних криптотекстів C_1 та C_2 завжди відповідає різним відкритим текстам X_1 та X_2.

Будучи алгоритмом шифрування з відкритим ключем, алгоритм Ель-Гамаля залежить як від закритого, так і від секретного ключів, які він генерує для шифрування даних. Таким чином, основною проблемою стає захист цих ключів і керування ними. Цю безпеку можна досягти, використовуючи певні апаратні модулі з програмним забезпеченням для керування ключами. Крім того, довжина ключів, які використовуються для шифрування даних, є настільки важливою для рівня безпеки, якого тільки можна досягти; отже, 64-бітний симетричний ключ зберігатиме дані в безпеці протягом тривалого часу. Алгоритм в основному залежить від простого числа, значення якого залежить від кількості бітів, що використовуються для генерації цього простого числа. Це також залежить від

закритого ключа, вхідного повідомлення та (розрізаної довжини) повідомлення блоку [19].

Розглянемо саму роботу алгоритму:

1) спочатку здійснюється генерація ключів (відкритий ключ та секретний ключ): вибирають деяке велике просте число p і велике ціле число q, причому $q < p$. Числа p і q можуть бути поширені серед групи користувачів; p та q є відкритим ключем;

2) вибирають число a з умовою $1 < a < p$. Число a є таємним ключем;

3) обчислюють $h = q^a mod\ p$. Число h є відкритим ключем.

Якщо користувач А хоче передати користувачу В повідомлення М, то він має виконати такі кроки:

1) користувач А має вибрати випадкове ціле число r з умовою $1 \leq r \leq p - 1$. Числа r та $(p-1)$ повинні бути взаємно прості, тобто їхнє НСД дорівнює 1;

2) зашифрувати повідомлення X за допомогою наступних формул: $c_1 = q^r mod\ p$ та $c_2 = h^r X\ mod\ p$, де X – відкритий текст та $X < p$. Пара чисел c_1 та c_2 є шифротекстом. Варто відмітити, що довжина шифротексту вдвічі більша довжини вихідного відкритого тексту X.

Користувач В, маючи свій таємний ключ а, розшифровує повідомлення за формулою :

$$X = c_2 (c_1^{-1})^a mod\ p. \tag{1.8}$$

При цьому дуже легко перевірити, що $(c_1^{-1})^a mod\ p = q^{ra} mod\ p$ і тому $c_2 (c_1^{-1})^a mod\ p = (h^r X) q^{ra} mod\ p = (q^{ra} X) q^{ra} mod\ p = X\ mod\ p$

Для практичних обчислень більше підходить наступна формула:

$$X = c_2 (c_1^{-1})^a mod\ p = c_2 (c_1)^{p-1-a} mod\ p \tag{1.9}$$

Схема шифрування на основі криптоалгоритму Ель-Гамаля представлена на рисунку 1.4.

Оскільки у схему Ель-Гамаля вводиться випадкова величина *r*, то цей шифр можна назвати шифром багатозначної заміни. У зв'язку з випадковістю вибору числа *r* таку схему називають ще схемою ймовірнісного шифрування. Ймовірнісний характер шифрування є перевагою для криптосистеми Ель-Гамаля, тому що у таких схемах спостерігається більша стійкість у порівнянні зі схемами із певним процесом шифрування.

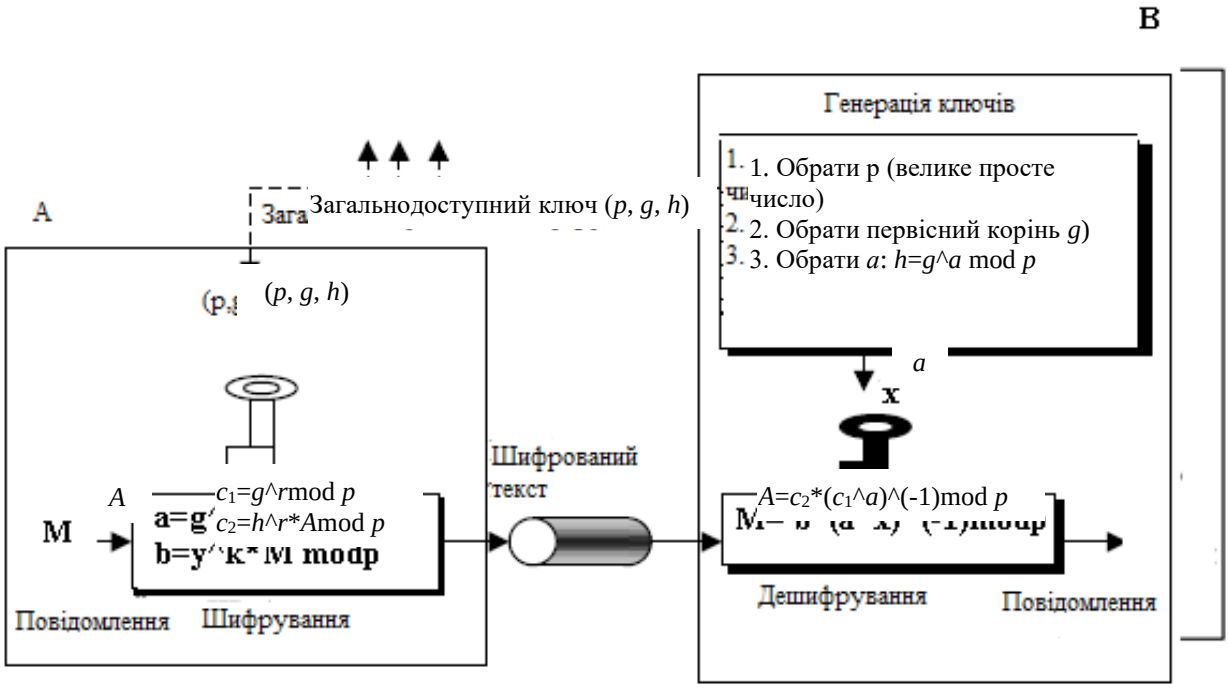

Рисунок 1.4 – Схема шифрування на основі криптоалгоритму Ель-Гамаля

Безпека алгоритму Ель-Гамаля полягає в складності обчислення дискретних логарифмів за великим простим модулем. Розв'язати цю задачу з логарифмуванням важко. Перевагою алгоритму Ель-Гамаля є генерація ключів за допомогою дискретних логарифмів. Метод шифрування та дешифрування використовує випадкові значення для кожного повідомлення, що робить результати шифрування різними для одного й того ж вихідного повідомлення. Недоліком цього алгоритму є те, що він потребує великої довжини відкритого тексту і вимагає процесора, здатного виконувати масштабні обчислення для масивних логарифмічних обчислень. Це дослідження намагається проаналізувати, який алгоритм краще виконує процес шифрування та дешифрування.

1.3 Принципи модулярного множення та застосування

Операція модулярного множення багаторозрядних чисел є однією з найважливіших в асиметричних системах захисту інформації, зокрема RSA та Ель-Гамаля. Саме результативність виконання цих дій визначає продуктивність процесів шифрування і розшифрування даних [20-23].

Математично ця операція визначається як знаходження *C* у виразі:

$$\mathrm{C} \equiv M^E (mod\ N) \tag{1.10}$$

де M - це основа повідомлення або його частина, E - експонента (ключ), а N - модуль. У реальних криптографічних застосунках ці три числа є надзвичайно великими, та, для прикладу, можуть мати довжину 2048, 3072 або 4096 біт.

Пряме обчислення M^E з подальшим взяттям залишку від ділення на N є абсолютно неможливим з обчислювальної точки зору. Проміжний результат M^E містив би таку кількість біт, що значно перевищує обсяги пам'яті сучасних комп'ютерів. Тому наукова інженерна спільнота розробила низку спеціалізованих алгоритмів, що дозволяють обчислювати кінцевий результат *C*, жодного разу не виходячи за межі величини модуля N (або його квадрату) на проміжних кроках.

Існує декілька підходів до реалізації модульного експоненціювання, які відрізняються продуктивністю та складністю реалізації [24, 25]. Серед них можна виокремити наступні.

1) класичні бінарні методи ("Square-and-Multiply"). Найбільш фундаментальною та інтуїтивно зрозумілою родиною алгоритмів для цієї задачі є бінарні методи, також відомі у світовій літературі як "Square-and-Multiply" (Піднеси-до-квадрату-та-помнож). Його було запропоновано ще в 1960-х роках у контексті обчислювальної математики, а пізніше адаптовано для криптографії в алгоритмах RSA та Diffie-Hellman. Принцип методу полягає у представленні показника степеня *E* у двійковій системі. Для кожного біта виконується операція

піднесення поточного результату до квадрату, а якщо біт дорівнює одиниці - додатково виконується множення на основу *M*. Метод використовується у більшості реалізацій RSA та Ель-Гамаля через простоту й відносну швидкодію;

2) метод прямого піднесення до степеня. Це найпростіший спосіб обчислення виразу, при якому число поступово множиться на себе в циклі. На кожному етапі результат одразу береться за модулем, щоб уникнути переповнення числового діапазону. Метод зручний у реалізації, проте потребує великої кількості множень, отже, менше ефективний при великих показниках. Також, попри свою простоту, він має досить обмежене практичне застосування в сучасній криптографії. У сучасних криптографічних протоколах, зокрема RSA, Ель-Гамаля або Diffie–Hellman, цей метод поступається місцем більш ефективним алгоритмам зі зниженою обчислювальною складністю;

3) метод зліва направо (Left-to-Right Binary Exponentiation). Метод зліва направо – це варіація класичного бінарного алгоритму, де обробка бітів показника виконується від старших до молодших. Він з'явився у 1980-х роках у контексті оптимізації обчислень для апаратних криптомодулів. Перевага підходу полягає в тому, що часткові результати одразу використовуються для подальших множень, що знижує кількість операцій та спрощує конвеєрну обробку. Метод активно використовується в апаратних реалізаціях криптографічних процесорів, зокрема для RSA-операцій з великими ключами;

4) метод справа наліво (Right-to-Left Binary Exponentiation). Метод справа наліво є альтернативним варіантом класичного алгоритму, запропонованим для зменшення обсягу проміжних обчислень і пам'яті. Уперше був детально описаний у літературі з чисельних методів у 1970-х роках. На відміну від підходу «зліва направо», цей метод обробляє біти показника від наймолодшого до найстаршого. Цей метод часто використовується у бібліотеках відкритого коду (OpenSSL, GMP), оскільки спрощує оптимізацію під конкретну архітектуру процесора;

5) метод Монтгомері. Метод Монтгомері, розроблений Пітером Монтгомері у 1985 році, є одним із найефективніших алгоритмів для обчислення

модульних множень у великих числах. Його ключова ідея полягає в тому, щоб уникнути операцій ділення на модуль, які є обчислювально дорогими. Замість цього всі обчислення виконуються у спеціальному “просторі Монтгомері”, де ділення замінюється зсувами та додаванням. Завдяки відсутності ділення метод Монтгомері суттєво пришвидшує операції в криптографічних бібліотеках і є стандартом де-факто для RSA, ECC і TLS-протоколів;

6) китайська теорема про залишки. Китайська теорема про залишки походить ще з III століття н.е. і вперше була описана китайським математиком Сунь-Цзи. У криптографії CRT використовується для оптимізації розрахунків у системах, де модуль є добутком кількох простих чисел. Ідея полягає в тому, щоб розбити складне обчислення за великим модулем на два або більше простіших обчислення за меншими модулями, а потім об’єднати результати. Цей метод забезпечує прискорення обчислень приблизно у 3–4 рази при розшифруванні RSA-повідомлень.

Узагальнюючи викладене, можна зробити висновок, що операції модулярного множення та модулярного експоненціювання є центральними обчислювальними процедурами в асиметричних криптографічних алгоритмах. Саме від ефективності реалізації цих операцій залежить швидкодія шифрування, розшифрування, формування цифрових підписів та генерування ключових параметрів [26]. Тому оптимізація модульних обчислень є одним із ключових напрямів розвитку сучасних криптографічних систем.

Еволюція апаратного забезпечення – поява багатоядерних процесорів, графічних прискорювачів (GPU, FPGA) та SIMD-інструкцій – зумовила появу нових, більш ефективних методів обчислення степеневих виразів за модулем [27]. До таких підходів належать векторно-модульні алгоритми, паралельне експоненціювання, оптимізація множення у системі залишкових класів (RNS) та використання китайської теореми про залишки (CRT). Застосування цих методів дозволяє не лише скоротити час виконання базових операцій, а й рівномірно розподіляти навантаження між апаратними ресурсами, що є критично важливим для систем із високою інтенсивністю криптографічних запитів.Окрема увага приділяється аспектам захисту від атак побічних каналів. Деякі алгоритми,

зокрема метод Монтгомері, мають спеціальні модифікації для забезпечення постійного часу виконання операцій, що значно ускладнює аналіз споживання енергії або часових затримок з боку потенційного порушника. Такі контрзаходи є обов’язковими під час розроблення криптографічних модулів для смарт-карт, мобільних пристроїв та вбудованих систем.Таким чином, різні підходи до реалізації модульного множення – від класичних бінарних методів до алгоритму Монтгомері та технік, заснованих на CRT, – формують комплекс інструментів, які забезпечують необхідний рівень продуктивності та криптостійкості.

У поєднанні вони дозволяють ефективно реалізувати асиметричні криптосистеми, такі як RSA, Ель-Гамаля, DSA, протоколи Diffie–Hellman, а також сучасні протоколи захисту інформації у мережевих комунікаціях (TLS/SSL). Зважаючи на постійне зростання вимог до безпеки та швидкодії, подальші дослідження в галузі модульних обчислень матимуть суттєвий вплив на розвиток криптографії та інформаційної безпеки загалом.

2 РЕАЛІЗАЦІЯ МОДУЛЯРНОГО ЕКСПОНЕНЦІЮВАННЯ В АСИМЕТРИЧНИХ КРИПТОСИСТЕМ

2.1. Модулярне експоненціювання

Модулярне експоненціювання є однією з фундаментальних операцій у криптографії та обчислювальній математиці. Його сутність полягає у визначенні остачі від ділення числа a^b на модуль c, тобто у виконанні виразу:

$$a^b \bmod c \tag{2.1}$$

Цей тип обчислень лежить в основі асиметричних криптосистем, цифрових підписів і протоколів шифрування, де потрібно виконувати великі обсяги степеневих операцій з великими числами в обмеженому числовому просторі.

Основна мета застосування модульної арифметики – зберегти числову стабільність і запобігти переповненню [28-30]. Замість роботи з надвеликими степенями, кожен проміжний результат редукується за модулем c, що гарантує, що всі подальші обчислення відбуваються в межах діапазону $[0, c-1]$. Таким чином, модульне експоненціювання є основою алгоритмів, які забезпечують надійність, відтворюваність і криптографічну стійкість сучасних інформаційних систем.

Якщо a і p є взаємно простими, то відповідно до малої теореми Ферма:

$$a^{p-1} \equiv 1 \pmod{p}, \tag{2.2}$$

а в загальному вигляді справджується співвідношення Ейлера:

$$a^{\varphi(p)} \equiv 1 \pmod{p}, \tag{2.3}$$

де $\varphi(p)$ – функція Ейлера, що визначає кількість натуральних чисел, менших за p і взаємно простих із ним.

Звідси випливає, що будь-яке піднесення до степеня можна звести до залишку від ділення показника b на $\varphi(p)$:

$$a^b \bmod p = a^{b \bmod \varphi(p)} \bmod p.$$

Ця властивість дозволяє значно зменшити обсяг обчислень, особливо при роботі з великими степенями в межах криптографічних задач.

Щоб реалізувати обчислення ефективніше, показник степеня b подається у двійковій системі числення:

$$b = (b_{n_0-1} b_{n_0-2} \dots b_1 b_0)_2,$$

де кожен біт b_i набуває значення 0 або 1. Тоді степінь можна подати у вигляді:

$$b = \sum_{i=0}^{n_0-1} b_i \cdot 2^i,$$

а вираз для обчислення експоненти матиме вигляд:

$$a^b \bmod p = \prod_{i=0}^{n_0-1} (a^{2^i})^{b_i} \bmod p. \qquad (2.4)$$

Така форма зручна для алгоритмічної реалізації, оскільки дає змогу використовувати лише операції піднесення до квадрату й вибіркового множення. На цьому принципі базуються всі основні методи експоненціювання, що розрізняються структурою виконання цих операцій.

Розглянуті математичні засади становлять основу для побудови різних алгоритмів модульного експоненціювання. Кожен із них пропонує власний спосіб зменшення кількості обчислень і підвищення ефективності. У подальших

підрозділах детально розглянуто їхні особливості, алгоритмічні структури та ефективність у практичних криптографічних системах.

2.1.1. Метод виділення квадрату

Метод піднесення до квадрату, також відомий як алгоритм square-and-multiply, є одним із найефективніших способів обчислення модульного експоненціювання., що значно скорочує кількість множень порівняно з традиційним прямим методом.

Основна ідея полягає у тому, що під час піднесення числа a до степеня x усі проміжні результати зводяться до квадрату, а множення виконуються лише тоді, коли відповідний біт показника степеня у двійковому представленні дорівнює одиниці. Таким чином, операція експоненціювання перетворюється на послідовність простих дій – піднесення до квадрату та умовного множення. Це дозволяє суттєво знизити обчислювальні витрати, особливо при великих показниках степеня.

Нехай необхідно обчислити:

$$N = a^x \bmod p.$$

Показник степеня xможна подати у двійковій формі:

$$x = (x_{n_0-1}x_{n_0-2}\dots x_1x_0)_2.$$

Тоді значення $a^x mod p$ можна визначити за формулою:

$$N = a^x \bmod p = \left(a^{x-2x_1}a_1^{x_1-2x_2}\dots a_i^{x_i-2x_{i+1}}\dots\right)\bmod p = \left(\prod_{i=0}^{[log_2 x]} a_i^{x_i-2x_{i+1}}\right)\bmod p, \qquad (2.5)$$

де $a_0=a$, $x_0=x$, $a_i = a_{i-1}^2 \bmod p$;

$$x_i = \left[\frac{x_{i-1}}{2}\right];$$

Фактично, на кожному кроці алгоритм зводить поточне значення до квадрату, після чого, якщо біт експоненти дорівнює 1, виконує додаткове множення на початкову основу.

Метод виділення квадрату можна подати у вигляді такого алгоритму:

1) подати експоненту x у двійковій системі числення;

2) встановити початкове значення $N = 1$;

3) для кожного біта x_i, починаючи з найстаршого, виконати:

- піднесення поточного результату до квадрату;
- якщо $x_i = 1$, виконати додаткове множення на основу a;
- після кожної операції виконати модульну редукцію за p.

4) після проходження всіх бітів отримане N буде остаточним результатом.

Завдяки тому, що кількість ітерацій дорівнює кількості бітів у показнику степеня, а множення виконуються лише для одиничних бітів, метод характеризується високою ефективністю. Його алгоритмічна складність становить $O(\log_2 x)$, що є істотним покращенням порівняно з прямим методом, де складність пропорційна самому значенню x.

Розглянемо приклад обчислення $23^{19} \bmod 29$ за методом виділення квадрату.

Показник степеня 19 у двійковому вигляді дорівнює $(10011)_2$.

