НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«До захисту допущено»
Науковий керівник кафедри
_____ Іван ДИЧКА
«___»_____2020 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою
«Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

спеціальності 121 Інженерія програмного забезпечення

на тему: «Модифікований метод та програмне забезпечення для асиметричного шифрування»

Виконав:
студент IV курсу, групи КП-62
Квітка Олександр Вячеславович _____

Керівник:
доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Консультант з нормоконтролю:
доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
доцент кафедри СПіСКС, к.т.н., доцент,
Клятченко Ярослав Михайлович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних та інформаційно-пошукових систем»

«ЗАТВЕРДЖУЮ»

Науковий керівник кафедри

_____ Іван ДИЧКА

«___»_____ 2019 р.

ЗАВДАННЯ

на дипломну роботу студенту

Квітці Олександру Вячеславовичу

1. Тема роботи «Модифікований метод та програмне забезпечення для асиметричного шифрування», керівник роботи Онай Микола Володимирович, доцент кафедри ПЗКС, к.т.н., доцент, затверджені наказом по університету від «25» травня 2020 р. №1181-с

2. Термін подання студентом роботи «___» червня 2020 р.

3. Вихідні дані до роботи:
   - алгоритми реалізації класичних методів асиметричного шифрування.

4. Зміст роботи:
   - аналіз існуючих рішень;
   - розроблення модифікованого методу асиметричного шифрування;
   - обґрунтування засобів реалізації;
   - розроблення програмного забезпечення для реалізації та аналізу продуктивності роботи запропонованого модифікованого методу шифрування.

5. Перелік обов'язкового ілюстративного матеріалу:
   - схема роботи класичного методу;
   - схема роботи модифікованого методу;
   - графіки, що ілюструють аналіз ефективності роботи методу;
   - графіки, що ілюструють порівняння модифікованого методу і класичного.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент кафедри ПЗКС, к.т.н., доцент		

7. Дата видачі завдання «31» жовтня 2019 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вивчення літератури за тематикою роботи та збір даних	16.11.2019	
2.	Проведення порівняльного аналізу методів асиметричного шифрування	09.12.2019	
3.	Підготовка матеріалів першого розділу дипломної роботи	30.12.2019	
4.	Розробка модифікованого методу асиметричного шифрування	16.01.2020	
5.	Підготовка матеріалів другого розділу дипломної роботи	10.02.2020	
6.	Підготовка тез доповіді на конференцію	20.02.2020	
7.	Підготовка матеріалів третього розділу дипломної роботи	10.03.2020	
8.	Підготовка матеріалів четвертого розділу дипломної роботи	11.04.2020	
9.	Підготовка графічної частини дипломної роботи	19.05.2020	
10.	Оформлення дипломної роботи	26.05.2020	

Студент Олександр КВІТКА

Керівник роботи Микола ОНАЙ

АНОТАЦІЯ

Дана дипломна робота присвячена розробленню модифікованого методу асиметричного шифрування.

У роботі виконано порівняльний аналіз існуючих рішень для асиметричного шифрування, проаналізовано особливості реалізації класичних методів асиметричного шифрування, обґрунтовано вибір технологій, таких як мова програмування, допоміжні бібліотеки та фреймворк для реалізації даного програмного забезпечення, що реалізує модифікований метод асиметричного шифрування.

В роботі проаналізовано один з класичних методів асиметричного шифрування, виявлено його слабкі сторони та вразливості. Запропоновано модифікований метод асиметричного шифрування, який дозволяє посилити слабкі місця класичного методу асиметричного шифрування, підвищити його криптостійкість та ускладнити злам даного методу.

Розроблено програмне забезпечення для дослідження методів асиметричного шифрування, за допомогою якого проведено дослідження запропонованого модифікованого та класичного методу. Також проаналізовано особливості реалізації даного методу з метою підвищення продуктивності його роботи.

ABSTRACT

This diploma work dedicated to development of modified asymmetric encryption method and software for its implementation.

During the writing of the diploma work was carried out a comparative analysis of the existing solutions for existing asymmetric encryption method, were analyzed the peculiarities of realization of classical methods of asymmetric encryption, justified the choice of technologies, such as programing language, auxiliary libraries and framework for realization of this software, which implements the modified method of asymmetric encryption.

This diploma work analyzes one of the classical methods of asymmetric encryption, reveals its weaknesses and vulnerabilities. A modified method of asymmetric encryption, which permits a certain weakness in the classic method of asymmetric ciphering, and the increased crypto-encryption and accelerates the evils of this method, has been proposed.

The software for analysis of asymmetric encryption methods were developed, by means of which the research of the offered modified and classical method was carried out. The peculiarities of the implementation of this method in order to increase the productivity of its work were also analyzed.

ЗМІСТ

СПИСОК ТЕРМІНІВ ТА СКОРОЧЕНЬ

DSS (Digital Signature Standard) – американський стандарт, що описує алгоритм для генерації цифрового підпису, що використовується для визначення дійсності підпису сторони, що її поставили.

Символ Лежандра – мультиплікативна функція, що використовується в теорії чисел, названа на честь Адрієна-Марі Лежандра.

Символ Якобі – мультиплікативна функція, що є узагальненням символу Лежандра і використовується для додатних цілих непарних чисел.

НСД – найбільший спільний дільник двох або більше невід'ємних чисел, найбільше натуральне число, на яке ці числа діляться без остачі.

PKCS (Public Key Cryptography Standards) – специфікації, що були розроблені RSA Security спільно з розробниками систем безпеки всього світу з метою прискорення розробки криптографії з відкритим ключем.

CPython – найбільш розповсюджена еталонна реалізація мови програмування Python, являє собою інтерпретатор байт-коду, написаний мовою C.

PSFL (Python Software Foundation License) – BSD-подібна пермісивна ліцензія на вільне програмне забезпечення, призначена для розповсюдження програмного проекту Python.

XML (eXtensible Markup Language) – здатна до розширення мова розмітки. Специфікація XML описує XML-документи і XML-процесори, тобто програми, що читають та обробляють XML-документи. Дана специфікація рекомендована Консорціумом Всесвітнього Павутиння W3S.

STL (Standard Template Library) – бібліотека стандартних шаблонів, набір узгоджених узагальнених алгоритмів, контейнерів, засобів доступу до їх вмісту і різноманітних допоміжних функцій у мові C++.

.NET Framework – програмна платформа компанії Microsoft, основою якої є загальномовне середовище виконання CLR.

CLR (Common Language Runtime) – середовище виконання байт-коду CIL, в якій компілюються програми, написані на мовах, що сумісні з платформою .NET.

FCL (Framework Class Library) – компонент .NET Framework, перша реалізація CLI.

Jinja2 – шаблонізатор мови програмування Python.

OpenGL ES 2 – графічний інтерфейс, розроблений для вбудованих систем.

MATLAB – пакет прикладних програм для вирішення задач технічного обчислення.

ВСТУП

Кількість проблем, пов’язаних з інформаційною безпекою, що виникають внаслідок розвитку інформаційно-телекомунікаційних технологій, невпинно зростають, тому проблема захисту інформації надзвичайно актуальна. Одним з основних рішень даної проблеми є використання криптографічних методів. Вони допомагають у забезпеченні цілісності та вірогідності передаваної інформації, у фінансовій сфері. Враховуючи сучасні вимоги до програмного забезпечення інформаційних систем, дана задача стає все більш проблематичною.

Асиметричну криптографію винайдено понад сорок років тому. Весь цей час вона невпинно розвивалась і тепер посідає місце поряд з блочним симетричним шифруванням. На відміну від останнього, асиметричне шифрування, яке ще називають шифруванням на відкритому ключі, ґрунтується власній унікальній ідеології, а отже стабільно займає нішу поміж інших систем інформаційної безпеки.

У даній роботі було прийнято рішення проаналізувати існуючі асиметричні методи шифрування для виявлення слабкостей та вразливостей, та створити модифікований метод асиметричного шифрування на основі одного з класичних методів. Планується підвищити стійкість до атак методами криптоаналізу, а також порівняти час виконання програмної реалізації методу асиметричного шифрування з класичними методами.

МЕТА ДОСЛІДЖЕННЯ ТА ПОСТАНОВКА ЗАДАЧІ

Метою дипломної роботи є розроблення модифікованого методу асиметричного шифрування.

Науково-практична задача, що розв’язується у даній дипломній роботі, включає наступні завдання:

1. Аналіз існуючих методів асиметричного шифрування.
2. Аналіз вразливостей до криптоаналітичних методів зламу.
3. Розроблення модифікованого методу асиметричного шифрування, що усуває або знижує виявлену вразливість.
4. Розроблення програмного забезпечення, що реалізує класичний та модифікований метод асиметричного шифрування.
5. Оцінка швидкості роботи програмного забезпечення, що реалізує модифікований метод асиметричного шифрування, та порівняння отриманих даних з відповідними показниками швидкості роботи алгоритму, що реалізує класичний метод асиметричного шифрування.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1. Класичні методи шифрування

На початку розвитку криптографії переважали перестановочні шифри, що змінюють порядок літер в повідомленні, а також підстановочні шифри, що містять систему заміни одних літер чи символів іншими. Ці класичні шифри забезпечували мінімальний захист [1].

Найбільш відомий серед перших підстановочних шифрів – це шифр Цезаря. Його суть полягала в заміні кожної літери алфавіту іншою літерою зі зсувом по алфавітному порядку. Гай Юлій Цезар використовував цей шифр зі зсувом в 3 позиції, для передачі повідомлень між ним і генералами під час своїх військових кампаній.

Всі вищезгадані шифри залишались повністю незахищеними від криптоаналізу з використанням частотного аналізу. Ситуація змінилась з винаходом поліалфавітного шифр. Досить популярний поліалфавітний шифр Віженера за допомогою ключового слова, що керує підстановкою літер в залежності від літери, що у ключовому слові використовується. В середині 1800-тих років, Чарльз Беббідж довів, що поліалфавітні шифри такого типу все ще залишились частково беззахисними перед частотним аналізом. Було доведено, що збереження ключа у секреті є достатньою умовою захисту інформації нормальною криптографічною схемою. В цьому полягає фундаментальний принцип криптографії, що був вперше сформований в 1883 році Огюстом Кекгофсом і носить його ім’я.

З винайденням електроніки, а згодом і цифрових комп’ютерів, можливості шифрування розширилися, з’явилися нові, більш складні шифри. Крім того, з’явилась можливість зашифрувати будь-які дані, що можна представляти у двійковому вигляді у комп’ютері, а не лише письмові тексти, як це було раніше.

Перші ідеї, що лягли в основу асиметричного шифру, з’явились 1976 року у роботі «Нові напрямки в сучасній криптографії» Вітфілда

Діффі та Мартіна Геллмана [2]. У своїй роботі вони запропонували абсолютно новий спосіб організації таємного зв’язку без попереднього обміну ключами, що був названий шифруванням з відкритим ключем. Даний спосіб використовує для шифрування та дешифрування різні ключі, які не дають можливості при наявності одного з них знайти інший. Завдяки цьому ключ шифрування може бути відкритим, так як це не відображається на стійкості шифру, і лише ключ дешифрування має знаходитись тільки у одержувача, тому криптосистеми з відкритим ключем називаються асиметричними (або несиметричними) криптосистемами.

На рис. 1.1 подано структурну схему криптосистеми з відкритим ключем.

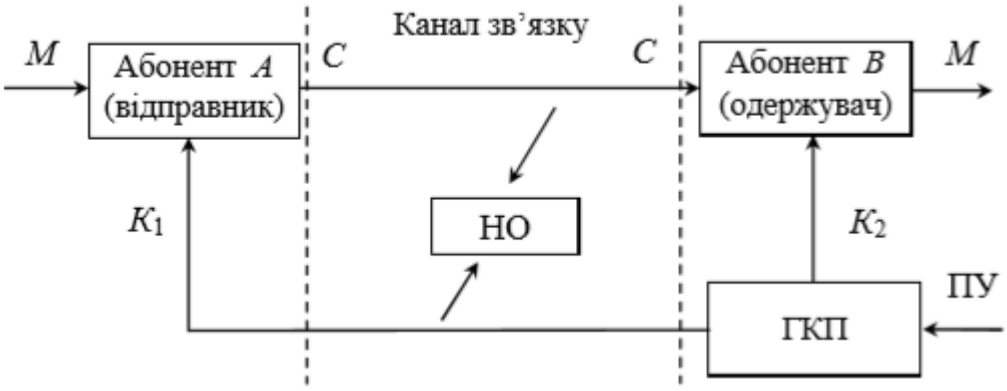


Рис. 1.1. Схема структури асиметричної криптосистеми

ГКП, тобто генератор ключової пари, створює пару ключів (К1, К2) в залежності від початкових умов (ПУ), що відомі тільки одержувачу повідомлення. Відкритий ключ К1 надсилається відправнику незахищеним каналом зв’язку. Відправник шифрує повідомлення М з використанням ключа К1. Зашифрований текст С (криптограма) пересилається через незахищений канал зв’язку одержувачеві, а він, в свою чергу, розшифровує криптограму, відновлюючи вихідне повідомлення, за допомогою секретного ключа К2. Канал зв'язку незахищений, тому несанкціонована особа (НО), маючи до них доступ, може перехопити криптограму С і

відкритий ключ К1. Також, НО може володіти алгоритмом шифрування. Проте єдине, чим вона не володіє – це ключ К2, без якого розшифрувати криптограму С не вийде.

Найбільш відомі асиметричні криптосистеми:

- криптосистема RSA;
- криптосистема Діффі-Геллмана;
- криптосистема Меркле-Геллмана;
- криптосистема Ель-Гамаля;
- криптосистема на еліптичних кривих;
- електронний цифровий підпис DSS.

1.2. Аналіз класичних асиметричних методів шифрування

1.2.1. Алгоритм Меркле-Геллмана

Криптосистема була розроблена Ральфом Меркле та Мартіном Геллманом у 1978 році. Вона стала першим асиметричним алгоритмом шифрування, що мала широке призначення. Криптосистема Меркле-Геллмана належить до ранцевих алгоритмів. Спершу алгоритм використовувався лише для шифрування повідомлень, але згодом Аді Шамір створив модифікацію відповідної криптосистеми задля підтримки засобів цифрового підпису. Безпека ранцевих алгоритмів базується на відомій математичній задачі про укладання ранця. В основі алгоритму лежить ідея шифрування повідомлення через розв’язок серії завдань стосовно укладання ранця. Розв’язання задачі з використанням ранцевого алгоритму для п’яти елементів досить легке. Щоб забезпечити практичну придатність, ранці повинні містити від 250 елементів і більше, кожен з яких має бути у діапазоні від 200 до 400 біт. Довжина модуля має перебувати у проміжку між 100 і 200 біт. Щоб отримати такі значення, на практиці використовують генератори випадкових послідовностей. Дана криптосистема була зламана Шаміром і Циппелом, які змогли виявити її

вразливі місця і відновити швидкозростаючу ранцеву послідовність через нормальну. Пізніше було створено ще багато криптосистем з використанням алгоритму укладання ранця, але жодна не виявилась надійною та стійкою до зламу.

Сутність криптографічного алгоритму Меркле-Геллмана, що ґрунтується на задачі «про наповнення ранця», зводиться до наступного [3]. Абонент, що одержує інформацію, обирає початковий вектор W = (w_1, w_2, …, w_n), що складається з елементів w_i, де i = 1, 2, …, n, які задовольняють вимоги швидкозростаючої послідовності. Потім одержувач обирає просте число p, що має значення більше ніж сума w_i, і певне число r (необов'язково просте), що задовольняє умовам $r \leq p - 1$ і НСД (r, p) = 1. Далі він формує відкритий ключ K = (k_1, k_2, …, k_n), використовуючи формулу

$$k_i \equiv (w_i r) \bmod p, \quad i = 1, 2, \ldots, n. \tag{1.1}$$

Значення елементів ключа передаються відправнику повідомлення через відкритий канал зв'язку. Значення початкового вектора W, а також чисел r та p зберігаються лише у одержувача повідомлення. Шифроване повідомлення M розбивається відправником на блоки по n символів:

$$M = (m_1, m_2, \ldots, m_n), \text{ де } m_i = \{1, 0\}, \ i = 1, 2, \ldots, n. \tag{1.2}$$

За допомогою ключа K, відправник шифрує повідомлення за формулою

$$C = K \cdot M = \sum_{i=1}^{n} k_i m_i \,. \tag{1.3}$$

Потім шифрований текст передається відправником через відкритий канал до одержувача. Одержувач розв'язує рівняння

$$re \equiv 1 \ (\bmod p) \tag{1.4}$$

відносно e, тобто він знаходить обернене значення для числа r за модулем p. Далі отримана криптограма перетворюється ним за формулою

$$C' \equiv (Ce) \bmod p. \tag{1.5}$$

C' потрібно привести до форми

$$C' = (Ce) \bmod p \equiv \left(\sum_{i=1}^{n} k_i \, m_i e \right) \bmod p \equiv \left(\sum_{i=1}^{n} m_i \, w_i re \right) \bmod p \equiv \sum_{i=1}^{n} w_i m_i \,. \quad (1.6)$$

Далі потрібно розв’язати завдання щодо визначення суми підмножини.

Шифр, що запропонував Аді Шамір, забезпечує обмін таємними повідомленнями через відкриту лінію зв’язку без використання захищених каналів та секретних ключів. Даний шифр повністю розв’язує задачу обміну повідомленнями, що закриті для читання, у випадку, якщо абоненти не мають змоги використовувати закриті лінії зв’язку. Але схема Шаміра має суттєвий недолік, оскільки повідомлення пересилається між абонентами тричі.

1.2.2. Алгоритм RSA

Через деякий час після появи ранцевого алгоритму Меркле-Геллмана було створено перший повноцінний алгоритм з відкритим ключем, використання якого підходило як для шифрування, так і для створення цифрових підписів, алгоритм RSA. Його назва складається з перших літер імен його винахідників, а саме Рональда Рівеста, Аді Шаміра і Леонарда Адлемана. Він був створений у 1978 році [4]. Розробникам вдалося достатньо ефективно реалізувати ідею односпрямованих функцій із секретом. Стійкість алгоритму RSA спирається на складність факторизації великих цілих чисел. Ключі, відкритий і закритий, є функціями двох великих простих чисел дуже великої розрядності. А відновлення вихідного тексту через шифрований текст і відкритий ключ є рівнозначним розкладанню числа на пару великих простих множника. Шифрування проводиться через операцію піднесення до степеня за модулем великого цілого числа. Дешифрування відбувається через обчислення функції Ейлера від даного великого числа, що вимагає знання розкладення числа на прості множники.

Стійкість алгоритму RSA залежить від трудомісткості розв'язку проблеми факторизації, тобто розкладання великих чисел на множники. Проте це не є коректно з технічної точки зору. Залежність безпеки RSA від проблеми факторизації великих чисел є гіпотетичним твердженням, оскільки теоретично можливе винайдення іншого способу зламу алгоритму RSA.

Сутність алгоритму RSA зводиться до наступного. Довільним способом обираються два великі прості числа p і q. Обчислюється їх добуток $n = pq$, а також функція Ейлера:

$$\varphi(n) = (p - 1)(q - 1). \tag{1.7}$$

Довільно обирається просте число e – ключ зашифровування, що задовольняє умовам $e < \varphi(n)$; НСД $(e, \varphi(n)) = 1$. Обчислюється число d – ключ розшифровування, що є оберненим до числа e, тобто

$$ed \equiv 1 \pmod{\varphi(n)}. \tag{1.8}$$

Пара чисел (e, n) відіграють роль відкритого ключа, що розміщується у загальнодоступному довіднику, а числа p і q зберігаються у секреті. Число d – це секретний ключ. Для шифрування повідомлення M спочатку розкладається на цифрові блоки, розміри яких є меншими за n, тобто якщо p та q – це 128-розрядні прості числа, то n буде містити не менш як 256 розрядів, відповідно кожен із блоків повідомлення m_i повинен бути довжиною близько 256 розрядів. Такої самої довжини будуть блоки c_i, з яких буде складатись зашифроване повідомлення C. Формула зашифровування буде мати вигляд

$$C \equiv M^e \pmod{n}. \tag{1.9}$$

Розшифроване повідомлення отримується через операцію піднесення до степеня d за модулем n отриманого шифрованого тексту C, тобто $\mathrm{M} \equiv C^d \pmod{n}$, оскільки

$$\begin{gathered} C^d \pmod{n} \equiv M^{ed} \pmod{n} \equiv \\ \equiv (M^{k(p-1)(q-1)+1}) \bmod n \equiv (MM^{k(p-1)(q-1)}) \bmod (pq) \equiv M. \end{gathered} \tag{1.10}$$

1.2.3. Алгоритм Рабіна

Алгоритм Рабіна створювався як модифікація алгоритму RSA. Його безпека ґрунтується на складності пошуку квадратного кореня за модулем складеного числа [5].

