ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Методи та підходи до криптографічного захисту даних»

на здобуття освітнього ступеня бакалавра
зі спеціальності *124 – Системний аналіз*
(код, найменування спеціальності)
освітньо-професійної програми *Системний аналіз*
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ *Владислав КАРПІЧ*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач(ка) вищої освіти гр. САД-41
КАРПІЧ Владислав
Ім'я, ПРІЗВИЩЕ
Керівник: МАСТАКОВ Олександр
ст. вик. каф. Ім'я, ПРІЗВИЩЕ

Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інформаційних систем та технологій

Ступінь вищої освіти Бакалавр

Спеціальність 124 – Системний аналіз

Освітньо-професійна програма Системний аналіз

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІПЗАС
_____Каміла СТОРЧАК
«____» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Карпіч Владислав Михайлович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Методи та підходи до криптографічного захисту даних»

керівник кваліфікаційної роботи Мастаков Олександр Сергійович, *ст. вик. каф*,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «___» _____2025р. № 36

2. Строк подання кваліфікаційної роботи «___» _____20___р.
3. Вихідні дані до кваліфікаційної роботи: чат-бот з елементами штучного інтелекту для автоматизації обслуговування клієнтів;
наукова та технічна література, експлуатаційна документація, нормативні

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Поставновка завдання: визначення потреби, вимог до функціональності та розроблення методів криптографічного захисту даних у сучасних інформаційних системах;
2. Архітектура системи: вибір криптографічних алгоритмів, їх математичні основи, аналіз безпеки та стійкості до атак;
3. Реалізація та тестування: моделювання механізмів шифрування та дешифрування, вибір технологічного стеку, збірка, тестування продуктивності та аналіз результатів.
5. Перелік ілюстративного матеріалу: *презентація*

6. Дата видачі завдання «___» _____2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Визначення потреб та вимог до чат-бота. Формулювання цілей та обґрунтування вибору технологій.	15.02.2025 – 28.02.2025.	
2	Аналіз наукової та технічної літератури. Сучасні підходи до розробки криптографічних алгоритмів.	01.03.2025 – 14.03.2025	
3	Огляд, порівняння та аналіз технологій: алгоритми RSA, AES, ECC, криптографічні протоколи, цифрові підписи.	15.03.2025 – 28.03.2025	
4	Розробка рекомендацій щодо використання алгоритму RSA для безпечного зберігання та передачі даних.	29.03.2025 – 11.04.2025	
5	Реалізація програмної частини (розробка та тестування програмного коду для шифрування та дешифрування RSA).	12.04.2025 – 25.04.2025	
6	Оформлення результатів дослідження. Написання та оформлення кваліфікаційної роботи.	26.04.2025 – 12.05.2025	
7	Підготовка доповіді до захисту.	13.05.2025 – 21.05.2025	
8			

Здобувач вищої освіти _____ Карпіч Владислав
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Мастаков Олександр
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина бакалаврської роботи: 62 сторінки, 21 рисунок, 37 джерел.

Об'єкт дослідження – методи криптографічного захисту інформації та їх застосування у сучасних інформаційних системах.

Предмет дослідження – алгоритми криптографічного захисту, їх математичні основи, механізми реалізації та стійкість до атак.

Мета роботи – Методи та підходи до криптографічного захисту даних, детальне дослідження алгоритму RSA, його практичного застосування та реалізація програмного рішення для шифрування та дешифрування інформації.

Методи дослідження:

1. Аналіз літературних джерел та нормативних документів – вивчення сучасних методів криптографічного захисту, їх класифікація та оцінка ефективності.
2. Методи математичного аналізу – дослідження основ криптографічних алгоритмів, зокрема RSA, його математичних принципів та вибору параметрів ключів. Емпіричні методи – дослідження ефективності різних реалізацій алгоритму RSA та аналіз можливих загроз, пов'язаних із його використанням.
3. Методи моделювання – розробка та тестування програмної реалізації RSA, оцінка її продуктивності та безпеки.
4. Порівняльний аналіз – оцінка ефективності алгоритму RSA у порівнянні з іншими криптографічними методами та його використання у сучасних інформаційних системах.

Галузь використання – кібербезпека.

КЛЮЧОВІ СЛОВА: КРИПТОГРАФІЯ, ЗАХИСТ ДАНИХ, ШИФРУВАННЯ, АЛГОРИТМ RSA, СИМЕТРИЧНЕ ТА АСИМЕТРИЧНЕ ШИФРУВАННЯ, КРИПТОАНАЛІЗ, ЕЛЕКТРОННИЙ ПІДПИС, ІНФОРМАЦІЙНА БЕЗПЕКА

ABSTRACT

Text Part of the Bachelor's Thesis: 62 pages, 21 figures, 37 sources.

Object of the research – methods of cryptographic information protection and their application in modern information systems.

Subject of the research – cryptographic protection algorithms, their mathematical foundations, implementation mechanisms, and resistance to attacks.

Purpose of the work – to analyze data protection methods, conduct an in-depth study of the RSA algorithm, its practical application, and the implementation of a software solution for encryption and decryption of information.

Research methods:

1. Analysis of literary sources and regulatory documents – studying modern cryptographic protection methods, their classification, and evaluation of their effectiveness.
2. Mathematical analysis methods – research of the foundations of cryptographic algorithms, particularly RSA, its mathematical principles, and key parameter selection.
Empirical methods – evaluation of the efficiency of different RSA implementations and analysis of potential threats associated with its use.
3. Modeling methods – development and testing of the RSA software implementation, performance, and security assessment.
4. Comparative analysis – evaluation of the effectiveness of the RSA algorithm in comparison with other cryptographic methods and its use in modern information systems.

Field of application – cybersecurity.

KEYWORDS: CRYPTOGRAPHY, DATA PROTECTION, ENCRYPTION, RSA ALGORITHM, SYMMETRIC AND ASYMMETRIC ENCRYPTION, CRYPTOANALYSIS, DIGITAL SIGNATURE, INFORMATION SECURITY

ЗМІСТ

ВСТУП

Актуальність дослідження. У сучасному цифровому світі інформація є одним із найцінніших ресурсів, а її захист – одним із ключових викликів кібербезпеки. З розвитком технологій, обсяг переданих і збережених даних стрімко зростає, що підвищує ризики несанкціонованого доступу, витоків та атак на інформаційні системи. У зв'язку з цим питання криптографічного захисту даних набуває особливої важливості.

Методи криптографічного захисту використовуються в різних галузях: від банківських та фінансових систем до електронного документообігу, захисту персональних даних, криптовалют та захищеного зв'язку. Ефективність алгоритмів шифрування, таких як RSA, AES, ECC, визначає рівень стійкості систем до сучасних атак, включаючи квантові загрози та криптоаналіз.

Дослідження спрямоване на глибокий аналіз криптографічних методів, оцінку їхньої безпеки та розробку ефективних механізмів шифрування даних, що відповідають сучасним викликам інформаційної безпеки. *Предмет дослідження* – алгоритми криптографічного захисту, їх математичні основи, механізми реалізації та стійкість до атак.

Мета роботи – Методи та підходи до криптографічного захисту даних, детальне дослідження алгоритму RSA, його практичного застосування та реалізація програмного рішення для шифрування та дешифрування інформації.

Наукові завдання:

– дослідити основні підходи до шифрування, класифікувати алгоритми на симетричні та асиметричні, розглянути їхні переваги та недоліки;

– вивчити математичні основи найбільш широко вживаних криптографічних алгоритмів, зокрема RSA, AES, ECC, та оцінити їхню криптостійкість у сучасних умовах;

– проаналізувати можливі вектори атак на криптографічні алгоритми, включаючи факторизацію чисел, атаки по часу виконання (timing attacks) та квантові загрози;

– оцінити ефективність алгоритму RSA у порівнянні з іншими методами шифрування за критеріями продуктивності та криптостійкості;

– розробити та протестувати програмну реалізацію алгоритму RSA для шифрування та дешифрування даних, проаналізувати її продуктивність та безпеку;

– сформувати рекомендації щодо вибору криптографічних методів для захисту інформації в сучасних інформаційних системах.

Методи дослідження:

1. Аналіз літературних джерел та нормативних документів – вивчення сучасних методів криптографічного захисту, їх класифікація та оцінка ефективності.

2. Методи математичного аналізу – дослідження основ криптографічних алгоритмів, зокрема RSA, його математичних принципів та вибору параметрів ключів. Емпіричні методи – дослідження ефективності різних реалізацій алгоритму RSA та аналіз можливих загроз, пов'язаних із його використанням.

3. Методи моделювання – розробка та тестування програмної реалізації RSA, оцінка її продуктивності та безпеки.

4. Порівняльний аналіз – оцінка ефективності алгоритму RSA у порівнянні з іншими криптографічними методами та його використання у сучасних інформаційних системах.

1 ТЕОРЕТИЧНІ ОСНОВИ КРИПТОГРАФІЧНОГО ЗАХИСТУ ІНФОРМАЦІЇ

1.1 Криптографія як ключовий елемент інформаційної безпеки

Криптографія – це наука та практика забезпечення конфіденційності, цілісності, автентичності та достовірності інформації шляхом її перетворення у форму, незрозумілу для сторонніх осіб, за допомогою спеціальних математичних методів та алгоритмів.

На рисунку 1.1 зображено приклад шифру Цезаря – одного з найпростіших методів симетричного шифрування, у якому кожна літера повідомлення зміщується на фіксовану кількість позицій у алфавіті [1].

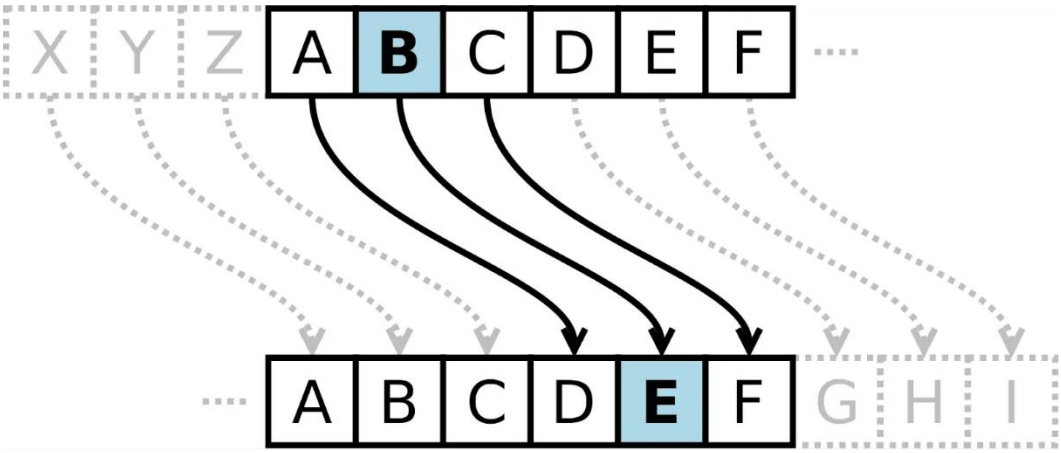


Рисунок 1.1 – Схематичне зображення шифру Цезаря

Одним із фундаментальних засобів забезпечення інформаційної безпеки є криптографія, яка забезпечує конфіденційність, цілісність, автентичність та незаперечність даних. Завдяки криптографічним методам стає можливим надійне зберігання та передача інформації навіть у середовищах із підвищеним ризиком перехоплення чи спотворення.