Алгоритм діє так:

1) початкове значення $N = 1$, $a = 23$;

2) зчитуємо біти зліва направо:

- перший біт 1 $\rightarrow N = N^2 \times 23 \bmod 29 = 23$;
- другий біт 0 $\rightarrow N = 23^2 \bmod 29 = 7$;
- третій біт 0 $\rightarrow N = 7^2 \bmod 29 = 20$;
- четвертий біт 1 $\rightarrow N = 20^2 \times 23 \bmod 29 = 25$.

3) отримуємо кінцевий результат $N = 25$.

Таким чином, $23^{19} \bmod 29 = 25$.

Цей приклад демонструє, як метод дозволяє уникати надлишкових множень, зберігаючи при цьому точність результату навіть для великих чисел. Такий підхід дозволяє ефективно оперувати з великими степенями, не виходячи за межі обчислювального простору модуля.

Метод широко використовується у ключових асиметричних криптосистемах, таких як RSA, Ель-Гамаля, Діффі-Гелмана, DSA, та Paillier; в апаратних криптографічних модулях (HSM, TPM), де необхідна висока швидкість модульних операцій. Також у ресурсозберігаючих системах із низьким енергоспоживанням, наприклад, в IoT-пристроях; при оптимізації обчислень на GPU чи FPGA, де квадратування може проводитися як паралельний процес.

2.1.2. Метод піднесення до кубу

Метод піднесення до кубу, відомий як cube-and-multiply algorithm, є вдосконаленим варіантом методу піднесення до квадрату, який оптимізує процес модульного експоненціювання. Його основна ідея полягає у використанні трійкового представлення експоненти, що дозволяє скоротити кількість арифметичних операцій під час виконання модульного експоненціювання.

Цей підхід особливо ефективний у випадках, коли структура показника степеня дає змогу мінімізувати кількість множень і піднесень до степеня, що є критично важливим для високопродуктивних криптографічних систем.

Як і в попередньому методі, потрібно обчислити значення:

$$N = a^x \bmod p$$

На відміну від двійкового алгоритму, тут експонента x представляється у трійковій системі числення:

$$x = (x_{n_0-1} x_{n_0-2} \dots x_1 x_0)_3,$$

де кожен коефіцієнт $x_i \in \{0,1,2\}$. Тоді піднесення до степеня можна записати у вигляді:

$$N = a^x \bmod p = \left(a^{x-3x_1} a_1^{x_1-3x_2} \ldots a_i^{x_i-3x_{i+1}} \ldots\right) \bmod p = \left(\prod_{i=0}^{[log_3 x]} a_i^{x_i-3x_{i+1}}\right) \bmod p, \qquad (2.6)$$

де $a_0=a$, $x_0=x$, $a_i = a_{i-1}^3 \bmod p$;

$$x_i = \left[\frac{x_{i-1}}{3}\right];$$

Основна ідея методу полягає у використанні операції піднесення до кубу замість квадрата. Це дозволяє у деяких ситуаціях скоротити кількість необхідних множень, що є важливим при роботі з великими експонентами та у багаторівневих схемах шифрування.

Загальна структура алгоритму має вигляд:

1) подати експоненту x у трійковій системі числення;

2) встановити початкове значення $N = 1$.

3) для кожного розряду x_i, починаючи з найстаршого:

- піднести поточне значення до кубу;
- якщо $x_i = 1$, виконати множення на основу a;
- якщо $x_i = 2$, виконати множення на a^2;
- після кожного кроку виконати модульну редукцію за p.

4) після обробки всіх трійкових цифр отримане N буде результатом $a^x \bmod p$.

Таким чином, метод використовує кубування як основну операцію замість квадратування, що дозволяє скоротити кількість кроків для певних класів експонент.

Серед переваг цього методу:

- зменшення кількості операцій завдяки використанню трійкового подання експоненти;

- висока сумісність із векторно-модульними обчисленнями та паралельною обробкою даних.

До недоліків: необхідність перетворення експоненти у трійкову систему, що ускладнює попередню підготовку; для довільних експонент метод часто не забезпечує значних переваг порівняно з методом піднесення до квадрату; дещо складніший для апаратної реалізації.

Таким чином, метод піднесення до кубу знаходить своє застосування в специфічних умовах, де його переваги можуть бути максимально використані.

2.1.3. Векторно-модульний метод

Векторно-модульний метод (Vector Modular Exponentiation) є одним із найпродуктивніших підходів до реалізації модулярного експоненціювання, яке є центральною операцією в асиметричних криптографічних системах, зокрема RSA, ElGamal, Paillier та інших схемах, що базуються на складності обчислення степеня за великим модулем [29].

Його поява пов’язана з потребою прискорити виконання криптографічних операцій у середовищах із великими обсягами даних та обмеженими часовими ресурсами. Сучасні апаратні платформи – від центральних процесорів до графічних прискорювачів і FPGA – забезпечують широкий спектр можливостей для паралельної обробки, що і стало передумовою виникнення цього підходу.

Сутність векторно-модульного методу полягає в поділі процесу піднесення до степеня за модулем на низку незалежних підзадач, кожна з яких може бути обчислена окремо, а потім їх результати комбінуються для отримання кінцевого значення. Така декомпозиція обчислення дозволяє виконувати часткові операції одночасно, тобто векторизовано, з використанням багатопотокових або багатоядерних обчислювальних систем.

Нехай необхідно обчислити вираз:

$$a^x \bmod p, x \leq \varphi(p),$$

де $\varphi(p)$– функція Ейлера від модуля p, тобто кількість чисел, взаємно простих із p і менших за нього.

Для цього формується проміжна матриця розміром $n_0 \times n_0$, де n_0– кількість бітів модуля p. У кожному стовпці цієї матриці зберігається результат:

$$A_i = a^{2^i} \bmod p,$$

тобто степені числа a, що відповідають двійковим позиціям показника степеня.

Показник xподається у двійковому вигляді:

$$x = (x_{n_0-1} x_{n_0-2} \dots x_0)_2.$$

Тоді значення $a^x \bmod p$можна обчислити як добуток тих елементів A_i, для яких $x_i = 1$:

$$a^x \bmod p = \prod (A_i)^{x_i} \bmod p. \quad (2.6)$$

Рекурентна формула для побудови елементів матриці має вигляд:

$$A_i = (A_{i-1})^2 \bmod p. \quad (2.7)$$

Такий підхід дозволяє уникнути надмірних множень і забезпечує структуровану логіку обчислення, яка добре підходить для розпаралелювання. Кожен елемент матриці можна розраховувати незалежно, використовуючи попередній результат, а потім усі обчислені значення поєднуються модульним множенням.

Для наочного прикладу розглянемо обчислення $23^{19} \bmod 29$. Спочатку будується матриця степенів числа 23:

$$23^1 \bmod 29 = 23,$$
$$23^2 \bmod 29 = 7,$$
$$23^4 \bmod 29 = 20,$$

$$23^8 \bmod 29 = 7,$$
$$23^{16} \bmod 29 = 23.$$

Число 19 у двійковому представленні дорівнює $(10011)_2$. Це означає, що для обчислення експоненти необхідно взяти ті степені, які відповідають одиничним бітам, тобто 2^0, 2^1і 2^4. Тоді:

$$23^{19} \bmod 29 = (23^{16} \times 23^3) \bmod 29 = (7 \times 7 \times 23) \bmod 29 = 25.$$

У реальних умовах, коли модуль p має велику розрядність (наприклад, 1024 або 2048 біт), створення повної матриці значень A_i є обчислювально затратним. Тому використовується векторно-модульна оптимізація, у межах якої обчислення окремих степенів $a^{2^i} \bmod p$ виконуються паралельно у кількох потоках.

Це дозволяє знизити часову складність алгоритму до: $O(\log n/k)$, де k– кількість паралельних потоків або апаратних ядер.

Для об’єднання часткових результатів використовують китайську теорему про залишки (CRT) або систему залишкових класів (RNS). Застосування таких методів забезпечує точність кінцевого результату та дозволяє проводити обчислення без втрати проміжних даних навіть при роботі з надвеликими числами.

Практично реалізований алгоритм векторно-модульного експоненціювання складається з кількох етапів:

1) розбиття задачі – визначення діапазонів бітів показника степеня, які будуть оброблятися окремими потоками;

2) паралельне піднесення до степеня – виконання часткових експоненціювань;

3) об’єднання результатів – поєднання часткових значень через модульне множення або CRT;

4) редукція – приведення отриманого результату до меж модуля.

Таке структурування дозволяє адаптувати метод під будь-яке апаратне середовище – від звичайного багатоядерного CPU до спеціалізованих криптографічних прискорювачів.

2.1.4. Використання системи залишкових класів

У процесі виконання модульного експоненціювання для великих чисел важливо мінімізувати обчислювальну складність і забезпечити можливість паралельної обробки даних.

Одним із найефективніших способів досягнення цього є застосування СЗК (англ. Residue Number System – RNS), яка ґрунтується на поданні чисел не у звичайній позиційній формі, а у вигляді набору залишків від ділення на попарно взаємно прості модулі.

Сутність СЗК полягає у тому, що будь-яке число N із діапазону

$$0 \leq N < P, P = \prod_{i=1}^{k} \quad p_i,$$

можна подати у вигляді вектора залишків:

$$N \leftrightarrow (n_1, n_2, \dots, n_k),$$

де $n_i = N \bmod p_i$, а p_i – взаємно прості модулі.

Таке представлення дозволяє виконувати основні арифметичні операції незалежно за кожним модулем, що усуває перенос розрядів і робить систему повністю паралельною.

Після завершення часткових обчислень відновлення початкового результату здійснюється за допомогою КТЗ, що гарантує існування єдиного числа N у межах добутку модулів P, для якого справджується рівність:

$$N \equiv n_i (\bmod p_i), i = 1,2, \dots, k.$$

Процес піднесення до степеня в СЗК можна подати як:

$$N = a^x \bmod p = (\sum_{i=1}^{k} \quad m_i M_i a_i) \bmod p, \qquad (2.8)$$

де

$$a_i = (a \bmod p_i)^x \bmod p_i, p = \prod_{i=1}^{k} \quad p_i, M_i = \frac{p}{p_i}, m_i = M_i^{-1} \bmod p_i,$$

а k – кількість модулів.

Така побудова дозволяє розділити операцію піднесення до степеня на кілька незалежних потоків, кожен із яких виконує обчислення у своєму модульному просторі

На фінальному етапі результати об'єднуються за допомогою КТЗ, формуючи єдине значення N у межах глобального модуля p.

Алгоритмічна послідовність:

1) вибір системи модулів. Обираються k взаємно простих модулів $p_1, p_2, \dots, p_k$, добуток яких перевищує максимальне можливе значення експоненційного результату;

2) обчислення залишків. Для основи aвизначаються залишки $a_i = a \bmod p_i$;

3) піднесення до степеня. Кожне a_iпідноситься до степеня xза модулем p_i;

4) відновлення результату. Отримані часткові результати $a_i^x \bmod p_i$ комбінуються у фінальний результат за формулою (2.1).

Для обчислення обернених елементів m_i використовується алгоритм Евкліда, який дозволяє знайти коефіцієнти зворотності навіть для великих чисел без необхідності прямого ділення.

Таким чином, СЗК забезпечує природну можливість паралельного виконання арифметичних операцій, що істотно скорочує час експоненціювання великих чисел у криптографічних застосуваннях.

2.1.5. Використання модифікованої досконалої форми системи залишкових класів

Модифікована досконала форма системи залишкових класів (МДФ) СЗК є подальшим розвитком класичної СЗК, спрямованим на підвищення швидкодії обчислень під час виконання модульних операцій великої розрядності [30].

Основна ідея МДФ СЗК полягає у спрощенні відновлення результату за китайською теоремою про залишки шляхом виключення громіздкої операції пошуку оберненого елемента для кожного модуля.

У класичній формі СЗК відновлення результату здійснюється за формулою:

$$N = a^x \bmod p = (\sum_{i=1}^{k} \quad m_i M_i a_i) \bmod p, \tag{2.9}$$

де

$$a_i = (a \bmod p_i)^x \bmod p_i, M_i = \frac{p}{p_i}, m_i = M_i^{-1} \bmod p_i.$$

Тут пошук обернених коефіцієнтів m_iвимагає застосування алгоритму Евкліда, що при великих модулях значно збільшує обчислювальну складність. У МДФ ця проблема усувається завдяки спеціальному вибору системи модулів.

Якщо модулі $p_1, p_2, \ldots, p_k$ підібрані так, що виконуються співвідношення:

$$m_i = M_i^{-1} \bmod p_i = \pm 1,$$

тобто обернені коефіцієнти дорівнюють ±1, то формулу (2.7) можна спростити, замінивши множення на просте додавання або віднімання. Це зменшує кількість арифметичних операцій і, відповідно, скорочує загальний час обчислень.

Розглянемо систему конгруенцій:

$$\begin{cases} x \bmod p_1 = b_1, \\ x \bmod p_2 = b_2, \\ x \bmod p_k = b_k, \end{cases} \tag{2.10}$$

де всі p_i– попарно взаємно прості модулі. Згідно з КТЗ, розв'язок системи можна подати у вигляді:

$$x = (\sum_{i=1}^{k} b_i M_i m_i) \bmod P,$$

де $P = \prod_{i=1}^{k} p_i$. У МДФ СЗК, завдяки вибору модулів, при якому $m_i = \pm 1$, ця формула набуває вигляду:

$$x = \left(\sum_{i=1}^{k} b_i M_i m_i\right) \bmod P, \text{де } m_i \in \{+1, -1\} \qquad (2.11)$$

Таким чином, відновлення результату не потребує виконання операції обернення за модулем, а здійснюється лише за допомогою додавання та множення (табл. 2.1).

Таблиця 2.1 – Порівняльна характеристика методів

Показник	**Класична СЗК**	**МДФ СЗК**
Необхідність пошуку M_i^{-1}	так	ні
Тип операцій при відновленні	множення, додавання	лише додавання або віднімання
Час обчислень	вищий	знижений
Складність реалізації	помірна	нижча
Апаратна реалізація	паралельна	оптимізована
Область застосування	загальні обчислення	криптосистеми, прискорене експоненціювання

Застосування МДФ СЗК дозволяє спростити реалізацію алгоритму модульного експоненціювання та значно зменшити кількість необхідних арифметичних операцій. Це робить МДФ СЗК перспективним напрямом для апаратних реалізацій асиметричних криптосистем, де обчислення з великими числами потребують максимальної оптимізації.

2.2. Алгоритмічне забезпечення криптосистеми RSA на основі оптимізованих методів модульного експоненціювання

Ключовою операцією в криптосистемі RSA є модульне експоненціювання вигляду

$$C = M^K \bmod N,\ M = C^k \bmod N\ , \tag{2.12}$$

де M – блок відкритого тексту;

C – шифротекст;

K – показник відкритого ключа;

k – показник секретного ключа;

$N = P * Q$ – модуль, що є добутком двох великих простих чисел P та Q. Саме обчислення степеня за великим модулем визначає обчислювальну вартість шифрування та розшифрування.

На рисунку 2.1 наведено узагальнену структурну схему криптоалгоритму RSA. Вона містить три логічні блоки: генерацію ключів, шифрування та розшифрування. В обох останніх блоках центральним елементом є модульне експоненціювання, в якому може використовуватись будь-який із наведених нижче методів.

Піднесення до степеня у криптосистемі RSA на основі методу виділення квадрату використовує двійкове подання показника степеня і дозволяє зменшити кількість операцій множення в порівнянні з прямим обчисленням K послідовних множень за модулем. У програмному лістингу *run_binpow.py* ця логіка реалізована в класі *PowerDecreaseCalculation*. Змінна a відповідає поточній основі, змінна *power* – показнику x, а змінна *add_mul* виконує роль накопичувача r. На кожній ітерації циклу *while* показник або ділиться на 2 (парний випадок), або зменшується на одиницю з наступним діленням на 2 (непарний випадок), а до *add_mul* додатково «підмножується» поточна основа a. Після кожного кроку основа замінюється її квадратом за модулем p.

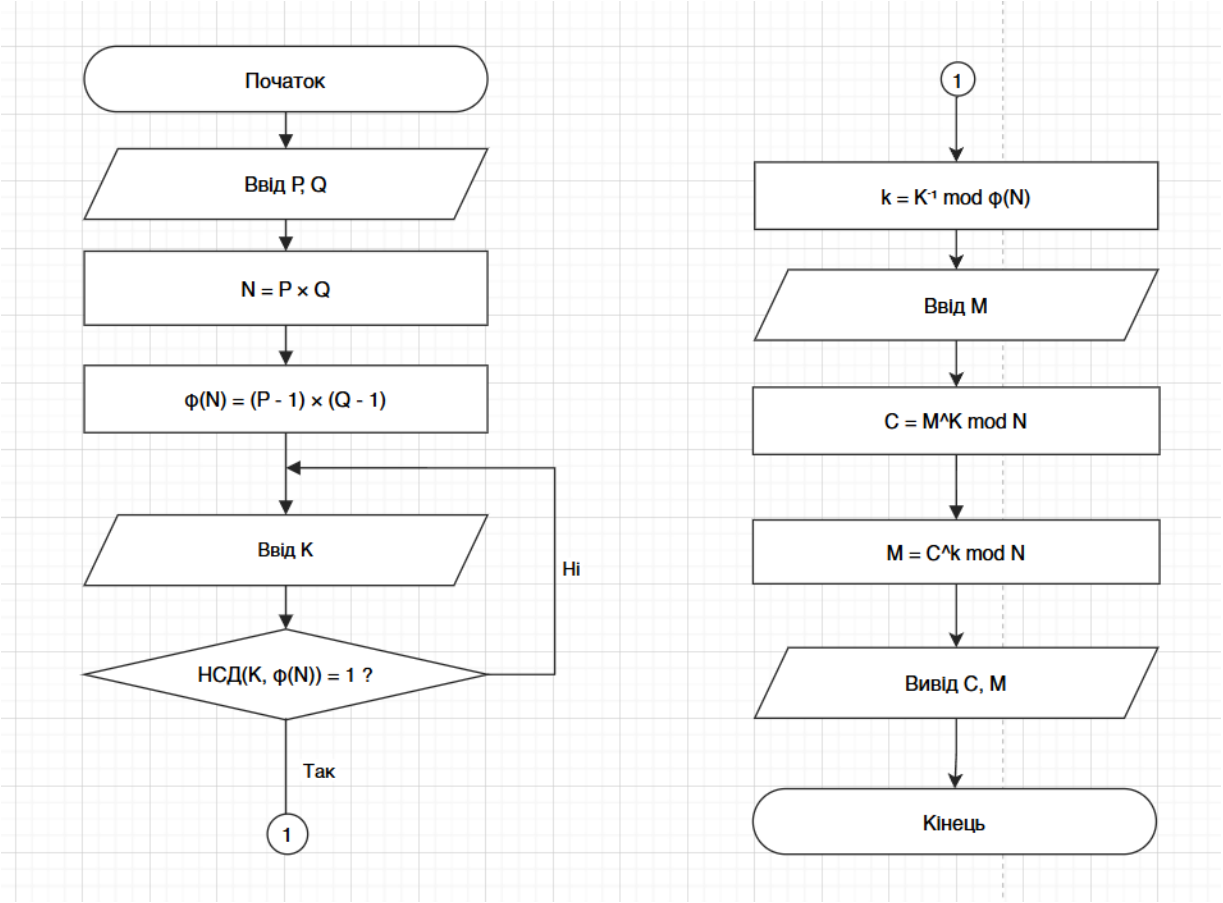

Рисунок 2.1 – Схема алгоритму RSA

Триарний або кубічний метод модульного експоненціювання у криптосистемі RSA ґрунтується на представленні показника степеня у троїчній системі числення. На відміну від бінарного методу, який працює з двома можливими станами біта (0 або 1), триарний підхід використовує три стани, що дозволяє зменшити кількість ітерацій циклу приблизно на третину за рахунок обробки більшої кількості інформації на одному кроці. У програмному лістингу *run_cubic.py* реалізація триарного методу подана в класі *PowerDecreaseCubicCalculation*. Логіка методу базується на використанні залишків від ділення показника степеня на 3 та послідовному оновленні проміжних значень:

- змінна *a* зберігає поточну основу;
- *a2* відповідає за попередньо обчислене $a^2 \bmod p$;
- *res* накопичує результат з урахуванням залишку *r*.