Сутність алгоритму Рабіна зводиться до наступного. Обираються два великі прості числа p та q, що порівняні з 3 за модулем 4. Дані прості числа постають в якості закритого ключа, а їх добуток $n = pq$ – в якості відкритого ключа:

$$p \equiv 3 \pmod 4 \equiv -1 \pmod 4; \tag{1.11}$$

$$q \equiv 3 \pmod 4 \equiv -1 \pmod 4. \tag{1.12}$$

Вважатимемо, що e є фіксоване і завжди становить 2, тоді шифрований текст повідомлення M обчислюється як

$$C \equiv M^2 \pmod n. \tag{1.13}$$

Розшифрування криптограми C. Введемо допоміжні величини x та y:

$$x \equiv C^l \pmod p, \tag{1.14}$$

$$y \equiv C^k \pmod q, \tag{1.15}$$

де $4l = p + 1$; $4k = q + 1$. Тоді для x^2 та y^2 одержуємо

$$\begin{gathered} x^2 \equiv C^{2k} \pmod p \equiv \left[(M^2)^{\frac{p+1}{4}}\right]^2 \bmod p \equiv \\ \equiv M^{p+1} \pmod p \equiv (M^{p-1}M^2) \bmod p \equiv \\ \equiv M^2 \pmod p; \end{gathered} \tag{1.16}$$

$$y^2 \equiv C^{2l} \pmod q \equiv \left[(M^2)^{\frac{q+1}{4}}\right]^2 \bmod q \equiv M^2 \pmod q. \tag{1.17}$$

Отримаємо чотири системи рівнянь для M_1, M_2, M_3, M_4:

$$\begin{cases} M_1 \equiv x(\bmod p); \\ M_1 \equiv y(\bmod q); \end{cases} \quad \begin{cases} M_2 \equiv x(\bmod p); \\ M_2 \equiv -y(\bmod q); \end{cases} \quad \begin{cases} M_3 \equiv -x(\bmod p); \\ M_3 \equiv y(\bmod q); \end{cases} \quad \begin{cases} M_4 \equiv -x(\bmod p); \\ M_4 \equiv -y(\bmod q). \end{cases}$$

Один з результатів M_1, M_2, M_3 та M_4 і буде повідомленням M. У разі якщо повідомлення написане словами, то обрати правильне M нескладно.

Проте якщо повідомлення є потоком випадкових бітів, що призначені для генерації ключів цифрового підпису, тоді визначити правильний варіант повідомлення M досить складно. Один із способів розв'язати дану проблему – це перед зашифровуванням додавати щось до повідомлення відомого заголовка.

1.2.4. Алгоритм Вільямса

Хью Вільямс оптимізував алгоритм Рабіна, додавши до нього зміни, котрі усувають неоднозначність приймання [6].

Сутність алгоритму Вільямса зводиться до наступного. Обираються два прості числа p та q такі, щоб $p \equiv 3 \pmod 4$, $q \equiv 3 \pmod 4$ і $n = pq$. Також обирається невелике ціле число s, для якого символ Якобі $J(s, n)$ дорівнює –1. Числа s і n надсилаються відправнику відкритим каналом. Секретним ключем є

$$k = \frac{1}{2}\left[\frac{1}{4}(p-1)(q-1)+1\right]. \tag{1.18}$$

Для зашифровування повідомлення M обчислюється C_1 таке, що $J(M, n) = (-1)^{C_1}$, і визначаються проміжні повідомлення

$$M' \equiv (s^{C_1}\mathrm{m}) \bmod n; \tag{1.19}$$

$$C_2 \equiv M' \pmod 2. \tag{1.20}$$

Криптограма обчислюється за аналогією алгоритму Рабіна:

$$C \equiv (M')^2 \bmod n. \tag{1.21}$$

Одержувач отримує три числа – C, C_1, C_2. Щоб розшифрувати C, одержувач обчислює M'':

$$\pm M'' \equiv C^k \pmod n; \tag{1.22}$$

$$M \equiv (s^{-C_1} M'') \bmod n. \tag{1.23}$$

Правильний знак M'' визначає C_2.

1.2.5. Алгоритм Ель-Гамаля

Даний алгоритм є альтернативою алгоритму RSA і при однаковій довжині ключа забезпечує той же рівень криптостійкості [7]. Безпека даного алгоритму ґрунтується на складності обчислення дискретних

логарифмів. Алгоритм Ель-Гамаля – це перший алгоритм асиметричного шифрування, що використовується як для повідомлень, так і для цифрових підписів, не обмежений патентами США.

Сутність алгоритму Ель-Гамаля зводиться до наступного. Учасники обміну інформаційними повідомленнями обирають просте число p і ціле число q, яке є первинним коренем за модулем p. Одержувач повідомлення генерує секретний ключ $k_a < p$ і за його допомогою обчислює відкритий ключ

$$Y_a \equiv q^{k_a} \pmod{p}. \tag{1.24}$$

Відправник генерує число $k_b < p$ і зашифровує повідомлення, що передається, M у наступний спосіб:

$$Y_b \equiv q^{k_b} (\mathrm{mod}\ p) \text{ і } C = M \oplus (Y_a^{k_b} (\mathrm{mod}\ p)). \tag{1.25}$$

Значенням M є послідовність двійкових символів, що передаються через канал зв'язку. Значення $Y_a^{k_b} (\mathrm{mod}\ p)$ перед підсумовуванням перетворюється на послідовність двійкових символів. Одержувач приймає повідомлення у формі Y_b і C та відновлює його:

$$M = (Y_b^{k_a} (\mathrm{mod}\ p)) \oplus C. \tag{1.26}$$

1.2.6. Алгоритм Діффі-Геллмана

Діффі й Геллман у 1976 році запропонували алгоритм, що створює криптографічні системи з відкритим ключем, що базується на складності обчислення дискретного логарифма [8]. Цим вона подібна до криптосистеми, що використовує алгоритм Ель-Гамаля. Алгоритм Діффі-Геллмана використовується задля розподілу ключів та генерування секретного ключа, проте цей алгоритм непридатний для шифрування повідомлення.

Також алгоритм Діффі-Геллмана використовується в комутативних кільцях. Існує варіант алгоритму, запропонований Шмулі та Кевіном МакКерлі, в якому модуль є складеним числом, а також його розширена версія, запропонована Міллером і Нілом Кобліцами, для використання з

еліптичними кривими. Ель-Гамаль при розробці свого алгоритму шифрування та цифрового підпису використовував основоположну ідею алгоритму Діффі-Геллмана. Також даний алгоритм працює в полі Галуа. Такий підхід використовується у ряді реалізацій, оскільки обчислення виконується набагато швидше, але важливо обирати достатньо велике поле, щоб забезпечити необхідну стійкість. Криптографічна стійкість алгоритму Діффі-Геллмана ґрунтується на складності проблеми дискретного логарифмування. Проте варто зазначити, що хоча вміння розв'язувати задачу дискретного логарифмування і дозволить зламати алгоритм Діффі-Геллмана, це не доводить зворотне твердження. Ще однією особливістю даного алгоритму є те, що він працює лише на лініях зв'язку, які надійно захищені від модифікацій, оскільки у тому випадку, коли в каналі можливо модифікувати дані, з'являється можливість для атаки «людина посередині». Сутність алгоритму Діффі-Геллмана зводиться до наступного. Згідно даного алгоритму, учасники обміну повідомленнями А та В домовляються про значення великого простого числа p і його простого дискретного кореня – числа a (рис. 1.2).

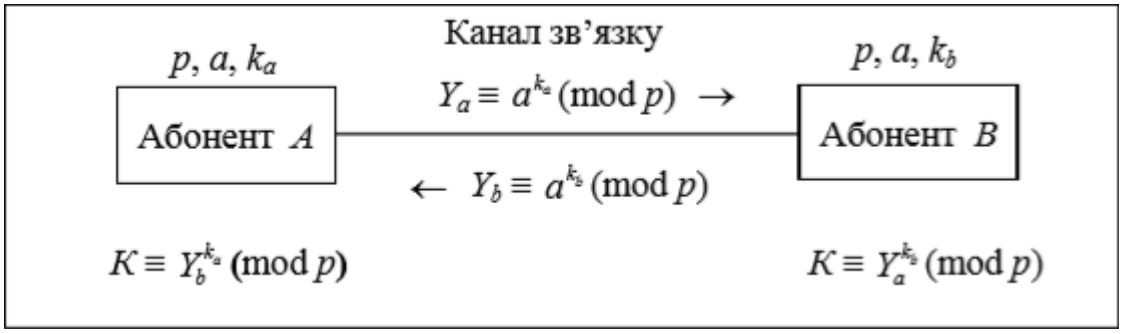


Рис. 1.2. Схема структури алгоритму Діффі-Геллмана

Абонент А обирає випадкове число k_a, а абонент В – випадкове число k_b таким способом, щоб виконувалися умови $1 < k_a < p - 1$ та $1 < k_b < p - 1$. Ці числа k_a та k_b зберігаються абонентами А та В у

секреті. Абонент А формує відкритий ключ за допомогою використання формули

$$Y_a \equiv a^{k_a} \pmod{p}. \quad (1.27)$$

Аналогічно абонент В формує відкритий ключ, використовуючи формулу

$$Y_b \equiv a^{k_b} \pmod{p}. \quad (1.28)$$

Після обміну відкритими ключами Y_a та Y_b абоненти обчислюють значення секретного числа K:

$$K \equiv Y_a^{k_b} \pmod{p} \equiv a^{k_a k_b} \pmod{p}; \quad (1.28)$$

$$K \equiv Y_b^{k_a} \pmod{p} \equiv a^{k_b k_a} \pmod{p}. \quad (1.29)$$

Отримане число K для потенційного зловмисника є секретним, тому що розв'язання рівнянь Y_a та Y_b для великих чисел є неможливим.

Алгоритм Діффі-Геллмана можна використовувати у випадку з трьома і більше учасниками.

1.2.7. Криптосистема на еліптичних кривих

Криптосистеми, що базуються на еліптичних кривих, належать до класу асиметричних криптосистем [9]. Їх безпека ґрунтується здебільшого на складності розв'язання задачі дискретного логарифмування у групі точок еліптичної кривої, що проходить над скінченним полем. Це зумовлює досить потужну криптостійкість, недосяжну для інших алгоритмів шифрування. Також існують криптосистеми на еліптичних кривих, що базуються на складності розкладання великих цілих чисел, у разі якщо еліптична крива задана над скінченним кільцем за складеним модулем, проте такі системи зустрічаються досить рідко. Крім того, варто зауважити, що криптостійкість – це відносне поняття, що пов'язане з наявністю оптимального відомого криптоаналітичного алгоритму.

Еліптичні криві – це математичний об'єкт, що може бути визначений над певним довільним полем. Переважно це певна множина точок (x, y), що задовольняють рівняння:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (1.30)$$

У криптографії як правило використовуються скінченні поля. Для точок, які знаходяться на еліптичній кривій, вводиться операція додавання, що відіграє ту саму роль, що й операція множення у алогоритмах RSA та Ель-Гамаля.

Першим завданням в даній системі є шифрування відкритого тексту повідомлення m, що має пересилатись у вигляді значення $x - y$ для точки P_m. Тут точка P_m буде представляти закодований в неї текст. Користувач А обирає особистий ключ n_A і генерує відкритий ключ за формулою

$$P_A = n_A G. \quad (1.31)$$

Для шифрування повідомлення обирається випадкове додатне число k і обчислюється значення $G_m = (kG, P_m + kP_B)$.

Перевагою криптосистем з використанням еліптичних кривих є висока швидкість опрацювання інформації. Звичайно, завдяки потужній криптостійкості, в криптосистемах на еліптичних кривих можливе використання ключів меншої довжини. Хоча швидкість обчислень, що є прийнятною для роботи в мережах, досягається лише при використанні спеціальних алгоритмів для швидкого обчислення і відповідних полів характеристик, що є цілком природним для асиметричних алгоритмів.

Криптосистеми на еліптичних кривих, так само як інші асиметричні криптосистеми, недоцільно використовувати для шифрування великого обсягу даних. Проте вони ефективно застосовуються у системах цифрового підпису і ключового обміну. Починаючи з 1998 року використання еліптичних кривих для розв’язання криптографічних задач, таких як цифровий підпис, було закріплено у стандартах США ANSI X9.62 і FIPS 186-2, а 2001 року аналогічний стандарт ГОСТ Р34.10-2001 було затверджено в Росії. Ухвалений український стандарт цифрового підпису з використанням еліптичних кривих називається ДСТУ 4145-2002. Варто

зазначити, що безпека даних систем цифрового підпису спирається не тільки на стійкість алгоритму на еліптичних кривих, а ще й на стійкість використовуваної геш-функції. Численні дослідження засвідчують, що криптосистеми з використанням еліптичних кривих перевершують інші криптосистеми з відкритим ключем за такими важливими параметрами як: швидкодія програмної і апаратної реалізації та міра захищеності з розрахунку на кожен біт ключа. Це можна пояснити тим, що для обчислення обернених функцій на еліптичних кривих існують лише алгоритми з експоненційним зростанням трудомісткості, а для звичайних систем використовуються субекспоненційні методи. В результаті, той рівень стійкості, який досягається, наприклад, в RSA при використанні 1024-бітових модулів, в криптосистемах на еліптичних кривих отримується при розмірі модуля 160 біт, що забезпечує набагато простішу програмну і апаратну реалізацію.

1.2.8. Електронний цифровий підпис

Одною з найбільш важливих сфер застосовування криптографії з відкритим ключем є цифровий підпис. Створено декілька методів побудови схем ЕЦП [10].

Перша схема – це шифрування електронного документа (ЕД) з використанням симетричних алгоритмів. Дана схема передбачає у системі наявність третьої особи (арбітра), якій довіряють обидва учасника обміну підписаними у подібний спосіб електронними документами.

Друга схема – шифрування ЕД з використанням асиметричних алгоритмів шифрування. Про факт підписання документа в даній схемі свідчить шифрування документа за допомогою секретного ключа його відправника. Така схема використовується досить рідко через те, що довжина документа може виявитись завеликою. Застосовування алгоритмів з відкритим ключем для шифрування повідомлень великої довжини є неефективним з точки зору швидкості роботи цих алгоритмів. Вони не передбачають наявності третьої сторони, проте третя сторона

може виступати в ролі сертифікаційного органу відкритих ключів користувачів.

Третя схема – найбільш розповсюджена – стала схема ЕЦП, тобто: зашифрування кінцевого результату опрацювання ЕД геш-функцією за допомогою асиметричного алгоритму. Схему структури такої побудови ЕЦП подано на рис. 1.3.

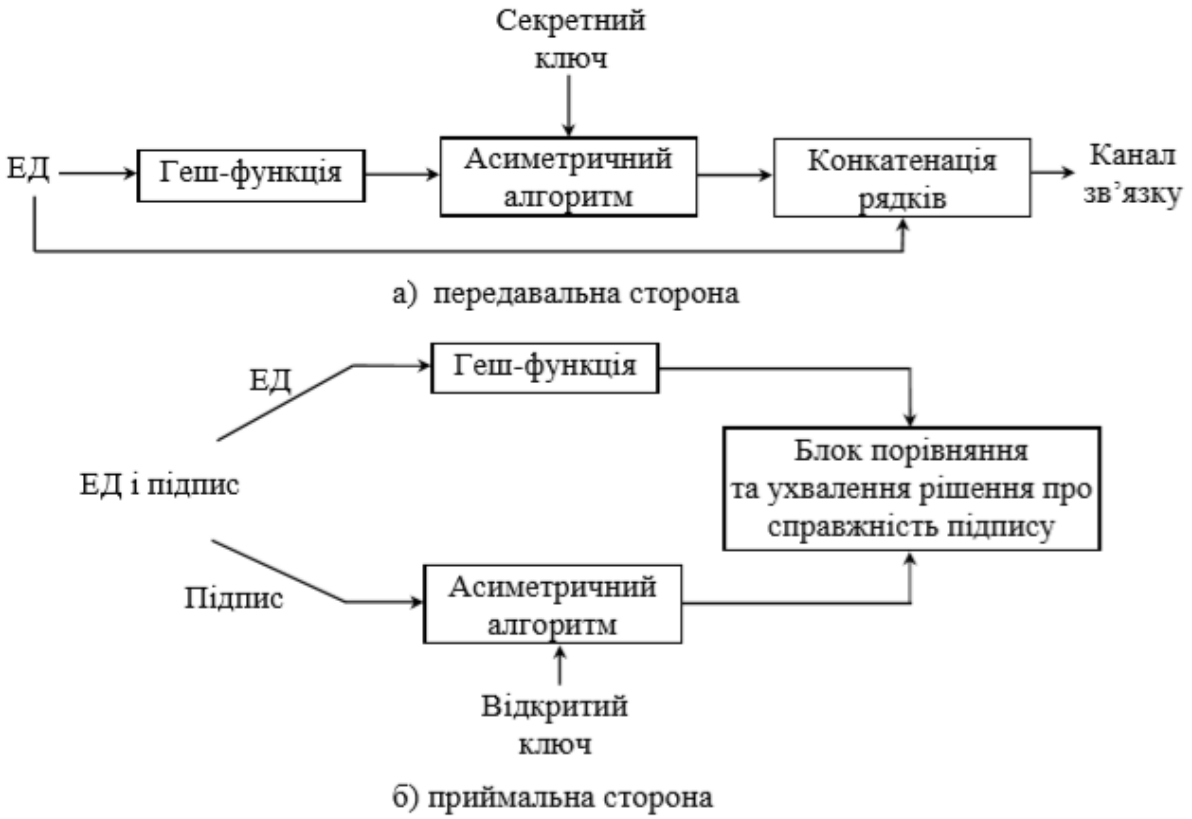


Рис. 1.3. Структурна схема побудови ЕЦП

Генерування підпису відбувається в наступний спосіб.

Учасник А обчислює геш-код від ЕД. Далі отриманий геш-код проходить перетворюється на нове значення, використовуючи власний секретний ключ, що і є ЕЦП, яке разом з ЕД надсилається учасникові В.

Учасник В отримує ЕД разом з ЕЦП та сертифікований відкритий ключ учасника А, а потім проводить розшифровування ЕЦП використанням отриманого відритого ключа. Сам ЕД обробляється геш-

функцією, після цього отримані результати порівнюються і, якщо вони співпадають, ЕЦП визнається справжнім, в іншому разі – хибним. Стійкість даного типу ЕЦП ґрунтується на стійкості асиметричних алгоритмів шифрування та геш-функцій.

Таким чином, можна відокремити такі властивості ЕЦП:

- однозначно ідентифікує людину, котра підписала цей документ;
- непідробність, оскільки розв'язати рівняння $Ek(S) = M$ має можливість лише власник секрету k;
- є непереносним на інший ЕД, виняток виникає лише у випадку, коли для використаної геш-функції виникають колізії;
- верифікація підпису відбувається на основі знання функції Ek;
- ЕД з ЕЦП може бути передана через відкриті канали, оскільки будь-яка зміна ЕД призведе до того, що цей факт виявиться під час процедури перевірки ЕЦП.

1.2.9. Існуючі модифікації класичних методів

Після появи класичних методів асиметричного шифрування з'явилось багато їх модифікацій.

Стаття у журналі ASRJETS пропонує модифікацію методу RSA, що полягає у збільшенні кількості приватних ключів до певного k з метою підвищення надійності методу [11]. Метод застосовується до матриць, тобто для кожного матричного елемента створюється окремий приватний ключ. Таким чином можна шифрувати зображення, як чорно-білі, так і кольорові.