Криптографія використовується в більшості сфер, де необхідна безпечна робота з даними – зокрема, в електронній комерції, банківських системах,

телекомунікаціях, державному управлінні, охороні здоров’я і зокрема у військовій справі (MilTech). Вона лежить в основі таких технологій, як цифрові підписи, сертифікати автентифікації, електронні паспорти, криптографічні протоколи (SSL/TLS, HTTPS) та захищені канали зв’язку [2].

Використання криптографії дозволяє значно знизити, але не виключити повністю, ризики пов’язані з несанкціонованим доступом до даних, їх підробкою або модифікацією. Застосування сучасних криптографічних алгоритмів, як-от RSA (Rivest-Shamir-Adleman), AES (Advanced Encryption Standard) або ECC (Elliptic Curve Cryptography), дозволяє будувати стійкі до атак системи, здатні протистояти як традиційним, так і новітнім загрозам, зокрема тим, що пов’язані з розвитком квантових технологій.

Основними цілями інформаційної безпеки, які досягаються шляхом застосування криптографічних методів, є конфіденційність, цілісність та автентичність даних. Кожна з цих властивостей відіграє ключову роль у захисті інформації в цифровому середовищі.

Конфіденційність означає забезпечення доступу до інформації лише верифікованим особам. Іншими словами, сторонні особи або системи не повинні мати можливості ознайомитися зі змістом повідомлення або даних. Зазвичай досягнення конфіденційності реалізується шляхом шифрування, при якому дані перетворюються у форму, незрозумілу для стороннього спостерігача без відповідного ключа.

Цілісність забезпечує впевненість у тому, що дані не були змінені під час зберігання або передачі. Навіть незначне спотворення інформації має бути виявлене, що особливо важливо для критично важливих систем. Для контролю цілісності використовуються хеш-функції (математичні алгоритми, які перетворюють вхідні дані довільної довжини у рядок фіксованого розміру), цифрові підписи та інші засоби верифікації.

Автентичність полягає у підтвердженні справжності джерела інформації, тобто гарантії того, що дані отримані саме від тієї сторони, яка заявлена як відправник. Ця властивість особливо важлива в електронних комунікаціях, де

високий ризик підміни учасника взаємодії. Засоби автентифікації включають цифрові сертифікати, електронні підписи та криптографічні протоколи обміну ключами [3].

Забезпечення цих трьох властивостей є важливим для функціонування будь-якої інформаційної системи, оскільки порушення хоча б однієї з них може призвести до втрати довіри, порушення безпеки або витоку конфіденційних даних.

Криптографія має багатовікову історію, що бере початок ще з давніх часів, коли виникла потреба у прихованій передачі повідомлень між воєначальниками, правителями та дипломатами. Із плином часу методи шифрування ускладнювались, переходячи від простих маніпуляцій із символами до складних математичних алгоритмів.

На рисунку 1.1 зображено приклад шифру Цезаря – одного з найпростіших методів симетричного шифрування, у якому кожна літера повідомлення зміщується на фіксовану кількість позицій у алфавіті [1].

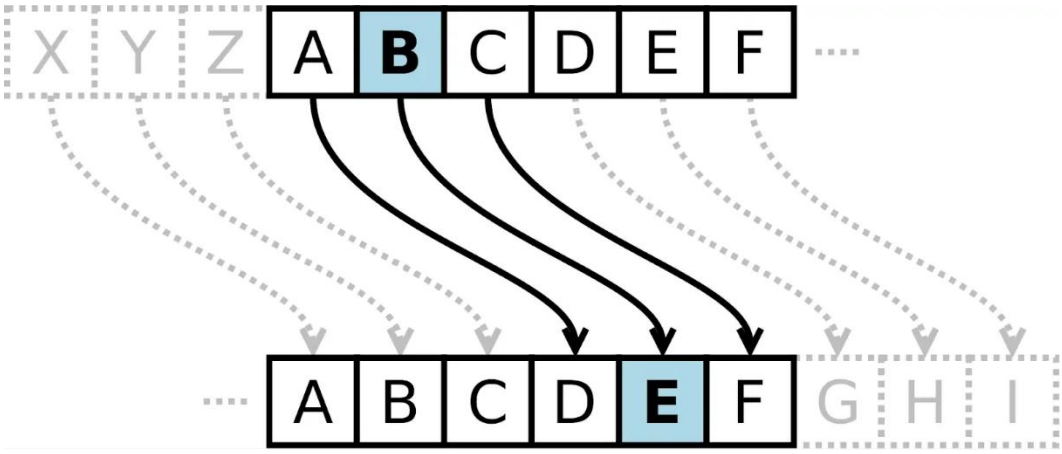


Рисунок 1.1 – Схематичне зображення шифру Цезаря

В майбутньому на заміну шифру Цезаря, був створений шифр Скитала (рис. 1.2), який використовувався у Стародавній Спарті. Він представляв собою дерев’яний циліндр, навколо якого намотувалась стрічка з текстом. Повідомлення ставало зрозумілим лише тоді, коли стрічку намотували на циліндр такої ж товщини, що й у відправника [4].

1	т	е	к	с	т
2	о	в	е	п	о
3	в	і	д	о	м
4	л	е	н	н	я

Рисунок 1.2 Шифр Скитала

У середньовіччі широко застосовувався шифр Віженера (рис. 1.3), що був вдосконаленням поліалфавітного шифру. Його ключовою особливістю було використання декількох шифрувальних алфавітів, що значно ускладнювало криптоаналіз у порівнянні з простими замінами [5].

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Рисунок 1.3 Квадрат Віженера

Під час Першої та Другої світових воєн криптографія отримала новий поштовх у розвитку. Найвідомішим прикладом є машина «Енігма» (рис. 1.4), яку

використовувала нацистська Німеччина для шифрування військових повідомлень. Завдяки зусиллям криптоаналітиків з Блетчлі-Парку, зокрема Алана Тюрінга, вдалося зламати шифр, що суттєво вплинуло на хід війни [6].

Рисунок 1.4 – машина «Енігма»

У другій половині XX століття відбувся перехід від механічних до електронних та математично обґрунтованих алгоритмів. Зокрема, у 1977 році був представлений алгоритм RSA, який став першим практичним прикладом асиметричного шифрування з використанням відкритих ключів.

У 2001 році Національний інститут стандартів і технологій США (NIST) затвердив алгоритм AES (Advanced Encryption Standard) як новий стандарт симетричного шифрування, який і досі залишається актуальним.

Сьогодні криптографія активно розвивається у напрямі еліптичних кривих (ECC), квантово-стійких алгоритмів, а також методів, що використовуються у блокчейні та цифрових валютах. Таким чином, історія криптографії відображає постійне вдосконалення засобів захисту інформації у відповідь на нові виклики та загрози [7].

Криптографія виконує – захист даних від несанкціонованого доступу, підробки та викрадення. Вона є одним із базових інструментів кібербезпеки, на якому ґрунтується захист конфіденційної інформації в різноманітних сферах.

У банківській сфері криптографія використовується для забезпечення безпечного здійснення фінансових транзакцій, захисту даних клієнтів, автентифікації користувачів та цифрового підпису платіжних документів. Усі сучасні системи інтернет-банкінгу та електронної комерції покладаються на криптографічні протоколи, такі як TLS/SSL, що забезпечують шифрування переданих даних.

Цифрові підписи, які базуються на асиметричній криптографії, дозволяють гарантувати автентичність і цілісність електронних документів. Вони широко застосовуються в електронному документообігу, судочинстві, державних реєстрах та комерційних системах. За допомогою цифрового підпису можна підтвердити, що документ був створений конкретною особою, а його зміст не зазнав змін.

Віртуальні приватні мережі (VPN), в таких мережах шифрування використовується для захисту трафіку, що передається між користувачем і сервером, забезпечуючи конфіденційність і анонімність під час роботи в інтернеті, особливо в публічних або незахищених мережах.

Таким чином, криптографія забезпечує фундаментальну основу захисту даних у кіберпросторі, є ключовим елементом сучасних систем безпеки та продовжує залишатися актуальною у контексті новітніх викликів, пов’язаних із розвитком цифрових технологій [8].

1.2 Основні поняття та методи криптографічного захисту

У сфері криптографічного захисту інформації широко використовуються базові поняття, які формують основу процесів забезпечення конфіденційності та безпеки даних. До таких понять належать шифрування, дешифрування та криптоаналіз.

Шифрування – це процес перетворення відкритого (придатного для аналізу) тексту у зашифрований (шифротекст) з метою приховування змісту інформації від сторонніх осіб. Цей процес здійснюється за допомогою спеціального алгоритму та ключа шифрування. Внаслідок шифрування повідомлення стає непридатним для розуміння без наявності відповідного ключа.

Дешифрування – це зворотний до шифрування процес, під час якого зашифроване повідомлення перетворюється назад у відкритий текст. Для успішного дешифрування зазвичай використовується або той самий ключ (у разі симетричної криптографії), або відповідний до відкритого – закритий ключ (у разі асиметричної криптографії).

Криптоаналіз – це сукупність методів і засобів, спрямованих на спробу розкриття зашифрованої інформації без знання ключа. Криптоаналіз вивчає вразливості криптографічних алгоритмів і дозволяє оцінити їхню стійкість до зламування. Класичні методи криптоаналізу включають перебір ключів (brute force), частотний аналіз, аналіз побічних каналів, а також атаки на основі часу виконання або статистичних закономірностей.

У криптографії існує декілька основних підходів до шифрування даних, серед яких найпоширенішими є симетричне, асиметричне та гібридне шифрування. Вибір методу залежить від вимог до безпеки, швидкодії, способу обміну ключами та характеру використання.

Симетричне шифрування передбачає використання одного і того ж ключа для шифрування та дешифрування інформації. Цей підхід є швидким і ефективним з точки зору обчислювальних ресурсів, однак потребує надійного та безпечного каналу для передачі ключа між сторонами. Прикладами є алгоритми AES (Advanced Encryption Standard), DES (Data Encryption Standard), Blowfish (симетричний блочний шифр з довжиною 32-448 біт ключа та 64 біт блоків).