У кожному циклі обидві змінні (*a* та *a2*) підносяться до кубу, що узгоджується з переходом до наступного розряду в троїчній системі. Такий підхід забезпечує меншу кількість ітерацій при великих експонентах і є ефективним компромісом між швидкодією та складністю обчислень.

Векторно-модульний метод реалізації операції піднесення до степеня за модулем, що ґрунтується на розпаралеленні обчислень. Його основна ідея полягає у поділі задачі експоненціювання на кілька незалежних підзадач, які можуть виконуватися одночасно різними обчислювальними потоками (ядрами процесора, блоками GPU або FPGA-елементами). Цей метод особливо ефективний при роботі з великими числами у криптосистемах типу RSA, оскільки дозволяє зменшити загальний час виконання шляхом розбиття експоненти або модуля на вектори часткових обчислень, результати яких потім комбінуються у підсумкове значення.

Метод СЗК дозволяє розпаралелити обчислення і виконувати модульні операції над меншими числовими значеннями, що істотно підвищує швидкодію при великих розрядах у криптосистемі RSA. У коді клас *RnsModularExponentiation* реалізує повний цикл операцій системи залишкових класів:

- розбиття основи на залишки за кожним модулем (*residues*);
- незалежне виконання експоненціювання для кожного підмодуля (*powers*);
- кінцевого результату за формулою китайської теореми про залишки (*result*).

Такий підхід дозволяє ефективно розпаралелити обчислення, що особливо корисно при роботі з великими ключами RSA (1024, 2048, 4096 біт). Крім того, метод СЗК знижує ризик переповнення змінних і спрощує реалізацію апаратного прискорення.

Метод МДФ СЗК є подальшим розвитком класичної СЗК. Його основна мета – мінімізувати надлишкові операції множення та зменшити часові витрати при виконанні модульного експоненціювання в криптосистемі RSA. Замість довільного набору модулів, як у звичайній СЗК, використовується строго

визначена комбінація, що мінімізує похибку округлення при відновленні результату та забезпечує оптимальний розподіл розрядності між модулями. Таким чином, метод МДФ СЗК демонструє найкращий баланс між швидкодією, стабільністю обчислень та апаратною реалізацією серед усіх оптимізованих форм системи залишкових класів.

2.3. Алгоритмічне забезпечення криптосистеми Ель-Гамаля на основі оптимізованих методів модульного експоненціювання

Ключовою операцією в криптосистемі Ель-Гамаля, як і в RSA, є модулярне експоненціювання, що використовується під час шифрування та розшифрування повідомлень. Для заданих параметрів p - великого простого числа, g - первісного елемента групи, a - секретного ключа, $h=q^a \bmod p$– відкритого параметра, обчислення під час шифрування та дешифрування зводяться до експоненційних операцій виду:

$$c_1 = q^r \bmod p, \quad c_2 = h^r X \bmod p, \quad X = c_2\,(c_1^{-1})^a \bmod p.$$

де p - простий модуль;

q - первісний елемент;

a - секретний ключ;

$h = q^a \bmod p$ - відкритий ключ;

r - випадкове число, що генерується для кожного шифрування;

X - блок відкритого тексту.

Саме піднесення до степеня за великим модулем визначає часову складність криптографічних процедур Ель-Гамаля. На рисунку 2.2 подано узагальнену структурну схему криптоалгоритму Ель-Гамаля. Вона містить три логічні блоки:

1) генерація ключів;
2) шифрування;
3) розшифрування.

Центральним елементом у кожному блоці є обчислення виразів вигляду $a^x \bmod p$ де a, x, p відповідають поточним параметрам алгоритму.

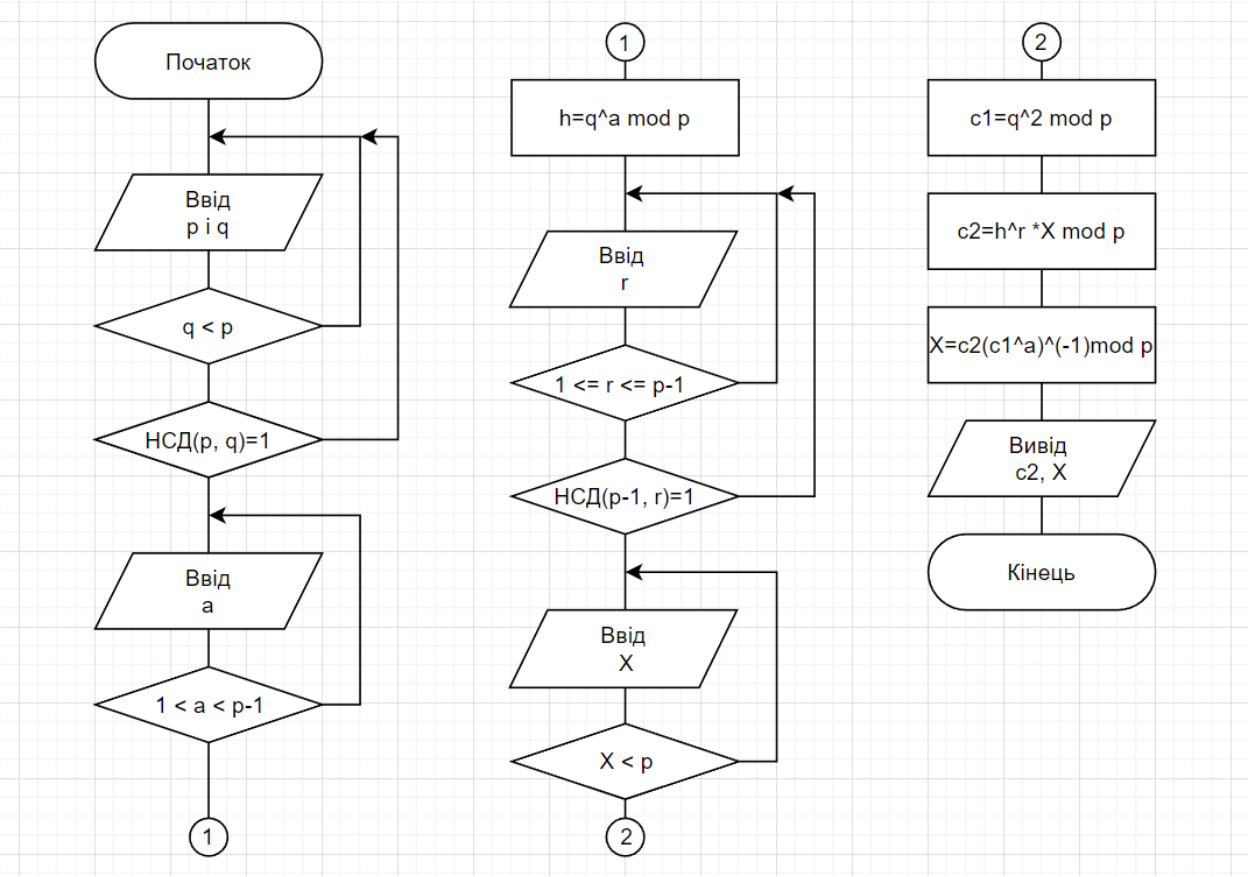

Рисунок 2.2 – Блок-схема криптоалгоритму Ель-Гамаля

Виконання операції модулярного експоненціювання у криптосистемі Ель-Гамаля за допомогою п’яти різних методів представлене у додатку Б. Піднесення до степеня за методом виділення квадратів використовує двійкове представлення показника степеня. Це дозволяє скоротити кількість операцій множення порівняно з прямим обчисленням. Відповідна реалізація у файлі run_elgamal_binpow.py. У ній застосовано клас PowerDecreaseCalculation, який циклічно зменшує показник x, поділяючи його на 2, а при непарному значенні – множить накопичувач на поточну основу a. Після кожного кроку основа зводиться у квадрат за модулем p. Такий підхід є базовим і використовується як еталон для порівняння інших методів.

Тріарний (кубічний) метод ґрунтується на поданні показника степеня в троїчній системі числення. Це дозволяє скоротити кількість ітерацій приблизно на третину,

оскільки кожен цикл розглядає три можливі стани. Реалізація наведена у файлі run_elgamal_cubic.py, у якому використано клас PowerDecreaseCubicCalculation. Він зберігає дві змінні – a і a2 = a^2 mod p, що дозволяє виконувати множення з урахуванням залишку при поділі на 3. Такий метод забезпечує кращий баланс між швидкодією та числовою стабільністю при великих розрядностях.

Векторно-модульний підхід (Vector Modular Exponentiation) використовує розпаралелення процесу обчислення степеня. Замість послідовного зведення у степінь уся операція розбивається на незалежні частини (вектори), які можуть обчислюватися одночасно – що особливо ефективно при реалізації на багатоядерних процесорах або GPU. Реалізація міститься у файлі run_elgamal_vecmod.py, де для кожної розрядності генеруються випадкові параметри, а результати усереднюються для визначення середнього часу виконання. Метод демонструє стабільну швидкодію у широкому діапазоні розрядностей і рівнів випадковості.

Класичний метод СЗК розбиває основу степеня на залишки від ділення за кількома взаємно простими модулями p_1, p_2, p_3. Далі для кожного залишку окремо виконується модульне піднесення до степеня, а результати комбінуються за допомогою китайської теореми про залишки. Код реалізовано у файлі run_elgamal_rns_classic.py, який використовує класи Rns та RnsPowerDecreaseCalculator. Такий підхід дозволяє ефективно розпаралелити обчислення та зменшити ризик переповнення при великих модулях.

Метод МДФ СЗК є вдосконаленням класичного підходу, що дозволяє уникнути операцій обернення завдяки попередньо сформованим базисним коефіцієнтам. У файлі run_elgamal_rns_mpf.py реалізовано клас MpfRns з попереднім розрахунком базисних множників для швидкого відновлення результату без обчислення обернених елементів. Це дає змогу значно прискорити обчислення при розрядностях понад 1024 біти, забезпечуючи найкращу масштабованість серед усіх розглянутих методів.

Більш детальний аналіз показує, що обчислювальна ефективність криптосистеми Ель-Гамаля визначається не лише вибором конкретного алгоритму експоненціювання, але й сукупністю додаткових чинників, серед яких архітектурні особливості апаратної платформи, структура групи, в якій здійснюються операції, а також статистичні властивості випадкових параметрів, що генеруються під час шифрування. Зокрема,

випадкове число r, яке є критичним компонентом стійкості алгоритму, має рівномірно розподілятися на множині $\{1,\ldots,p-2\}$. Нерівномірність або передбачуваність цього параметра може значно знизити криптостійкість алгоритму, що, у свою чергу, актуалізує проблему поєднання систем шифрування з надійними генераторами псевдовипадкових чисел (PRNG) та апаратними джерелами ентропії.

Також важливою особливістю є те, що криптосистема Ель-Гамаля може працювати не лише у мультиплікативній групі цілих чисел за модулем p, але й у групах точок еліптичних кривих, де ступінь складності обчислювальних операцій змінюється досить суттєво. У такому випадку методи модульного експоненціювання трансформуються у процедури скалярного множення, але фундаментальна роль оптимізованих алгоритмів обчислення залишається незмінною.

Це підкреслює універсальність та широту застосування оптимізаційних технік, які розглядаються в цьому підрозділі.Ще одним важливим аспектом алгоритмічної оптимізації є забезпечення стійкості до атак на реалізацію (implementation attacks), до яких належать атаки таймінгу, диференціального споживання енергії (DPA), простого споживання енергії (SPA), аналізу електромагнітного випромінювання тощо. Певні методи модульного експоненціювання, такі як традиційні бінарні алгоритми, можуть демонструвати варіативність у послідовності виконуваних операцій залежно від значень бітів показника степеня. Така поведінка здатна створювати інформаційні витоки, які можуть бути використані зловмисником для відновлення секретного ключа a.

Тому сучасні криптографічні реалізації все частіше використовують алгоритми з (a) вирівняним часом виконання (constant-time), (b) маскуванням проміжних даних, або (c) рандомізацією порядку виконання окремих операцій. Оптимізовані методи Монтгомері, СЗК та МДФ СЗК мають відповідні модифікації для підтримання таких вимог.Важливо зазначити, що практична реалізація криптосистеми Ель-Гамаля повинна враховувати той факт, що модульні операції становлять більшу частину обчислювальної складності процесу шифрування. Розбиття експоненційної операції на множину незалежних підзадач, характерне для векторно-модульних методів та підходів на основі СЗК, дозволяє застосовувати багатопотокові архітектури процесорів з максимальною ефективністю.

Це особливо важливо для систем, де одночасно обробляється значна кількість криптографічних сесій, наприклад, у серверах автентифікації, VPN-шлюзах, хмарних сервісах або інфраструктурі електронного підпису.У цьому контексті вдосконалений підхід МДФ СЗК (модифікована декомпозиція факторів із системою залишкових класів) демонструє особливу цінність, оскільки дозволяє скоротити кількість дорогих операцій обернення у класичній CRT-реконструкції, натомість замінюючи їх на обчислення, придатні до попередньої підготовки (precomputation).

Це забезпечує суттєве скорочення часу обробки повідомлень при роботі з надвеликими модулями (4096 біт і більше), що є критично важливим для систем довготривалого зберігання конфіденційних даних та високонадійних криптографічних протоколів.Отже, вибір методу модульного експоненціювання у криптосистемі Ель-Гамаля має комплексний характер і визначається цілим рядом факторів: можливостями апаратної платформи, необхідною швидкодією, вимогами до захисту від побічних каналів, розрядністю ключів та обсягом передаваних даних.

Відсутність універсального алгоритму, що забезпечує оптимальність в усіх умовах, обумовлює використання комбінованих та адаптивних підходів, у яких різні методи доповнюють один одного. Таким чином, оптимізоване алгоритмічне забезпечення криптосистеми Ель-Гамаля формується як багаторівнева система, що поєднує математичні, інженерні та прикладні рішення для досягнення максимальної ефективності та криптостійкості.

3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ МОДУЛЯРНОГО ЕКСПОНЕНЦІЮВАННЯ ЗА ДОПОМОГОЮ PYTHON

3.1. Реалізація алгоритмів модулярного експоненціювання в криптосистемі RSA

Існує декілька підходів до реалізації модульного експоненціювання, які відрізняються продуктивністю та складністю реалізації. Усі дослідження було зроблено на комп'ютері Lenovo Legion 5 15ACH6 із процесором AMD Ryzen 5 5600H (3.3 GHz), 16 GB оперативної пам'яті. Програмну реалізацію здійснено мовою Python 3.4.0, вибір якої обумовлено її відкритою архітектурою, підтримкою об'єктно-орієнтованого та функціонального програмування, а також зручністю повторного використання коду. Python належить до інтерпретованих мов високого рівня з динамічною семантикою, що робить її оптимальним інструментом для розроблення та тестування алгоритмів, які потребують обробки великих чисел. Крім того, мова підтримує кілька парадигм програмування (об'єктно-орієнтовану, процедурну, функціональну й аспектно-орієнтовану) та має вбудовані механізми роботи з багаторозрядними значеннями (до 4096 біт), що забезпечує можливість реалізації криптографічних обчислень різної складності.

Під час вибору методів модулярного експоненціювання основну увагу було зосереджено на забезпеченні співставного порядку часу виконання для різних алгоритмів, а також на можливості їх застосування до чисел великої розрядності (до 4096 біт). З огляду на це, низку методів було свідомо виключено з дослідження через надмірну обчислювальну складність або нестійкість при великих числах. Зокрема, метод прямого піднесення до степеня не застосовувався, оскільки вже при обробці 16-бітних значень відбувалося переповнення розрядної сітки, що робило подальші обчислення некоректними. Також не розглядалися методи послідовного множення справа наліво чи зліва направо, які вимагають виконання (x–1) множень, що істотно збільшує загальний час розрахунку. Аналогічно, класичний підхід системи залишкових класів (СЗК), у якому модуль p подається як добуток кількох простих чисел ($p = p_1 p_2 \ldots p_n$), не

використовувався через необхідність факторизації модуля, що призводить до суттєвого ускладнення алгоритму та збільшення часової складності.

Для проведення експериментальних досліджень обрано наступні методи: бінарний, пониження степеня за допомогою кубів, векторно-модульний, системи залишкових класів та модифікованої досконалої форми системи залишкових класів.

Метод пониження степеня за допомогою квадратів або бінарний метод базується на поетапному зниженні степеня за рахунок повторного піднесення основи до квадрату, що дозволяє значно скоротити кількість множень порівняно з прямими підрахунками. Його можна описати наступною формулою:

$$N = a^x \bmod p = \left(a^{x-2x_1} a_1^{x_1-2x_2} \ldots a_i^{x_i-2x_{i+1}} \ldots\right) \bmod p = \left(\prod_{i=0}^{[\log_2 x]} a_i^{x_i-2x_{i+1}}\right) \bmod p,$$

де $a_0=a$, $x_0=x$, $a_i = a_{i-1}^2 \bmod p$;

$$x_i = \left[\frac{x_{i-1}}{2}\right].$$

Метод пониження степеня за допомогою кубів (3-арний), фактично є узагальненням попереднього, але використовує кубічне піднесення, завдяки чому ще більше зменшується кількість ітерацій при обчисленнях великих степенів. Його можна записати наступним чином:

$$N = a^x \bmod p = \left(a^{x-3x_1} a_1^{x_1-3x_2} \ldots a_i^{x_i-3x_{i+1}} \ldots\right) \bmod p = \left(\prod_{i=0}^{[\log_3 x]} a_i^{x_i-3x_{i+1}}\right) \bmod p,$$

де $a_0=a$, $x_0=x$, $a_i = a_{i-1}^3 \bmod p$;

$$x_i = \left[\frac{x_{i-1}}{3}\right].$$

Векторно-модульний метод – поєднує ідеї класичного піднесення до квадрату з елементами векторизації, коли проміжні степені формуються заздалегідь, що забезпечує ефективність при великих розрядностях і дозволяє реалізувати паралельну обробку підмножин степенів. У межах експериментів

обчислення виконуються в одному потоці, однак сама логіка побудови вектора створює базу для потенційної оптимізації в апаратних реалізаціях. Метод можна описати наступним виразом:

$$a^x \bmod p = \prod (A_i)^{x_i} \bmod p,$$

де

$$A_i = a^{2^i} \bmod p,$$

а $x = (x_{n_0-1} x_{n_0-2} \dots x_0)_2$ - бінарне представлення показника степеня.