У журналі IJRTE було опубліковано статтю про модифікацію методу RSA, що розширює модуль n, щоб зробити його стійким до факторизації [12]. Сутність модифікації полягає у додаткових функціях, що приховують значення e та n, таким чином шифрування виходить дворівневим. Недоліком методу є збільшення часу виконання.

Видання Національного Технічного Інституту міста Руркла, що у Індії, випустило примірник, що детально описує модифікований метод RSA з використанням Китайської теореми залишків [13]. Модифікація направлена на збільшення швидкості роботи методу, а також підвищення складності зламу.

Стаття у журналі IJSRET пропонує модифікацію методу RSA, що використовує 4 прості числа та зберігає ключі у хмарному сховищі даних [14]. Метод використовує спеціальні прості числа Прота і Мерсенна. Даний метод програє оригінальному методу RSA в часі виконання.

1.3. Результати проведеного аналізу

В результаті проведення аналізу було отримано наступні дані про сутність криптосистем з відкритим ключем шифрування, знання яких дозволяє займатись розробкою нових криптосистем:

- схему асиметричного шифрування;
- математичні задачі, на складності яких можуть базуватися алгоритми, такі як:
    - задача про укладання ранця;
    - задача факторизації великих цілих чисел;
    - задача пошуку квадратного кореня за модулем складеного числа;
    - задача обчислювання дискретних логарифмів.

Крім того, було проаналізовано ряд переваг та недоліків окремих існуючих алгоритмів, а також особливості їх реалізації.

2. РОЗРОБЛЕННЯ МОДИФІКОВАНОГО МЕТОДУ АСИМЕТРИЧНОГО ШИФРУВАННЯ

2.1. Аналіз класичного методу шифрування Вільямса

Розглянемо алгоритм Вільямса, який покладено в основу розроблюваного модифікованого методу [15].

Для будь-якої системи бажана наявність доказу, що її злам має обчислювальну складність аналогічну обчислювальній складності задачі, яку на даний момент неможливо вирішити за означений час. Криптосистема Вільямса, як і RSA, спирається на припущення про складність факторизації великих чисел. Було доведено, що для будь-якого зламу шифротексту за допомогою криптоаналізу, маючи лише відкритий ключ, необхідно провести факторизацію, тобто вирішити рівняння $pq = n$ відносно p та q. Обчислювальна складність задачі факторизації невідома, проте вважається, що вона досить висока.

Основний принцип методу полягає в тому, щоб піднести по модулю до степені число у формі

$$\alpha = a + b\sqrt{c}, \tag{2.1}$$

де a, b, c – цілі числа, а не просто ціле число, як у алгоритмі RSA.

Визначимо функції:

$$X_i(\alpha) = (\alpha^i + \bar{\alpha}^i) / 2; \tag{2.2}$$

$$Y_i(\alpha) = b\,(\alpha^i + \bar{\alpha}^i) / (\alpha + \bar{\alpha}), \tag{2.3}$$

де $\bar{\alpha}$ – спряжене до α число, що дорівнює $a - b\sqrt{c}$. Тоді функція, що дозволить виразити степінь числа α буде мати наступний вигляд:

$$\alpha^i = X_i(\alpha) + \sqrt{c}\, Y_i(\alpha). \tag{2.4}$$

Математична сутність методу закладена в лемі: нехай n добуток двох простих непарних чисел p і q, a, b, c – цілі числа, що задовольняють

порівняння $a^2 - cb^2 = 1 (\text{mod } n)$. Нехай символи Лежандра $\delta_p = \left(\frac{c}{p}\right)$ і $\delta_q = \left(\frac{c}{q}\right)$ задовольняють порівняння:

$$\delta_p = -p \text{ (mod 4)}; \quad (2.5)$$

$$\delta_q = -q \text{ (mod 4)}. \quad (2.6)$$

Також, НСД(cb, n) = 1 та символ Якобі $\left(\frac{2(a+1)}{n}\right) = 1$. Позначимо

$$m = (p - \delta_p)(q - \delta_q) / 4, \quad (2.7)$$

а також e та d, що відповідають рівності

$$ed = (m + 1) / 2 \text{ (mod } n). \quad (2.8)$$

Тоді при таких допущеннях

$$a^{2ed} = \pm a \text{ (mod } n). \quad (2.9)$$

Використовуючи дану лему, генеруються ключі шифрування. Спершу обираються два простих непарних числа p та q, обчислюється їх добуток n. За допомогою підбору обирається число c таке, що символи Лежандра δ_p і δ_q задовольняють умовам, викладеним у лемі. Також підбором обирається число s, таке що символ Якобі $\left(\frac{s^2-c}{n}\right) = -1$ і НСД(s, n) = 1. Далі обчислюємо число $m = (p - \delta_p)(q - \delta_q) / 4$. Обираємо число d таке що НСД(d, m) = 1. Обчислюємо e за наступною формулою:

$$e = \frac{m+1}{2} d^{-1} \text{ (mod } m). \quad (2.10)$$

Отримуємо ключі n, e, c, s – відкритий, p, q, m, d – закритий.

Шифрування відбувається наступним чином. Повідомлення подається у вигляді числа M, $1 < M < n$. Обчислюється γ і b_1: якщо символ Якобі $\left(\frac{M^2-c}{n}\right) = 1$, то $b_1 = 0$ і $\gamma = M + \sqrt{c}$, інакше $b_1 = 1$ і $\gamma = (M + \sqrt{c})(s + \sqrt{c})$. Число α отримується за формулою $\alpha = \frac{\gamma}{\bar{\gamma}}$, де $\bar{\gamma} = M - \sqrt{c}$. Далі обчислюється:

$$X_e(\alpha) = (\alpha^e + \bar{\alpha}^e) / 2, \quad (2.11)$$

$$Y_e(\alpha) = (\alpha^e - \bar{\alpha}^e) / (\alpha + \bar{\alpha}), \quad (2.12)$$

і з них отримуємо $E = \frac{X_e(\alpha)}{Y_e(\alpha)}$. Також обчислюється $b_2 = a \bmod 2$.

У якості шифротексту відправляється три числа (E, b_1, b_2).

Дешифрування відбувається наступним чином. Спочатку обчислюється $\alpha^{2e} = \frac{E+\sqrt{c}}{E-\sqrt{c}}$, далі $\alpha^{2ed} = X_d(\alpha^{2e}) + Y_d(\alpha^{2e})\sqrt{c} = \pm\alpha'$. Число b_2 показує, який знак потрібно обрати, а число b_1 – чи потрібно результат множити на $\frac{s-\sqrt{c}}{s+\sqrt{c}}$. В результаті отримаємо число α'. Вихідний текст повідомлення M отримаємо за формулою:

$$M = \frac{\alpha'+1}{\alpha'-1}\sqrt{c}. \qquad (2.13)$$

На рис. 2.1 подано структурну схему класичного методу шифрування Вільямса.

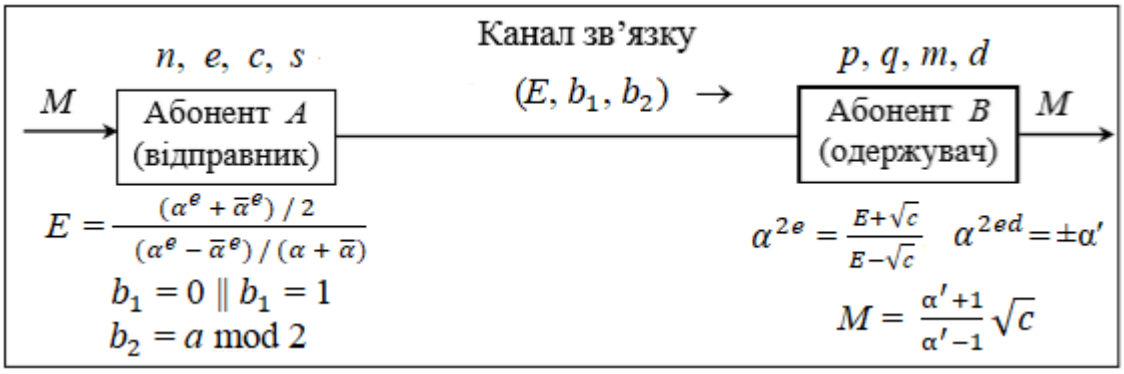


Рис. 2.1. Структурна схема класичного методу шифрування Вільямса

Після генерації ключа основні обчислення відбуваються при піднесенні числа α до степені, що відбувається за O(log n) множень по модулю, кожне з яких відбувається за O(log n) операцій додавання, які в свою чергу відбуваються за O(log n) операцій додавання однакової швидкості. Тобто вся операція відбувається за O(log^3 n), що є рівним часу піднесення до степеня цілого числа по модулю, що використовується у RSA.

2.2. Вразливості класичного методу шифрування Вільямса

Класична криптосистема Вільямса вразлива до атаки на основі підібраного шифротексту [16]. Це атака, при якій криптоаналітик збирає інформацію про шифр шляхом підбору зашифрованого тексту і отримання його розшифрування при невідомому ключі. Використовуючи отримані дані, криптоаналітик може відтворити секретний ключ.

Для проведення атаки криптоаналітику потрібен доступ до дешифруючого пристрою. Щоб отримати доступ, він може виступити в якості учасника обміну повідомленнями та перевіряти їх на відповідність стандарту шифрування, аналізувати повідомлення про помилки або вимірювати час відповіді отримувача повідомлень, аналізуючи чи є помилка форматною. Таким чином, на основі відповідей від пристрою та знань про формат відкритого повідомлення, що відповідає певному стандарту, криптоаналітик може обчислити повідомлення $m = c^d \pmod{n}$.

Крім того, існують алгоритми, що мінімізують кількість шифротекстів, необхідних для отримання відкритого повідомлення. На першому етапі отримується шифротекст c_0, що відповідає повідомленню m_0. Далі знаходяться числа s_i, для яких шифротексти виду $c_0 \cdot (s_i)^e \bmod n$ задовольняють стандарт PKCS. Для кожного знайденого s_i, використовуючи попередні знання про m_0, криптоаналітик зможе обчислити рід інтервалів, в яких може міститись m_0. На останньому етапі знаходіться ряд, що складається лише з одного інтервала. Це можливо зробити, коли криптоаналітик володіє достатньою кількістю інформації про m_0, щоб вибрати потрібне s_i, Інтервал поступово звужується до одного можливого числа.

Сам метод атаки проводиться наступним чином. Обирається число φ таке, що символ Якобі $\left(\frac{\varphi^2 - c}{n}\right) = -1$. Далі потрібно зашифрувати φ, але таким чином, ніби символ Якобі дорівнював +1 і $b_1 = 0$. На виході отримується шифротекст $(E,\ 0,\ b_2)$. Після цього отриманий шифротекст

дешифрується, отримується певне число $\psi \neq \varphi$. Тоді з ймовірністю $\delta > 0$ криптоаналітик може отримати $\varphi - \psi = p \pmod{n}$ або $\varphi - \psi = q \pmod{n}$, що дозволяє легко провести факторизацію, обчислити секретний ключ та зламати криптосистему. За умови, що довжина n не менше 512 біт, таку ймовірність зламу δ оцінюють як $0.18 \cdot 2^{-16} < \delta < 0.97 \cdot 2^{-8}$. Для успішного зламу криптоаналітик в середньому використовує 2^{13} шифротекстів.

Для боротьби із порушенням конфіденційності обміну повідомленнями та підвищення надійності криптосистеми використовується декілька підходів.

Перший підхід – стандартизація ключів шифрування, де вимагається генерувати ключі певної довжини. З нарощуванням обчислювальної потужності криптоаналітиків довжину доводиться збільшувати, що негативно відображається на швидкості обміну повідомленнями та швидкості проведення обчислень для їх шифрування та дешифрування.

Одним з таких стандартів є PKCS. Це специфікація, створена з метою прискорення розробки криптографії з відкритим ключем. Розробники цього стандарту також займаються створенням модифікацій алгоритмів, що використовують відкритий ключ шифрування.

Існує ряд підходів, що не використовувались у стандартах. Серед них модифікація ключів шифрування додатковою функцією, окреме надсилання відкритих ключів, що ускладнює часовий аналіз повідомлень, або переведення ключів у іншу систему числення (як правило двійкову).

2.3. Аналіз запропонованого модифікованого методу шифрування Вільямса

Запропонований модифікований метод полягає в тому, щоб збільшити кількість множників для обчислення n з метою ускладнення задачі факторизації для зламу криптосистеми.

Спершу обираються три простих непарних числа p, q, та f, обчислюється їх добуток n. За допомогою підбору обирається число c таке, що символи Лежандра δ_p, δ_q і δ_f задовольняють порівняння:

$$\delta_p = -p \pmod 4; \tag{2.14}$$

$$\delta_q = -q \pmod 4; \tag{2.15}$$

$$\delta_f = -f \pmod 4. \tag{2.16}$$

Також підбором обирається число s, таке що символ Якобі $\left(\frac{s^2-c}{n}\right) = -1$ і НСД(s, n) = 1. Далі обчислюємо число

$$m = (p - \delta_p)(q - \delta_q)(f - \delta_f) / 8. \tag{2.17}$$

Обираємо число d таке що НСД(d, m) = 1. Обчислюємо e за наступною формулою:

$$e = \frac{m+1}{2} d^{-1} \pmod m. \tag{2.18}$$

Отримуємо ключі n, e, c, s – відкритий, p, q, f, m, d – закритий. Подальші дії виконуються так само як і у класичному методі шифрування.

Шифрування відбувається наступним чином. Повідомлення подається у вигляді числа M, $1 < M < n$. Обчислюється γ і b_1: якщо символ Якобі $\left(\frac{M^2-c}{n}\right) = 1$, то $b_1 = 0$ і $\gamma = M + \sqrt{c}$, інакше $b_1 = 1$ і $\gamma = (M + \sqrt{c})(s + \sqrt{c})$. Число α отримується за формулою $\alpha = \frac{\gamma}{\bar{\gamma}}$, де $\bar{\gamma} = M - \sqrt{c}$. Далі обчислюється:

$$X_e(\alpha) = (\alpha^e + \bar{\alpha}^e) / 2, \tag{2.19}$$

$$Y_e(\alpha) = (\alpha^e - \bar{\alpha}^e) / (\alpha + \bar{\alpha}), \tag{2.20}$$

і з них отримуємо $E = \frac{X_e(\alpha)}{Y_e(\alpha)}$. Також обчислюється $b_2 = a \bmod 2$.

У якості шифротексту відправляється три числа (E, b_1, b_2).

Дешифрування відбувається наступним чином. Спочатку обчислюється $\alpha^{2e} = \frac{E+\sqrt{c}}{E-\sqrt{c}}$, далі $\alpha^{2ed} = X_d(\alpha^{2e}) + Y_i(\alpha^{2e})\sqrt{c} = \pm\alpha'$. Число b_2 показує, який знак потрібно обрати, а число b_1 – чи потрібно результат

множити на $\frac{s-\sqrt{c}}{s+\sqrt{c}}$. В результаті отримаємо число α'. Вихідний текст повідомлення M отримаємо за формулою

$$M = \frac{\alpha'+1}{\alpha'-1}\sqrt{c}. \tag{2.21}$$

На рис. 2.2 подано структурну схему запропонованого модифікованого методу шифрування.

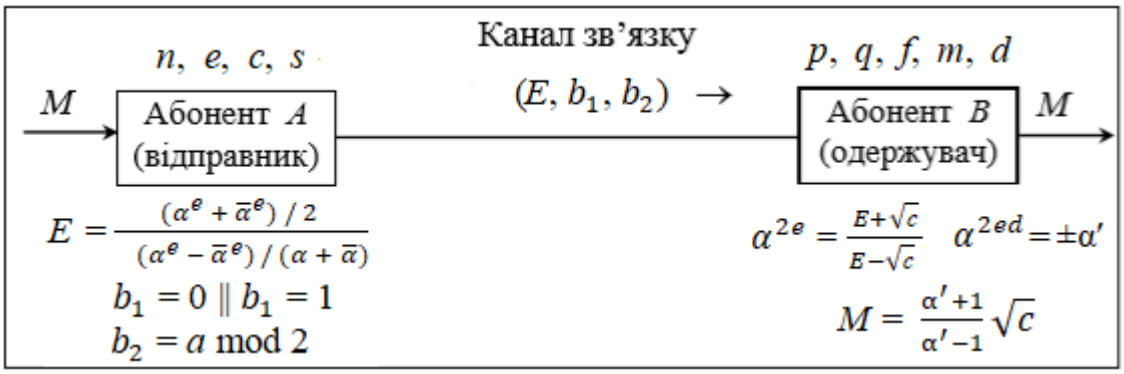


Рис. 2.2. Структурна схема модифікованого методу шифрування

При застосуванні описаного вище методу атаки на основі підібраного шифротексту, криптоаналітик отримає з ймовірністю $\delta > 0$ одне з наступних чисел:

$$\varphi - \psi = p \pmod{n}, \tag{2.22}$$

$$\varphi - \psi = q \pmod{n}, \tag{2.23}$$

$$\varphi - \psi = f \pmod{n}. \tag{2.24}$$

Маючи одне з трьох чисел, криптоаналітик після факторизації отримає добуток двох інших чисел. Для успішного зламу доведеться повторювати всю процедуру атаки. Таким чином, ймовірність зламу зменшується квадратично. За умови, що довжина n не менше 512 біт, таку ймовірність зламу δ можна оцінити як $0.18 \cdot 2^{-32} < \delta < 0.97 \cdot 2^{-16}$.

Це дозволить зменшити довжину ключів, завдяки чому зменшиться складність обчислення та підніметься швидкість обміну повідомленнями.

Недоліком запропонованого модифікованого методу порівняно з класичним є зменшення складності прямої факторизації відкритого ключа n. Складність факторизації відкритого ключа n класичного методу можна оцінити наступним чином [17]. Нехай P та Q – натуральні числа заданої довжини, $P = p_m\ p_{m-1}\ \dots\ p_2\ p_1$, $Q = q_m\ q_{m-1}\ \dots\ q_2\ q_1$ – їх двійкове представлення, $N = P \cdot Q$. Очевидно, N складається з $2m$ двійкових розрядів. Тоді існує 2^{2m-2} варіантів добутку $N = P \cdot Q$. Довжина m визначається як початково задана довжина ключа n, поділена на 2. Тоді кількість варіантів добутку дорівнює 2^{n-2}. Модифікований метод генерує натуральні прості числа довжиною $\frac{n}{3}$, відповідно кількість варіантів добутку $N = P \cdot Q \cdot F$ буде дорівнювати 2^{n-3}. Відповідно падіння криптостійкості можна оцінити як $\frac{2^{n-2}}{2^{n-3}}$, тобто у 2 рази.

2.4. Висновки

В першому підрозділі даного розділу проаналізовано класичний метод шифрування Вільямса та його математичне підґрунтя. Також розглянуто його швидкодію.

В другому розділі проаналізовано вразливості класичного методу шифрування Вільямса, а саме атаку на основі підбору шифротексту, її математичне підґрунтя, ймовірність успіху атаки та основні підходи для подолання даної вразливості.

В третьому розділі проаналізовано запропонований модифікований метод шифрування Вільямса як спосіб подолання вразливості до атаки на основі підбору шифротексту. Також проведено оцінку зменшення ймовірності зламу при використанні запропонованого модифікованого методу, його переваги і недоліки порівняно з класичним методом шифрування Вільямса.

3. ОБГРУНТУВАНЯ ВИБОРУ ЗАСОБІВ РЕАЛІЗАЦІЇ

3.1. Обґрунтування вибору мови програмування

Серед мов програмування для реалізації криптосистеми розглянемо найбільш популярні мови, що відповідають тенденціям використання у розвитку криптографії:

- Python;
- C;
- C++;
- C#;
- Java;
- JavaScript.

Розглянемо переваги і недоліки цих мов і порівняємо.

3.1.1. Python

Python – високорівнева об'єктно-орієнтована інтерпретована мова програмування загального призначення [18]. Вона орієнтована на підвищення продуктивності розробника і читання коду. Також можна відзначити мінімалістичний синтаксис ядра Python, а стандартна бібліотека містить великий обсяг корисних функцій.