Асиметричне шифрування, або шифрування з відкритим ключем, використовує пару ключів: відкритий (для шифрування) і закритий (для дешифрування). Цей метод спрощує процес обміну ключами, оскільки відкритий ключ може бути переданий відкрито. Найбільш відомими прикладами є RSA

(Rivest-Shamir-Adleman), ElGamal (асиметричний шифр, може використовуватися для цифрових підписів), ECC (Elliptic Curve Cryptography). Асиметричне шифрування зазвичай повільніше за симетричне.

Гібридне шифрування поєднує переваги обох підходів: асиметричне шифрування використовується для захищеного обміну симетричним ключем, а самі дані шифруються симетричним алгоритмом. Такий підхід широко застосовується у протоколах безпечної передачі даних, зокрема в TLS/SSL.

Таблиця 1.1 – Порівняльна таблиця методів шифрування

Критерій	**Симетричне шифрування**	**Асиметричне шифрування**	**Гібридне шифрування**
Кількість ключів	Один спільний ключ	Пара: відкритий і закритий ключі	Обидва типи ключів
Швидкодія	Висока	Нижча через складність обчислень	Висока (основна частина – симетричне шифрування)
Безпека обміну ключами	Вразлива	Надійна	Надійна
Складність реалізації	Нижча	Вища	Найвища
Застосування	Шифрування файлів, VPN	Цифрові підписи, обмін ключами	Інтернет-протоколи (TLS, HTTPS, PGP)

У сучасній криптографії важливу роль відіграють поняття, що пов’язані з обробкою, зберіганням та використанням ключів. Саме криптографічний ключ є критичним елементом у процесі шифрування й дешифрування даних. Розуміння цих понять є необхідним для оцінки рівня безпеки інформаційної системи.

Відкритий ключ (public key) – це ключ, який може бути відкрито переданий іншим користувачам без ризику для безпеки. Він використовується для

шифрування повідомлень або перевірки цифрового підпису. Застосовується в асиметричних алгоритмах, таких як RSA, де кожен учасник має унікальну пару ключів [9].

Закритий ключ (private key) – це ключ, який повинен зберігатися у таємниці. Він використовується для дешифрування даних, зашифрованих відкритим ключем, або для створення цифрового підпису. Втрата або компрометація закритого ключа призводить до серйозних ризиків безпеки.

Секретний ключ (shared key) – це ключ, який використовується в симетричному шифруванні як для шифрування, так і для дешифрування повідомлень. Він має бути відомим лише учасникам комунікації та передаватися через надійний канал.

Ключовий простір – це множина всіх можливих значень ключів, які можуть бути використані в межах певного криптографічного алгоритму. Розмір ключового простору визначає стійкість алгоритму до атаки повним перебором (brute force). Чим більший ключовий простір, тим більше часу потрібно зловмиснику для знаходження правильного ключа шляхом перебору.

Довжина ключа – це кількість бітів, які використовуються для представлення ключа. Вона безпосередньо впливає на рівень криптостійкості алгоритму. Наприклад, для алгоритму RSA безпечним вважається ключ довжиною щонайменше 2048 бітів.

Керування ключами – це сукупність процедур і засобів, що забезпечують генерацію, розповсюдження, зберігання, зміну та видалення ключів. Надійна система керування ключами є необхідною умовою для підтримання загального рівня криптографічного захисту [9].

Життєвий цикл ключа охоплює всі етапи існування ключа – від його створення до моменту виведення з експлуатації. Нерегламентоване або надто тривале використання одного і того ж ключа може призвести до зниження рівня безпеки системи.

На рис. 1.3 наведено класифікацію криптографічних ключів, що використовуються у симетричних та асиметричних методах шифрування.

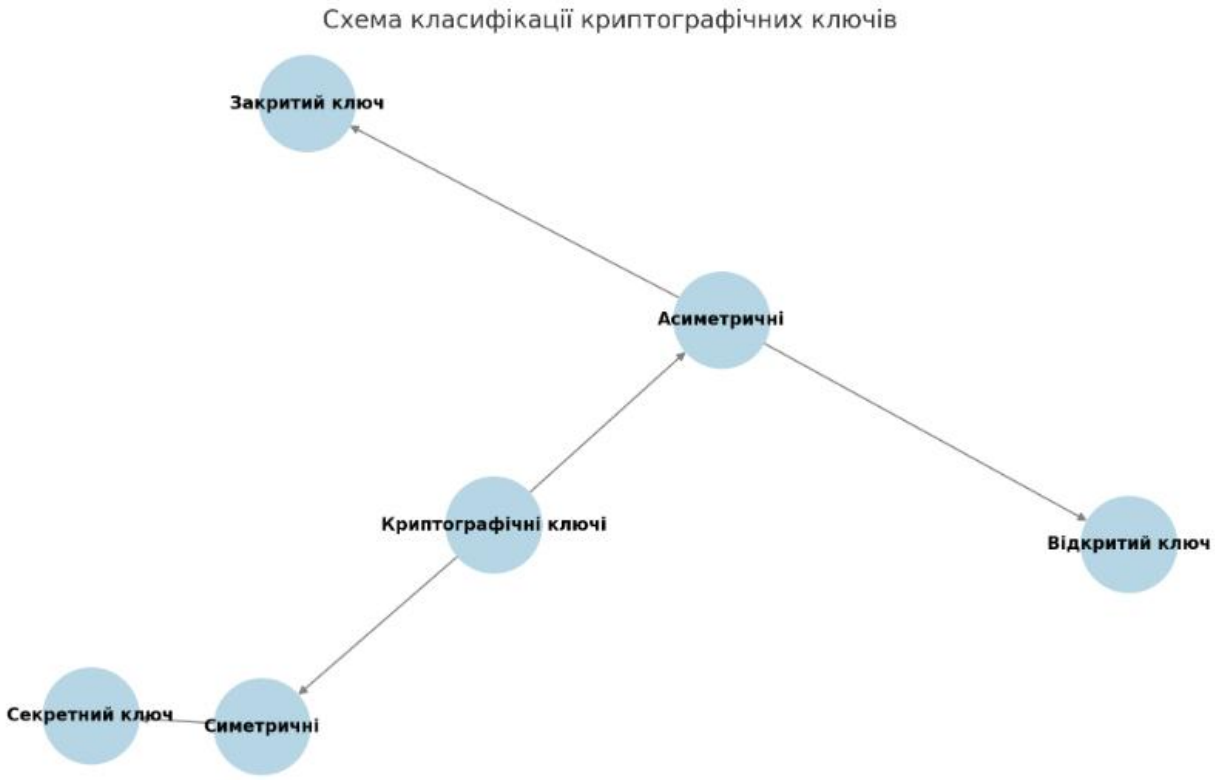


Рисунок 1.3 – Класифікація криптографічних ключів

1.3 Огляд сучасних алгоритмів шифрування та їх класифікація

Сучасні криптографічні алгоритми класифікуються за різними ознаками: типом ключа (симетричні, асиметричні), функціональним призначенням (шифрування, хешування, підпис), а також за способом обробки даних. За останньою ознакою виділяють три основні класи:

1. Блочні алгоритми (block ciphers), працюють із фіксованими за розміром блоками вхідних даних (наприклад, 64 або 128 бітів), які шифруються окремо. Якщо обсяг вхідних даних перевищує розмір блоку, використовується режим шифрування, що визначає послідовність обробки блоків.

Типові приклади:

– AES (Advanced Encryption Standard) – сучасний стандарт симетричного шифрування з блоком 128 біт і довжиною ключа 128, 192 або 256 біт;

– DES (Data Encryption Standard) – історично значимий алгоритм з 56-бітним ключем; нині вважається застарілим;

– Blowfish, Twofish – швидкі й гнучкі альтернативи AES.

2. Поточні (стрімінгові) алгоритми (stream ciphers) – обробляють дані по одному біту або байту. Кожен елемент вихідного потоку шифрується в реальному часі, що дозволяє використовувати такі алгоритми у випадках, коли потрібна мінімальна затримка, наприклад у передачі відео або аудіо.

Типові приклади:

– RC4 – відомий стрімінговий алгоритм, нині визнаний небезпечним через численні вразливості;

– ChaCha20 – сучасний безпечний стрімінговий шифр, рекомендований у протоколах TLS (замість RC4).

3. Хеш-функції (cryptographic hash functions) не шифрують дані в класичному розумінні, але є важливою частиною криптографії. Вони перетворюють довільне за розміром вхідне повідомлення у фіксований за довжиною хеш-код (хеш-суму), який виступає як унікальний відбиток інформації.

Типові приклади:

– SHA-2 (SHA-256, SHA-512) – основний стандарт для безпечного хешування;

– SHA-3 – нова специфікація, що базується на sponge-конструкції;

– MD5, SHA-1 – раніше популярні функції, нині не рекомендуються через виявлені колізії.

У сучасній криптографічній практиці використовується велика кількість алгоритмів, кожен із яких має власні особливості, рівень безпеки та призначення. Серед них найбільш відомими та поширеними є алгоритми RSA, AES, ECC, ElGamal, DES та SHA.

Rivest–Shamir–Adleman (RSA) є одним із найперших та найпоширеніших алгоритмів асиметричного шифрування. Його стійкість базується на складності факторизації добутку двох великих простих чисел. Алгоритм використовує пару ключів – відкритий і закритий, що дозволяє забезпечити як шифрування, так і цифровий підпис. Незважаючи на відносно повільну роботу порівняно з

симетричними алгоритмами, RSA широко застосовується для захищеного обміну ключами та в цифрових сертифікатах (наприклад, у протоколі TLS/SSL) [10].

Advanced Encryption Standard (AES) – сучасний симетричний блочний алгоритм шифрування, що став офіційним стандартом шифрування в США (NIST) з 2001 року. Він підтримує розмір ключа 128, 192 або 256 біт та працює з блоками по 128 біт. AES вирізняється високою продуктивністю, ефективністю та криптостійкістю, що робить його одним із найпопулярніших алгоритмів у сфері захисту інформації. Застосовується у VPN, Wi-Fi, мобільних додатках, файлових шифраторах тощо.

Elliptic Curve Cryptography (ECC) – це клас алгоритмів асиметричного шифрування, що базується на використанні еліптичних кривих над скінченими полями. Основною перевагою ECC є висока криптостійкість при меншій довжині ключа порівняно з RSA, що зменшує обчислювальні витрати та потребу в пам’яті. Цей алгоритм набуває все більшої популярності в мобільних пристроях, інтернеті речей (IoT) та сучасних криптографічних протоколах.

Алгоритм ElGamal є ще одним прикладом асиметричної криптосистеми, криптостійкість якої ґрунтується на складності дискретного логарифмування. Він підтримує як шифрування, так і цифрові підписи. Хоча ElGamal має деякі обмеження щодо розміру зашифрованих повідомлень, його математична простота й адаптивність до різних систем забезпечили йому широке застосування, зокрема у відкритому стандарті PGP.