Метод системи залишкових класів (СЗК) – передбачає розбиття основи степеня на залишки від ділення на кілька взаємно простих модулів. Далі для кожного залишку застосовується бінарний метод зниження степеня, після чого результати поєднуються в одне число шляхом відновлення у десятковій системі. Для відновлення використовується алгоритм Евкліда та китайська теорема про залишки. Цей метод можна описати наступним виразом:

$$N = a^x \, mod \, p = \left(\sum_{i=1}^{k} m_i M_i a_i \right) mod \, p,$$

де $a_i = \left(a \, mod \, p_i\right)^x \, mod \, p_i$;

$p = \prod_{i=1}^{k} p_i$, $M_i = \frac{p}{p_i}$;

$m_i = M_i^{-1} \, mod \, p_i$;

k – кількість модулів.

Метод МДФ СЗК – відрізняється від класичного підбором модулів таким чином, щоб виключити необхідність знаходження оберненого елемента та виконання додаткових множень на базисні числа m_i. Це спрощення дозволяє істотно скоротити час виконання операції, що є особливо помітним при роботі з великими розрядностями.

Для проведення експериментів параметри $a^x \bmod p$ формувалися відповідно до вибраної розрядності чисел n та кількості одиничних бітів у їхньому двійковому поданні, що визначається вагою Хемінга. Значення цих параметрів генерувалися за допомогою випадкової функції RND, яка приймає значення в інтервалі від 0 до 1. Для моделювання використовувалися дискретні рівні параметра випадковості: 0, 10, 30, 50 та 80. Показник Δ=0 відповідає максимальній вазі Хемінга, тобто коли всі біти чисел $a=2^n-1$ та $x=2^{n-3}-1$ є одиничними. Зі збільшенням Δ відбувається поступове випадкове обнулення певної частини молодших бітів, що дозволяє змоделювати різні ступені "розрідженості" чисел у двійковій формі. Такий підхід дає змогу дослідити, як зміна кількості одиничних розрядів у вхідних даних впливає на швидкість виконання модулярного експоненціювання.

Перед проведенням експериментальних замірів усі сторонні процеси в операційній системі було вимкнено, щоб уникнути впливу фонових завдань на час виконання програм. Для підвищення точності вимірювань кожен алгоритм модулярного експоненціювання було реалізовано як окремий Python-скрипт: run_binpow.py - для бінарного методу, run_cubic.py - для 3-арного, run_vecmod.py - для векторно-модульного методу, run_rns_classic.py - для класичного СЗК, та run_rns_mpf.py - для МДФ СЗК. Усі програми побудовані на спільному принципі: спочатку генерується набір випадкових вхідних параметрів a, x, p заданої розрядності, після чого виконується обчислення виразу $a^x \bmod p$ із фіксацією часу в мікросекундах. Для кожного набору параметрів проводилося кілька запусків, після чого визначався середній час виконання. Результати автоматично виводилися у табличному форматі, де для кожного методу подано значення часу виконання при різних розрядностях і рівнях параметра випадковості Δ.

Для кожного із п'ятьох реалізованих методів використовувався однаковий принцип побудови модуля p, який формувався як добуток трьох попарно взаємно простих чисел $p_1,\ p_2,\ p_3$ підібраних відповідно до МДФ СЗК, де $p_2 = 2p_1 - 1$, а $p_3 = 2p_1 + 1$. Така конструкція забезпечує відсутність перетинів між підмодулями та дозволяє відновлювати результат без обчислення обернених

елементів, що особливо корисно при великих розрядностях. Загальний модуль p перевищував значення, характерні для розрядності n, у середньому у 8–16 разів, що гарантувало стабільність обчислень і відсутність переповнення. Для кожного методу виконувалося не менше десяти ітерацій на кожній розрядності, після чого визначався середній час виконання операції модулярного експоненціювання в мікросекундах (мкс). Отримані результати виводилися автоматично у вигляді таблиць для кожного окремого методу, що відображали залежність часу виконання від розрядності та параметра випадковості Δ. Програмний код реалізації алгоритмів наведено в додатку А, тоді як у даному розділі на рисунках 3.1 – 3.5 подано скріншоти результатів роботи розробленого програмного засобу.

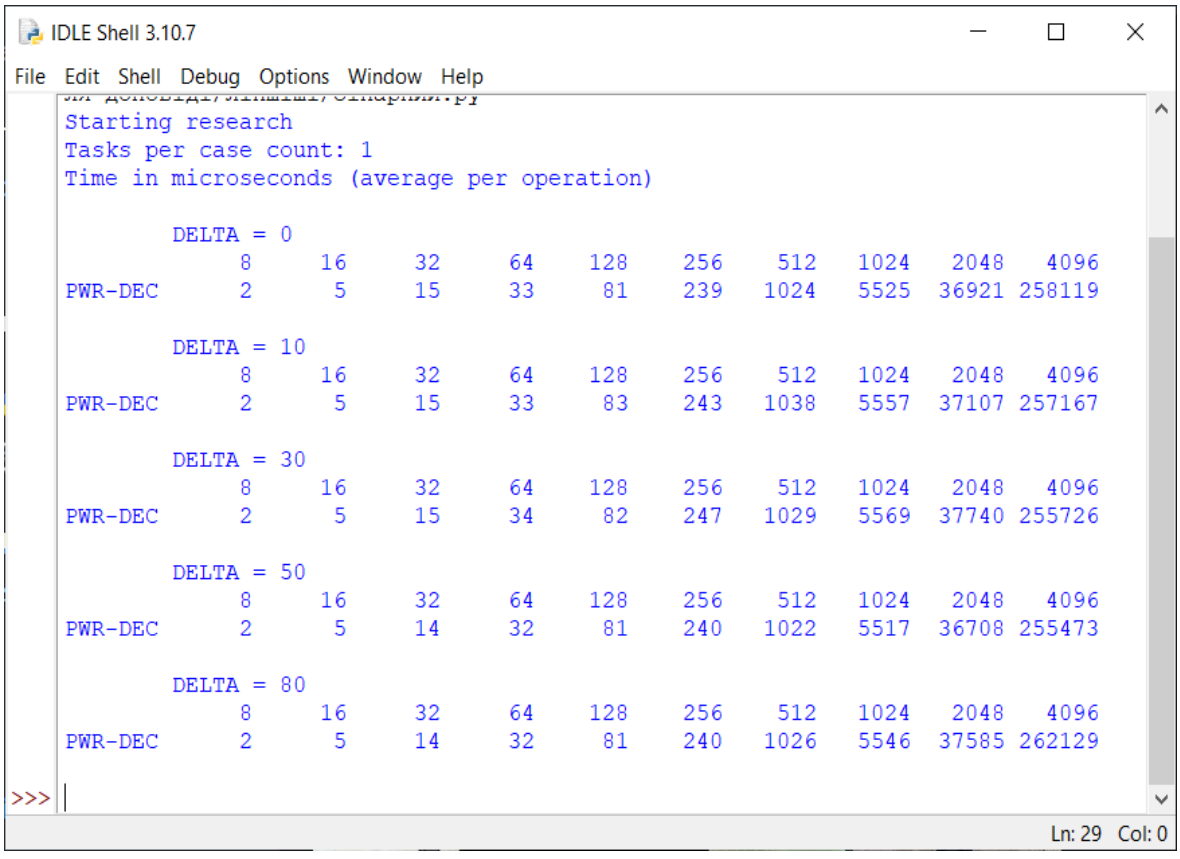

Рисунок 3.1 – Результат запуску розробленої програми для бінарного методу

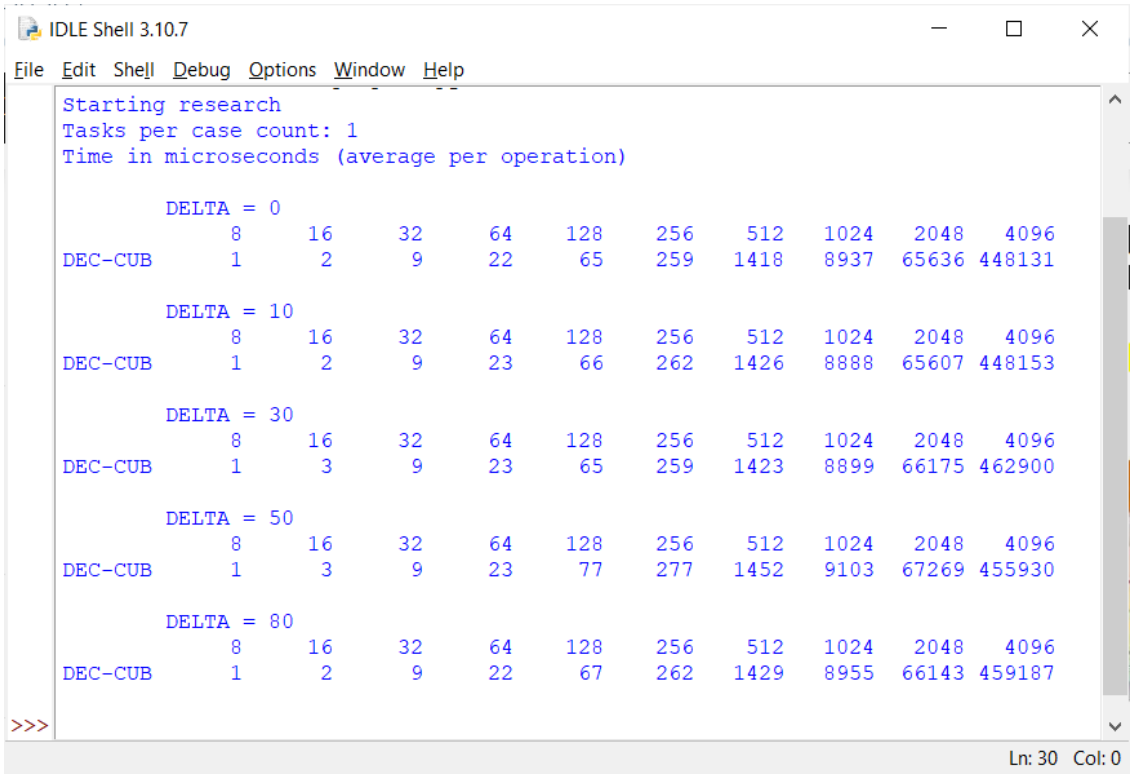

Рисунок 3.2 – Результат запуску розробленої програми для триарного методу

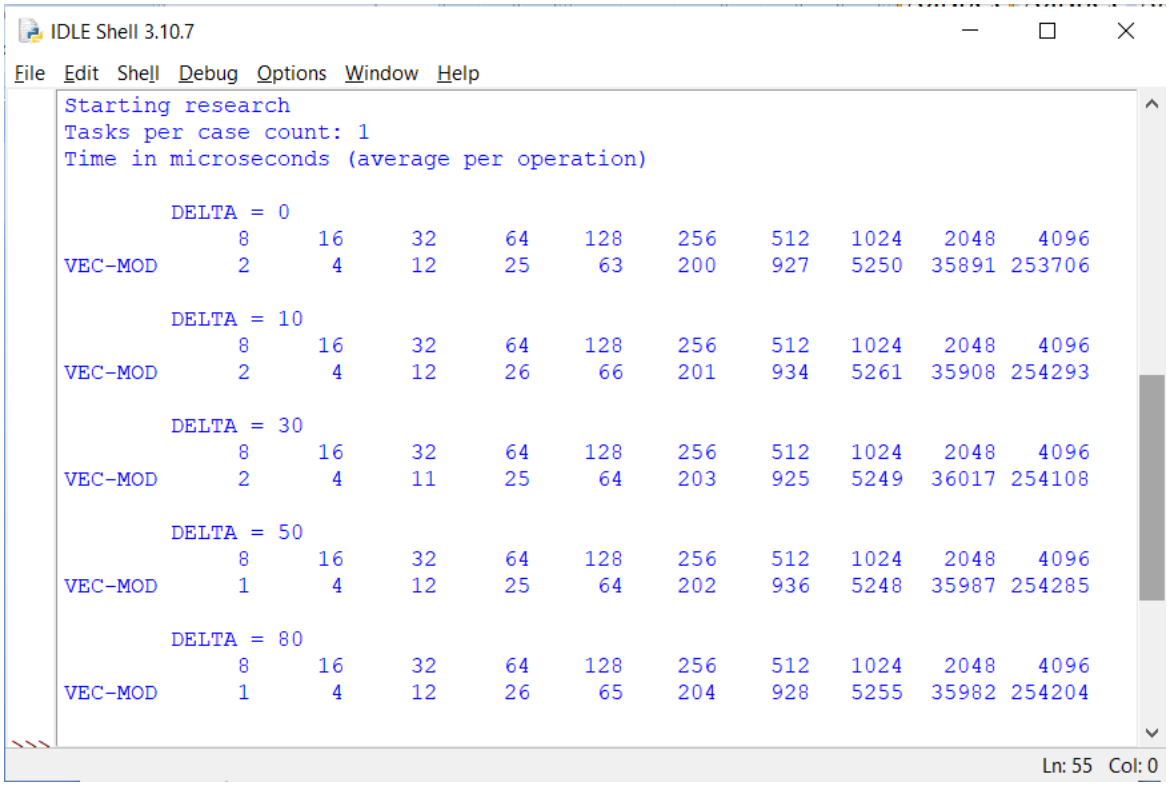

Рисунок 3.3 – Результат запуску розробленої програми для векторно-модульного методу

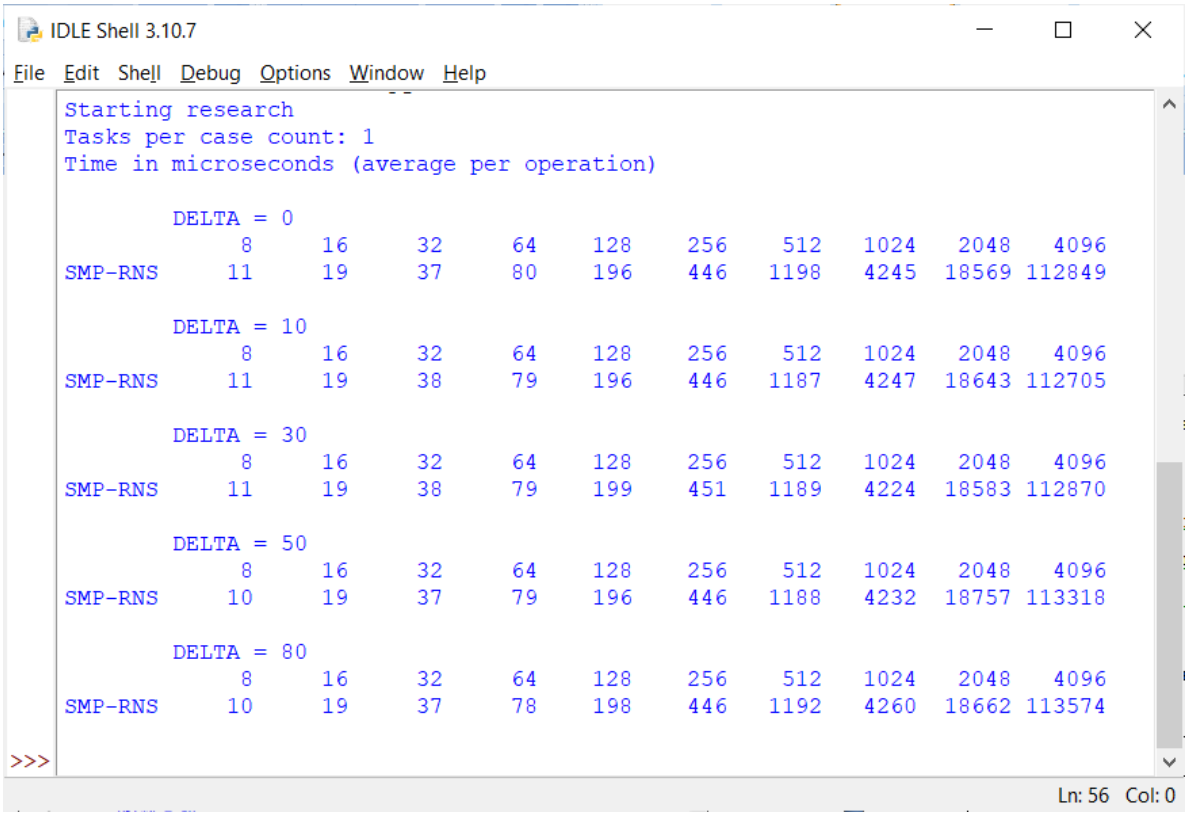

Рисунок 3.4 – Результат запуску розробленої програми для СЗК методу

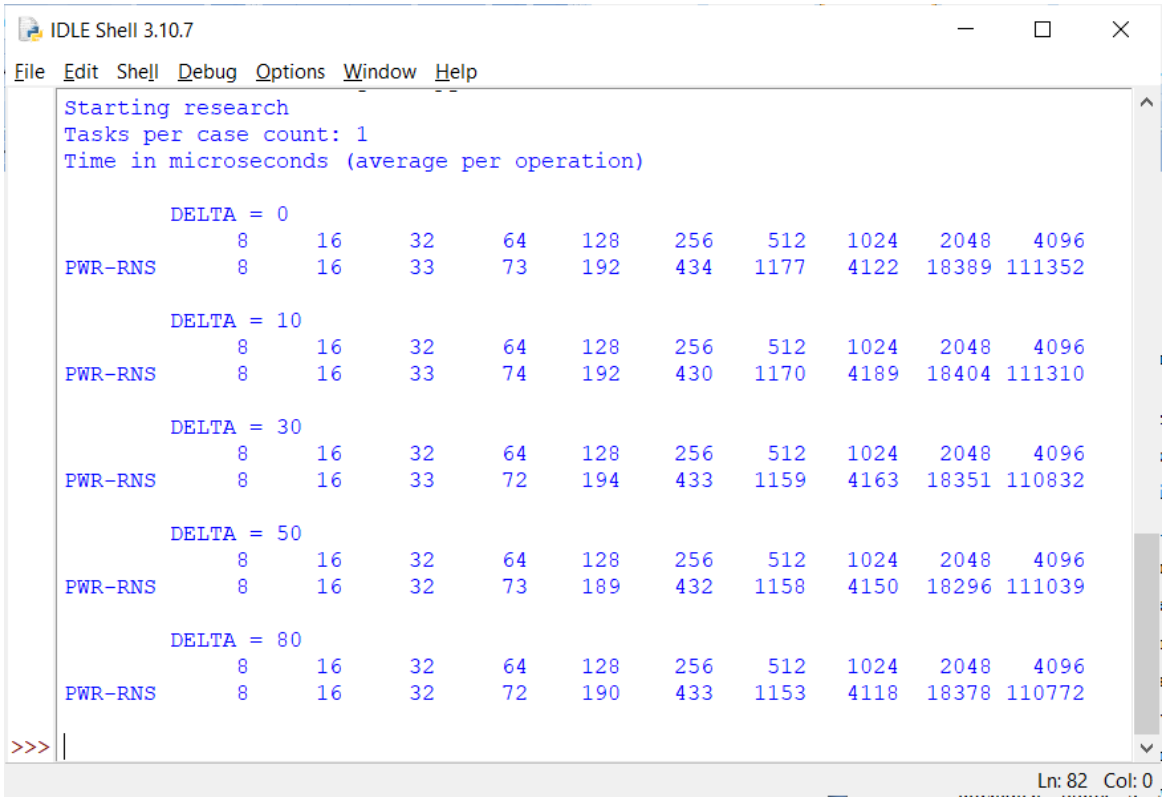

Рисунок 3.5 – Результат запуску розробленої програми для МДФ СЗК методу

Для порівняння результатів у таблиці 3.1 наведено час виконання операцій модулярного експоненціювання.