Python підтримує велику кількість різноманітних парадигм програмування, зокрема: процедурне, функціональне, об'єктно-орієнтоване, імперативне і аспектно-орієнтоване програмування [19]. Серед основних архітектурних рис можна відзначити динамічну типізацію, автоматичне керування пам'яттю, повну інтроспекцію часу виконання, механізм обробки виключень, підтримку високорівневих структур даних, а також багатопоточні обчислення. Підтримується розбиття програм на модулі, що також можуть об'єднуватися в пакети. Крім того, є інтерактивний режим роботи.

Інтерпретатор даної мови CPython, що можна розширювати функціями та типами даних, розробленими на C, C++ або на іншій мові, яку можна викликати із C, підтримується більшістю сучасних платформ. Поширення під вільною ліцензією PSFL дозволяє використовувати і розповсюджувати Python без обмежень в будь-яких додатках.

Мова Python дуже швидко розвивається. Версії з оновленими мовними властивостями виходять у середньому раз на два чи три роки. Python – поширена, стабільна мова. Вона широко використовується багатьма великими компаніями в проектах найрізноманітніших напрямків, виконуючи функції як основної мови програмування, так і для створення розширень та інтеграції додатків. Останніми роками Python випереджає у популярності використання інші мови програмування, що протягом довгого часу посідали перші місця рейтингів, такі як C, C ++ і Java.

З моменту початку розробки у 1980-тих роках, мова зазнала різких змін. Використовуваний нині Python 3.0, представлений у 2008 році, порівняно з попередниками, має конструктивний дизайн, який уникає дублювання модулів і конструкцій. На даний момент, найновішою версією є Python 3.8.

Мова Python має наступні переваги:

- велика кількість корисних модулів, що дають можливості для роботи з мережевими протоколами, серверами та клієнтами, крос-платформними застосунками, графічним інтерфейсом, а також засобами для роботи з математичними функціями;
- портованість – мова працює майже на всіх відомих платформах, таких як Microsoft Windows, всі варіанти UNIX, Mac OS, iPhone OS, Palm OS, Amiga та Android, при цьому на всіх основних платформах Python має підтримку характерних для них специфічних технологій;

- продуктивність – інтеграція процесів, покращені можливості керування та система тестування модулів сприяють підвищенню швидкості для більшості програм та їх продуктивності;
- легка інтеграція – мова має потужні можливості керування, такі як виклик функцій безпосередньо через C, C++ або Java через Jython. Також Python обробляє XML та інші мови розмітки, оскільки працює на всіх сучасних операційних системах за допомогою одного і того ж байт-коду;
- простий синтаксис;
- дозволяє працювати з цілими числами довільного розміру.

Недоліки мови Python:

- оскільки Python є інтерпретованою мовою, швидкість виконання написаних на ній програм невисока порівняно з неінтерпретованими мовами;
- відсутність статичної типізації не дозволяє реалізувати механізм перевантаження функцій на етапі компіляції;
- помилки, що не виявляються на етапі компіляції, а лише під час виконання, що ускладнює тестування програми.

3.1.2. C

C – універсальна, компільована, процедурна, статично типізована, імперативна мова програмування [20]. Є однією з найпопулярніших мов програмування за кількістю вже написаних на ній програм, хоча спершу вона була розроблена для написання лише системного програмного забезпечення.

Компілятори мови C існують для багатьох операційних систем. Вони достатньо прості і реалізують направлення мови на мінімалізм. C забезпечує низькорівневий доступ до оперативної пам’яті і не вимагає великої динамічної підтримки.

Багато високорівневих мов програмування мають компілятори, інтерпретатори та бібліотеки, написані на C. Також деякі з них використовують C як проміжну мову.

Як основний засіб свого розширення дана мова використовує бібліотеки, що являють собою набір функцій.

Серед переваг мови C потрібно відзначити:

- швидкість виконання;
- реалізація на великій кількості апаратних платформ.

Недоліки даної мови:

- складність мови, що сприяє написанню складного, проблемного і заплутаного коду;
- різноманітні допущення у мові дозволяють програмі компілюватися з багатьма невиявленими помилками, що призводить до непередбачуваної поведінки програми;
- примітивна підтримка модульності;
- відсутність контролю над адресною арифметикою, а також контролю ініціалізації змінних і динамічної пам’яті сприяє вразливості програм;
- брак можливостей у роботі з багатьма прикладними задачами.

3.1.3. C++

C++ – статично типізована, компільована мова програмування загального призначення [21]. Підтримує багато парадигм програмування, такі як процедурне, об’єктно-орієнтоване та узагальнене програмування.

Мова використовується для системного програмування, написання драйверів, серверних та клієнтських програм, а також відеоігор. Синтаксично мова подібна до мови C. C++ багато в чому є розширеною версією мови C, зокрема додає об’єктно-орієнтовані можливості, вводячи класи, що забезпечують інкапсуляцію, успадкування та поліморфізм.

Стандартна бібліотека C++ заснована на Стандартній Бібліотеці Шаблонів (STL), що надає такі важливі інструменти як контейнери та ітератори.

Переваги мови C++:

- швидкість роботи програм мовою C++ практично не поступається швидкості роботи програм, написаних мовою C;
- мова підходить для розробки для найрізноманітніших систем і платформ;
- можливість роботи на низькому рівні з адресами, портами і пам’яттю;
- підтримка різноманітних стилів програмування.

Серед недоліків мови C++ потрібно відзначити:

- можливості, що зменшують безпеку програм і сприяють виникненню помилок, вразливостей і непередбачуваної роботи програми (подібно до мови C);
- примітивна підтримка модульності;
- неінтуїтивне перетворення типів даних;
- відсутня підтримка функціонального програмування.

3.1.4. C#

C# – об’єктно-орієнтована мова програмування для платформи Microsoft .NET Framework, має безпечну систему типізації та C-подібний синтаксис [22]. Мова підтримує строгу статичну типізацію, перевантаження операторів, поліморфізм, делегати, атрибути, події, анонімні функції, ітератори. Перейнявши багато особливостей від своїх попередників, мова C# не підтримує деякі моделі, що зарекомендували себе як проблемні.

На сьогоднішній день мова програмування C# є одною із найбільш потужних та затребуваних у найрізноманітніших сферах, вона швидко розвивається. C# створювалася як мова компонентного програмування, і

вона спрямована на можливість повторного використання створених компонентів.

Серед переваг мови C# варто відзначити:

- C# є потужною об'єктною мовою, що має можливості універсалізації й спадкування;
- C# створювався паралельно з каркасом .NET Framework і у повній мірі враховує всі його можливості, такі як FCL та CLR;
- C# містить кращі риси мов C/C++;
- завдяки каркасу .NET Framework, який став надбудовою над операційною системою, програмісти мови C# отримують ті ж переваги роботи з віртуальною машиною, що й програмісти мови Java, навіть більше, оскільки виконавче середовище CLR являє собою компілятор проміжної мови, у той час як віртуальна машина мови Java є інтерпретатором байт-коду;
- потужна бібліотека каркасів підтримує зручну побудову різних типів додатків на мові C#, що дозволяє легко будувати Web-служби та інші види компонентів, достатньо просто зберігати й отримувати інформацію з бази даних та інших сховищ даних;
- реалізація, яка сполучає побудову надійного й ефективного коду, є дуже важливим чинником, що сприяє успіху C#.

3.1.5. Java

Java – строго типізована об’єктно-орієнтована мова програмування [23]. Програми, написані даною мовою, транслюються в байт-код, що виконується віртуальною машиною Java – програмою, що опрацьовує байт-код, передаючи інструкції обладнанню, в якості інтерпретатора.

Java майже повністю запозичила синтаксис C та C++, а також об’єктну модель C++. Проте в ній усунуто можливість появи конфліктних

систуацій, зменшено відповідальність програміста за коректність роботи програми. Через орієнтацію передусім на платформну незалежність, Java має менше низькорівневих можливостей, а також нижчу швидкість виконання та збільшений обсяг використання пам’яті порівняно з мовами C та C++.

Переваги мови Java:

- незалежність від архітектури і переносимість;
- безпека від низькорівневих помилок;
- висока продуктивність виконання;
- автоматичне керування пам’яттю.

Недоліки мови:

- низька швидкість виконання програм;
- значне використання ресурсів пам’яті.

3.1.6. JavaScript

JavaScript – це прототипна, динамічна, об’єктно-орієнтована мова програмування [24]. Вона підтримує об’єктно-орієнтований, функціональний та імперативний стилі програмування. Найчастіше JavaScript використовується для створення сценаріїв веб-сторінок, проте мова також підходить для написання веб-застосунків, серверної сторони, а також стаціонарних та мобільних додатків.

Серед її архітектурних особливостей варто відзначити автоматичне керування пам’яттю, слабку динамічну типізацію, прототипне програмування та функції як об’єкти першого класу. Існує великий вибір бібліотек, що значно полегшує використання мови.

Переваги мови JavaScript:

- високий рівень абстракції;
- крос-браузерність;
- автоматичне приведення типів і збирання сміття;
- обробка винятків;

- спрощений синтаксис.

Недоліки мови:

- невисока швидкість роботи програм, оскільки мова є інтерпретатором;
- складність відлагодження;
- виконання програм у браузерах.

3.1.7. Порівняння проаналізованих мов

Порівняємо можливості описаних вище мов та їх відповідність до написання криптосистеми. Python, JavaScript, Java і C# мають захист від низькорівневих порушень роботи програми. Перші два мають спрощений синтаксис порівняно з C-подібними мовами, проте вони програють у швидкості виконання. Python наділений найбільш підходящим і в той же час простим функціоналом для роботи з математичними функціями та великими числами, що необхідно при створенні криптосистеми.

Згідно проаналізованих даних, було вирішено використовувати мову Python, адже вона має найбільш зручні інструменти для швидкої та якісної розробки криптосистеми.

3.2. Обґрунтування вибору бібліотек та платформи для реалізації та аналізу продуктивності роботи запропонованого модифікованого методу шифрування

3.2.1. Обґрунтування вибору бібліотек

Для аналізу розробленого програмного забезпечення, що реалізує запропонований модифікований метод шифрування та порівняння ефективності його роботи з класичним методом шифрування необхідно використати бібліотеки, що дозволяють будувати графіки та працювати з великими числами.

Matplotlib – бібліотека, написана мовою програмування Python для візуалізації даних двовимірної та тривимірної графіки [25]. Зображення,

що генеруються в різноманітних форматах можуть використовуватись в інтерактивній графіці, наукових публікаціях та графічному інтерфейсі користувача. Matplotlib являє собою гнучкий пакет, що легко конфігурується. Підтримуються всі види графіків та діаграм.

NumPy – відкритий модуль, написаний мовою програмування Python, що являє собою загальні математичні та числові операції у вигляді швидких функцій, об'єднаних у високорівневі пакети [26]. Функції оптимізовані для роботи з багатовимірними масивами. NumPy розглядається як альтернатива до MATLAB.

3.2.2. Обґрунтування вибору платформи

Для розширення можливостей мови Python та підвищення зручності розробки існує велика кількість засобів. Основними є:

- Jupiter Notebook;
- Django;
- Flask;
- Pyramid;
- Kivy.

Jupiter Notebook – це графічна веб-оболонка для інтерактивних обчислень [27]. Вона являє собою веб-додаток з відкритим кодом, що дозволяє створювати та обмінюватися документами з різноманітним вмістом, таким як: живий код, візуалізація, рівняння та текст. Використання включає очищення та перетворення даних, чисельне і статистичне моделювання, візуалізацію даних, машинне навчання та багато іншого. Jupiter Notebook дозволяє поєднати код, текст та діаграми на одній сторінці та легко передавати їх іншим користувачам.

За замовчуванням ноутбук Jupiter Notebook постачається з ядром IPython. IPython – це інтерактивна оболонка мови програмування Python, що забезпечує розширену інтроспекцію, додатковий командний синтаксис, підсвічування коду та автоматичне доповнення. Також вона може

інтерактивно керувати паралельними кластерами, використовуючи асинхронні статуси зворотних викликів. Основні преваги даної платформи:

- комплексна інтроспекція об'єктів;
- збереження історії введення та самої сесії протягом усіх сеансів;
- кешування вихідних результатів;
- додатковий командний синтаксис;
- доступ до системної оболонки;
- підключення безлічі клієнтів до одного обчислювального ядра і, завдяки своїй архітектурі, робота в паралельному кластері.

Django – відкритий високорівневий фреймворк для розробки веб-додатків, написаний мовою Python [28]. Істотною архітектурною відмінністю даного фреймворку є побудова веб-додатку з окремих модульних частин. Він дозволяє додати більшість стандартних функцій єдиним пакетом замість пошуку окремих бібліотек. Також Django спрощує типові задачі під час розробки веб-додатку, такі як авторизація, аутентифікація, URL-маршрутизація, конфігурування сервера, міграція схеми даних та адміністрування створеного веб-додатку, доступ до бази даних. У фреймворку присутні система опису шаблонів з тегами та наслідуванням, гнучка система кешування і фільтрації. Серед недоліків даного фреймворку варто зазначити надмірну монолітність

Flask – фреймворк для створення веб-додатків мовою програмування Python [29]. В його основі полягає інструментарій Werkzeug, а також шаблонізатор Jinja2. Даний фреймворк відноситься до категорії мікрофреймворків, тобто мінімалістичних каркасів для веб-додатків, що не вимагають спеціальних засобів чи бібліотек. Сам по собі Flask не містить рівень абстракції для роботи з формами, сервером, базою даних та іншими компонентами веб-додатку, проте він має підтримку розширень, що забезпечують додаткові можливості для розробки. Таким чином, даний

фреймворк компенсує звужену функціональність порівняно з full-stack фреймворками. Функціонал без додаткових розширень містить сервер для розробки і відлагоджувач, шаблони Jinja2, підтримку юніт-тестів і сесій на стороні клієнта. Основними недоліками даного фреймворку є погана підтримка модульності та роботи із запитами.

Pyramid – фреймворк для розробки веб-додатків з відкритим кодом, написаний мовою Python [30]. Даний фреймворк відрізняється гнучкістю, легко розширяється. Функціонал зосереджений на забезпеченні таких можливостей як опрацювання маршрутів, система подій, проста авторизація. Перевагами фреймворку Pyramid є:

- можливість написати додаток у одному файлі;
- заснована на декораторах конфігурація;
- гнучке налаштування відлагодження;
- додаткові модулі та шаблони, що здатні розширятись.

Також варто відзначити універсальність, фреймворк однаково ефективно працює з великими і невеликими додатками.

Kivy – бібліотека з відкритим кодом, що написана мовою Python [31]. Дана бібліотека призначена для створення крос-платформних додатків, дозволяє легко створювати графічний інтерфейс. Серед переваг даного фреймворку:

- широка підтримка вводу для подій, що стосуються мультитачів, характерних для миші, клавіатури та ОС;
- графічна бібліотека, що використовує лише OpenGL ES 2 та заснована на об'єкті Vertex Buffer і шейдерах;
- широкий спектр віджетів, які підтримують мультитач;
- проміжна мова Kv, яка використовується для легкого проектування користувальницьких віджетів.

Після аналізу найпопулярніших фреймворків для розробки додатків мовою Python було обрано Jupiter Notebook, як найбільш універсальний

інструмент, що дозволяє розміщувати код, текст та діаграми на одній сторінці та легко передавати їх іншим користувачам.

3.3. Висновки

У даному розділі проаналізовано переваги та недоліки шістьох мов програмування, що за статистикою частіше за інші використовуються розробниками програмного забезпечення. Внаслідок аналізу обрано інтерпретовану мову програмування Python, як мову, що має найбільше зручних інструментів для роботи з великими числами, для аналізу продуктивності роботи алгоритму та полегшення реалізації математичних операцій.

Для аналізу продуктивності роботи запропонованого модифікованого методу шифрування та порівняння з класичними методами шифрування було обрано модулі NumPy та Matplotlib, що дозволять ефективно проаналізувати великий обсяг даних та проілюструвати їх відповідними графіками.

Також було проаналізовано 5 найпопулярніших фреймворків для розробки мовою Python, в наслідок аналізу було обрано Jupiter Notebook, через його універсальність, модульність, зручність та відповідність розроблюваному програмному забезпеченню.

4. РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ ЗАПРОПОНОВАНОГО МОДИФІКОВАНОГО МЕТОДУ АСИМЕТРИЧНОГО ШИФРУВАННЯ

4.1. Особливості реалізації запропонованого модифікованого методу шифрування

Запропонований модифікований метод шифрування було реалізовано в окремому модулі, який містить функції, що реалізують необхідні алгоритми. Також було реалізовано модуль генерації великих простих чисел, модуль математичних алгоритмів, що використовуються у процесі шифрування, а також модуль, що реалізує операції прямої та зворотної конвертацію чисел в байти.

Для генерації простих чисел великої розрядності було вирішено використати тест Міллера-Рабіна. Це поліноміальний тест на простоту числа. Він спирається на перевірку ряду рівностей, що виконуються для простих чисел. Результат тесту не може строго довести, що число просте, але ймовірність цього досить висока. Використання тесту Міллера-Рабіна показало найкращий час роботи порівняно з детермінованими тестами на простоту. Середній час виконання при генерації числа розміром 512 біт склав 0,5 секунд.

В лістингу 1 наведено текст програми, що реалізує основні функції для генерації великих простих чисел.

Лістинг 1. Модуль генерації великих простих чисел

```
import utils


def get_prime(nbits):
    assert nbits > 3
    while True:
        integer = utils.read_random_odd_int(nbits)

        if is_prime(integer):
            return integer
```

```
def is_prime(number):

    if number < 10:
        return number in {2, 3, 5, 7}

    if not (number & 1):
        return False

    k = get_primality_testing_rounds(number)

    return miller_rabin_primality_testing(number, k + 1)


def get_primality_testing_rounds(number):

    bitsize = utils.bit_size(number)

    if bitsize >= 1536:
        return 3
    if bitsize >= 1024:
        return 4
    if bitsize >= 512:
        return 7

    return 10


def miller_rabin_primality_testing(n, k):

    if n < 2:
        return False

    d = n - 1
    r = 0
    while not (d & 1):
        r += 1
        d >>= 1

    for _ in range(k):
        # Generate random integer a, where 2 <= a <= (n - 2)
        a = utils.randint(n - 3) + 1
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == 1:
                # n is composite.
                return False
            if x == n - 1:
                # Exit inner loop and continue with next witness.
                break
        else:
            # If loop doesn't break, n is composite.
            return False
        return True
```

Згенеровані числа використовуються у функції генерації ключів шифрування. У лістингу 2 наведено реалізацію генерації ключів шифрування.

Лістинг 2. Генерація ключів шифрування

```
def gen_keys(nbits):
    # Regenerate p and q values, until calculate_keys doesn't raise a
    # ValueError.
    while True:
        (p, q, f) = find_p_q_f(nbits // 3)
        try:
            (c, s, n) = find_c_s(p, q, f)
            (w, e, d) = find_w_e_d(p, q, f, c)
            break
        except ValueError:
            pass
    return (n, e, c, s), d


def find_p_q_f(nbits):
    # Make sure that p and q aren't too close or the factoring programs can
    # factor n.
    shift = nbits // 16
    pbits = nbits + shift
    qbits = nbits - shift
    fbits = nbits
    # Choose the two initial primes
    while True:
        p = pg.get_prime(pbits)
        if (p - 3) % 4 == 0 and (p + 1) % 4 == 0:
            break
    while True:
        q = pg.get_prime(qbits)
        if q != p and (q - 3) % 4 == 0 and (q + 1) % 4 == 0:
            break
    while True:
        f = pg.get_prime(fbits)
        if f != p and f != q and (f - 3) % 4 == 0 and (f + 1) % 4 == 0:
            break
    return p, q, f


def find_c_s(p, q, f):
    n = p * q * f
    c = DEFAULT_EXPONENT
    while (p + m_alg.legendre_symbol(c, p)) % 4 != 0 or
            (q + m_alg.legendre_symbol(c, q)) % 4 != 0 or
                 (f + m_alg.legendre_symbol(c, f)) % 4 != 0:
        c += 1
    s = DEFAULT_EXPONENT
    while m_alg.jacobi_symbol((s ** 2) - c, n) != -1 or
                                       math.gcd(s, n) != 1:
        s += 1
    return c, s, n


def find_w_e_d(p, q, c):
```

```
    w = int((p - m_alg.legendre_symbol(c, p)) *
                            (q -m_alg.legendre_symbol(c, q)) / 4)
    e = DEFAULT_EXPONENT
    while math.gcd(e, w) != 1:
        e += 1
    d = int((int((w + 1) / 2) % w) / e)
    return w, e, d
```

Для підбору випадкових чисел замість використання алгоритму генерації випадкових чисел було обрано використовувати задане стартове значення числа і перевіряти його на відповідність умовам, поступово збільшуючи. Такий підхід значно підвищив швидкість виконання функції.