Data Encryption Standard (DES) – перший офіційний симетричний стандарт шифрування, розроблений у 1970-х роках. Він використовує 56-бітний ключ і працює з 64-бітними блоками даних. Станом на сьогодні DES вважається застарілим через недостатню довжину ключа, яка не забезпечує належного рівня безпеки проти атаки повним перебором. У більшості сучасних систем DES замінено на AES або його вдосконалену версію – Triple DES (3DES).

Secure Hash Algorithm (SHA) – це сімейство криптографічних хеш-функцій, призначених для створення цифрового відбитка (хешу) повідомлення. Алгоритми SHA не призначені для шифрування, однак широко використовуються у цифрових

підписах, зберіганні паролів, контролі цілісності даних. Найпоширенішими є SHA-2 (SHA-256, SHA-512), а також новітній SHA-3, який має іншу математичну структуру та підвищену стійкість до атак. Раніше поширені MD5 та SHA-1 нині не рекомендуються через знайдені колізії [11].

У табл. 1.2 наведено порівняльну характеристику поширених криптографічних алгоритмів за типом шифрування, довжиною ключа, стійкістю до атак, швидкодією та основними сферами застосування. Це дозволяє наочно оцінити переваги та недоліки кожного алгоритму з урахуванням практичних потреб інформаційної безпеки.

Таблиця 1.2 – Порівняльна характеристика поширених криптографічних алгоритмів

Алгоритм	Тип шифрування	Довжина ключа (типова)	Стійкість до атак	Швидкодія	Основні сфери застосування
RSA	Асиметричне	2048–4096 біт	Висока (при достатній довжині ключа)	Низька	Цифрові підписи, передача ключів, сертифікати
AES	Симетричне	128/192/256 біт	Висока	Висока	VPN, Wi-Fi, захист файлів, бази даних
ECC	Асиметричне	160–521 біт	Дуже висока при коротших ключах	Вища за RSA при однаковій безпеці	IoT, мобільні пристрої, блокчейн
ElGamal	Асиметричне	≥1024 біт	Висока	Низька	PGP, електронна пошта, цифровий підпис
DES	Симетричне	56 біт	Низька (застарілий)	Середня	Історичне значення, деякі апаратні пристрої
SHA-2	Хеш-функція	256/512 біт	Висока (SHA-2), SHA-1 – зламана	Дуже висока	Контроль цілісності, цифрові підписи, паролі

1.4 Порівняльний аналіз симетричних та асиметричних методів шифрування

У табл. 1.3 наведено порівняння основних характеристик симетричних та асиметричних методів шифрування, зокрема таких, як швидкодія, рівень надійності, складність реалізації та спосіб управління ключами. Це дає змогу обґрунтовано обирати метод відповідно до завдань інформаційної безпеки.

Таблиця 1.3 – Порівняння симетричного та асиметричного шифрування

Критерій	Симетричне шифрування	Асиметричне шифрування
Швидкодія	Висока, швидке шифрування та дешифрування	Низька, особливо при великих ключах
Надійність	Висока при надійному збереженні секретного ключа	Висока, завдяки складним математичним задачам
Ключі	Один спільний ключ для обох сторін	Пара ключів: відкритий і закритий
Управління ключами	Складне, вимагає безпечного обміну	Просте: відкритий ключ можна вільно поширювати
Складність реалізації	Нижча	Вища (через математичну складність та обчислення з великими числами)

Симетричне та асиметричне шифрування мають свої переваги й обмеження, що обумовлює доцільність їх використання у різних контекстах інформаційної безпеки.

Симетричне шифрування вирізняється високою швидкодією при шифруванні та дешифруванні даних, що є особливо важливим для обробки великих обсягів інформації. Алгоритми цього типу, як правило, є простими в реалізації та вимагають менше обчислювальних ресурсів, що дозволяє використовувати їх у системах з обмеженою потужністю. Однак основним недоліком симетричного шифрування є необхідність безпечного обміну секретним ключем між сторонами. У масштабованих системах, де взаємодіє багато користувачів, виникає проблема збереження та управління великою кількістю унікальних ключів. Крім того, у разі компрометації ключа порушується конфіденційність усіх даних, зашифрованих із його використанням.

Асиметричне шифрування, навпаки, не потребує безпечного каналу для передавання ключів, оскільки відкритий ключ може бути вільно розповсюджений. Це значно спрощує процедури обміну та управління ключами, особливо у багатокористувацьких середовищах. Окрім того, асиметричні алгоритми дозволяють реалізовувати функціонал цифрового підпису, що забезпечує автентичність і незаперечність переданих даних. Водночас асиметричне шифрування має нижчу швидкодію порівняно з симетричними методами, що

зумовлено складністю математичних обчислень та необхідністю використання довших ключів для досягнення належного рівня безпеки. Через це асиметричні алгоритми, як правило, не застосовуються для шифрування великих масивів даних, а використовуються переважно для захищеного обміну ключами, які надалі використовуються в симетричних системах.

На практиці вибір між симетричним та асиметричним шифруванням залежить від конкретних умов, задач та вимог до безпеки, швидкодії й масштабу. Нижче наведено типові сценарії, в яких використовується кожен із підходів.

Симетричне шифрування [11]:

– Шифрування файлів та баз даних. Наприклад, у системах зберігання даних або резервного копіювання застосовуються алгоритми AES для швидкого та ефективного захисту великого обсягу інформації.

– VPN-з’єднання. У віртуальних приватних мережах симетричне шифрування (AES, ChaCha20) забезпечує безпечну передачу даних між клієнтом і сервером у режимі реального часу.

– Захист бездротових мереж (Wi-Fi). Стандарти WPA2 та WPA3 використовують симетричні алгоритми для шифрування трафіку між маршрутизатором і користувачем.

Асиметричне шифрування:

– Цифрові підписи. У системах електронного документообігу (наприклад, «Дія», електронний кабінет платника податків) використовується RSA або ECC для створення цифрового підпису, який підтверджує автентичність і незмінність документа.

– Обмін ключами в TLS/SSL. Під час встановлення захищеного з’єднання (наприклад, при переході на вебсайт через HTTPS), асиметричне шифрування використовується для безпечної передачі симетричного ключа між клієнтом і сервером.

– Шифрування електронної пошти. Протоколи PGP або S/MIME застосовують асиметричні алгоритми для захисту вмісту електронних листів, забезпечуючи конфіденційність і автентичність комунікації.

Гібридні системи:

– Протокол TLS. Використовує RSA або ECDH для початкового обміну ключами, після чого основна передача даних відбувається за допомогою AES.

– Програми для обміну повідомленнями (наприклад, Signal, WhatsApp). Використовують гібридні криптосистеми, що поєднують асиметричний обмін ключами та симетричне шифрування повідомлень.

– Таким чином, симетричне шифрування доцільне у випадках, де потрібна висока швидкість і вже є встановлений довірений зв’язок, тоді як асиметричне – у відкритих або багатокористувацьких середовищах, де важливо забезпечити безпечний обмін ключами та цифрову ідентифікацію.

Висновок до розділу 1

Розглянуто теоретичні основи криптографічного захисту інформації, що є фундаментальною складовою сучасної системи інформаційної безпеки. Було визначено, що криптографія забезпечує реалізацію таких ключових властивостей, як конфіденційність, цілісність, автентичність та незаперечність даних, які є критично важливими в умовах зростання кіберзагроз.

Проведено аналіз основних понять криптографії, таких як шифрування, дешифрування, криптоаналіз, ключі та хеш-функції. Здійснено класифікацію сучасних алгоритмів шифрування на блочні, поточні та хеш-функції, з детальним описом їхніх характеристик та механізмів роботи. Особливу увагу приділено огляду та порівнянню найбільш відомих криптографічних алгоритмів, таких як RSA, AES, ECC, ElGamal, DES та SHA-2, а також їх практичним сферам застосування.

Порівняльний аналіз симетричних та асиметричних методів шифрування дозволив виявити їхні переваги, недоліки та доцільність використання у різних прикладних сценаріях. Було встановлено, що оптимальним підходом у більшості систем захисту є використання гібридної криптографії, яка поєднує високу швидкодію симетричних алгоритмів із безпечним обміном ключами, забезпеченим асиметричними методами.

2 ДОСЛІДЖЕННЯ АЛГОРИТМУ RSA ТА ЙОГО ЗАСТОСУВАННЯ

2.1 Історичний розвиток та принцип роботи RSA

Алгоритм RSA був створений у 1977 році трьома американськими науковцями – Роном Рівестом, Аді Шаміром та Леонардом Адлеманом, які на той час працювали в Массачусетському технологічному інституті (MIT). Назва алгоритму утворена з перших літер прізвищ його авторів (Rivest–Shamir–Adleman).

RSA став першим практичним алгоритмом, що реалізував концепцію асиметричної криптографії – шифрування з відкритим ключем. Вперше цю концепцію було запропоновано роком раніше, у 1976 році, Вітфілдом Діффі та Мартіном Хеллманом, які виклали ідею передачі ключів через незахищені канали, однак не мали ефективної реалізації. Саме RSA надав реальний математичний механізм для реалізації цієї ідеї, спираючись на обчислювальну складність факторизації великих чисел [12].

Принцип дії RSA ґрунтується на властивостях числової теорії, зокрема:

– складності обернення добутку двох великих простих чисел (тобто задачі факторизації);

– використанні функції Ейлера;

– застосуванні модульних експонент.

RSA є алгоритмом з публічним (відкритим) ключем. Це означає, що кожен користувач має пару ключів:

– відкритий ключ, який може використовувати будь-хто для шифрування повідомлення;

– закритий ключ, який є таємним і використовується для розшифрування.

Таким чином, будь-хто може зашифрувати повідомлення для конкретного користувача, але розшифрувати його зможе лише власник закритого ключа. Це кардинально відрізняє RSA від симетричних методів, де той самий ключ використовувався і для шифрування, і для дешифрування, створюючи проблему безпечного розповсюдження ключів.

Із моменту свого створення RSA став наріжним каменем сучасної цифрової безпеки. Його застосування охоплює:

– електронну пошту (зокрема, підписи та шифрування через PGP, S/MIME);

– протоколи безпеки в Інтернеті (SSL/TLS);

– цифрові сертифікати (наприклад, у стандарті X.509);

– електронні підписи;

– банківські системи;

– блокчейн-технології та криптовалюти.

RSA також відіграє важливу роль у реалізації цифрових підписів, що дозволяє гарантувати справжність і цілісність переданих даних, а також ідентифікувати відправника.