Таблиця 3.1 - Час виконання операції модулярного експоненціювання різними способами у криптосистемі RSA

Розрядність	8	16	32	64	128	256	512	1024	2048	4096
Δ=0										
1	2	5	15	33	81	239	1024	5525	36921	258119
2	1	2	9	22	65	259	1418	8937	65636	448131
3	2	4	12	25	63	200	927	5250	35891	253706
4	11	19	37	80	196	446	1198	4245	18569	112849
5	8	16	33	73	192	434	1177	4122	18389	111352
Δ=10										
1	2	5	15	33	83	243	1038	5557	37107	257167
2	1	2	9	23	66	262	1426	8888	65607	448153
3	2	4	12	26	66	201	934	5261	35908	254293
4	11	19	38	79	196	446	1187	4247	18643	112705
5	8	16	33	74	192	430	1170	4189	18404	111310
Δ=30										
1	2	5	15	34	82	247	1029	5569	37740	255726
2	1	3	9	23	65	259	1423	8899	66175	462900
3	2	4	11	25	64	203	925	5549	36017	254108
4	11	19	38	79	199	451	1189	4224	18583	112870
5	8	16	33	72	194	433	1159	4163	18351	110832
Δ=50										
1	2	5	14	32	81	240	1022	5517	36708	255473
2	1	3	9	23	77	277	1452	9103	67269	455930
3	1	4	12	25	64	202	936	5248	25987	254285
4	10	19	37	79	196	446	1188	4232	18757	113318
5	8	16	32	73	189	432	1158	4150	18296	111039
Δ=80										
1	2	5	14	32	81	240	1026	5546	37585	262129
2	1	2	9	22	67	262	1429	8955	66143	459187
3	1	4	12	26	65	204	928	5255	35982	254202
4	10	19	37	78	198	446	1192	4260	18662	113574
5	8	16	32	72	190	433	1153	4118	18378	110772

У першому стовпчику таблиці подано нумерацію методів модулярного експоненціювання: 1 — метод квадратів, 2 — метод кубів, 3 — СЗК, 4 — векторно-модульний метод, 5 — МДФ СЗК.

Згідно з таблицею 3.1, для малих розрядностей (до 512 біт) найшвидшими залишаються класичні методи – 1 та 2. Починаючи з 1024 біт, ефективність переходить до алгоритмів 4, 5, забезпечує найменший час виконання завдяки розпаралелюванню обчислень. Метод 3 займає проміжну позицію за стабільністю та швидкодією.

На рисунку 3.6 подано порівняльний аналіз часу виконання операції модульного експоненціювання при розрядності 4096 біт для п’яти реалізацій алгоритму.

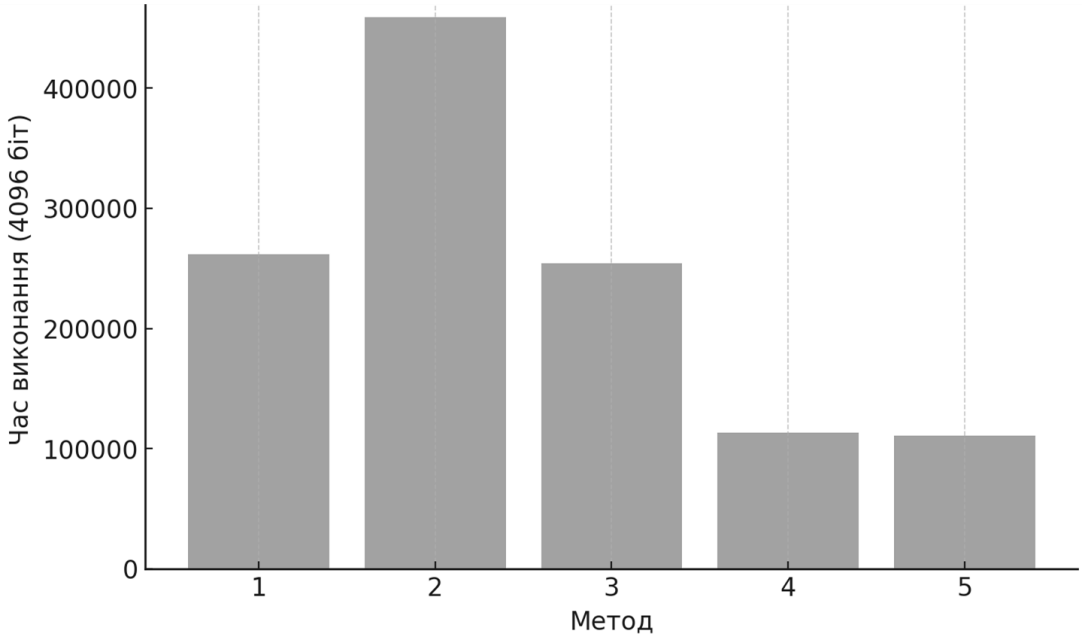

Рисунок 3.6 – Порівняння методів при розрядності 4096 Біт (Δ=80)

Як видно з рисунку 3.6, що всі методи демонструють значне зростання обчислювальних витрат у високих розрядностях, однак між ними зберігається помітна різниця у швидкодії.

3.2. Реалізація алгоритмів модулярного експоненціювання в криптосистемі Ель-Гамаля

Криптосистема Ель-Гамаля, як і RSA, ґрунтується на обчисленні модульного експоненціювання, однак застосовується для симетричних операцій

шифрування та розшифрування з використанням випадкової основи. На відміну від RSA, де ключі формуються через добуток двох простих чисел, в Ель-Гамаля вся обчислювальна логіка зосереджена навколо операцій виду:

$$c_1 = q^r \bmod p, \quad c_2 = h^r X \bmod p, \quad X = c_2\,(c_1^{-1})^a \bmod p.$$

де p - простий модуль;

q - первісний елемент;

a - секретний ключ;

$h = q^a mod\ p$ - відкритий ключ;

r - випадкове число, що генерується для кожного шифрування;

X - блок відкритого тексту.

Процес шифрування в Ель-Гамаля передбачає використання випадкового числа, що генерується для кожного повідомлення, внаслідок чого однакові відкриті тексти за однакових ключів породжують різні шифротексти. Така особливість забезпечує ймовірнісний характер шифрування та підвищує криптографічну стійкість алгоритму. Основні обчислення під час генерації ключів, шифрування та розшифрування зводяться до багаторазового виконання операцій модульного піднесення до степеня за великим простим модулем, що безпосередньо впливає на загальну продуктивність криптосистеми. Саме тому ефективність реалізації модулярного експоненціювання відіграє ключову роль в практичному застосуванні криптосистеми Ель-Гамаля, особливо при використанні великих розрядностей модуля, характерних для сучасних криптографічних систем.

Для реалізації експоненціювання в межах криптосистеми Ель-Гамаля було використано ті самі модулі, що й для RSA:

- run_binpow.py - для бінарного методу;
- run_cubic.py - для триарного методу;
- run_vecmod.py - для векторно-модульного підходу;
- run_rns_classic.py - для класичної СЗК;
- run_rns_mpf.py - для МДФ СЗК.

Кожен метод було протестовано на однакових наборах даних, що включали випадкові значення p, q, a, r, q заданої розрядності та різних рівнів параметра випадковості Δ.

Для оцінки ефективності вимірювався середній час обчислення виразу. Для кожного методу виконується серія тестів при розрядностях 8–4096 біт та різних рівнях параметра випадковості Δ. Приклад запуску програми для бінарного методу показаного на рисунку 3.5, реалізовано для інших методів (рисунки 3.7–3.11). Програмний код реалізації алгоритмів наведено в додатку А, тоді як у даному розділі подано скріншоти результатів роботи розробленого програмного засобу.

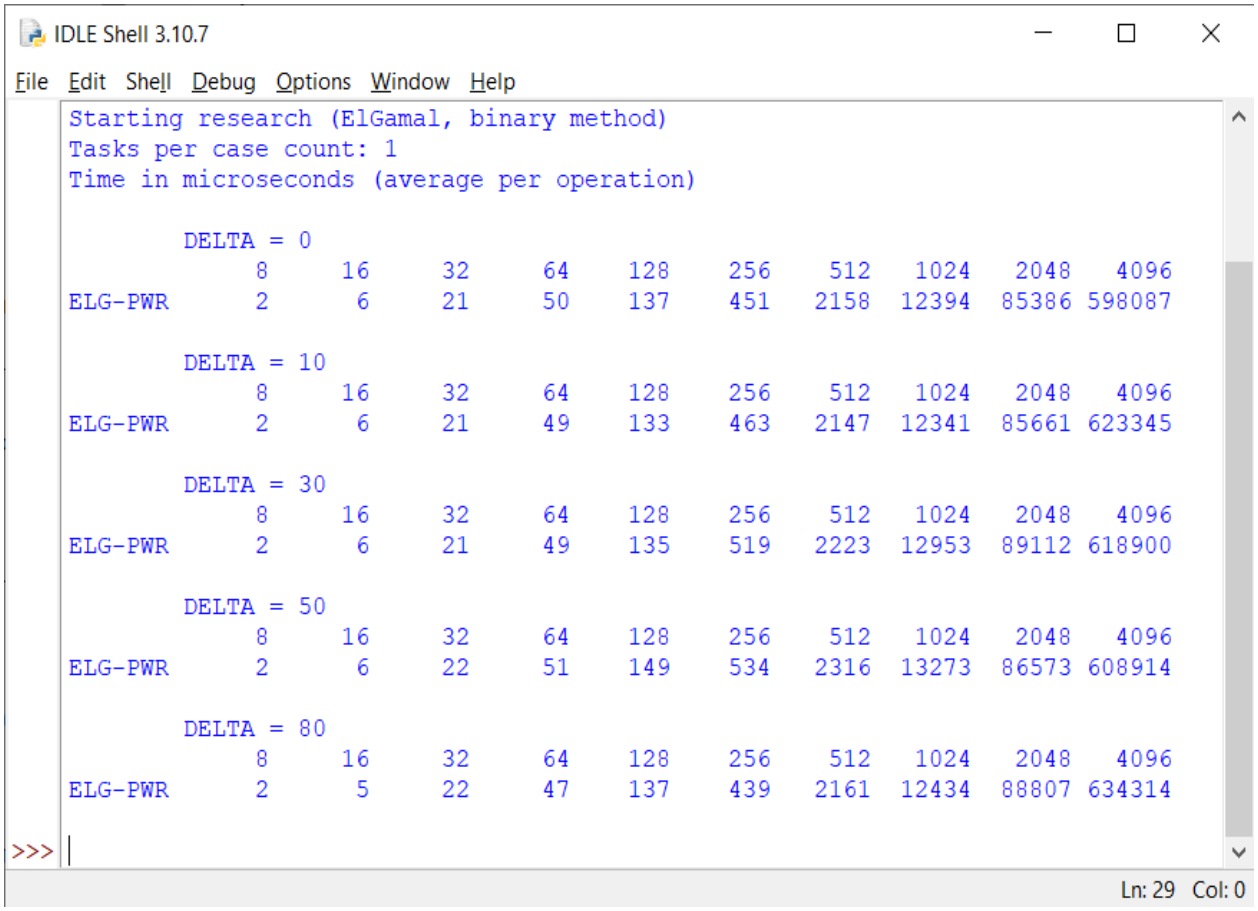

Рисунок 3.7 – Результат запуску розробленої програми для бінарного методу

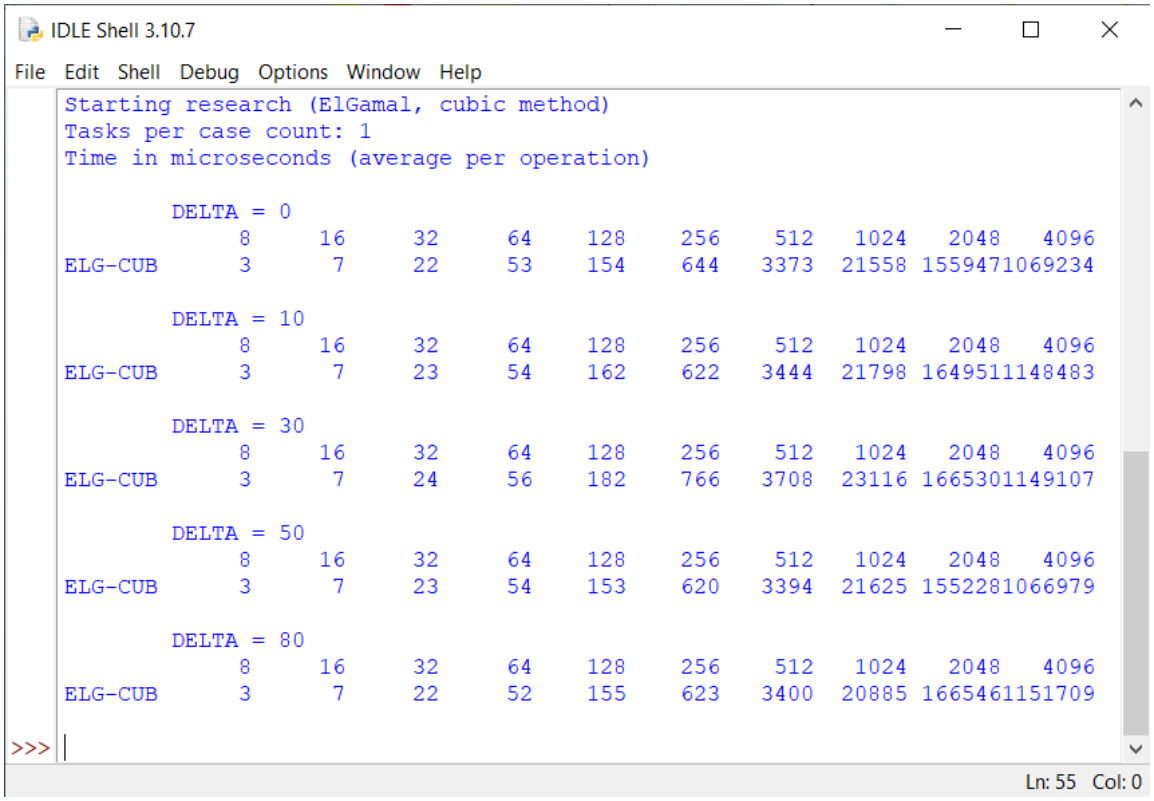

Рисунок 3.8 – Результат запуску розробленої програми для триарного методу

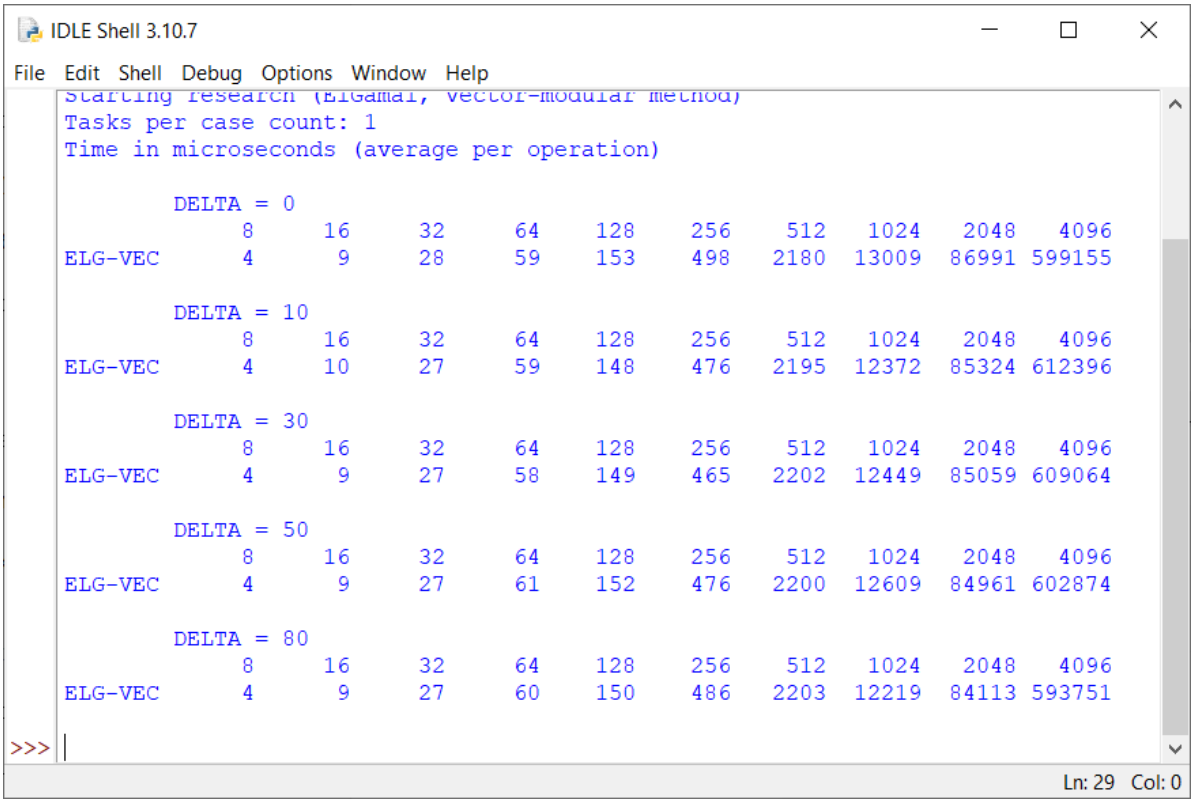

Рисунок 3.9 – Результат запуску розробленої програми для векторно-модульного методу