Функція приймає в якості параметра довжину ключа. Кожне з трьох простих чисел генеруються з довжиною, що є третиною від довжини ключа. Таким чином загальний обсяг даних та довжина ключів, з якими відбуваються обчислення, не збільшує свою довжину порівняно з класичним методом шифрування, зберігаючи при цьому підвищену криптостійкість.

При генерації ключів використовуються алгоритми, що реалізують обчислення математичних функцій, таких як символ Лежандра та його узагальнення – символ Якобі. Це мультиплікативні функції, що часто використовуються у теорії чисел. Дані функції разом з додатковими математичними операціями реалізовані у окремому модулі. Текст програми реалізації символів Лежандра та Якобі наведено у лістингу 3.

Лістинг 3. Реалізація обчислення символа Лежандра та символа Якобі

```
import math


def jacobi_symbol(a, b):
    if math.gcd(a, b) != 1:
        return 0
    r = 1
    if a < 0:
        a = -a
        if b % 4 == 3:
            r = -r
    while a != 0:
        t = 0
```

```
        while a % 2 == 0:
            t += 1
            a = a / 2
        if t % 2 == 1:
            if b % 8 == 3 or b % 8 == 5:
                r = -r
        if a % 4 == 3 and b % 4 == 3:
            r = -r
        c = a
        a = b % c
        b = c
    return r


def legendre_symbol(a, b):
    if a >= b or a < 0:
        return legendre_symbol(a % b, b)
    elif a == 0 or a == 1:
        return a
    elif a == 2:
        if b % 8 == 1 or b % 8 == 7:
            return 1
        else:
            return -1
    elif a == b - 1:
        if b % 4 == 1:
            return 1
        else:
            return -1
    elif not is_prime(a):
        factors = factorize(a)
        product = 1
        for pi in factors:
            product *= legendre_symbol(pi, b)
        return product
    else:
        if ((b - 1) / 2) % 2 == 0 or ((a - 1) / 2) % 2 == 0:
            return legendre_symbol(b, a)
        else:
            return (-1) * legendre_symbol(b, a)


def is_prime(a):
    return all(a % i for i in range(2, a))


def factorize(n):
    factors = []
    p = 2
    while True:
        while n % p == 0 and n > 0:
            factors.append(p)
            n = n / p
        p += 1
        if p > n / p:
            break
    if n > 1:
        factors.append(int(n))
    return factors
```

Після генерації ключі передаються методу шифрування і дешифрування. Їх реалізація наведена у лістингу 4.

Лістинг 4. Реалізація шифрування та дешифрування

```
def ol_encrypt(message, public):
    n, e, c, s = public

    if message < 0:
        raise ValueError('Only non-negative numbers are supported')

    if message > n:
        raise OverflowError("The message %i is too long for n=%i" %
(message, n))

    if m_alg.jacobi_symbol(pow(message, 2) - c, n) == 1:
        b1 = 0
        y = message + math.sqrt(c)
        inv_y = message - math.sqrt(c)

    elif m_alg.jacobi_symbol(pow(message, 2), n) == 1:
        b1 = 1
        y = (message + math.sqrt(c)) * (s + math.sqrt(c))
        inv_y = (message - math.sqrt(c)) * (s - math.sqrt(c))

    else:
        raise OverflowError("Wrong keys for message %i " % (message, n))

    a = y / inv_y
    inv_a = inv_y / y
    x = int((pow(a, e) + pow(inv_a, -e)) / 2)
    y = int((pow(a, e) - pow(inv_a, -e)) / (2 * math.sqrt(c)))
    b2 = a % 2
    b0 = int(x / y)

    return b0, b1, b2


def decrypt(crypto, private):
    c, s, d, n = private
    c0, c1, c2 = crypto

    a = (pow(c0, 2) + c) / (pow(c0, 2) - c)
    x = int((pow(a, d) + pow(inv_a, -d)) / 2)
    y = int((pow(a, d) - pow(inv_a, -d)) / (2 * math.sqrt(c)))
    a1 = x + y * math.sqrt(c)

    if a1 < 0 < c2 < 0 or a1 < 0 < c2:
         a1 = -a1
    if c1 == 1:
        a1 *= ((s - math.sqrt(c))/(s + math.sqrt(c)))

    message = int((a1 + 1) / (a1 - 1)) * math.sqrt(c)
    return message
```

4.2. Порівняння розробленого модифікованого методу шифрування з класичним

Для перевірки роботи запропонованого модифікованого методу асиметричного шифрування замірялась швидкість обчислень під час генерації ключів, шифрування та дешифрування для реалізацій як модифікованого методу шифрування, так і класичного. При перевірці використовувались ключі довжиною 128 біт, 265 біт, 512 біт, 1024 біта і 2048 біт. Також окремо порівнювався час роботи реалізації модифікованого методу шифрування ключами різної довжини.

Результати проведеного аналізу наведено у вигляді графіків.

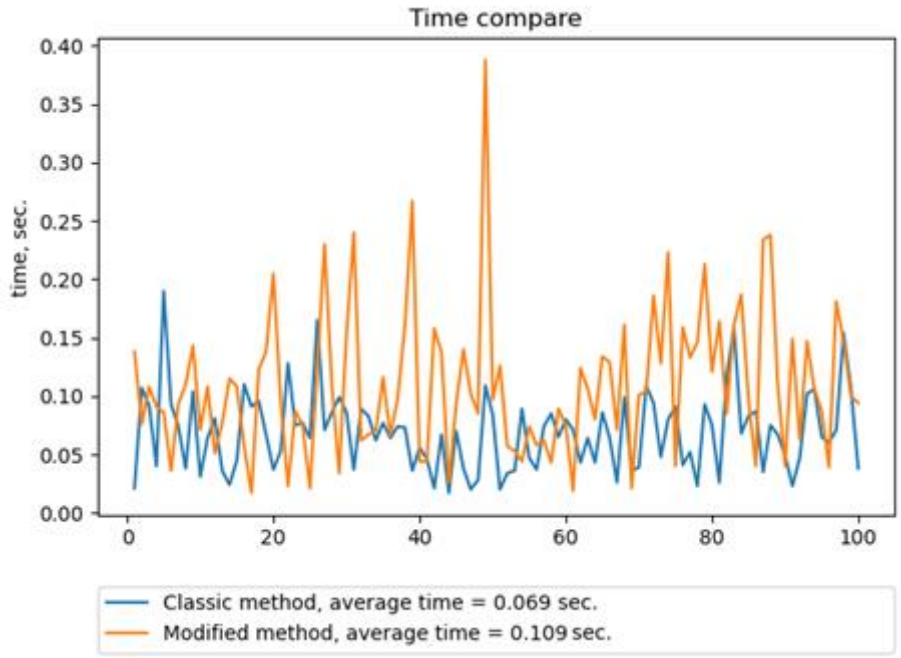


Рис. 4.1. Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 128 біт

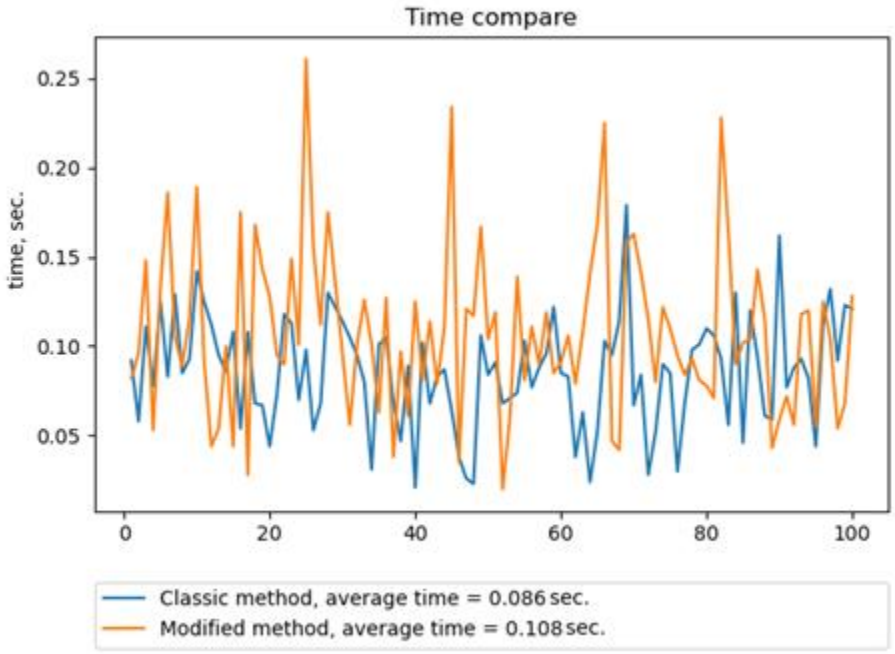


Рис. 4.2. Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 256 біт

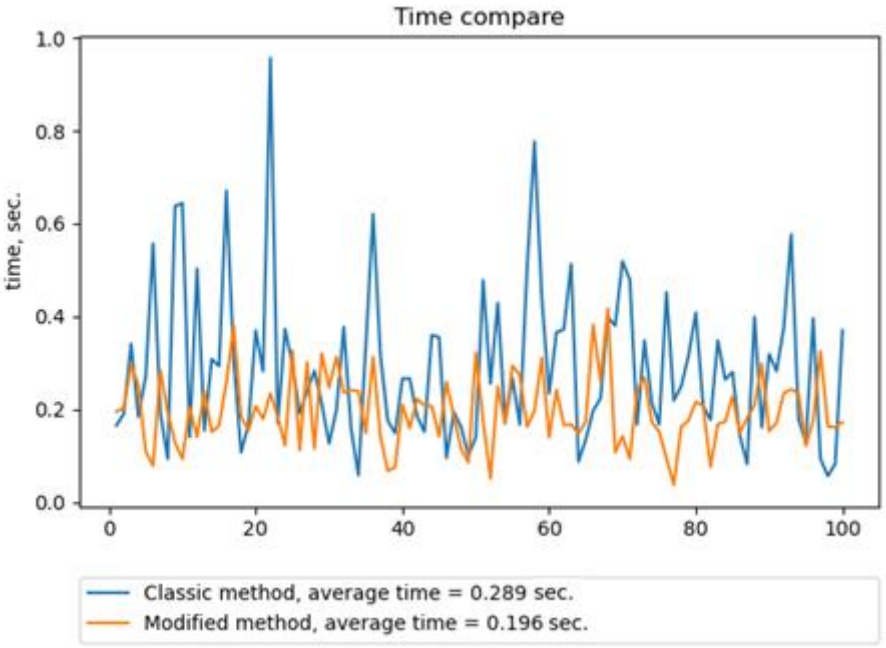


Рис. 4.3. Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 512 біт

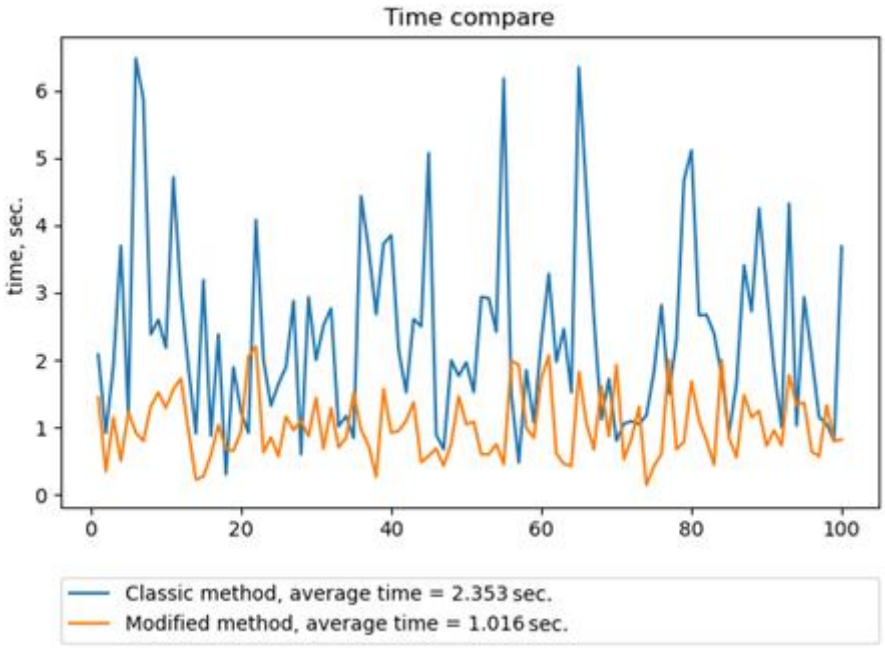


Рис. 4.4. Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування довжиною 1024 біт

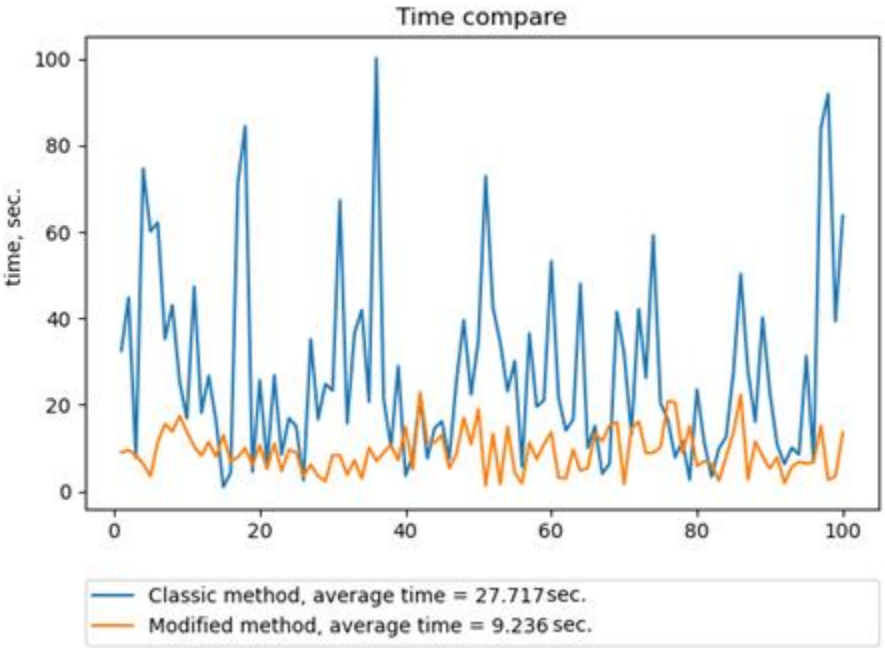


Рис. 4.5. Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 2048 біт

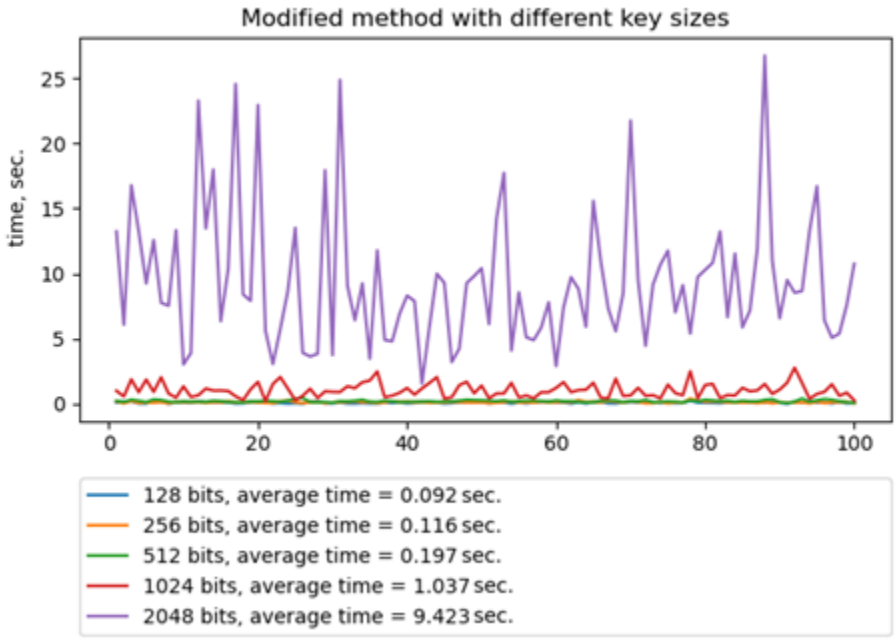


Рис. 4.6. Графіки часу роботи алгоритму, що реалізує модифікований метод шифрування з ключами довжиною 128, 256, 512, 1024 і 2048 біт

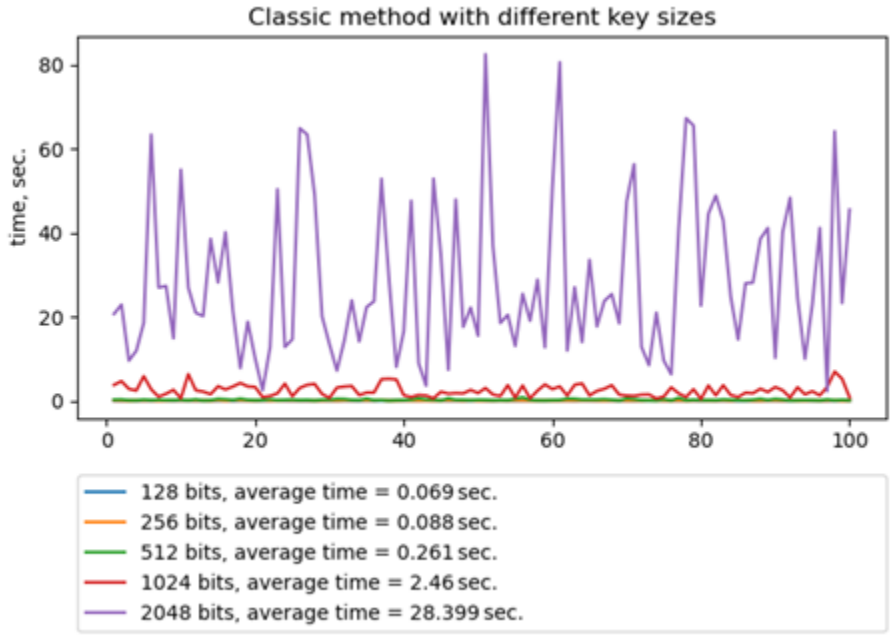


Рис. 4.7. Графіки часу роботи алгоритму, що реалізує класичний метод шифрування з ключами довжиною 128, 256, 512, 1024 і 2048 біт

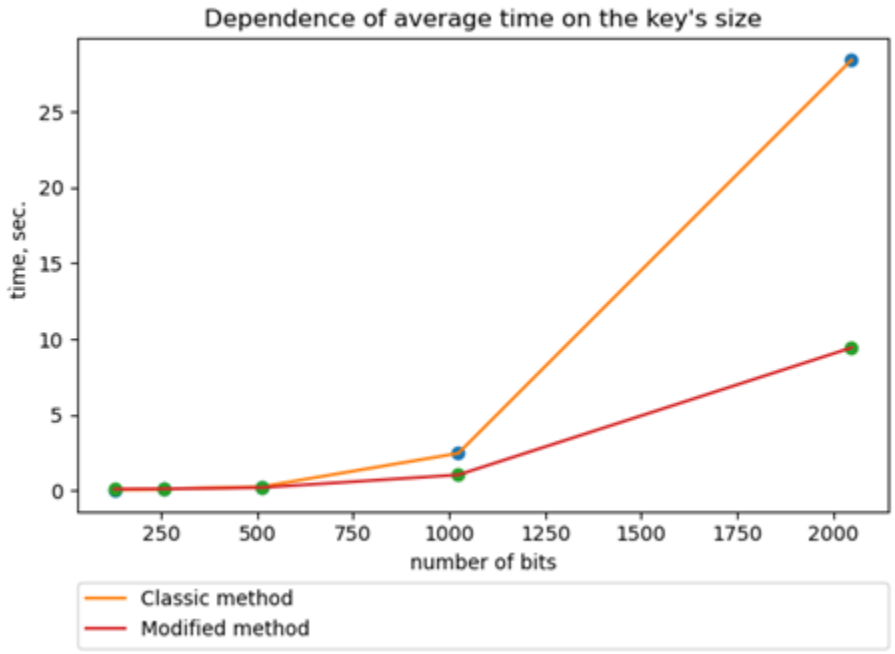


Рис. 4.8. Графік залежності середнього часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування, від довжини ключів

Проаналізувавши отримані графіки, можна зробити наступні висновки. На ключах невеликої довжини алгоритм, що реалізує модифікований метод, працює повільніше, проте на ключах після 500 біт він показує кращий час роботи, ніж реалізація класичного методу шифрування. Ключ довжиною 512 біт генерується в середньому на 28,5% швидше, ключ довжиною 1024 біта – у 2 рази швидше, а ключ довжиною 2048 біт – у 3 рази.