Попри свою популярність, RSA має деякі обмеження [12]:

– великі розміри ключів (для сучасного рівня безпеки – щонайменше 2048 біт);

– відносно повільна швидкість шифрування порівняно з симетричними алгоритмами.

У зв’язку з розвитком квантових обчислень з'являються певні загрози для RSA, оскільки квантові алгоритми (наприклад, алгоритм Шора) можуть потенційно зруйнувати стійкість RSA до факторизації. Тому ведуться активні дослідження в галузі постквантової криптографії.

З моменту свого створення в 1977 році алгоритм RSA пройшов довгий шлях розвитку – від академічної новинки до одного з ключових елементів інформаційної безпеки. Його поява була революційною у криптографії, оскільки він став першим практичним прикладом асиметричного шифрування.

У наступні десятиліття RSA набув широкого поширення в багатьох сферах: від захисту електронної пошти та банківських транзакцій до використання в Інтернет-протоколах безпеки. Алгоритм постійно вдосконалювався, зростали довжини ключів, з’являлися нові стандарти, реалізації та сценарії використання. З

розвитком квантових обчислень розпочалися активні дослідження в напрямку постквантової криптографії, яка потенційно має замінити RSA у майбутньому. Основні етапи розвитку алгоритму RSA наведено в таблиці 2.1.

Таблиця 2.1 – Еволюція алгоритму RSA: ключові віхи

Рік	**Подія**
1976	Діффі та Хеллман запропонували концепцію криптографії з відкритим ключем
1977	Створено алгоритм RSA (Рівест, Шамір, Адлеман, MIT)
1983	RSA вперше публічно запатентовано у США
1993	RSA інтегровано у протокол SSL для безпеки Інтернету
2000-ті	Масове впровадження RSA у цифрові сертифікати та електронну комерцію
2010+	Перехід до ключів довжиною 2048 біт і більше
2020+	Дослідження постквантової криптографії та потенційна загроза від квантових комп’ютерів

2.2 Математичні основи та вибір параметрів ключів

Алгоритм RSA базується на використанні математичних операцій з великими простими числами, зокрема – на складності факторизації добутку двох таких чисел. Основними етапами його функціонування є генерація ключів, шифрування та дешифрування.

Генерація ключів:

– вибираються два великі прості числа p та q, які повинні бути випадковими та відрізнятись одне від одного;

– обчислюється добуток цих чисел: $n = p \cdot q$. Число n виступає модулем як у відкритому, так і в закритому ключі.

– обчислюється функція Ейлера: $\varphi(\mathrm{n}) = (\mathrm{p} - 1)(\mathrm{q} - 1)$;

– вибирається ціле число e, яке є взаємно простим з $\varphi(n)$ та виконує умови $1 < e < \varphi(n)$. Це число є відкритим експонентом шифрування;

– обчислюється число d, що є мультиплікативним оберненим до e за модулем $\varphi(n)$, тобто $d \cdot e \equiv 1 mod\ \varphi(n)$. Число d є секретним експонентом дешифрування.

У результаті утворюється відкритий ключ (e, n) та закритий ключ (d, n).

Шифрування повідомлення в RSA виконується за допомогою відкритого ключа. Повідомлення m попередньо перетворюється у числову форму, після чого обчислюється шифртекст c за формулою [13]:

$$c = m^e mod\ n$$

Для розшифрування повідомлення використовується закритий ключ. Зашифроване повідомлення c підноситься до ступеня d за модулем n, після чого відновлюється оригінальне повідомлення m:

$$c = c^d mod\ n$$

Завдяки використанню пари ключів і складності обернення операції без знання секретного експонента d, алгоритм RSA забезпечує високий рівень захисту інформації від несанкціонованого доступу.

2.3 Атаки на RSA та методи захисту від криптоаналізу

Попри високу надійність RSA, цей алгоритм не є абсолютно захищеним від криптоаналітичних атак. Його безпека базується на складності математичної задачі факторизації великих чисел, однак на практиці існує низка методів, що дозволяють частково або повністю скомпрометувати систему за певних умов. Розглянемо основні типи атак на RSA.

Однією з головних загроз для RSA є факторизація модуля n, який утворюється добутком двох простих чисел p та q. Якщо зловмиснику вдається

розкласти n на множники, він може легко обчислити закритий ключ d, а отже – дешифрувати будь-яке зашифроване повідомлення.

Серед найвідоміших методів факторизації:

– Метод квадратичного решета (Quadratic Sieve);

– Метод числового решета (Number Field Sieve – NFS) – найефективніший відомий метод для дуже великих чисел.

Чим більший розмір ключа, тим складніше провести таку атаку. Тому ключі довжиною менше ніж 1024 біти вважаються вразливими і більше не рекомендуються для використання.

Атаки на основі слабких або некоректних параметрів ключів, спрямовані на помилки у виборі параметрів RSA:

– Якщо прості числа p та q мають невелику різницю або мають спільні біти, це знижує стійкість алгоритму.

– Використання надто малого значення експоненти e (наприклад, $e = 3$) без додаткових заходів безпеки може призвести до атаки Поверса (Low-Exponent Attack), особливо якщо повідомлення не доповнено заповненням (padding).

– Використання одного і того ж n для кількох пар ключів також може бути небезпечним.

Атаки по сторонньому каналу (side-channel) можуть ґрунтуватися на аналізі часу, який витрачається на операції шифрування або дешифрування. Наприклад:

– Якщо одна операція займає більше часу через певні математичні особливості (наприклад, обчислення $m^d \, mod \, n$), це може дати зловмиснику підказки щодо приватного ключа.

– До таких атак належить Timing Attack, вперше описана Полом Керчерсом у 1996 році.

Атаки через сторонні канали (Side-Channel Attacks), цей тип атак базується не на криптографічних слабкостях, а на фізичних характеристиках пристрою, який виконує криптографічні операції. До прикладів можна віднести:

– Аналіз споживання електроенергії під час виконання обчислень (Power Analysis);

– Електромагнітне випромінювання;

– Помилки при обчисленнях (Fault Injection Attack).

Такі атаки особливо актуальні для вбудованих пристроїв, смарт-карт та IoT-систем, де ресурсів обмежено, а реалізація захисту часто слабша.

Однією з найсерйозніших загроз для алгоритму RSA у XXI столітті є розвиток квантових обчислень. Класичні алгоритми криптоаналізу, що використовуються сьогодні, потребують надзвичайно багато часу та обчислювальних ресурсів для факторизації великих чисел. Саме на цьому ґрунтується стійкість RSA. Проте з появою квантових комп'ютерів ситуація може докорінно змінитися [14].

У 1994 році Пітер Шор розробив алгоритм Шора (Shor's Algorithm), який здатен виконувати факторизацію великих чисел експоненційно швидше, ніж класичні методи. У випадку практичного впровадження потужного квантового комп'ютера цей алгоритм зробить можливим розрахунок приватного ключа RSA за відносно короткий час, що зруйнує криптографічну безпеку алгоритму.

Станом на сьогодні:

– масштабних квантових комп'ютерів, здатних зламати RSA-ключі у 2048 біт, ще не існує;

– однак розробки у цій сфері активно просуваються, зокрема провідними технологічними компаніями (IBM, Google, Intel, D-Wave);

– тому вже зараз важливо враховувати майбутні ризики та готуватись до перехідного періоду постквантової криптографії.

У відповідь на ці виклики міжнародні організації, такі як Національний інститут стандартів і технологій США (NIST), проводять відкриті конкурси на розробку нових постквантових криптографічних алгоритмів, які будуть стійкими до атак квантових комп'ютерів. Одним з напрямків є заміна RSA на алгоритми, що базуються на решітках, кодах або багатозмінних рівняннях.

Таким чином, хоча RSA залишається надійним інструментом на сьогоднішній день, його довгострокове використання потребує переосмислення у зв’язку з технологічним прогресом [14].

2.4 Використання RSA у криптографічних протоколах та цифровому підписі

Застосування RSA в протоколі SSL/TLS

Одним із ключових прикладів практичного застосування алгоритму RSA є його інтеграція в криптографічні протоколи, зокрема SSL/TLS (Secure Sockets Layer / Transport Layer Security). Ці протоколи відповідають за захищену передачу даних у мережевих середовищах, таких як інтернет. Хоча первинно було розроблено SSL, сьогодні він практично повністю витіснений його більш захищеною і вдосконаленою версією — TLS. Незважаючи на це, назви обох протоколів часто вживаються як синоніми.

У рамках SSL/TLS протоколу RSA відіграє ключову роль у двох основних аспектах: аутентифікації сервера та обміні симетричним ключем. Під час встановлення з’єднання клієнт ініціює запит до сервера, який у відповідь надсилає цифровий сертифікат. Цей сертифікат містить відкритий ключ сервера та підписаний приватним ключем довіреного центру сертифікації. Клієнт, використовуючи відкритий ключ цього центру, перевіряє автентичність сертифіката і, відповідно, ідентичність сервера.

Після підтвердження автентичності відбувається обмін симетричним ключем. Саме на цьому етапі RSA активно застосовується: клієнт генерує випадковий симетричний ключ, зашифровує його відкритим ключем сервера (RSA-шифруванням) і передає йому. Сервер, у свою чергу, розшифровує повідомлення за допомогою свого приватного ключа. В результаті обидві сторони мають спільний секретний ключ, який надалі використовується для симетричного шифрування даних протягом сесії.

Важливо зазначити, що сам алгоритм RSA не застосовується для шифрування великих обсягів даних через значну обчислювальну складність. Замість цього він

використовується на початкових етапах — для захищеного обміну ключами та аутентифікації.

У новіших версіях TLS, зокрема TLS 1.3, відбулося часткове відходження від RSA на користь більш сучасних та ефективних криптографічних методів, зокрема алгоритмів на основі еліптичних кривих (наприклад, X25519). Це підвищує загальну продуктивність і стійкість до потенційних атак.

Таким чином, алгоритм RSA залишається важливим елементом у криптографічній інфраструктурі SSL/TLS, особливо в аспектах аутентифікації та обміну ключами, хоча в сучасних протоколах його застосування поступово скорочується на користь новітніх криптографічних рішень [13].

Застосування RSA в протоколах SSH та PGP

Алгоритм RSA знайшов широке застосування в мережевих протоколах, зокрема у SSH (Secure Shell) та PGP (Pretty Good Privacy), які забезпечують захист даних у процесі дистанційного доступу та електронної комунікації.

SSH — це протокол, який використовується для безпечного доступу до віддалених комп’ютерних систем через командну оболонку. Одним з ключових етапів встановлення захищеного з’єднання є асиметричний обмін ключами, де застосовується RSA. На цьому етапі генеруються ключові пари, і клієнт із сервером обмінюються відкритими ключами. За допомогою RSA шифрується симетричний ключ, який надалі використовується для шифрування всіх даних у межах SSH-сеансу.