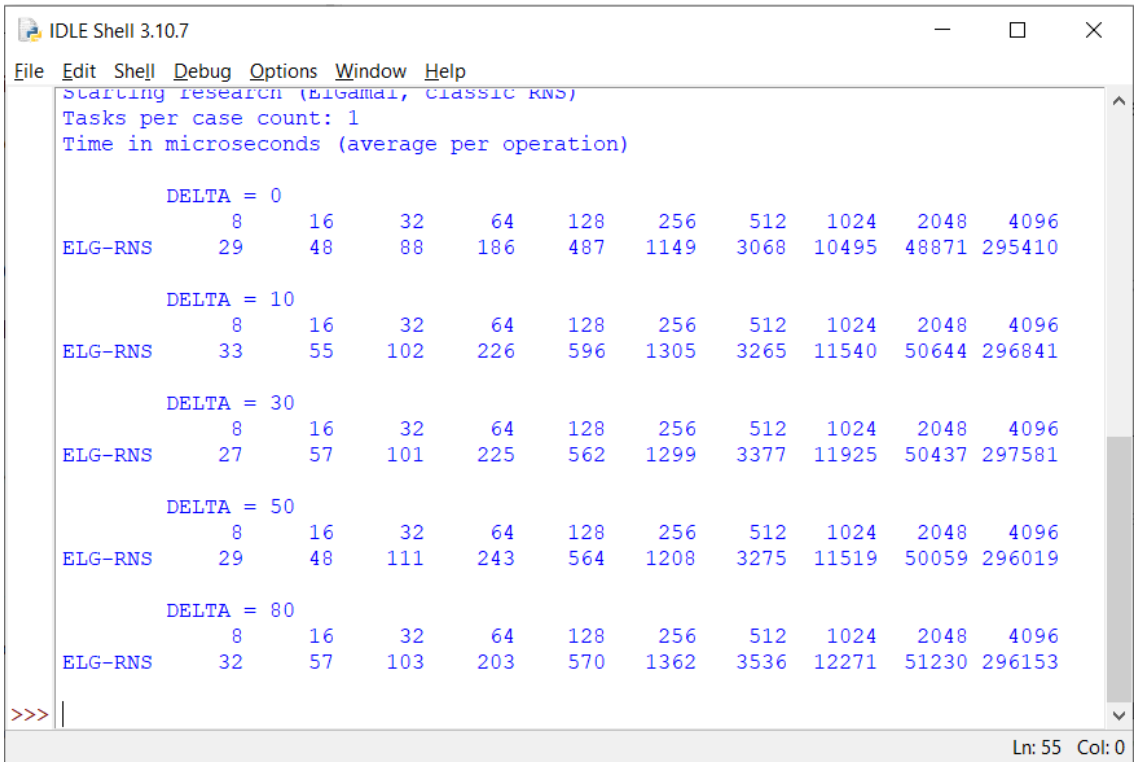

Рисунок 3.10 – Результат запуску розробленої програми для методу СЗК

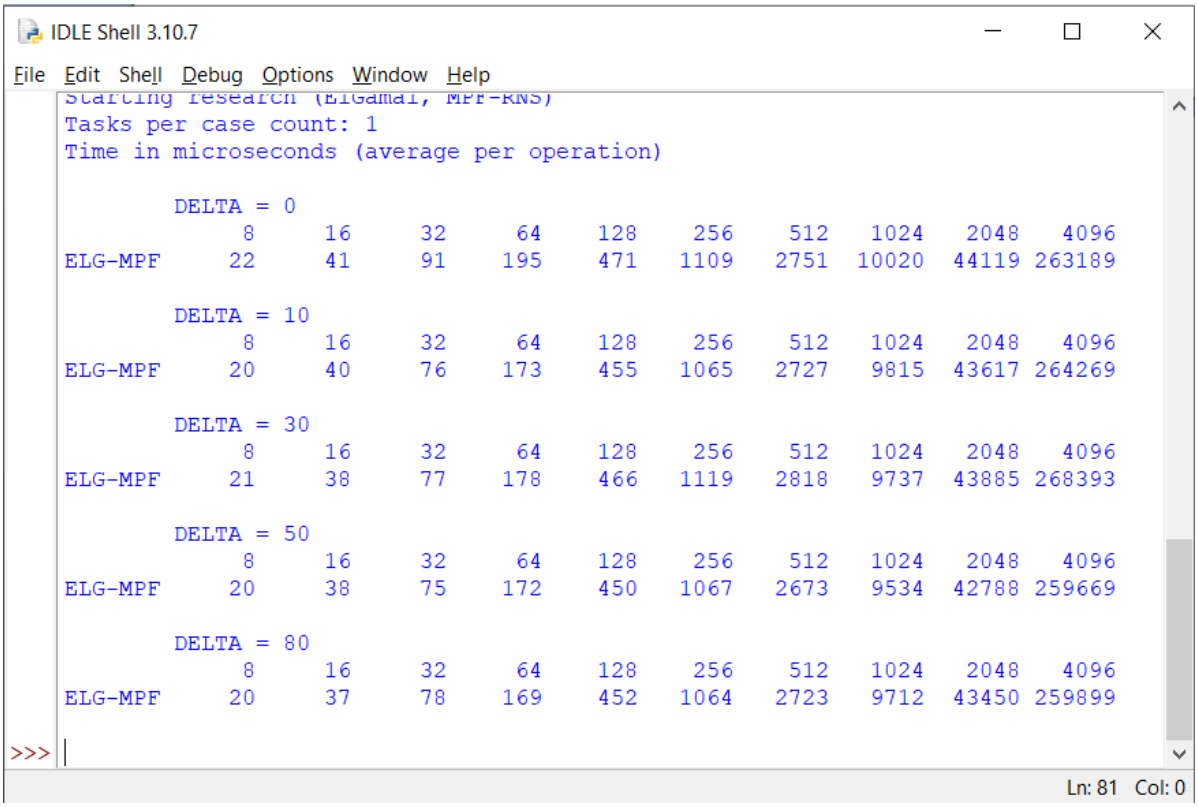

Рисунок 3.11 – Результат запуску розробленої програми для методу МДФ СЗК

У таблиці 3.2 наведено час виконання операцій модулярного експоненціювання для різних методів.

Таблиця 3.2 - Час виконання операції модулярного експоненціювання різними способами у криптосистемі Ель-Гамаля

Розрядність	8	16	32	64	128	256	512	1024	2048	4096
Δ=0										
1	2	6	21	50	137	451	2158	12394	85386	598087
2	3	7	22	53	154	644	3373	21558	155947	1069234
3	4	9	28	59	153	498	2180	13009	86991	599155
4	29	48	88	186	487	1149	3068	10495	48871	295410
5	22	41	91	195	471	1109	2751	10020	44119	263189
Δ=10										
1	2	6	21	49	133	463	2147	12341	85661	623345
2	3	7	23	54	162	622	3444	21798	164951	1148483
3	4	10	27	59	148	476	2195	12372	85324	612396
4	33	55	102	226	596	1305	3265	11540	50644	296841
5	20	40	76	173	455	1065	2727	9815	43617	264269
Δ=30										
1	2	6	21	49	135	519	2223	12953	89112	618900
2	3	7	24	56	182	766	3708	23116	166530	1149107
3	4	9	27	58	149	465	2202	12449	85059	609064
4	27	57	101	225	562	1299	3377	11925	50437	297581
5	21	38	77	178	466	1119	2818	9737	43885	268393
Δ=50										
1	2	5	22	51	149	534	2316	13273	86573	608914
2	3	7	23	54	153	620	3394	21625	155228	1066979
3	4	9	27	61	152	476	2200	12609	84961	602874
4	32	57	103	203	570	1362	3536	12271	51230	296153
5	20	38	75	172	450	1067	2673	9534	42788	259669
Δ=80										
1	2	5	22	47	137	439	2161	12434	88807	634314
2	3	7	22	52	155	623	3400	20885	166546	1151709
3	4	9	27	60	150	486	2203	12219	84113	593751
4	32	57	103	203	570	1362	3536	12271	51230	296153
5	20	37	78	169	452	1064	2723	9712	43450	259899

Згідно з таблицею 3.2, для малих розрядностей (до 256 біт) найменший час виконання демонструють класичні підходи – 1 та 2 методи. Починаючи з 512–1024 біт, помітно зростає ефективність методів 4 та 5, які завдяки розпаралелюванню обчислень забезпечують меншу тривалість операцій. МДФ СЗК стабільно демонструє найнижчі часові показники при 2048–4096 бітах, підтверджуючи перевагу модульного рознесення та усунення інверсій при великих розрядностях. Метод 3 займає проміжну позицію, зберігаючи стабільність часу виконання у всіх діапазонах Δ.

На рисунку 3.12 подано порівняння часу виконання модульного експоненціювання в криптосистемі Ель-Гамаля (Δ = 0) для різних методів і різних розрядностей модуля.

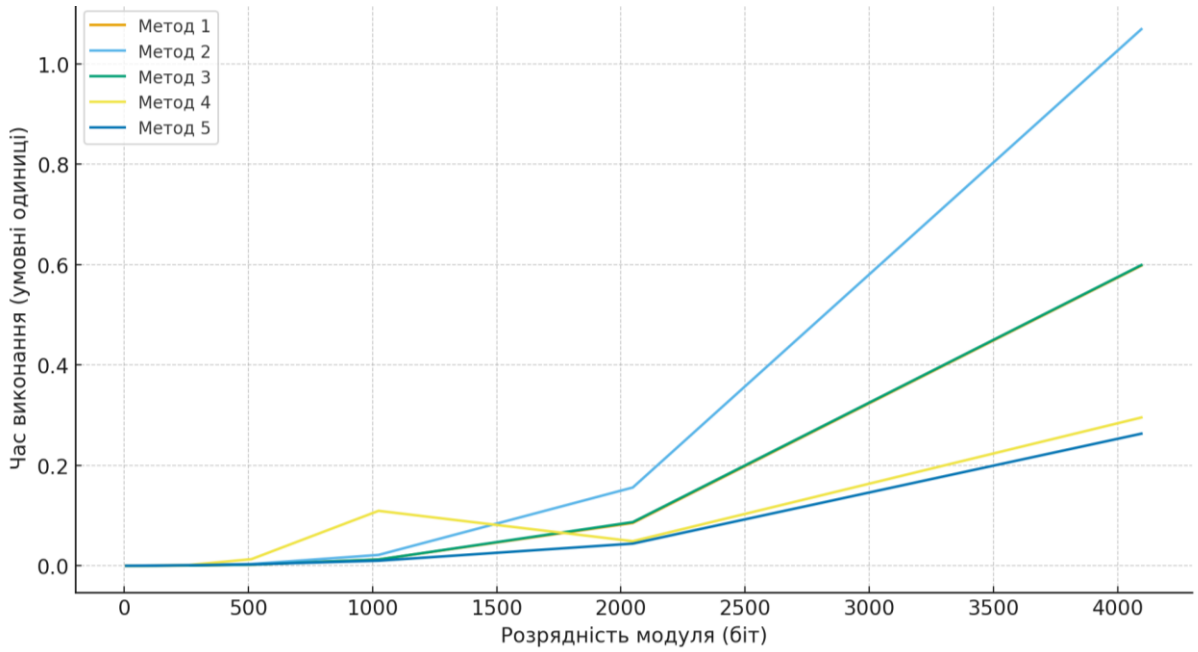

Рисунок 3.12 – Порівняння методів при розрядності при Δ=0

Як видно з рисунка, зі збільшенням довжини модуля час виконання зростає для всіх алгоритмів, проте швидкість зростання істотно відрізняється. Проведене дослідження підтверджує, що вибір алгоритму суттєво впливає на продуктивність криптографічних операцій при великих модулях.

Порівняльний аналіз результатів експериментальних досліджень для криптосистем RSA та Ель-Гамаля показав, що час виконання операцій

модулярного експоненціювання зростає зі збільшенням розрядності модуля для всіх розглянутих методів. При цьому базові методи характеризуються більшими часовими витратами, тоді як оптимізовані підходи забезпечують кращу продуктивність.

Встановлено, що криптосистема Ель-Гамаля за однакових параметрів, як правило, має вищі обчислювальні витрати порівняно з RSA, однак використання оптимізованих методів модулярного експоненціювання дозволяє істотно підвищити ефективність обчислень і забезпечити практичну придатність обох криптосистем при великих розрядностях модуля.

ВИСНОВКИ

У кваліфікаційній роботі розглянуто алгоритми модулярного експоненціювання в асиметричних криптосистемах та оцінено їх ефективність. За результатами виконаного дослідження отримано такі результати:

1. Здійснено аналіз теоретичних основ асиметричних криптосистем RSA та Ель-Гамаля, що дало змогу визначити їх математичні властивості, особливості побудови ключів та залежність стійкості від складності операцій модульного експоненціювання. Установлено, що продуктивність криптографічних процедур безпосередньо визначається вибором методу піднесення до степеня.

2. Досліджено та систематизовано оптимізовані алгоритми модульного експоненціювання, зокрема метод виділення квадрату, метод виділення кубу, векторно-модульний метод, класичний підхід із використанням системи залишкових класів (СЗК) та модифіковану досконалу форму СЗК.

3. Розроблено алгоритмічне забезпечення криптосистем RSA та Ель-Гамаля на основі оптимізованих методів модульного експоненціювання, що дало змогу формально описати кожен етап шифрування та розшифрування з урахуванням вибраних алгоритмів та побудувати структурні схеми реалізації.

4. Створено програмний засіб на мові Python для дослідження ефективності різних методів модульного експоненціювання, який забезпечує автоматизоване вимірювання часу виконання, порівняння результатів та аналіз масштабованості.

5. На основі експериментальних досліджень отримано порівняльні результати, які підтвердили, що методи, засновані на системі залишкових класів та її модифікованій формі, демонструють найвищу продуктивність при великих значеннях модулів, тоді як класичні бінарні методи залишаються оптимальними для середніх розрядностей. Векторно-модульний підхід забезпечує стабільний приріст продуктивності за умови можливості паралельних обчислень..

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Digicert Developer Portal. Asymmetric Algorithms. [Електронний ресурс]. – Режим доступу: https://dev.digicert.com/en/trustcore-sdk/nanocrypto/asymmetric-algorithms.html

2. IBM Think. Asymmetric Encryption. [Електронний ресурс]. – Режим доступу: https://www.ibm.com/think/topics/asymmetric-encryption

3. ScienceDirect. Asymmetric Cryptography. [Електронний ресурс]. – Режим доступу: https://www.sciencedirect.com/topics/computer-science/asymmetric-cryptography

4. SpringerLink. Book: Introduction to Cryptography and Coding Theory. [Електронний ресурс]. – Режим доступу: https://link.springer.com/book/10.1007/978-3-642-04101-3

5. Kasianchuk, M.M.; Karpinsky, M.; Kazmirchuk, S. Методологія опрацювання багаторозрядних чисел в асиметричних криптосистемах. – Захист інформації, 2019. – Т.21, №2. – С. 65–73.

6. Kozaczko, D.; Kasianchuk, M.; Yakymenko, I.; Ivasiev, S. Vector Module Exponential in the Remaining Classes System. – Proceedings of IDAACS–2015, Warsaw, Poland. – V.1. – P. 161–163.

7. Nykolaychuk, Ya. M.; Kasianchuk, M. M.; Yakymenko, I. Z. Theoretical Foundations of the Modified Perfect Form of Residue Number System. – Cybernetics and Systems Analysis, 2016. – P. 219–223.

8. Азаров, Д.С. Злочини у сфері комп'ютерної інформації (кримінально-правове дослідження): монографія. – К.: Атіка, 2007. – 304 с.

9. Глинська, М.Л.; Лісковецький, Д.В.; Івасьєв, С.В. Кібербезпека та комп'ютерно-інтегровані технології (КБКІТ–2019). – Збірник матеріалів наукової конференції. – Тернопіль, 2019. – С. 21–24.

10. Максим'юк А., Касянчук М. Алгоритми та програмний засіб для дослідження модулярного експоненціювання в асиметричних криптосистемах. Матеріали XIV міжнародної науково-практичної конференції «Безпека

інформаційних технологій: ITSEC-2025». Тернопіль (Україна), 2025. С.126.

11. Максим'юк А., Касянчук М. Дослідження алгоритмів модулярного експоненціювання в асиметричних криптосистемах. Матеріали IV Міжнародної науково-практичної конференції «Кібербезпека державних інституцій та подолання кризових станів». Київ-Тернопіль-Краків. 2025. С. 465–466.

12. Бурячок, В.Л. Кібернетична безпека – головний фактор сталого розвитку сучасного інформаційного суспільства. – Сучасна спеціальна техніка, 2011. – №3 (26). – С. 104–114.

13. Ivasiev, S.; Yakymenko, I.; Kasianchuk, M.; Shevchuk, R.; Tymoshenko, L. The Method of Factorizing Multi-Digit Numbers Based on the Operation of Adding Odd Numbers. – Advanced Computer Information Technology (ACIT–2018), Ceske Budejovice, 2018. – P. 232–235.

14. Zadiraka, V.; Yakymenko, I.; Kasianchuk, M.; Ivasiev, S. Theoretical and Numerical Krestenson's Basis and Its Application to Cryptographic Problems. – In: Computer Technologies in Information Security. – Ternopil: Kart-blansh, 2015. – P. 216–260.

15. Даник Ю.Г.; Бугайчук К.Л. Основи кібербезпеки. – Житомир: ЖВІ НАУ, 2018. – 320 с.

16. Stallings, W. Cryptography and Network Security: Principles and Practice. – 7th ed. – Pearson, 2017.

17. Волокітін, А.В.; Маношкин, А.П.; Солдатенков, А.В.; Савченко, С.А.; Петров, Ю.А. Інформаційна безпека державних організацій і комерційних фірм. – К.: Юніор, 2012. – 303 с.

18. Persson, A.; Bengtsson, L. Forward and Reverse Converters and Moduli Set Selection in Signed-Digit Residue Number Systems. – J. Signal Process. Syst., 2009. – Vol. 56(1). – P. 1–15.

19. Івасьєв, С.В.; Лісковецький, Д.В.; Шпак, В.Б. Метод збереження простих великорозрядних чисел у базисі Радемахера. – Матеріали конференції «Автоматизація та комп'ютерно-інтегровані технології» (АКІТ–2019). – Тернопіль, 2019. – С. 156–160.

20. Касянчук, М.М. Теорія та математичні закономірності досконалої форми системи залишкових класів. – Праці міжнародного симпозіуму «Питання оптимізації обчислень (ПОО–XXXV)». – Т.1. Київ–Кацивелі, 2009. – С. 306–310.

21. Nykolaychuk, Ya.M.; Kasyanchuk, M.M.; Yakymenko, I.Z. Теорія алгоритмів перетворень китайської теореми про залишки в матрично-розмежованому базисі Радемахера–Крестенсона. – Вісник НУ «Львівська політехніка», 2011. – №688. – С. 118–124.

22. Касянчук, М.М.; Якименко, І.З.; Волинський, О.І.; Івасьєв, С.В. Теоретичні основи аналітики та алгоритми оптимізації обчислень простих чисел. – ПНМК–2010. – В.6, т.1. – С. 33–36.

23. Касянчук, М.М.; Якименко, І.З.; Івасьєв, С.В.; Момотюк, О.В. Експериментальне дослідження програмної реалізації методів пошуку оберненого елемента за модулем. – Інформатика та математичні методи в моделюванні, 2017. – Т.7, №3. – С. 178–186.

24. Поляков О.М.; Корченко О.Г. Інформаційна безпека: теорія і практика. – К.: НАУ, 2016. – 256 с.

25. Tsmots, I.; Teslyuk, V.; Teslyuk, T.; Ihnatyev, I. Basic Components of Neuronetworks with Parallel Vertical Group Data Processing – CSIT 2017. – P. 558–576.

26. Menezes, A.J.; van Oorschot, P.C.; Vanstone, S.A. Handbook of Applied Cryptography. – Boca Raton: CRC Press, 1996. – Chapter 2: Number Theory and Modular Arithmetic.

27. Мельник А.О. Основи криптографії та захисту інформації. – Львів: Видавництво Львівської політехніки, 2011. – С. 112–118.

28. Бабак В.П.; Григорович В.М. Криптографічний захист інформації. – К.: Кондор, 2008. – Розд. 3: Модульна арифметика та експоненціювання.