Результат можна пояснити тим, що основний час роботи алгоритму витрачається на генерацію простих чисел і залежить від їх розрядності. Алгоритм, що реалізує класичний метод шифрування, генеруючи ключ, що є складеним числом, довжиною n біт, генерує два прості числа довжиною $\frac{n}{2}$ біт. За алгоритмом реалізації модифікованого методу створюється складений ключ довжиною n біт з трьох простих чисел довжиною $\frac{n}{3}$ біт. Відповідно, числа меншої розрядності генеруються

швидше. Через це алгоритм модифікованого методу на довгих ключах працює швидше.

4.3. Висновки

У даному розділі проаналізовано розроблене програмне забезпечення, що реалізує модифікований метод асиметричного шифрування. Також було зроблено порівняння швидкості роботи алгоритмів, що реалізують модифікований метод і класичний. Це було зроблено шляхом заміру використаного часу при виконанні обчислень для генерації ключів, шифрування та дешифрування повідомлення.

У результаті проведеного дослідження виявлено, що запропонований модифікований метод асиметричного шифрування працює швидше на довгих ключах від 500 біт і більше.

ВИСНОВКИ

У даній роботі розроблено асиметричний метод шифрування, що є модифікацією класичного методу асиметричного шифрування Вільямса. Модифікація ґрунтується на використанні 3 простих чисел замість двох для обчислення ключів шифрування. Модифікація направлена на зменшення вразливості до атаки на основі підібраного шифротексту.

Під час виконання роботи було проаналізовано класичні методи асиметричного шифрування, а також існуючі модифікації класичних методів асиметричного шифрування. Особливо докладно проаналізовано метод шифрування Вільямса, його вразливості та недоліки, зокрема атаку на основі підібраного шифротексту. Проведено аналіз інструментів та технологій для проведення аналізу розробленого методу.

Результати дослідження розробленого модифікованого методу шифрування та класичного методу шифрування показали, що він працює швидше для ключів довжиною понад 500 біт.

Серед недоліків модифікованого методу наявна втрата стійкості до прямих атак, що основані на факторизації відкритого ключа n, у 2 рази.

Доцільним застосуванням модифікованого методу є використання довгих ключів шифрування. Таким чином, модифікований метод збереже свої переваги перед класичним методом, такі як швидкість роботи та підвищена стійкість до атаки на основі підбору шифротексту, а також буде достатньо стійким до атаки через факторизацію відкритого ключа завдяки його великій довжині.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Криптографія [Електронний ресурс] — Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Криптографія#Історія_криптографії
2. Diffie W., Hellman M. E. New Directions in Cryptography IEEE Trans. Inf. Theory / F. Kschischang — IEEE, 1976.
3. Онацкий А. В., Йона Л. Г. Асимметричные методы шифрования. – Модуль 2 Криптографические методы защиты информации в телекоммуникационных системах и сетях: учеб. пособие, 2011.
4. Rivest R., Shamir A., Adleman L. A method for obtaining digital signatures and public-key cryptosystems, Commun. ACM — New York City: ACM, 1978.
5. Arto Salomaa. Public-Key Cryptography, 1990.
6. Hugh C. Williams. Some Public-Key Crypto-Functions as Intractable as Factorization, 1985.
7. W.Stallings. Cryptography and Network Security, 2nd Edition, 1999.
8. Behrouz Forouzan. Cryptography and Network Security
9. Darrel Hankerson, Alfred Menezes, Scott Vanstone. Guide to Elliptic Curve Cryptography, 2004.
10. Douglas, R. Stinson. Cryptography - Theory and Practice, CRC Press, 1995.
11. Harsh Saha. Modified RSA Cryptosystem, 2015.
12. Carlo A. Intila, Bobby Gerardo, Ruji P. Medina. Modified Rsa Algorithm Based On Key Generation, 2019.
13. Sangeeta Patel, Partha P. Nayak. A Novel Method Of Encryption Using Modified Rsa Algorithm And Chinese Remainder Theorem.
14. Aditya Kumar. Improved RSA Algorithm based on cloud database using Proth Number and Mersenne Prime Number, 2019.
15. Криптосистема Уильямса [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Криптосистема_Уильямса

16. Атака на основе подобранного шифротекста [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/ Атака_на_основе_подобранного_шифротекста
17. Ш.Т.Ишмухаметов, Р.Г.Рубцова. О Сложности Задачи Факторизации Натуральных Чисел
18. Python [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Python
19. Марк Саммерфилд. Python на практике. — Переклад з англійської. — М.: ДМК Пресс, 2014.
20. C (язык программирования) [Електронний ресурс] — Режим доступу до ресурсу:https://ru.wikipedia.org/wiki/Си_(язык_программирования)
21. C++ (язык программирования) [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/C%2B%2B
22. C Sharp [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/C_Sharp
23. Java [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Java
24. JavaScript [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/JavaScript
25. NumPy [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/NumPy
26. Matplotlib [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/ Matplotlib
27. Project Jupyter [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Project_Jupyter
28. Django [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Django
29. Flask (веб-фреймворк) [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Flask_(веб-фреймворк)

30. Pyramid (программный каркас) [Електронний ресурс] — Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Pyramid_(программный_каркас)
31. Kivy (framework) [Електронний ресурс] — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Kivy_(framework)

ДОДАТКИ

Додаток 1

Копії графічних матеріалів

Діаграма класів реалізації модифікованого методу асиметричного шифрування

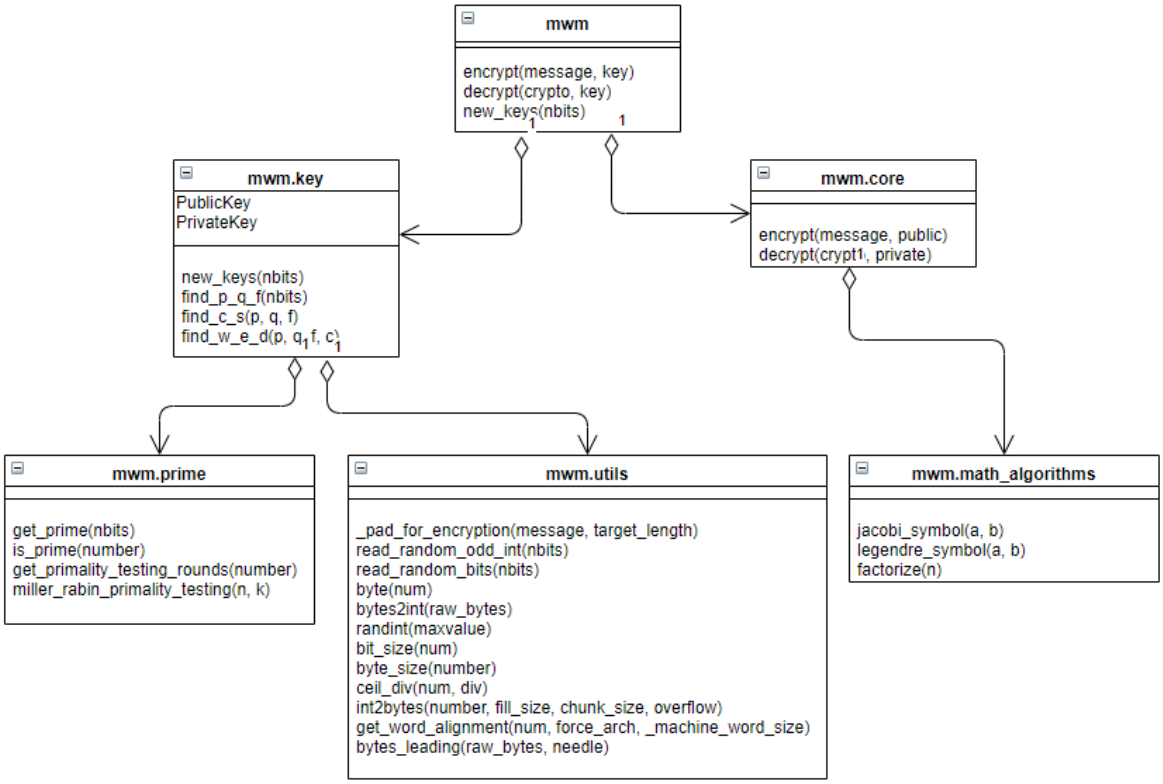


Схема класичного методу шифрування вільямса

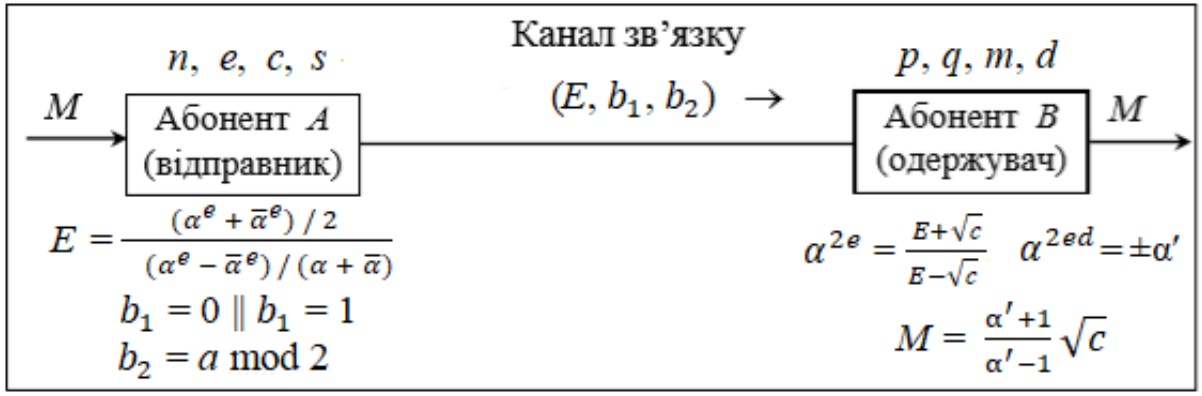
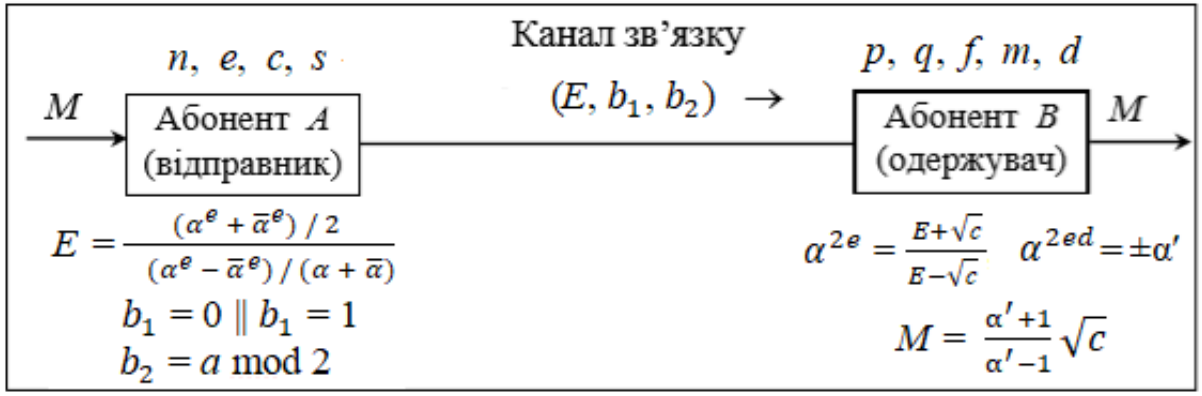
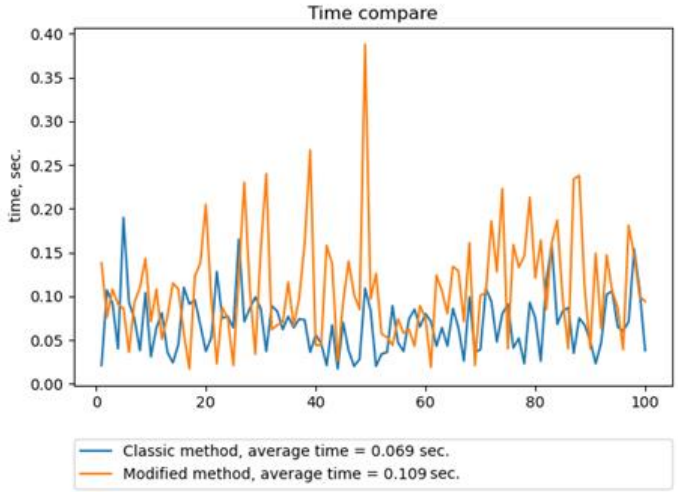


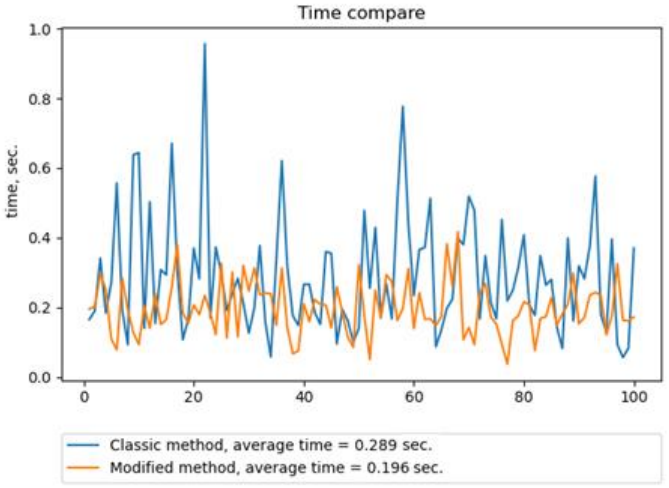
Схема модифікованого методу шифрування вільямса



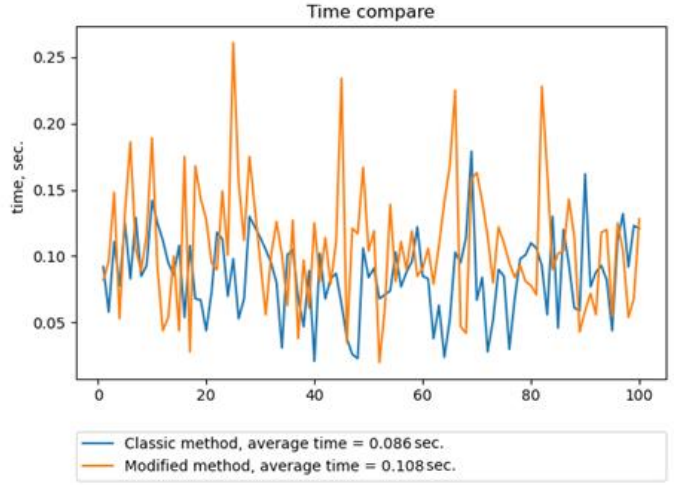
Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 128 біт



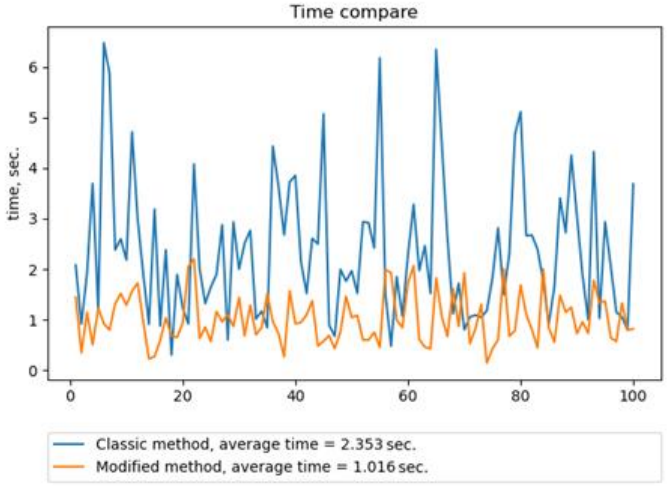
Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 512 біт



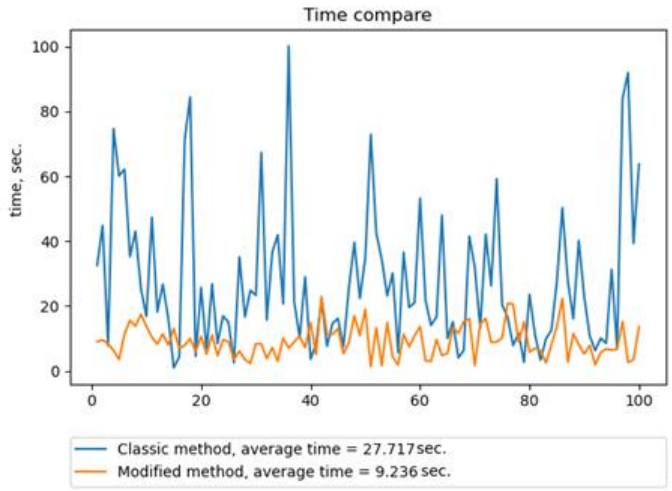
Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 256 біт



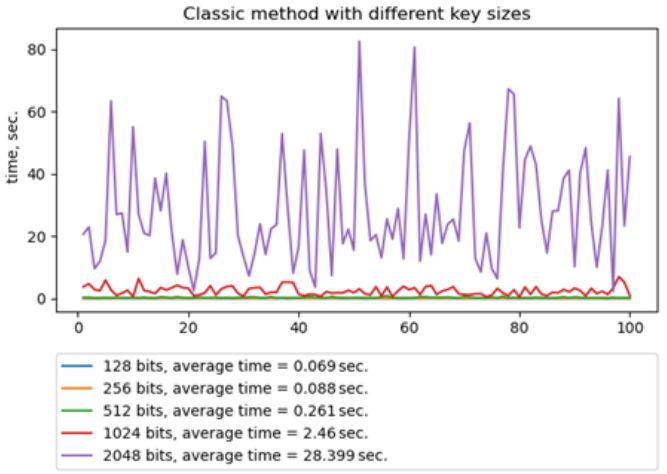
Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 1024 біт



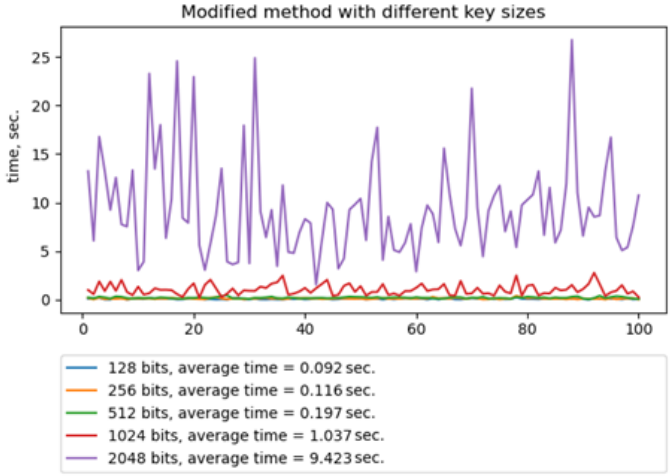
Графіки часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування з ключами довжиною 2048 біт