PGP, у свою чергу, призначений для захищеного обміну електронними листами та файлами. RSA у цьому контексті використовується як для шифрування повідомлень, так і для створення цифрових підписів. Користувачі PGP можуть створювати ключові пари, за допомогою яких шифруються симетричні ключі (що використовуються для безпосереднього шифрування даних), а також генеруються підписи для перевірки автентичності відправника.

Після етапу асиметричного обміну RSA-ключами обидва протоколи переходять до використання симетричних алгоритмів шифрування для забезпечення ефективності обробки великих обсягів інформації:

– У SSH найбільш поширеними алгоритмами є AES (Advanced Encryption Standard), Triple DES або Blowfish. Усі передані дані в межах сесії шифруються цими алгоритмами з використанням спільно узгодженого симетричного ключа.

– PGP також поєднує асиметричне і симетричне шифрування: фактичне повідомлення шифрується за допомогою симетричного алгоритму (часто AES), а вже симетричний ключ шифрується відкритим ключем RSA. Крім того, RSA використовується для генерації та перевірки цифрових підписів, що гарантує цілісність та справжність повідомлення.

Отже, в обох протоколах — SSH і PGP — RSA виконує фундаментальні функції обміну ключами, аутентифікації та підпису, закладаючи основу для подальшого використання інших криптографічних засобів шифрування [14].

Застосування RSA в електронному цифровому підписі

Електронний цифровий підпис (ЕЦП) є важливим елементом сучасних криптографічних систем, що забезпечує автентичність, цілісність та безвідмовність електронних повідомлень і документів. Одним з найпоширеніших алгоритмів, який використовується для створення цифрових підписів, є RSA (Rivest-Shamir-Adleman).

Цифровий підпис — це особливий криптографічний механізм, який дозволяє підтвердити, що повідомлення було створене певним відправником, а його вміст не було змінено після відправлення. Оскільки RSA базується на парі ключів — відкритому та закритому — він ідеально підходить для реалізації електронного підпису.

Процес створення та перевірки ЕЦП з використанням RSA включає кілька ключових етапів:

1. Генерація ключової пари. Створюється пара RSA-ключів: приватний (який зберігається в таємниці) та публічний (який може бути відкрито поширений).

2. Підпис повідомлення. Власник підпису обчислює хеш (стисле криптографічне представлення) повідомлення та шифрує його своїм приватним ключем RSA. Отриманий результат і є цифровим підписом.

3. Перевірка підпису. Одержувач повідомлення, використовуючи публічний ключ відправника, розшифровує підпис і отримує хеш. Потім він самостійно обчислює хеш повідомлення та порівнює його з розшифрованим. Якщо значення збігаються — підпис вважається дійсним.

Завдяки RSA цифровий підпис набуває таких важливих властивостей:

- Автентичність – підтвердження, що підпис створено саме власником приватного ключа.
- Цілісність – гарантія того, що повідомлення не було змінено після його підписання.
- Невідмовність – відправник не може згодом заперечити факт підписання повідомлення.
- Конфіденційність (опосередковано) – хоча RSA не використовується безпосередньо для шифрування змісту повідомлення, він може поєднуватися з іншими алгоритмами (наприклад, AES), що забезпечують конфіденційність, дозволяючи підписувати вже зашифровані дані.

Таким чином, RSA виступає як потужний інструмент для реалізації цифрових підписів у різних криптографічних протоколах, сприяючи захисту даних у цифровому середовищі, а також підтвердженню достовірності авторства електронних повідомлень і документів [14].

Висновок до розділу 2

Досліджено криптографічний алгоритм RSA, його історичне становлення, принципи функціонування та математичне підґрунтя. Розглянуто основні етапи генерації ключів, включаючи вибір простих чисел та визначення публічного і приватного ключів.

Описано найпоширеніші типи атак, такі як факторизаційні атаки, атаки через побічні канали та інші, а також наведено сучасні методи захисту від криптоаналізу.

Проаналізовано практичне застосування RSA у сучасних криптографічних протоколах, зокрема SSL/TLS, SSH, PGP, а також його роль у створенні електронного цифрового підпису. Зроблено висновок, що RSA залишається одним із найнадійніших алгоритмів для забезпечення конфіденційності, автентичності та цілісності даних у цифрових системах.

3 РЕАЛІЗАЦІЯ ТА ОЦІНКА АЛГОРИТМУ RSA

3.1 Проблеми реалізації та технічні аспекти RSA

Алгоритм RSA, незважаючи на свою популярність і високу надійність у сфері захисту інформації, має ряд особливостей, що ускладнюють його практичну реалізацію. Основна складність полягає в роботі з великими простими числами та складними обчислювальними процесами, які потребують значних ресурсів. Ці технічні аспекти безпосередньо впливають як на криптостійкість алгоритму, так і на його ефективність у реальних програмних або апаратних рішеннях. Розгляд ключових проблем реалізації RSA дозволяє краще зрозуміти обмеження алгоритму та вимоги до його безпечного використання в сучасних системах [15].

Проблема генерації простих чисел у RSA

Однією з перших і найважливіших проблем при реалізації RSA є **вибір великих простих чисел** p і q, які служать основою для обчислення модуля $n=p*q$. Надійність шифрування безпосередньо залежить від складності факторизації цього модуля, тому числа мають бути не лише простими, а й досить великими — зазвичай довжиною від 1024 до 4096 біт. Водночас процес генерації таких чисел є **обчислювально складним**, особливо при використанні надійних методів перевірки простоти.

Найпростіший, але практично неефективний підхід — це **перевірка простоти числа шляхом ділення на всі менші числа**, що є цілком непридатним для великих бітних довжин. Уже для 32-бітних чисел такий метод стає обчислювально затратним, не кажучи вже про криптографічні розміри.

Існує ряд алгоритмів, які дозволяють **перевірити простоту великих чисел з високим рівнем імовірності**, й більшість з них легко реалізуються у програмних засобах.

Серед таких методів можна виокремити:

- **Оптимізований перебір дільників**
- **Решето Ератосфена**

- **Тест Міллера-Рабіна**
- **Алгоритм AKS**

Оптимізований перебір дільників є одним з найдавніших способів перевірки простоти числа. Його коріння сягають ще античних часів — зокрема, він був відомий у стародавній Греції та Єгипті. Наприклад, **решето Ератосфена**, створене в III столітті до н.е., є прикладом методу, що дозволяв виявляти прості числа шляхом відсіювання їх кратних.

У сучасному вигляді оптимізований перебір дільників застосовує кілька покращень:

- Перевірка **лише до квадратного кореня з числа**. Якщо число має дільники, то хоча б один із них буде не більший за $\sqrt{n}$.
- Використання **лише простих чисел** як можливих дільників, що значно скорочує кількість перевірок.

Цей підхід лишається корисним у теоретичному розумінні та при роботі з відносно малими числами. Однак для криптографічних задач, де розміри чисел сягають тисяч біт, застосовуються **ймовірнісні алгоритми перевірки простоти**, такі як **тест Міллера-Рабіна**, **тест Соловея-Штрассена** або **алгоритм AKS**, які забезпечують більшу ефективність і прийнятну точність [16].

Алгоритм решета Ератосфена

Серед класичних методів визначення простих чисел одним з найвідоміших є **решето Ератосфена** — ефективний і візуально зрозумілий алгоритм, що дозволяє знайти всі прості числа до заданого значення n. Метод отримав свою назву на честь давньогрецького вченого Ератосфена з Кірени, який жив у III столітті до нашої ери й прославився не лише в математиці, але й у географії та астрономії.

Хоча подібні методики використовувалися ще до його часу, саме Ератосфен запропонував **систематизовану процедуру**, яка базується на послідовному викреслюванні складених чисел з набору натуральних чисел.

Суть алгоритму полягає в наступному: для кожного числа, починаючи з 2, викреслюються всі його кратні зі списку чисел від 2 до n. Числа, що не були викреслені протягом цього процесу, є простими.

Кроки виконання алгоритму:

1. Створити список чисел від 2 до n.
2. Початково вважати всі числа простими.
3. Починаючи з 2, залишити його в списку як просте, а всі його кратні — викреслити.
4. Перейти до наступного не викресленого числа і повторити крок 3.
5. Повторювати дії до завершення списку.

Після завершення роботи алгоритму в списку залишаться лише **прості числа**.

Алгоритм решета Ератосфена демонструє не лише свою історичну цінність, а й **достатньо високу ефективність**: його часова складність становить $\theta(n * \log \log n)$, що робить його придатним для знаходження простих чисел у помірно великих межах. Проте, через значні витрати пам'яті, він менш ефективний для надвеликих чисел, які застосовуються в RSA, тому в криптографії частіше використовуються більш сучасні методи [16].

Тест Міллера–Рабіна як ефективний метод перевірки простоти

Серед найефективніших методів перевірки простоти великих чисел важливе місце займає **тест Міллера-Рабіна** — один із найшвидших і широко застосовуваних **ймовірнісних алгоритмів**. Його особливість полягає в тому, що він не дає 100% гарантії простоти, але дозволяє з високою ймовірністю зробити висновок про те, чи є число простим або складеним.

Цей тест ґрунтується на **малій теоремі Ферма** та вводить поняття **свідка простоти.** Якщо для заданого числа n знайдеться хоча б одне число α, яке не відповідає визначеним умовам, то n вважається складеним. Якщо жоден із випадкових "свідків" не виявить складеності числа, воно вважається простим — з певною, заздалегідь відомою ймовірністю помилки.

Перевага методу полягає у можливості **багаторазового повторення тесту з різними базовими числами** α. Кожна нова ітерація знижує ймовірність помилкового визначення складеного числа як простого. Таким чином, **збільшення кількості ітерацій** суттєво підвищує достовірність результату.

У порівнянні з іншими тестами на простоту — такими як тест Соловея-Штрассена або тест Люка-Лемера — **тест Міллера-Рабіна демонструє значно кращу швидкодію**, особливо при роботі з великими числами, які характерні для криптографічних алгоритмів. Його **асимптотична складність становить** $\theta(k * \log n^3)$, де k — кількість обраних баз (ітерацій).

Попри свою ймовірнісну природу, цей тест активно використовується в криптографічній практиці, оскільки при належній кількості ітерацій він забезпечує **надійну перевірку простоти чисел**, необхідних для генерації ключів у RSA [17].

Детермінований алгоритм перевірки простоти: AKS

Одним з найважливіших проривів у сфері теорії чисел за останні десятиліття став алгоритм AKS (Agrawal–Kayal–Saxena), розроблений у 2002 році індійськими науковцями Маніндро Агравалом, Нітіном Каялом і Ніппоном Саксеною. На відміну від більшості практичних алгоритмів перевірки простоти, які є ймовірнісними, AKS є детермінованим — тобто таким, що гарантує точну відповідь на запитання, чи є задане число простим або складеним.