29. Николайчук, Я.М.; Волинський, О.І.; Кулина, С.В. Теоретичні основи побудови спецпроцесорів у базисі Крестенсона. – Вісник Хмельницького національного університету, 2007. – №3, Т.І(93). – С. 85–90.

30. Якименко, І.З.; Касянчук, М.М.; Тимошенко, Л.М.; Івасьєв, С.В.; Николайчук, Я.М. Алгоритм знаходження системи модулів модифікованої досконалої форми СЗК – МНПК СІЕТ, 2014. – С. 115–117.

ДОДАТОК А

Програмні реалізації для криптосистеми RSA

1. Бінарний метод

```
# run_binpow.py
from datetime import datetime
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
  def generate(self, task_count=1):
        return [
            Task(
                self.random_number(self.bits_count - 3),
                self.random_number(self.bits_count),
                self.moduli_generator.multiply,
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
```

```
        a = task.a
        power = task.x
        add_mul = 1
        if power == 0:
            return 1 % task.p
        while power > 1:
            if power % 2 == 0:
                power //= 2
            else:
                power = (power - 1) // 2
                add_mul *= a
                if add_mul > task.p:
                    add_mul %= task.p
            a = a ** 2 % task.p
        return (a * add_mul) % task.p
    def calc_repeats(bits: int) -> int:
        if bits <= 32:
            return 5000
        if bits <= 128:
            return 2000
        if bits <= 512:
            return 500
        if bits <= 2048:
            return 200
        return 100
    def run_test(tasks, calculator, bits):
        repeats = calc_repeats(bits)
        result = None
        t1 = perf_counter_ns()
        for _ in range(repeats):
            for task in tasks:
                result = calculator.calculate(task)
        t2 = perf_counter_ns()
        avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
        return avg_time_us, result
    if __name__ == "__main__":
        name = "PWR-DEC"  # бінарний
        bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
        input_deltas = [0, 10, 30, 50, 80]
        tasks_count = 1
        def _create_table_printer():
            rows = [name]
            columns = [str(b) for b in bits_count]
            return TablePrinter(rows, columns, 7)
        def _run_tests_for_delta(input_delta):
            result_table = _create_table_printer()
            for bits in bits_count:
                tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
                calc = PowerDecreaseCalculation(None, bits)
                t_us, _ = run_test(tasks, calc, bits)
                result_table.set_data(name, str(bits), str(int(t_us)))
            result_table.print()
        print("Starting research")
        print("Tasks per case count: %d" % tasks_count)
        print("Time in microseconds (average per operation)\n")
        for d in input_deltas:
            print("\tDELTA = %d" % d)
            _run_tests_for_delta(d)
```

2. Триарний метод

```
# run_cubic.py
from datetime import datetime
from random import random
from time import perf_counter_ns
```

```
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        return [
            Task(
                self.random_number(self.bits_count - 3),
                self.random_number(self.bits_count),
                self.moduli_generator.multiply,
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCubicCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
        a = task.a % task.p
        x = task.x
        if x == 0:
            return 1 % task.p
        a2 = (a * a) % task.p
        res = 1
        while x > 0:
            r = x % 3
            x //= 3
            if r == 1:
                res = (res * a) % task.p
```

```
            elif r == 2:
                res = (res * a2) % task.p
            a = (a * a * a) % task.p
            a2 = (a2 * a2 * a2) % task.p
        return res
def calc_repeats(bits: int) -> int:
    if bits <= 32:
        return 5000
    if bits <= 128:
        return 2000
    if bits <= 512:
        return 500
    if bits <= 2048:
        return 200
    return 100
def run_test(tasks, calculator, bits):
    repeats = calc_repeats(bits)
    result = None
    t1 = perf_counter_ns()
    for _ in range(repeats):
        for task in tasks:
            result = calculator.calculate(task)
    t2 = perf_counter_ns()
    avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
    return avg_time_us, result
if __name__ == "__main__":
    name = "DEC-CUB"
    bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
    input_deltas = [0, 10, 30, 50, 80]
    tasks_count = 1
    def _create_table_printer():
        rows = [name]
        columns = [str(b) for b in bits_count]
        return TablePrinter(rows, columns, 7)
    def _run_tests_for_delta(input_delta):
        result_table = _create_table_printer()
        for bits in bits_count:
            tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
            calc = PowerDecreaseCubicCalculation(None, bits)
            t_us, _ = run_test(tasks, calc, bits)
            result_table.set_data(name, str(bits), str(int(t_us)))
        result_table.print()
    print("Starting research")
    print("Tasks per case count: %d" % tasks_count)
    print("Time in microseconds (average per operation)\n")
    for d in input_deltas:
        print("\tDELTA = %d" % d)
        _run_tests_for_delta(d)
```

3. Векторно-модульний метод

```
# run_vecmod.py
from datetime import datetime
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
```

```
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        return [
            Task(
                self.random_number(self.bits_count - 3),
                self.random_number(self.bits_count),
                self.moduli_generator.multiply,
            )
            for _ in range(task_count)
        ]
class VectorModularCalculator:
        def __init__(self, number_factory_class=None, bits_count=8):
        self.bits_count = bits_count
    def calculate(self, task: Task):
        a = task.a % task.p
        x = task.x
        if x == 0:
            return 1 % task.p
        powers = []
        cur = a
        while (1 << len(powers)) <= x:
            powers.append(cur)
            cur = (cur * cur) % task.p
        res = 1
        bit_index = 0
        while x > 0:
            if x & 1:
                res = (res * powers[bit_index]) % task.p
            x >>= 1
            bit_index += 1
        return res
def calc_repeats(bits: int) -> int:
    if bits <= 32:
        return 5000
    if bits <= 128:
```

```
            return 2000
        if bits <= 512:
            return 500
        if bits <= 2048:
            return 200
        return 100
    def run_test(tasks, calculator, bits):
        repeats = calc_repeats(bits)
        result = None
        t1 = perf_counter_ns()
        for _ in range(repeats):
            for task in tasks:
                result = calculator.calculate(task)
        t2 = perf_counter_ns()
        avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
        return avg_time_us, result
    if __name__ == "__main__":
        name = "VEC-MOD"
        bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
        input_deltas = [0, 10, 30, 50, 80]
        tasks_count = 1
        def _create_table_printer():
            rows = [name]
            columns = [str(b) for b in bits_count]
            return TablePrinter(rows, columns, 7)
        def _run_tests_for_delta(input_delta):
            result_table = _create_table_printer()
            for bits in bits_count:
                tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
                calc = VectorModularCalculator(None, bits)
                t_us, _ = run_test(tasks, calc, bits)
                result_table.set_data(name, str(bits), str(int(t_us)))
            result_table.print()
        print("Starting research")
        print("Tasks per case count: %d" % tasks_count)
        print("Time in microseconds (average per operation)\n")
        for d in input_deltas:
            print("\tDELTA = %d" % d)
            _run_tests_for_delta(d)
```

4. СЗК метод

```
# run_rns_classic.py
from datetime import datetime
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
```

```
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        return [
            Task(
                self.random_number(self.bits_count - 3),
                self.random_number(self.bits_count),
                self.moduli_generator.multiply,
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
        a = task.a
        power = task.x
        add_mul = 1
        if power == 0:
            return 1 % task.p
        while power > 1:
            if power % 2 == 0:
                power //= 2
            else:
                power = (power - 1) // 2
                add_mul *= a
                if add_mul > task.p:
                    add_mul %= task.p
            a = a ** 2 % task.p
        return (a * add_mul) % task.p
class Rns:
    def __init__(self, number, moduli_generator):
        self.moduli_generator = moduli_generator
        self.number = [number % m for m in self.moduli_generator.moduli]
    def egcd(self, a, b):
        x, y, u, v = 0, 1, 1, 0
        while a != 0:
            q, r = b // a, b % a
            m, n = x - u * q, y - v * q
            b, a, x, y, u, v = a, r, u, v, m, n
        return b, x, y
    def _reciprocal(self, a, m):
        gcd, x, y = self.egcd(a, m)
        if gcd != 1:
            return None
```

```
        else:
            return x % m
    def get_number(self):
        n = 0
        M = self.moduli_generator.multiply
        for p, ni in zip(self.moduli_generator.moduli, self.number):
            Mi = M // p
            inv = self._reciprocal(Mi, p)
            n += ni * Mi * inv
        return n % M
    def multiply(self, argument):
        for i in range(len(self.moduli_generator.moduli)):
            self.number[i] = (
                self.number[i] * argument.number[i]
            ) % self.moduli_generator.moduli[i]
        return self
class RnsModuliGenerator:
    def __init__(self, bits=8):
        k = 2 ** int((bits + 0.2) / 3.0 + 1)
        p2 = k + 1 if k % 2 == 0 else k
        c = 2 ** (bits // 4)
        self.moduli = (p2 - c, p2, p2 + c)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class RnsNumberFactory:
    def __init__(self, initial_bits=8):
        self._moduli_generator = RnsModuliGenerator(initial_bits)
    def create_number(self, number):
        return Rns(number, self._moduli_generator)
class RnsPowerDecreaseCalculator:
    def __init__(self, rns_class_factory, bits_count):
        self.rns_factory = rns_class_factory(bits_count)
        self.power_decrease = PowerDecreaseCalculation()
    def calculate(self, task):
        a = self.rns_factory.create_number(task.a)
        for i in range(len(a.number)):
            sub_task = Task(a.number[i], task.x, a.moduli_generator.moduli[i])
            a.number[i] = self.power_decrease.calculate(sub_task)
        return a.get_number() % task.p
def calc_repeats(bits: int) -> int:
    if bits <= 32:
        return 5000
    if bits <= 128:
        return 2000
    if bits <= 512:
        return 500
    if bits <= 2048:
        return 200
    return 100  # для 4096 і вище
def run_test(tasks, calculator, bits):
    repeats = calc_repeats(bits)
    result = None
    t1 = perf_counter_ns()
    for _ in range(repeats):
        for task in tasks:
            result = calculator.calculate(task)
    t2 = perf_counter_ns()
    avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
    return avg_time_us, result
if __name__ == "__main__":
    name = "SMP-RNS"  # звичайна СЗК
    bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
    input_deltas = [0, 10, 30, 50, 80]
    tasks_count = 1
```

```
    def _create_table_printer():
        rows = [name]
        columns = [str(b) for b in bits_count]
        return TablePrinter(rows, columns, 7)
    def _run_tests_for_delta(input_delta):
        result_table = _create_table_printer()
        for bits in bits_count:
            tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
            calc = RnsPowerDecreaseCalculator(RnsNumberFactory, bits)
            t_us, _ = run_test(tasks, calc, bits)
            # округляємо до цілого мкс для таблиця
            result_table.set_data(name, str(bits), str(int(t_us)))
        result_table.print()
    print("Starting research")
    print("Tasks per case count: %d" % tasks_count)
    print("Time in microseconds (average per operation)\n")
    for d in input_deltas:
        print("\tDELTA = %d" % d)
        _run_tests_for_delta(d)
```

5. МДФ-СЗК метод

```
# run_rns_mpf.py
from datetime import datetime
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
        self.base_numbers = []
        for m in self.moduli:
            b = self.multiply // m
            if b % m != 1:
                b *= -1
            self.base_numbers.append(b)
class TaskGenerator:
```

```
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        return [
            Task(
                self.random_number(self.bits_count - 3),
                self.random_number(self.bits_count),
                self.moduli_generator.multiply,
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
        a = task.a
        power = task.x
        add_mul = 1
        if power == 0:
            return 1 % task.p
        while power > 1:
            if power % 2 == 0:
                power //= 2
            else:
                power = (power - 1) // 2
                add_mul *= a
                if add_mul > task.p:
                    add_mul %= task.p
            a = a ** 2 % task.p
        return (a * add_mul) % task.p
class MpfRns:
    def __init__(self, number, moduli_generator: MpfRnsModuliGenerator):
        self.moduli_generator = moduli_generator
        self.number = [number % m for m in moduli_generator.moduli]
    def get_number(self):
        # відновлення без інверсій
        s = 0
        for ni, bi in zip(self.number, self.moduli_generator.base_numbers):
            s += ni * bi
        return s % self.moduli_generator.multiply
    def multiply(self, argument):
        for i, ni in enumerate(self.number):
            self.number[i] = (
                self.number[i] * argument.number[i]
            ) % self.moduli_generator.moduli[i]
        return self
class MpfRnsNumberFactory:
    def __init__(self, initial_bits=8):
        self._moduli_generator = MpfRnsModuliGenerator(initial_bits)
    def create_number(self, number):
        return MpfRns(number, self._moduli_generator)
class RnsPowerDecreaseCalculator:
    def __init__(self, rns_class_factory, bits_count):
        self.rns_factory = rns_class_factory(bits_count)
        self.power_decrease = PowerDecreaseCalculation()
    def calculate(self, task):
        a = self.rns_factory.create_number(task.a)
        for i in range(len(a.number)):
            sub_task = Task(a.number[i], task.x, a.moduli_generator.moduli[i])
```

```
            a.number[i] = self.power_decrease.calculate(sub_task)
        return a.get_number() % task.p
def calc_repeats(bits: int) -> int:
    if bits <= 32:
        return 5000
    if bits <= 128:
        return 2000
    if bits <= 512:
        return 500
    if bits <= 2048:
        return 200
    return 100  # для 4096 і вище
def run_test(tasks, calculator, bits):
    repeats = calc_repeats(bits)
    result = None
    t1 = perf_counter_ns()
    for _ in range(repeats):
        for task in tasks:
            result = calculator.calculate(task)
    t2 = perf_counter_ns()
    avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
    return avg_time_us, result
if __name__ == "__main__":
    name = "PWR-RNS"
    bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
    input_deltas = [0, 10, 30, 50, 80]
    tasks_count = 1
    def _create_table_printer():
        rows = [name]
        columns = [str(b) for b in bits_count]
        return TablePrinter(rows, columns, 7)
    def _run_tests_for_delta(input_delta):
        result_table = _create_table_printer()
        for bits in bits_count:
            tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
            calc = RnsPowerDecreaseCalculator(MpfRnsNumberFactory, bits)
            t_us, _ = run_test(tasks, calc, bits)
            # округляємо до цілого мкс для таблиця
            result_table.set_data(name, str(bits), str(int(t_us)))
        result_table.print()
    print("Starting research")
    print("Tasks per case count: %d" % tasks_count)
    print("Time in microseconds (average per operation)\n")
    for d in input_deltas:
        print("\tDELTA = %d" % d)
        _run_tests_for_delta(d)
```

1. Бінарний метод

```
# run_elgamal_binpow.py
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    a^x mod p, де в Ель-Гамаля:
      a – може відповідати q, h або C1,
      x – a, r або (p-1-a),
      p – модуль.
    """
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m  # загальний модуль p
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        p = self.moduli_generator.multiply
        return [
            Task(
                self.random_number(self.bits_count - 3),  # a (q, h, C1)
                self.random_number(self.bits_count),      # x (a, r, ...)
                p
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
```

```
            pass
        def calculate(self, task):
            a = task.a % task.p
            power = task.x
            add_mul = 1
            if power == 0:
                return 1 % task.p
            while power > 1:
                if power % 2 == 0:
                    power //= 2
                else:
                    power = (power - 1) // 2
                    add_mul *= a
                    if add_mul > task.p:
                        add_mul %= task.p
                a = (a * a) % task.p
            return (a * add_mul) % task.p
    def calc_repeats(bits: int) -> int:
        if bits <= 32:
            return 5000
        if bits <= 128:
            return 2000
        if bits <= 512:
            return 500
        if bits <= 2048:
            return 200
        return 100
    def run_test(tasks, calculator, bits):
        repeats = calc_repeats(bits)
        result = None
        t1 = perf_counter_ns()
        for _ in range(repeats):
            for task in tasks:
                result = calculator.calculate(task)
        t2 = perf_counter_ns()
        avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
        return avg_time_us, result
    if __name__ == "__main__":
        name = "ELG-PWR"
        bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
        input_deltas = [0, 10, 30, 50, 80]
        tasks_count = 1
        def _create_table_printer():
            rows = [name]
            columns = [str(b) for b in bits_count]
            return TablePrinter(rows, columns, 7)
        def _run_tests_for_delta(input_delta):
            result_table = _create_table_printer()
            for bits in bits_count:
                tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
                calc = PowerDecreaseCalculation(None, bits)
                t_us, _ = run_test(tasks, calc, bits)
                result_table.set_data(name, str(bits), str(int(t_us)))
            result_table.print()
        print("Starting research (ElGamal, binary method)")
        print("Tasks per case count: %d" % tasks_count)
        print("Time in microseconds (average per operation)\n")
        for d in input_deltas:
            print("\tDELTA = %d" % d)
            _run_tests_for_delta(d)
```

2. Триарний метод

```
# run_elgamal_cubic.py
from random import random
```

```
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        p = self.moduli_generator.multiply
        return [
            Task(
                self.random_number(self.bits_count - 3),  # a (q, h, C1)
                self.random_number(self.bits_count),      # x (a, r, ...)
                p
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCubicCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
        a = task.a % task.p
        x = task.x
        if x == 0:
            return 1 % task.p
        a2 = (a * a) % task.p
        res = 1
        while x > 0:
            r = x % 3
            x //= 3
```

```
            if r == 1:
                res = (res * a) % task.p
            elif r == 2:
                res = (res * a2) % task.p
            a = (a * a * a) % task.p
            a2 = (a2 * a2 * a2) % task.p
        return res
def calc_repeats(bits: int) -> int:
    if bits <= 32:
        return 5000
    if bits <= 128:
        return 2000
    if bits <= 512:
        return 500
    if bits <= 2048:
        return 200
    return 100
def run_test(tasks, calculator, bits):
    repeats = calc_repeats(bits)
    result = None
    t1 = perf_counter_ns()
    for _ in range(repeats):
        for task in tasks:
            result = calculator.calculate(task)
    t2 = perf_counter_ns()
    avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
    return avg_time_us, result
if __name__ == "__main__":
    name = "ELG-CUB"
    bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
    input_deltas = [0, 10, 30, 50, 80]
    tasks_count = 1
    def _create_table_printer():
        rows = [name]
        columns = [str(b) for b in bits_count]
        return TablePrinter(rows, columns, 7)
    def _run_tests_for_delta(input_delta):
        result_table = _create_table_printer()
        for bits in bits_count:
            tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
            calc = PowerDecreaseCubicCalculation(None, bits)
            t_us, _ = run_test(tasks, calc, bits)
            result_table.set_data(name, str(bits), str(int(t_us)))
        result_table.print()
    print("Starting research (ElGamal, cubic method)")
    print("Tasks per case count: %d" % tasks_count)
    print("Time in microseconds (average per operation)\n")
    for d in input_deltas:
        print("\tDELTA = %d" % d)
        _run_tests_for_delta(d)
```