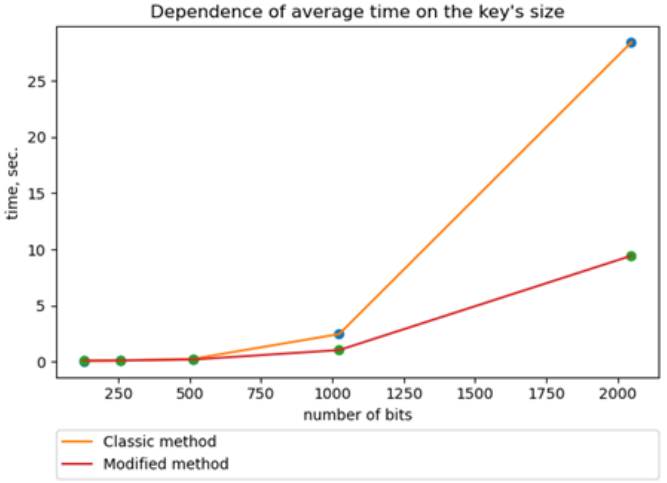
Графіки часу роботи алгоритму, що реалізує класичний метод шифрування з ключами довжиною 128, 256, 512, 1024 і 2048 біт



Графіки часу роботи алгоритму, що реалізує модифікований метод шифрування з ключами довжиною 128, 256, 512, 1024 і 2048 біт



Графік залежності середнього часу роботи алгоритмів, що реалізують класичний та модифікований методи шифрування, від довжини ключів