Основна ідея алгоритму полягає у використанні алгебраїчних властивостей чисел, зокрема властивостей циклічних груп та поліноміальних рівностей. Алгоритм досліджує тотожності, які повинні виконуватись для всіх простих чисел, зокрема базується на узагальненні малої теореми Ферма до поліноміальної форми. Саме це дозволяє AKS отримати доказовий (математично обґрунтований) результат без потреби у статистичних або ймовірнісних оцінках.

Попри свої теоретичні переваги, алгоритм має високу часову складність, що робить його малопридатним для використання в реальних криптографічних системах. Початкова оцінка складності становила $\theta(n^{7.5})$, де n — кількість біт у числі. У подальшому ця складність була оптимізована, однак у практичних застосуваннях AKS все ще поступається більш швидким тестам, таким як тест Міллера-Рабіна чи тест Соловея-Штрассена.

Попри це, алгоритм AKS має велике значення для математичної теорії. Його поява стала відповіддю на багатовікове запитання — чи можливо створити поліноміальний за часом детермінований тест простоти, що не потребує

допоміжних гіпотез. Цей результат не тільки поглибив розуміння структури простих чисел, а й відкрив нові перспективи для досліджень у сфері криптографії, кодування та обчислювальної теорії.

Знаходження взаємно простих чисел у RSA

Під час реалізації алгоритму RSA важливим етапом є **вибір відкритого експонента** e, яка повинна бути **взаємно простою** з числом $\varphi(n)$, тобто найбільший спільний дільник (НСД) між e і $\varphi(n)$ має дорівнювати 1. Це є критичною умовою для забезпечення правильності обчислення закритого ключа *d*, який має бути мультиплікативно оберненим до *e* за модулем $\varphi(n)$.

Для перевірки взаємної простоти двох чисел необхідно знайти їх **найбільший спільний дільник**. Найпростіший підхід — **послідовний перебір можливих дільників** і перевірка умови $\gcd(\alpha, b) = 1$, однак цей метод є неефективним при роботі з великими числами.

Набагато швидший і ефективніший спосіб — це **алгоритм Евкліда**, який широко використовується в криптографії та теорії чисел. Його перевага полягає у простоті реалізації та високій швидкодії навіть для великих вхідних значень.

Основна ідея алгоритму Евкліда:

Алгоритм базується на тому, що:

$$\gcd(\alpha, b) = \gcd(b, a \bmod b) \tag{3.1}$$

Це означає, що НСД не змінюється, якщо більше число замінити залишком від ділення на менше. Алгоритм виконується ітеративно або рекурсивно до тих пір, поки один із аргументів не стане рівним нулю [17].

Псевдокод алгоритму:

1. Нехай *a* і *b* — два вхідних числа, де $a > b$
2. Повторювати:

– Temp = a mod b

– A = b

– B = temp

3. Коли b = 0, результатом є *a* — НСД.

Складність алгоритму:

У найгіршому випадку, коли a і b є послідовними числами Фібоначчі, кількість ітерацій буде пропорційною b, і складність оцінюється як $\theta(b)$. Проте при оптимізації через модульну операцію, алгоритм має **логарифмічну складність** — $\theta(\log \min(a, b))$., що робить його дуже ефективним для застосування в RSA.

Таким чином, алгоритм Евкліда є незамінним інструментом для перевірки взаємної простоти чисел під час генерації відкритого ключа RSA. Його використання забезпечує надійність та математичну коректність криптографічної системи [18-20].

Модульне піднесення до ступеня та довга арифметика в RSA

При практичному впровадженні алгоритму RSA виникає критична потреба у використанні **довгої (великої) арифметики** — спеціальних методів обробки чисел, розмір яких значно перевищує можливості стандартних типів даних у комп'ютерних системах. Це зумовлено тим, що **криптостійкість RSA прямо залежить від довжини ключів**, яка зазвичай складає 1024, 2048 або навіть 4096 бітів.

Довга арифметика — це набір алгоритмів і структур даних, що дозволяють виконувати базові арифметичні операції (додавання, множення, піднесення до ступеня, ділення тощо) над числами, які не вміщуються в стандартні типи (наприклад, 32-бітні чи 64-бітні цілі). Зазвичай такі числа представляються як **масиви цифр або байтів**, а операції над ними реалізуються на основі класичних підходів, адаптованих до багаторозрядної структури.

В реалізації RSA особливо актуальною є операція **піднесення до** ступеня **за модулем**, яка використовується як для шифрування, так і для розшифрування даних. Наївне піднесення числа до великого ступеня (наприклад, e = 65537) без проміжної обробки призводить до чисел гігантських розмірів. Так, при роботі з 2048-бітним повідомленням довжина результату може сягати понад **16 мегабайт пам'яті**, що є обчислювально неефективним і ресурсозатратним.

Щоб цього уникнути, використовується метод **піднесення до** ступеня **за модулем** — тобто обчислення виразу [19]:

$$C = M^e mod\ n \tag{3.2}$$

Цей підхід дозволяє **на кожному кроці множення зменшувати проміжний результат**, обчислюючи залишок від ділення на модуль n. Це суттєво обмежує зростання числа та дозволяє ефективно управляти розрядністю. З математичної точки зору, коректність цього підходу гарантується теоремами модульної арифметики.

У межах довгої арифметики також важливо реалізовувати **ефективне множення великих чисел**. Існує кілька алгоритмів:

- **Множення "в стовпчик"** (класичне, складність $\theta(k^2)$);
- **Множення Карацуби** (швидше, складність $\theta(k^{1.58})$);
- **Множення з використанням швидкого перетворення Фур'є (FFT)** (складність до $\theta(k * logk)$).

Для освітніх або практичних цілей зазвичай застосовується **класичне множення в стовпчик**, оскільки його простіше реалізувати й дослідити. Цей метод передбачає поетапне множення кожної цифри одного числа на всі цифри іншого, із наступним підсумовуванням проміжних результатів — або по рядках, або по стовпцях, залежно від реалізації.

Таким чином, **довга арифметика є фундаментальною складовою** алгоритму RSA. Вона забезпечує можливість обробки чисел великої розрядності, дозволяє виконувати модульні обчислення і є необхідною умовою для надійної та ефективної реалізації криптографічних операцій [20].

3.2 Використання програмних засобів для реалізації алгоритму

Мова програмування С у реалізації RSA Мова програмування С є мовою загального призначення, яка завдяки своїм унікальним властивостям отримала широке визнання серед розробників і науковців. Створена у 1972 році, вона швидко стала популярною завдяки поєднанню високої ефективності, переносності та низькорівневого контролю над апаратними ресурсами [21].

Однією з ключових переваг мови С є її низькорівневість, що дозволяє програмістам безпосередньо взаємодіяти з пам'яттю та апаратною частиною

комп’ютера. Це є важливою умовою для реалізації високопродуктивних та ресурсоощадних алгоритмів, зокрема в галузі криптографії. Завдяки цим можливостям C широко застосовується для розробки системного програмного забезпечення, а також для обчислень, які потребують високої точності та контролю.

Система типізації в C дає змогу чітко визначати типи даних і керувати їх використанням, що підвищує безпеку й стабільність програм. Крім того, універсальність компіляторів C забезпечує високу переносність коду між різними апаратними платформами та операційними системами, що робить її зручною для кросплатформенної розробки.

У науковому середовищі мова C активно використовується для реалізації криптографічних алгоритмів, зокрема RSA. Її ефективність дозволяє створювати високошвидкісний код для обробки великих чисел, реалізації бітових операцій, довгої арифметики та модульних обчислень — основних етапів шифрування та розшифрування. Окрім того, C підтримує підключення до спеціалізованих бібліотек, таких як GMP (GNU Multiple Precision Arithmetic Library), яка суттєво спрощує роботу з багаторозрядними числами та криптографічними функціями.

Таким чином, мова програмування C залишається одним із найефективніших інструментів для реалізації алгоритмів криптографії, зокрема RSA. Її низькорівневі можливості, висока продуктивність і гнучкість дозволяють створювати надійні та швидкі обчислювальні рішення, що відповідають вимогам сучасних наукових задач [21].

Бібліотека GMP як інструмент обробки великих чисел у RSA

GMP (GNU Multiple Precision Arithmetic Library) — це високопродуктивна бібліотека, спеціально розроблена для виконання **арифметичних обчислень з числами довільної точності**. Її функціональність охоплює широкий спектр математичних операцій, що виходять за межі можливостей стандартних типів даних у більшості мов програмування.

Основна перевага GMP полягає у можливості **опрацьовувати великі числа**, які неможливо представити засобами типових цілих або дійсних типів (наприклад, `int` чи `float`). Це є особливо актуальним у таких галузях, як **криптографія**,

чисельні методи, **символьна математика**, а також у наукових обчисленнях, де точність та розрядність мають вирішальне значення.

Бібліотека надає **високоефективні реалізації** таких операцій, як:

- додавання, віднімання, множення, ділення;
- піднесення до ступеня та взяття залишку;
- порівняння, перетворення типів, генерація випадкових чисел;
- робота з **цілими**, **раціональними** числами та **числами з плаваючою комою**;
- **бітові маніпуляції**, що є важливими при реалізації криптографічних протоколів.

GMP побудована на базі оптимізованих алгоритмів, таких як **множення Карацуби**, **швидке перетворення Фур'є**, **алгоритм Барретта** та інші, що забезпечує **високу швидкодію** навіть при обробці багаторозрядних чисел.

Крім того, GMP є **кросплатформенною** — її можна використовувати в різних середовищах та операційних системах. Відкритий вихідний код і активна підтримка спільноти сприяють постійному вдосконаленню бібліотеки.

У контексті реалізації **алгоритму RSA**, GMP є **ключовим інструментом**. Вона дозволяє:

- ефективно працювати з великими простими числами;
- виконувати операції модульного піднесення до ступеня;
- реалізовувати генерацію ключів, шифрування та розшифрування з великою точністю.

Таким чином, GMP не лише спрощує реалізацію складних криптографічних обчислень, а й забезпечує їх **стабільність, точність та високу продуктивність**, що робить її незамінною для реалізації RSA в середовищі мови програмування C [22].

Компілятор GCC у реалізації RSA

GCC (GNU Compiler Collection) — це один із найпоширеніших і найпотужніших компіляторів для мови програмування **C**, що розробляється в рамках проєкту GNU. Його основне призначення — **перетворення вихідного коду**

на виконувані програми, сумісні з різними операційними системами та апаратними архітектурами.