3. Векторно-модулярний метод

```
# run_elgamal_vecmod.py
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
```

```
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        p = self.moduli_generator.multiply
        return [
            Task(
                self.random_number(self.bits_count - 3),  # a (q, h, C1)
                self.random_number(self.bits_count),      # x (a, r, ...)
                p
            )
            for _ in range(task_count)
        ]
class VectorModularCalculator:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
        a = task.a % task.p
        x = task.x
        if x == 0:
            return 1 % task.p
        powers = []
        cur = a
        while (1 << len(powers)) <= x:
            powers.append(cur)
            cur = (cur * cur) % task.p
        res = 1
        bit_index = 0
        while x > 0:
            if x & 1:
                res = (res * powers[bit_index]) % task.p
            x >>= 1
            bit_index += 1
        return res
def calc_repeats(bits: int) -> int:
    if bits <= 32:
```

```
        return 5000
    if bits <= 128:
        return 2000
    if bits <= 512:
        return 500
    if bits <= 2048:
        return 200
    return 100  # для 4096 і вище
def run_test(tasks, calculator, bits):
    repeats = calc_repeats(bits)
    result = None
    t1 = perf_counter_ns()
    for _ in range(repeats):
        for task in tasks:
            result = calculator.calculate(task)
    t2 = perf_counter_ns()
    avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
    return avg_time_us, result
if __name__ == "__main__":
    name = "ELG-VEC"  # векторно-модульний метод для Ель-Гамаля
    bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
    input_deltas = [0, 10, 30, 50, 80]
    tasks_count = 1
    def _create_table_printer():
        rows = [name]
        columns = [str(b) for b in bits_count]
        return TablePrinter(rows, columns, 7)
    def _run_tests_for_delta(input_delta):
        result_table = _create_table_printer()
        for bits in bits_count:
            tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
            calc = VectorModularCalculator(None, bits)
            t_us, _ = run_test(tasks, calc, bits)
            result_table.set_data(name, str(bits), str(int(t_us)))
        result_table.print()
    print("Starting research (ElGamal, vector-modular method)")
    print("Tasks per case count: %d" % tasks_count)
    print("Time in microseconds (average per operation)\n")
    for d in input_deltas:
        print("\tDELTA = %d" % d)
        _run_tests_for_delta(d)
```

4. Метод СЗК

```
# run_elgamal_rns_classic.py
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
```

```
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m  # загальний модуль p
class TaskGenerator:
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        p = self.moduli_generator.multiply
        return [
            Task(
                self.random_number(self.bits_count - 3),
                self.random_number(self.bits_count),
                p
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
        a = task.a
        power = task.x
        add_mul = 1
        if power == 0:
            return 1 % task.p
        while power > 1:
            if power % 2 == 0:
                power //= 2
            else:
                power = (power - 1) // 2
                add_mul *= a
                if add_mul > task.p:
                    add_mul %= task.p
            a = a ** 2 % task.p
        return (a * add_mul) % task.p
class Rns:
    def __init__(self, number, moduli_generator):
        self.moduli_generator = moduli_generator
        self.number = [number % m for m in self.moduli_generator.moduli]
    def egcd(self, a, b):
        x, y, u, v = 0, 1, 1, 0
        while a != 0:
            q, r = b // a, b % a
            m, n = x - u * q, y - v * q
            b, a, x, y, u, v = a, r, u, v, m, n
        return b, x, y
    def _reciprocal(self, a, m):
        gcd, x, y = self.egcd(a, m)
```

```
        if gcd != 1:
            return None
        else:
            return x % m
    def get_number(self):
        n = 0
        M = self.moduli_generator.multiply
        for p, ni in zip(self.moduli_generator.moduli, self.number):
            Mi = M // p
            inv = self._reciprocal(Mi, p)
            n += ni * Mi * inv
        return n % M
    def multiply(self, argument):
        for i in range(len(self.moduli_generator.moduli)):
            self.number[i] = (
                self.number[i] * argument.number[i]
            ) % self.moduli_generator.moduli[i]
        return self
class RnsModuliGenerator:
    def __init__(self, bits=8):
        k = 2 ** int((bits + 0.2) / 3.0 + 1)
        p2 = k + 1 if k % 2 == 0 else k
        c = 2 ** (bits // 4)
        self.moduli = (p2 - c, p2, p2 + c)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
class RnsNumberFactory:
    def __init__(self, initial_bits=8):
        self._moduli_generator = RnsModuliGenerator(initial_bits)
    def create_number(self, number):
        return Rns(number, self._moduli_generator)
class RnsPowerDecreaseCalculator:
    def __init__(self, rns_class_factory, bits_count):
        self.rns_factory = rns_class_factory(bits_count)
        self.power_decrease = PowerDecreaseCalculation()
    def calculate(self, task):
        a = self.rns_factory.create_number(task.a)
        for i in range(len(a.number)):
            sub_task = Task(a.number[i], task.x, a.moduli_generator.moduli[i])
            a.number[i] = self.power_decrease.calculate(sub_task)
        return a.get_number() % task.p
def calc_repeats(bits: int) -> int:
    if bits <= 32:
        return 5000
    if bits <= 128:
        return 2000
    if bits <= 512:
        return 500
    if bits <= 2048:
        return 200
    return 100  # для 4096 і вище
def run_test(tasks, calculator, bits):
    repeats = calc_repeats(bits)
    result = None
    t1 = perf_counter_ns()
    for _ in range(repeats):
        for task in tasks:
            result = calculator.calculate(task)
    t2 = perf_counter_ns()
    avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
    return avg_time_us, result
if __name__ == "__main__":
    name = "ELG-RNS"  # звичайна СЗК для Ель-Гамаля
    bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
```

```
    input_deltas = [0, 10, 30, 50, 80]
    tasks_count = 1
    def _create_table_printer():
        rows = [name]
        columns = [str(b) for b in bits_count]
        return TablePrinter(rows, columns, 7)
    def _run_tests_for_delta(input_delta):
        result_table = _create_table_printer()
        for bits in bits_count:
            tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
            calc = RnsPowerDecreaseCalculator(RnsNumberFactory, bits)
            t_us, _ = run_test(tasks, calc, bits)
            result_table.set_data(name, str(bits), str(int(t_us)))
        result_table.print()
    print("Starting research (ElGamal, classic RNS)")
    print("Tasks per case count: %d" % tasks_count)
    print("Time in microseconds (average per operation)\n")
    for d in input_deltas:
        print("\tDELTA = %d" % d)
        _run_tests_for_delta(d)
```

5. МДФ-СЗК метод

```
# run_elgamal_rns_mpf.py
from random import random
from time import perf_counter_ns
class TablePrinter:
    def __init__(self, rows, columns, cell_width=7):
        self.rows = rows
        self.cols = columns
        self.cell_width = cell_width
        self.data = {r: {c: "" for c in self.cols} for r in self.rows}
    def set_data(self, row, col, val):
        self.data[row][col] = val
    def print(self):
        header = "".ljust(self.cell_width) + "".join(
            c.rjust(self.cell_width) for c in self.cols
        )
        print(header)
        for r in self.rows:
            line = r.ljust(self.cell_width) + "".join(
                self.data[r][c].rjust(self.cell_width) for c in self.cols
            )
            print(line)
        print()
class Task:
    def __init__(self, a, x, p):
        self.a = a
        self.x = x
        self.p = p
class MpfRnsModuliGenerator:
    def __init__(self, bits):
        k = 2 ** (int((bits - 2) / 3.0) + 1) + 1
        p1 = k + 1
        self.moduli = (p1, 2 * p1 - 1, 2 * p1 + 1)
        self.multiply = 1
        for m in self.moduli:
            self.multiply *= m
        self.base_numbers = []
        for m in self.moduli:
            b = self.multiply // m
            if b % m != 1:
                b *= -1
            self.base_numbers.append(b)
class TaskGenerator:
```

```
    def __init__(self, bits_count=8, delta=0):
        self.bits_count = bits_count
        self.delta = delta
        self.moduli_generator = MpfRnsModuliGenerator(bits_count)
    def random_number(self, bits_count):
        max_number = (1 << bits_count) - 1
        delta_diff = round(random() * round(bits_count * self.delta / 100))
        return max_number - delta_diff
    def generate(self, task_count=1):
        p = self.moduli_generator.multiply
        return [
            Task(
                self.random_number(self.bits_count - 3),
                self.random_number(self.bits_count),
                p
            )
            for _ in range(task_count)
        ]
class PowerDecreaseCalculation:
    def __init__(self, number_factory_class=None, bits_count=8):
        pass
    def calculate(self, task):
        a = task.a
        power = task.x
        add_mul = 1
        if power == 0:
            return 1 % task.p
        while power > 1:
            if power % 2 == 0:
                power //= 2
            else:
                power = (power - 1) // 2
                add_mul *= a
                if add_mul > task.p:
                    add_mul %= task.p
            a = a ** 2 % task.p
        return (a * add_mul) % task.p
class MpfRns:
    def __init__(self, number, moduli_generator: MpfRnsModuliGenerator):
        self.moduli_generator = moduli_generator
        self.number = [number % m for m in moduli_generator.moduli]
    def get_number(self):
        s = 0
        for ni, bi in zip(self.number, self.moduli_generator.base_numbers):
            s += ni * bi
        return s % self.moduli_generator.multiply
    def multiply(self, argument):
        for i, ni in enumerate(self.number):
            self.number[i] = (
                self.number[i] * argument.number[i]
            ) % self.moduli_generator.moduli[i]
        return self
class MpfRnsNumberFactory:
    def __init__(self, initial_bits=8):
        self._moduli_generator = MpfRnsModuliGenerator(initial_bits)
    def create_number(self, number):
        return MpfRns(number, self._moduli_generator)
class RnsPowerDecreaseCalculator:
    def __init__(self, rns_class_factory, bits_count):
        self.rns_factory = rns_class_factory(bits_count)
        self.power_decrease = PowerDecreaseCalculation()
    def calculate(self, task):
        a = self.rns_factory.create_number(task.a)
        for i in range(len(a.number)):
            sub_task = Task(a.number[i], task.x, a.moduli_generator.moduli[i])
```

```
            a.number[i] = self.power_decrease.calculate(sub_task)
        return a.get_number() % task.p
def calc_repeats(bits: int) -> int:
    if bits <= 32:
        return 5000
    if bits <= 128:
        return 2000
    if bits <= 512:
        return 500
    if bits <= 2048:
        return 200
    return 100  # для 4096 і вище
def run_test(tasks, calculator, bits):
    repeats = calc_repeats(bits)
    result = None
    t1 = perf_counter_ns()
    for _ in range(repeats):
        for task in tasks:
            result = calculator.calculate(task)
    t2 = perf_counter_ns()
    avg_time_us = (t2 - t1) / (repeats * len(tasks) * 1000.0)
    return avg_time_us, result
if __name__ == "__main__":
    name = "ELG-MPF"  # МДФ-СЗК для Ель-Гамаля
    bits_count = (8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
    input_deltas = [0, 10, 30, 50, 80]
    tasks_count = 1
    def _create_table_printer():
        rows = [name]
        columns = [str(b) for b in bits_count]
        return TablePrinter(rows, columns, 7)
    def _run_tests_for_delta(input_delta):
        result_table = _create_table_printer()
        for bits in bits_count:
            tasks = TaskGenerator(bits, input_delta).generate(tasks_count)
            calc = RnsPowerDecreaseCalculator(MpfRnsNumberFactory, bits)
            t_us, _ = run_test(tasks, calc, bits)
            result_table.set_data(name, str(bits), str(int(t_us)))
        result_table.print()
    print("Starting research (ElGamal, MPF-RNS)")
    print("Tasks per case count: %d" % tasks_count)
    print("Time in microseconds (average per operation)\n")
    for d in input_deltas:
        print("\tDELTA = %d" % d)
        _run_tests_for_delta(d)
```

ДОДАТОК Б
Копії публікацій

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
UNIVERSITY OF THE NATIONAL EDUCATION COMMISSION, POLAND
TECHNICAL UNIVERSITY IN PRAGUE, CZECH REPUBLIC
Наукова школа "Кібербезпека"
Навчально-науковий інститут Кібербезпеки та захисту інформації ДУІКТ
Кафедра кібербезпеки ЗУНУ
ГО «АСОЦІАЦІЯ СПЕЦІАЛІСТІВ КІБЕРБЕЗПЕКИ»
ГО «АВТОМАТИЗАЦІЯ І КІБЕРБЕЗПЕКА»

ITSec-2025

Безпека інформаційних технологій

МАТЕРІАЛИ

XIV Міжнародної науково-технічної конференції

22-24 травня 2025
м. Тернопіль (Україна)

Алгоритми та програмний засіб для дослідження модулярного експоненціювання в асиметричних криптосистемах

УДК 004.41

Анжеліна Максим'юк[1], Михайло Касянчук[2]
Західноукраїнський національний університет,
[1]maksymjukanjelina@gmail.com, [2]kasyanchuk@ukr.net

Асиметричні криптосистеми є фундаментальним елементом сучасної криптографії, які забезпечують конфіденційність, цілісність та доступність даних [1]. Однак ефективність цих систем значною мірою залежить від продуктивності операцій модулярного експоненціювання, що лежать в основі алгоритмів шифрування та розшифрування. Існуючі підходи до оптимізації модулярного експоненціювання характеризуються різними показниками ефективності, що вимагає детального порівняльного аналізу.

Метою роботи є розробка програмного засобу для дослідження та порівняльного аналізу алгоритмів модулярного експоненціювання в асиметричних криптосистемах RSA та Ель-Гамаля, визначення їх ефективності за критеріями часової та обчислювальної складності, а також формування практичних рекомендацій щодо їх застосування.

Наукова новизна та практичне значення дослідження полягає у розробці та реалізації програмного засобу на основі мови програмування Python, який дозволяє досліджувати ефективність різних алгоритмів модулярного експоненціювання (зокрема векторно-модулярного методу, методу виділення квадрату, методу виділення кубу та прямого піднесення до степеня) у контексті практичного застосування в асиметричних криптосистемах.

Для досягнення поставленої мети було розроблено програмний засіб на мові Python, що реалізує основні алгоритми модулярного експоненціювання та забезпечує їх порівняльний аналіз. У рамках дослідження:

1) проаналізовано теоретичні основи асиметричних криптосистем та модулярного експоненціювання;
2) реалізовано криптосистеми RSA та Ель-Гамаля з використанням різних методів модулярного експоненціювання;
3) розроблено архітектуру програмного засобу, що дозволяє оцінювати часову та обчислювальну складність алгоритмів;
4) проведено порівняльний аналіз ефективності реалізованих методів модулярного експоненціювання.

У результаті дослідження було реалізовано програмний інструмент для вивчення особливостей модулярного експоненціювання в асиметричних криптосистемах. Проведений аналіз продемонстрував, що вибір алгоритму експоненціювання істотно впливає на продуктивність криптографічних операцій. Отримані результати можуть бути використані для оптимізації криптографічних рішень у реальних інформаційних системах.

1. Nykolaychuk Ya.M., Yakymenko I.Z., Vozna N.Ya., and Kasianchuk M.M. Residue Number System Asymmetric Cryptoalgorithms. Cybernetics and Systems Analysis. 2022, Vol. 58, No. 4, P.611-618.

Державна служба спеціального зв'язку
та захисту інформації України

Національний технічний університет України
"Київський політехнічний інститут імені Ігоря Сікорського"

Західноукраїнський національний університет

Університет комісії народної освіти в Кракові

Інститут спеціального зв'язку та захисту інформації
Національного технічного університету України
"Київський політехнічний інститут імені Ігоря Сікорського"

ПРОГРАМА
IV Міжнародної науково-практичної конференції "Кібербезпека державних інституцій та подолання кризових станів"

18 листопада 2025 року

Київ – Тернопіль – Краків

Анжеліна МАКСИМ'ЮК;
Михайло КАСЯНЧУК, д.т.н., професор

ДОСЛІДЖЕННЯ АЛГОРИТМІВ МОДУЛЯРНОГО ЕКСПОНЕНЦІЮВАННЯ В АСИМЕТРИЧНИХ КРИПТОСИСТЕМАХ

Анотація. У роботі розглянуто особливості реалізації модулярного експоненціювання, проведено порівняльний аналіз алгоритмів різних типів і визначено їх ефективність залежно від розрядності чисел. Результати дослідження можуть бути використані для підвищення швидкодії асиметричних криптографічних систем.

Summary. The paper considers the peculiarities of modular exponentiation implementation, conducts a comparative analysis of different types of algorithms, and determines their efficiency depending on the number of digits. The research results can be used to improve the performance of asymmetric cryptographic systems.

Ключові слова: модулярне експоненціювання, асиметрична криптосистема, криптосистема RSA, бінарний метод, модифікована СЗК.

Модулярне експоненціювання є однією з найважливіших математичних операцій, що лежать в основі роботи асиметричних криптосистем, зокрема алгоритму RSA. Її ефективність безпосередньо впливає на швидкодію процесів шифрування, дешифрування та цифрового підпису. У зв'язку з цим виникає потреба у виборі таких методів обчислення, які забезпечують оптимальний баланс між швидкістю виконання й обсягом необхідних обчислювальних ресурсів.

Під час вибору методів модулярного експоненціювання основну увагу було зосереджено на забезпеченні співставного порядку часу виконання для різних алгоритмів, а також на можливості їх застосування до чисел великої розрядності (до 4096 біт). З огляду на це, низку методів було свідомо виключено з дослідження через надмірну обчислювальну складність або нестійкість при великих числах. Зокрема, не застосовувались методи прямого піднесення до степеня, послідовного множення справа наліво чи зліва направо, класичний підхід системи залишкових класів (СЗК).

Для проведення експериментальних досліджень обрано наступні методи: бінарний, пониження степеня за допомогою кубів, системи залишкових класів та модифікованої досконалої форми системи залишкових класів.

Бінарний метод (метод квадратів) – передбачає поетапне зниження степеня через послідовне зведення в квадрат, що дозволяє суттєво скоротити кількість множень.

Триарний метод (метод кубів) – модифікація попереднього, де використовується зведення до кубу, що підвищує швидкість при великих показниках степеня.

Метод СЗК – полягає в розбитті числа на залишки за кількома взаємно простими модулями, виконанні піднесення до степеня для кожного залишку окремо та подальшому відновленні результату за китайською теоремою про залишки.

Метод модифікованої досконалої форми (МДФ) СЗК – вдосконалений підхід, що усуває потребу пошуку оберненого елемента, завдяки чому час виконання зменшується, особливо при великій розрядності.

Для кожного методу було проведено серію експериментів з різними розрядностями (від 8 до 4096 біт). Для підвищення достовірності результатів програма запускалася кілька разів, після чого обчислювався середній час виконання операції. Крім того, досліджено вплив ваги Хемінга (кількості одиничних бітів у двійковому поданні числа) на тривалість обчислення.

Результати показали, що при малих розрядностях (до 256 біт) найвищу швидкість забезпечують бінарний та триарний методи, тоді як СЗК та МДФ СЗК демонструють перевагу при обробці великих чисел (від 1024 біт і вище). Це свідчить про те, що ефективність алгоритму безпосередньо залежить від розрядності вхідних даних і структури обчислюваних чисел.

Отримані результати підтвердили доцільність використання методів на основі СЗК у високопродуктивних криптографічних застосуваннях, де важливим є оброблення великих обсягів даних із мінімальними часовими витратами. У той же час, при роботі з меншою розрядністю перевагу варто надавати бінарному або триарному алгоритмам, які забезпечують найменшу затримку обчислення.

Висновки. Проведене дослідження показало, що час виконання операції модулярного експоненціювання істотно залежить від вибору алгоритмічного підходу та розрядності чисел. Результати експериментів можуть бути використані для оптимізації криптографічних систем, які реалізують алгоритм RSA та подібні схеми шифрування.