Додаток 2

Текст програми

```
from demo import Demo

scenario = int(input("Enter scenario number: "))
number = int(input("Number of cryptic tries: "))
nbits = int(input("Number of bits: "))

d = Demo(scenario, number, nbits)

import random
import analyse
import modified_method as mm

MIN_SIZE = 2
MAX_SIZE = 1000
DEFAULT_BITS = 512

class Demo:

    def __init__(self, scenario_number, n, nbits=DEFAULT_BITS,
min_size=MIN_SIZE, max_size=MAX_SIZE):
        self.messages = self.message_generator(n, min_size, max_size)
        if scenario_number == 1:
            self.scenario1(nbits)
        elif scenario_number == 2:
            self.scenario2()
        elif scenario_number == 3:
            self.scenario3()
        elif scenario_number == 4:
            self.scenario4(nbits)

    @staticmethod
    def message_generator(n, min_size, max_size):
        messages = []
        for i in range(1, n + 1):
            messages.append(random.randrange(min_size, max_size))
        return messages

    def scenario1(self, nbits):
        time_classic = analyse.time_classic(self.messages, nbits)
        time_modified = analyse.time_modified(self.messages, nbits)
        analyse.draw_compare(time_classic, time_modified, "Time compare")

    def scenario2(self):
        _time = []
        _time.append(analyse.time_modified(self.messages, 128))
        _time.append(analyse.time_modified(self.messages, 256))
        _time.append(analyse.time_modified(self.messages, 512))
        _time.append(analyse.time_modified(self.messages, 1024))
        _time.append(analyse.time_modified(self.messages, 2048))
        analyse.draw_many(_time, "Modified method with different key
sizes")

    def scenario3(self):
        _time = []
        _time.append(analyse.time_classic(self.messages, 128))
        _time.append(analyse.time_classic(self.messages, 256))
        _time.append(analyse.time_classic(self.messages, 512))
```

```
        _time.append(analyse.time_classic(self.messages, 1024))
        _time.append(analyse.time_classic(self.messages, 2048))
        analyse.draw_many(_time, "Classic method with different key sizes")

    def scenario4(self, nbits):
        message = int(input("Message: "))
        public, private = mm.gen_keys(nbits)
        print("Public key: ", public, " Private key: ", private)
        crypto = mm.encrypt(message, public)
        enc_message = mm.decrypt(crypto, private)
        print("Encrypted message: ", crypto, " Decrypted message: ",
enc_message)

import matplotlib.pyplot as plt
from datetime import datetime
import classic_metod as cm
import modified_method as mm

def draw_one_graph(_time, title):
    n = len(_time)
    x = []
    for i in range(1, n + 1, 1):
        x.append(i)
    plt.plot(x, _time)
    plt.title(title)
    plt.ylabel('time, sec.')
    plt.show()

def draw_compare(time1, time2, title):
    if len(time1) != len(time2):
        return ValueError
    n = len(time1)
    x = []
    for i in range(1, n + 1, 1):
        x.append(i)
    average_classic_time = calculate_average_time(time1)
    average_modified_time = calculate_average_time(time2)
    plt.plot(x, time1, label="Classic method, average time = " +
str(average_classic_time) + " sec.")
    plt.plot(x, time2, label="Modified method, average time = " +
str(average_modified_time) + " sec.")
    plt.title(title)
    plt.ylabel('time, sec.')
    plt.legend(bbox_to_anchor=(0, -0.15, 1, 0), loc=1, mode="expand",
borderaxespad=0)
    plt.show()

def draw_many(_time, title):
    n = len(_time[0])
    x = []
    for i in range(1, n + 1, 1):
        x.append(i)
    average_modified_time_1 = calculate_average_time(_time[0])
    average_modified_time_2 = calculate_average_time(_time[1])
    average_modified_time_3 = calculate_average_time(_time[2])
    average_modified_time_4 = calculate_average_time(_time[3])
    average_modified_time_5 = calculate_average_time(_time[4])
```

```
    plt.plot(x, _time[0], label="128 bits, average time = " +
str(average_modified_time_1) + " sec.")
    plt.plot(x, _time[1], label="256 bits, average time = " +
str(average_modified_time_2) + " sec.")
    plt.plot(x, _time[2], label="512 bits, average time = " +
str(average_modified_time_3) + " sec.")
    plt.plot(x, _time[3], label="1024 bits, average time = " +
str(average_modified_time_4) + " sec.")
    plt.plot(x, _time[4], label="2048 bits, average time = " +
str(average_modified_time_5) + " sec.")
    plt.title(title)
    plt.ylabel('time, sec.')
    plt.legend(bbox_to_anchor=(0, -0.15, 1, 0), loc=1, mode="expand",
borderaxespad=0)
    plt.show()

def draw_by_points(time1, time2, bits):
    plt.title("Dependence of average time on the key's size")
    plt.plot(bits, time1, 'o')
    plt.plot(bits, time1, label="Classic method")
    plt.plot(bits, time2, 'o')
    plt.plot(bits, time2, label="Modified method")
    plt.ylabel('time, sec.')
    plt.xlabel('number of bits')
    plt.legend(bbox_to_anchor=(0, -0.15, 1, 0), loc=1, mode="expand",
borderaxespad=0)
    plt.show()

def calculate_average_time(_time):
    n = len(_time)
    s = 0.0
    for t in _time:
        s += t
    return round(s / n, 3)

def time_classic(messages, nbits):
    times = []
    for m in messages:
        start_time = datetime.now()
        public, private = cm.gen_keys(nbits)
        crypto = cm.encrypt(m, public)
        enc_message = cm.decrypt(crypto, private)
        times.append((datetime.now() - start_time).total_seconds())
    return times

def time_modified(messages, nbits):
    times = []
    for m in messages:
        start_time = datetime.now()
        public, private = mm.gen_keys(nbits)
        crypto = mm.encrypt(m, public)
        enc_message = mm.decrypt(crypto, private)
        times.append((datetime.now() - start_time).total_seconds())
    return times

import math
import prime_generator as pg
```

```
import math_algorithms as m_alg

DEFAULT_EXPONENT = 65537

def encrypt(message, public):
    n, e, c, s = public
    if message < 0:
        raise ValueError('Only non-negative numbers are supported')
    if message > n:
        raise OverflowError("The message %i is too long for n=%i" %
(message, n))

    if m_alg.jacobi_symbol(pow(message, 2) - c, n) == 1:
        b1 = 0
        y = message + math.sqrt(c)
        inv_y = message - math.sqrt(c)
    elif m_alg.jacobi_symbol(pow(message, 2) - c, n) == -1:
        b1 = 1
        y = (message + math.sqrt(c)) * (s + math.sqrt(c))
        inv_y = (message - math.sqrt(c)) * (s - math.sqrt(c))
    else:
        raise ValueError
    a = y / inv_y
    b2 = a % 2
    b0 = m_alg.culc_func(message, e, n, a)
    return b0, b1, b2

def decrypt(crypto, private):
    c, s, d, n = private
    c0, c1, c2 = crypto
    a = (pow(c0, 2) + c) / (pow(c0, 2) - c)
    inv_a = (pow(c0, 2) - c) / (pow(c0, 2) + c)
    x = int((pow(a, d) + pow(inv_a, -d)) / 2)
    y = int((pow(a, d) - pow(inv_a, -d)) / (2 * math.sqrt(c)))
    a1 = x + y * math.sqrt(c)
    if a1 < 0 < c2 < 0 or a1 < 0 < c2:
        a1 = -a1
    if c1 == 1:
        a1 *= ((s - math.sqrt(c)) / (s + math.sqrt(c)))
    message = m_alg.culc_func(c0, d, n, a1)
    return message


def gen_keys(nbits):
    # Regenerate p and q values, until calculate_keys doesn't raise a
    # ValueError.
    while True:
        (p, q) = find_p_q(nbits // 2)
        try:
            (c, s, n) = find_c_s(p, q)
            (e, d, w) = find_w_e_d(p, q, c)
            break
        except ValueError:
            pass
    return (n, e, c, s), (c, s, d, n)

def find_p_q(nbits):
```

```
    # Make sure that p and q aren't too close or the factoring programs can
    # factor n.
    shift = nbits // 16
    pbits = nbits + shift
    qbits = nbits - shift
    # Choose the two initial primes
    while True:
        p = pg.get_prime(pbits)
        if (p - 3) % 4 == 0 and (p + 1) % 4 == 0:
            break
    while True:
        q = pg.get_prime(qbits)
        if q != p and (q - 3) % 4 == 0 and (q + 1) % 4 == 0:
            break
    return p, q

def find_c_s(p, q):
    n = p * q
    c = DEFAULT_EXPONENT
    while (p + m_alg.legendre_symbol(c, p)) % 4 != 0 or (q +
m_alg.legendre_symbol(c, q)) % 4 != 0:
        c += 1
    s = DEFAULT_EXPONENT
    while m_alg.jacobi_symbol((s ** 2) - c, n) != -1 or math.gcd(s, n) !=
1:
        s += 1
    return c, s, n

def find_w_e_d(p, q, c):
    w = ((p - m_alg.legendre_symbol(c, p)) * (q - m_alg.legendre_symbol(c,
q))) / 4
    e, d = m_alg.calculate_keys(p, q, exponent=DEFAULT_EXPONENT)
    return e, d, w
import math
import prime_generator as pg
import math_algorithms as m_alg

DEFAULT_EXPONENT = 65537

def encrypt(message, public):
    n, e, c, s = public
    if message < 0:
        raise ValueError('Only non-negative numbers are supported')

    if message > n:
        raise OverflowError("The message %i is too long for n=%i" %
(message, n))
    if m_alg.jacobi_symbol(pow(message, 2) - c, n) == 1:
        b1 = 0
        y = message + math.sqrt(c)
        inv_y = message - math.sqrt(c)
    elif m_alg.jacobi_symbol(pow(message, 2) - c, n) == -1:
        b1 = 1
        y = (message + math.sqrt(c)) * (s + math.sqrt(c))
        inv_y = (message - math.sqrt(c)) * (s - math.sqrt(c))
    else:
        raise ValueError
```

```
    a = y / inv_y
    b2 = a % 2
    b0 = m_alg.culc_func(message, e, n, a)
    return b0, b1, b2

def decrypt(crypto, private):
    c, s, d, n = private
    c0, c1, c2 = crypto
    a = (pow(c0, 2) + c) / (pow(c0, 2) - c)
    inv_a = (pow(c0, 2) - c) / (pow(c0, 2) + c)
    x = int((pow(a, d) + pow(inv_a, -d)) / 2)
    y = int((pow(a, d) - pow(inv_a, -d)) / (2 * math.sqrt(c)))
    a1 = x + y * math.sqrt(c)
    if a1 < 0 < c2 < 0 or a1 < 0 < c2:
        a1 = -a1
    if c1 == 1:
        a1 *= ((s - math.sqrt(c)) / (s + math.sqrt(c)))
    message = pow(c0, d, n)
    return message

def gen_keys(nbits):
    # Regenerate p and q values, until calculate_keys doesn't raise a
    # ValueError.
    while True:
        (p, q, f) = find_p_q_f(nbits // 3)
        try:
            (c, s, n) = find_c_s(p, q, f)
            (e, d, w) = find_w_e_d(p, q, f, c)
            break
        except ValueError:
            pass
    return (n, e, c, s), (c, s, d, n)

def find_p_q_f(nbits):
    # Make sure that p and q aren't too close or the factoring programs can
    # factor n.
    shift = nbits // 16
    pbits = nbits + shift
    qbits = nbits - shift
    fbits = nbits
    # Choose the two initial primes
    while True:
        p = pg.get_prime(pbits)
        if (p - 3) % 4 == 0 and (p + 1) % 4 == 0:
            break
    while True:
        q = pg.get_prime(qbits)
        if q != p and (q - 3) % 4 == 0 and (q + 1) % 4 == 0:
            break
    while True:
        f = pg.get_prime(fbits)
        if f != p and f != q and (f - 3) % 4 == 0 and (f + 1) % 4 == 0:
            break
    return p, q, f

def find_c_s(p, q, f):
    n = p * q * f
```

```
    c = DEFAULT_EXPONENT
    while (p + m_alg.legendre_symbol(c, p)) % 4 != 0 \
            or (q + m_alg.legendre_symbol(c, q)) % 4 != 0 or (f +
m_alg.legendre_symbol(c, f)) % 4 != 0:
        c += 1
    s = DEFAULT_EXPONENT
    while m_alg.jacobi_symbol((s ** 2) - c, n) != -1 or math.gcd(s, n) !=
1:
        s += 1
    return c, s, n

def find_w_e_d(p, q, f, c):
    w = (p - m_alg.legendre_symbol(c, p)) * (q - m_alg.legendre_symbol(c,
q)) * (f - m_alg.legendre_symbol(c, f)) / 8
    e, d = m_alg.mod_calculate_keys(p, q, f, exponent=DEFAULT_EXPONENT)
    return e, d, w

import math


def jacobi_symbol(a, b):
    if math.gcd(a, b) != 1:
        return 0
    r = 1
    if a < 0:
        a = -a
        if b % 4 == 3:
            r = -r
    while a != 0:
        t = 0
        while a % 2 == 0:
            t += 1
            a = a / 2
        if t % 2 == 1:
            if b % 8 == 3 or b % 8 == 5:
                r = -r
        if a % 4 == 3 and b % 4 == 3:
            r = -r
        c = a
        a = b % c
        b = c
    return r


def legendre_symbol(a, b):
    if a >= b or a < 0:
        return legendre_symbol(a % b, b)
    elif a == 0 or a == 1:
        return a
    elif a == 2:
        if b % 8 == 1 or b % 8 == 7:
            return 1
        else:
            return -1
    elif a == b - 1:
        if b % 4 == 1:
            return 1
```

```
        else:
            return -1
    elif not is_prime(a):
        factors = factorize(a)
        product = 1
        for pi in factors:
            product *= legendre_symbol(pi, b)
        return product
    else:
        if not ((b - 1) // 2) % 2 == 0 or ((a - 1) // 2) % 2:
            return legendre_symbol(b, a)
        else:
            return (-1) * legendre_symbol(b, a)


def is_prime(a):
    return all(a % i for i in range(2, a))


def factorize(n):
    factors = []
    p = 2
    while True:
        while n % p == 0 and n > 0:
            factors.append(p)
            n = n / p
        p += 1
        if p > n / p:
            break
    if n > 1:
        factors.append(int(n))
    return factors


def calculate_keys(p, q, exponent):
    phi_n = (p - 1) * (q - 1)

    try:
        d = inverse(exponent, phi_n)
    except NotRelativePrimeError as ex:
        raise NotRelativePrimeError(
            exponent, phi_n, ex.d,
            msg="e (%d) and phi_n (%d) are not relatively prime
(divider=%i)" %
                (exponent, phi_n, ex.d))

    if (exponent * d) % phi_n != 1:
        raise ValueError("e (%d) and d (%d) are not mult. inv. modulo "
                         "phi_n (%d)" % (exponent, d, phi_n))

    return exponent, d


def mod_calculate_keys(p, q, f, exponent):
    phi_n = (p - 1) * (q - 1) * (f - 1)

    try:
```

```
        d = inverse(exponent, phi_n)
    except NotRelativePrimeError as ex:
        raise NotRelativePrimeError(
            exponent, phi_n, ex.d,
            msg="e (%d) and phi_n (%d) are not relatively prime
(divider=%i)" %
                (exponent, phi_n, ex.d))

    if (exponent * d) % phi_n != 1:
        raise ValueError("e (%d) and d (%d) are not mult. inv. modulo "
                         "phi_n (%d)" % (exponent, d, phi_n))

    return exponent, d


def inverse(x, n):
    (divider, inv, _) = extended_gcd(x, n)

    if divider != 1:
        raise NotRelativePrimeError(x, n, divider)

    return inv


def extended_gcd(a, b):
    # r = gcd(a,b) i = multiplicitive inverse of a mod b
    #      or      j = multiplicitive inverse of b mod a
    # Neg return values for i or j are made positive mod b or a
respectively
    # Iterateive Version is faster and uses much less stack space
    x = 0
    y = 1
    lx = 1
    ly = 0
    oa = a  # Remember original a/b to remove
    ob = b  # negative values from return results
    while b != 0:
        q = a // b
        (a, b) = (b, a % b)
        (x, lx) = ((lx - (q * x)), x)
        (y, ly) = ((ly - (q * y)), y)
    if lx < 0:
        lx += ob  # If neg wrap modulo orignal b
    if ly < 0:
        ly += oa  # If neg wrap modulo orignal a
    return a, lx, ly  # Return only positive values


def culc_func(a, b, c, d):
    return pow(a, b, c)


class NotRelativePrimeError(ValueError):
    def __init__(self, a, b, d, msg=None):
        super(NotRelativePrimeError, self).__init__(
            msg or "%d and %d are not relatively prime, divider=%i" % (a,
b, d))
```

```
        self.a = a
        self.b = b
        self.d = d

import utils


def get_prime(nbits):
    assert nbits > 3  # the loop wil hang on too small numbers

    while True:
        integer = utils.read_random_odd_int(nbits)

        # Test for primeness
        if is_prime(integer):
            return integer


def is_prime(number):

    # Check for small numbers.
    if number < 10:
        return number in {2, 3, 5, 7}

    # Check for even numbers.
    if not (number & 1):
        return False

    # Calculate minimum number of rounds.
    k = get_primality_testing_rounds(number)

    # Run primality testing with (minimum + 1) rounds.
    return miller_rabin_primality_testing(number, k + 1)


def get_primality_testing_rounds(number):
    # Calculate number bitsize.
    bitsize = utils.bit_size(number)
    # Set number of rounds.
    if bitsize >= 1536:
        return 3
    if bitsize >= 1024:
        return 4
    if bitsize >= 512:
        return 7
    # For smaller bitsizes, set arbitrary number of rounds.
    return 10


def miller_rabin_primality_testing(n, k):
    # prevent potential infinite loop when d = 0
    if n < 2:
        return False

    # Decompose (n - 1) to write it as (2 ** r) * d
    # While d is even, divide it by 2 and increase the exponent.
    d = n - 1
```

```
    r = 0

    while not (d & 1):
        r += 1
        d >>= 1

    # Test k witnesses.
    for _ in range(k):
        # Generate random integer a, where 2 <= a <= (n - 2)
        a = utils.randint(n - 3) + 1

        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue

        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == 1:
                # n is composite.
                return False
            if x == n - 1:
                # Exit inner loop and continue with next witness.
                break
        else:
            # If loop doesn't break, n is composite.
            return False

    return True


import os
import binascii
from struct import pack


def _pad_for_encryption(message, target_length):
    max_msglength = target_length - 11
    msglength = len(message)

    if msglength > max_msglength:
        raise OverflowError('%i bytes needed for message, but there is
only'
                            ' space for %i' % (msglength, max_msglength))

    # Get random padding
    padding = b''
    padding_length = target_length - msglength - 3

    # We remove 0-bytes, so we'll end up with less padding than we've asked
for,
    # so keep adding data until we're at the correct length.
    while len(padding) < padding_length:
        needed_bytes = padding_length - len(padding)

        # Always read at least 8 bytes more than we need, and trim off the
rest
        # after removing the 0-bytes. This increases the chance of getting
```

```
        # enough bytes, especially when needed_bytes is small
        new_padding = os.urandom(needed_bytes + 5)
        new_padding = new_padding.replace(b'\x00', b'')
        padding = padding + new_padding[:needed_bytes]

    assert len(padding) == padding_length

    return b''.join([b'\x00\x02',
                     padding,
                     b'\x00',
                     message])


def read_random_odd_int(nbits):

    value = read_random_int(nbits)

    # Make sure it's odd
    return value | 1


def read_random_int(nbits):
    randomdata = read_random_bits(nbits)
    value = bytes2int(randomdata)

    # Ensure that the number is large enough to just fill out the required
    # number of bits.
    value |= 1 << (nbits - 1)

    return value


def read_random_bits(nbits):
    nbytes, rbits = divmod(nbits, 8)

    # Get the random bytes
    randomdata = os.urandom(nbytes)

    # Add the remaining random bits
    if rbits > 0:
        randomvalue = ord(os.urandom(1))
        randomvalue >>= (8 - rbits)
        randomdata = byte(randomvalue) + randomdata

    return randomdata


def byte(num):
    return pack("B", num)


def bytes2int(raw_bytes):
    return int(binascii.hexlify(raw_bytes), 16)


def randint(maxvalue):
    b_size = bit_size(maxvalue)
```

```
    tries = 0
    while True:
        value = read_random_int(b_size)
        if value <= maxvalue:
            break

        if tries % 10 == 0 and tries:
            # After a lot of tries to get the right number of bits but
still
            # smaller than maxvalue, decrease the number of bits by 1.
That'll
            # dramatically increase the chances to get a large enough
number.
            b_size -= 1
        tries += 1

    return value


def bit_size(num):
    try:
        return num.bit_length()
    except AttributeError:
        raise TypeError('bit_size(num) only supports integers, not %r' %
type(num))


def byte_size(number):
    if number == 0:
        return 1
    return ceil_div(bit_size(number), 8)


def ceil_div(num, div):
    quanta, mod = divmod(num, div)
    if mod:
        quanta += 1
    return quanta


def int2bytes(number, fill_size=None, chunk_size=None, overflow=False):
    if number < 0:
        raise ValueError("Number must be an unsigned integer: %d" % number)

    if fill_size and chunk_size:
        raise ValueError("You can either fill or pad chunks, but not both")

    # Ensure these are integers.
    number & 1

    raw_bytes = b''

    # Pack the integer one machine word at a time into bytes.
    num = number
    word_bits, _, max_uint, pack_type = get_word_alignment(num)
    pack_format = ">%s" % pack_type
```

```
    while num > 0:
        raw_bytes = pack(pack_format, num & max_uint) + raw_bytes
        num >>= word_bits
    # Obtain the index of the first non-zero byte.
    zero_leading = bytes_leading(raw_bytes)
    if number == 0:
        raw_bytes = b'\x00'
    # De-padding.
    raw_bytes = raw_bytes[zero_leading:]

    length = len(raw_bytes)
    if fill_size and fill_size > 0:
        if not overflow and length > fill_size:
            raise OverflowError(
                    "Need %d bytes for number, but fill size is %d" %
                    (length, fill_size)
            )
        raw_bytes = raw_bytes.rjust(fill_size, b'\x00')
    elif chunk_size and chunk_size > 0:
        remainder = length % chunk_size
        if remainder:
            padding_size = chunk_size - remainder
            raw_bytes = raw_bytes.rjust(length + padding_size, b'\x00')
    return raw_bytes


def get_word_alignment(num, force_arch=64, _machine_word_size=64):
    max_uint64 = 0xffffffffffffffff
    max_uint32 = 0xffffffff
    max_uint16 = 0xffff
    max_uint8 = 0xff

    if force_arch == 64 and _machine_word_size >= 64 and num > max_uint32:
        # 64-bit unsigned integer.
        return 64, 8, max_uint64, "Q"
    elif num > max_uint16:
        # 32-bit unsigned integer
        return 32, 4, max_uint32, "L"
    elif num > max_uint8:
        # 16-bit unsigned integer.
        return 16, 2, max_uint16, "H"
    else:
        # 8-bit unsigned integer.
        return 8, 1, max_uint8, "B"


def bytes_leading(raw_bytes, needle=b'\x00'):
    leading = 0
    # Indexing keeps compatibility between Python 2.x and Python 3.x
    _byte = needle[0]
    for x in raw_bytes:
        if x == _byte:
            leading += 1
        else:
            break
    return leading
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

МОДИФІКОВАНИЙ МЕТОД ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ АСИМЕТРИЧНОГО ШИФРУВАННЯ

Виконав: Квітка Олександр Вячеславович

Керівник: доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович

Київ – 2020

АКТУАЛЬНІСТЬ

- Довжина ключів асиметричних алгоритмів постійно зростає.
- З 2013 року більшість браузерів перестали підтримувати сертифікати з довжиною ключів менше 2048 біт.
- Збільшення ключів значно уповільнює роботу алгоритмів шифрування.

Тому створення модифікації методу асиметричного шифрування, що працює швидше, не втрачаючи криптостійкості, є актуальним.

МЕТА РОБОТИ

Метою дипломної роботи є розроблення модифікованого методу асиметричного шифрування.

ЗАВДАННЯ

1. Проаналізувати класичні методи асиметричного шифрування та їх існуючі модифікації.
2. Проаналізувати вразливості існуючих методів.
3. Розробити модифікований метод асиметричного шифрування, що усуває або знижує знайдену вразливість.
4. Розробити програмне забезпечення, що реалізує розроблений модифікований метод.
5. Порівняти час виконання реалізації модифікованого методу та класичного.

ПРЕДМЕТ ДОСЛІДЖЕННЯ

Процес генерації ключів, шифрування та дешифрування у схемі асиметричного шифрування

ОБ'ЄКТ ДОСЛІДЖЕННЯ

Методи асиметричного шифрування

МАТЕМАТИЧНЕ ПІДҐРУНТЯ ДЛЯ МЕТОДУ АСИМЕТРИЧНОГО ШИФРУВАННЯ ВІЛЬЯМСА. ГЕНЕРАЦІЯ КЛЮЧІВ

1. $\delta_p = \left(\frac{c}{p}\right) = -p \pmod 4$,

   $\delta_q = \left(\frac{c}{q}\right) = -q \pmod 4$

2. $J\left(\frac{s^2-c}{n}\right) = -1$, НСД(s, n) = 1

3. $m = (p - \delta_p)(q - \delta_q) / 4$

4. НСД(d, m) = 1

5. $e = \frac{m+1}{2} d^{-1} \pmod m$

Відкритий ключ (n, e, c, s), закритий ключ (p, q, m, d)

МАТЕМАТИЧНЕ ПІДҐРУНТЯ ДЛЯ МЕТОДУ АСИМЕТРИЧНОГО ШИФРУВАННЯ ВІЛЬЯМСА

Шифрування

6. $J\left(\frac{M^2-c}{n}\right)$
7. $\gamma 1 = M + \sqrt{c};$
   $\gamma 2 = (M + \sqrt{c})\,(s + \sqrt{c})$
8. $\alpha = \frac{\gamma}{\bar{\gamma}},\ \bar{\gamma} = M - \sqrt{c}$
9. $E = \frac{(\alpha^e + \bar{\alpha}^e)\,/\,2}{(\alpha^e - \bar{\alpha}^e)\,/\,(\alpha + \bar{\alpha})}$

Надсилається (E, b_1, b_2)

Дешифрування

10. $\alpha^{2e} = \frac{E+\sqrt{C}}{E-\sqrt{C}}$
11. $\frac{s-\sqrt{C}}{s+\sqrt{C}}$
12. $M = \frac{\alpha'+1}{\alpha'-1}\sqrt{c}.$

СХЕМА КЛАСИЧНОГО МЕТОДУ ШИФРУВАННЯ ВІЛЬЯМСА

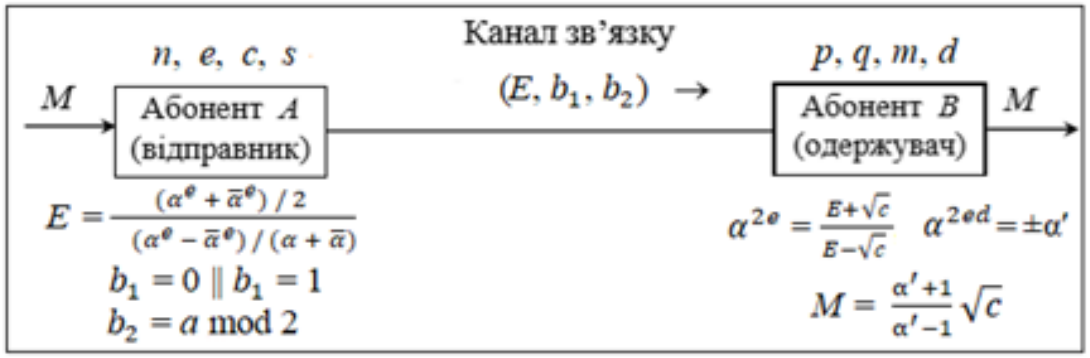
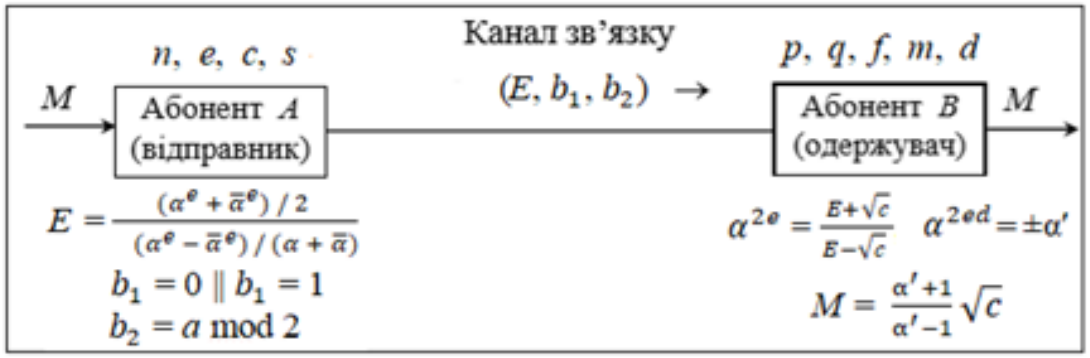


СХЕМА МОДИФІКОВАНОГО МЕТОДУ ШИФРУВАННЯ ВІЛЬЯМСА



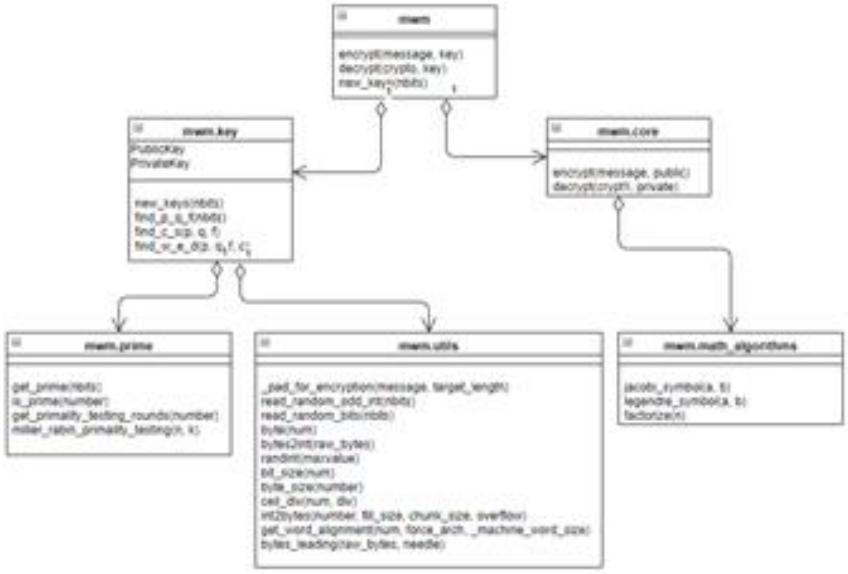
ПІДВИЩЕННЯ СТІЙКОСТІ ДО ЗЛАМУ

При застосуванні методу атаки на основі підібраного шифротексту, криптоаналітик отримає з ймовірністю δ одне з трьох чисел замість двох, що недостатньо для факторизації, тому атаку доведеться повторювати. Завдяки цьому ймовірність успішної атаки дорівнює квадрату ймовірності зламу класичного методу.

МОВА ПРОГРАМУВАННЯ

Для програмної реалізації методу шифрування було розглянуто наступні мови.

ДІАГРАМА КЛАСІВ РЕАЛІЗАЦІЇ МОДИФІКОВАНОГО МЕТОДУ АСИМЕТРИЧНОГО ШИФРУВАННЯ



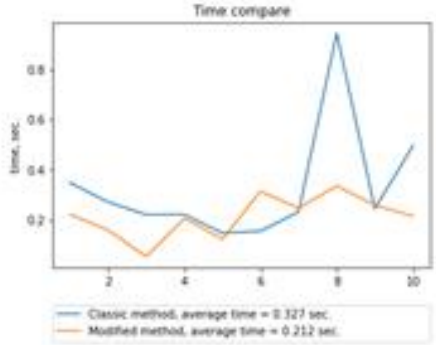
ІНТЕРФЕЙС

Для створення інтерфейсу було обрано веб-оболонку Jupiter Notebook, тому що вона дозволяє легко візуалізувати дані довільного формату. Інтерфейс надає можливість:

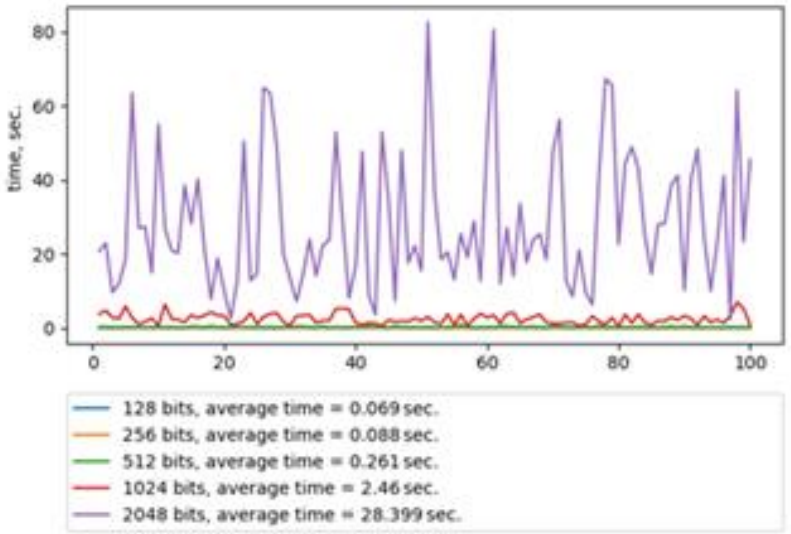
- порівнювати роботу класичного та модифікованого методів;
- переглядати згенеровані ключі та закодоване повідомлення.

ПРИКЛАД ІНТЕРФЕЙСУ

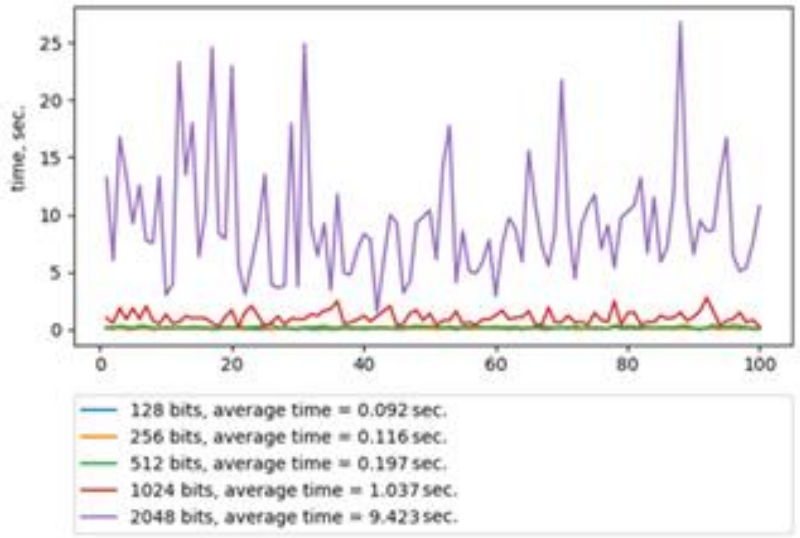
```
Scenario 1: choose number of cryptic tries and length of keys to compare time of working of classic and modified methods
Scenario 2: choose number of cryptic tries to compare time of working of modified method, using keys with different length
Scenario 3: choose number of cryptic tries to compare time of working of classic method, using keys with different length
Scenario 4: enter message and length of keys to see generated keys, encrypted message and decrypted message
Enter scenario number: 1
Number of cryptic tries: 10
Number of bits: 512
```



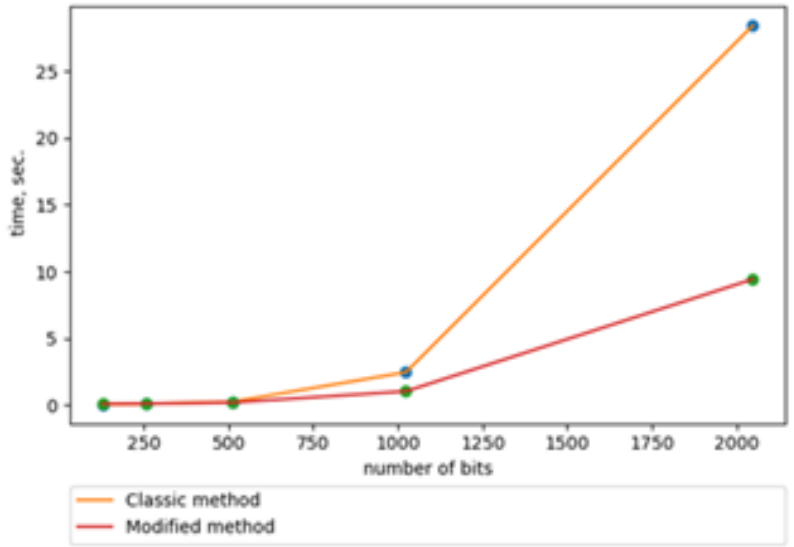
ПОРІВНЯННЯ ЧАСУ РОБОТИ КЛАСИЧНОГО МЕТОДУ З РІЗНОЮ ДОВЖИНОЮ КЛЮЧІВ



ПОРІВНЯННЯ ЧАСУ РОБОТИ МОДИФІКОВАНОГО МЕТОДУ З РІЗНОЮ ДОВЖИНОЮ КЛЮЧІВ



ЗАЛЕЖНІСТЬ СЕРЕДНЬОГО ЧАСУ РОБОТИ КЛАСИЧНОГО ТА МОДИФІКОВАНОГО МЕТОДІВ ВІД ДОВЖИНИ КЛЮЧІВ



НАУКОВА НОВИЗНА

Запропоновано модифікований асиметричний метод шифрування, який відрізняється від існуючого тим, що його час роботи зменшується в 3 рази для ключа довжиною 2048 біт, при тому, що ймовірність зламу за допомогою атаки на основі підбору шифротексту дорівнює квадрату ймовірності зламу існуючого методу.

ВИСНОВКИ

1. Розроблено програмне забезпечення для проведення дослідження методів асиметричного шифрування.
2. Час роботи модифікованого методу асиметричного шифрування менший у 3 рази, ніж у класичного, використовуючи довжину ключів 2048 біт.
3. Ймовірність успішної атаки на основі підбору шифротексту дорівнює квадрату ймовірності зламу класичного методу.
4. Складність прямої атаки через факторизацію відкритого ключа впала у 2 рази.

Дякую за увагу!