У контексті реалізації **алгоритму RSA**, компілятор GCC виступає ключовим інструментом, який дозволяє **перетворити програмний код на мові C у ефективну виконувану програму**. Це особливо важливо для задач, пов'язаних з криптографією, де має значення не лише функціональність, а й **оптимізація використання ресурсів**, швидкість обчислень та точність реалізації алгоритмів.

Однією з переваг GCC є **вбудована система оптимізації**, яка дозволяє генерувати більш швидкий та ефективний машинний код. Оптимізація охоплює як швидкодію, так і розмір виконуваного файлу, що може бути важливим при розробці ресурсозалежних застосунків або кросплатформених рішень.

GCC також підтримує **широкий спектр платформ**, включаючи Linux, Windows, macOS, а також вбудовані системи. Це забезпечує **переносність коду** — реалізація алгоритму RSA, скомпільована за допомогою GCC, може бути з легкістю адаптована для виконання на іншій архітектурі без значних змін у коді.

Крім цього, GCC забезпечує **сумісність із численними бібліотеками**, включно з **GMP**, яка використовується для обробки багаторозрядних чисел у криптографії. Це спрощує процес побудови комплексних систем, де ефективна робота з великими числами є критичною.

Таким чином, **GCC є важливою складовою** в процесі реалізації криптографічних алгоритмів, зокрема RSA. Його використання дозволяє **забезпечити переносність, продуктивність та стабільність роботи** програм, написаних мовою програмування C [23].

Використання текстового редактора Vim у реалізації RSA

Vim (Vi Improved) — це потужний, розширюваний текстовий редактор, який широко використовується розробниками завдяки своїй ефективності, гнучкості та підтримці різноманітних платформ. Заснований на класичному редакторі **Vi**, Vim поєднує в собі традиційні можливості UNIX-середовища з сучасними інструментами продуктивності.

Vim відзначається **режимом командного керування**, який дозволяє швидко виконувати широкий спектр операцій без використання миші. Це особливо важливо в середовищах, де ефективність роботи з кодом має ключове значення, зокрема під час реалізації обчислювально складних алгоритмів, таких як RSA.

Основні переваги використання Vim:

1) **Ефективність**: завдяки системі команд та можливості працювати повністю з клавіатури, Vim дозволяє **швидко редагувати, переміщуватись та маніпулювати текстом** без зайвих дій. Це значно підвищує продуктивність під час написання коду та роботи з великими текстовими файлами.

2) **Розширюваність**: Vim має **активну спільноту користувачів і розробників**, які створюють численні плагіни, скрипти та розширення. Це дозволяє налаштувати середовище під конкретні потреби, наприклад, додати підтримку для **мови C**, інтеграцію з **GCC** або автодоповнення синтаксису бібліотеки **GMP**.

3) **Кросплатформеність**: редактор підтримується на більшості сучасних операційних систем — **Linux, Windows, macOS**, що робить його універсальним інструментом для розробки незалежно від робочої платформи.

4) **Продуктивність і функціональність**: Vim підтримує **підсвічування синтаксису**, **автоматичне вирівнювання**, **автодоповнення**, **роботу з декількома файлами** одночасно, що сприяє зручному програмуванню та супроводженню коду.

У контексті реалізації **RSA на мові програмування C**, використання Vim дозволяє **зосередитись на коді**, прискорити редагування, контролювати структуру програмного коду і легко інтегруватися з компілятором GCC та іншими інструментами розробки [24].

3.3 Розробка програмної реалізації та тестування ефективності

У цьому розділі детально розглянуто **принцип роботи програмного забезпечення**, що реалізує криптографічний алгоритм RSA, а також наведено **повний опис функціональних можливостей** системи. Розроблена програма

орієнтована на застосування у **вбудованих системах**, таких як безпілотні літальні апарати, робототехнічні комплекси та автоматизовані системи керування.

Головними вимогами до такого модуля є [25]:

- **автономність** роботи;
- **низькорівнева інтеграція** з апаратним забезпеченням;
- **простота взаємодії** з іншими компонентами операційної системи.

Для реалізації цих цілей була обрана операційна система **Linux**, яка надає зручні інструменти низькорівневої взаємодії завдяки підтримці **концепції «все є файл»**.

Концепція «все є файл» у Linux

Ця концепція передбачає, що всі системні ресурси — файли, каталоги, пристрої введення-виводу, мережеві сокети, потоки тощо — представлені у вигляді **файлоподібних об'єктів**, з якими можна взаємодіяти за допомогою **єдиного інтерфейсу файлової системи**. Такий підхід значно спрощує програмування та дозволяє модулю шифрування працювати з будь-яким ресурсом як з простим файлом — читати, записувати, передавати дані тощо.

Завдяки цьому, реалізоване програмне забезпечення для шифрування RSA може бути легко **інтегроване у вбудоване середовище** та взаємодіяти з апаратними компонентами без необхідності створення окремих драйверів чи складних інтерфейсів [26].

Архітектура програмного забезпечення

Програма складається з **трьох взаємодіючих модулів**, кожен з яких виконує окрему функцію та може **працювати автономно**, що підвищує гнучкість системи та дозволяє повторно використовувати окремі її частини. Загальна структура модулів наведена на **рис. 3.1 [27]**.

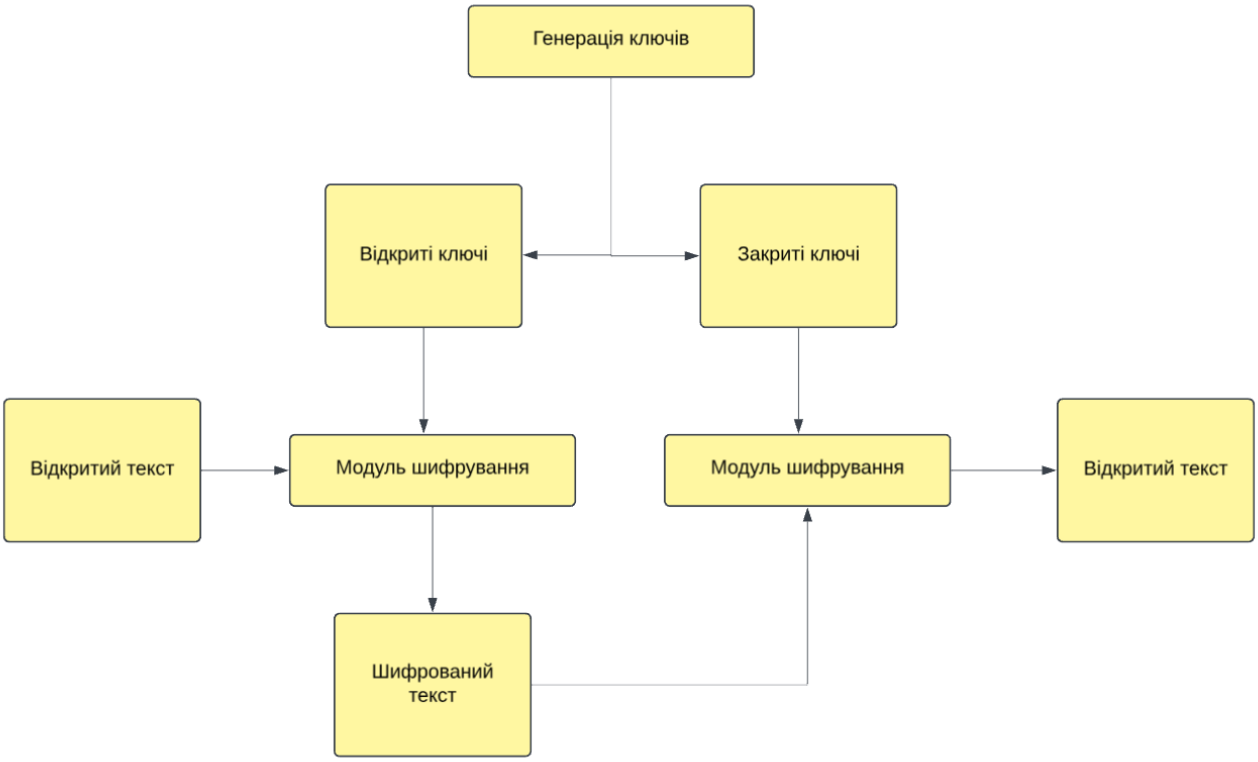


Рисунок 3.1 – Структурна схема програмного модуля шифрування RSA

Перший модуль програми відповідає за генерацію пари криптографічних ключів: відкритого та закритого. У процесі роботи він використовує сучасні криптографічні алгоритми та математичні методи, які базуються на випадкових параметрах, що забезпечують високу стійкість системи до атак. Згенеровані ключі зберігаються у двох окремих файлах: відкритий ключ призначений для шифрування даних, тоді як закритий ключ є конфіденційним і використовується виключно для розшифрування. Такий підхід дозволяє забезпечити безпечне управління ключами та запобігти несанкціонованому доступу до зашифрованої інформації [27].

Використання окремих файлів для зберігання відкритого та закритого ключів у реалізації RSA дозволяє підвищити рівень безпеки та забезпечити контрольований доступ до ключових матеріалів. Такий підхід також спрощує керування ключами і забезпечує гнучкість при використанні програмного забезпечення.

Другий модуль програми відповідає за процес шифрування даних: через аргументи командного рядка він приймає два параметри — файл з відкритим ключем та файл із текстом, який потрібно зашифрувати. У результаті роботи модуля формується новий файл, що містить зашифрований текст, який може бути переданий у наступні етапи обробки вбудованої системи через стандартні засоби операційної системи Linux. Цей процес гарантує конфіденційність і захист даних, перетворюючи відкритий текст у захищений формат, незрозумілий для сторонніх осіб [27].

Третій модуль виконує функцію дешифрування: він приймає на вхід файл із закритим ключем та файл із зашифрованими даними, після чого розшифровує інформацію і зберігає її у новому файлі у відкритому вигляді. Це дозволяє відновити вихідний текст і гарантує цілісність та доступність інформації для авторизованих користувачів. Така архітектура є особливо ефективною у системах вбудованих пристроїв, де важливо забезпечити безпечну передачу та обробку даних. У подальшому розглянемо принцип роботи кожного з модулів окремо.

Механізм створення ключової пари для RSA

Модуль генерації ключів відповідає за створення пари криптографічних ключів — відкритого та закритого — з урахуванням заданої користувачем довжини N біт. Гнучкість реалізації модуля полягає в тому, що параметр довжини ключа може змінюватися в залежності від вимог безпеки або обчислювальних можливостей конкретної вбудованої системи.

Модуль реалізовано у вигляді послідовного виконання набору функцій, кожна з яких відповідає за певний етап процесу генерації ключів. Цей процес передбачає генерацію великих простих чисел, обчислення модуля та функції Ейлера, вибір відкритої експоненти та обчислення оберненого значення для закритого ключа. Структура роботи модуля зображена на рисунку 3.2 [28-29].

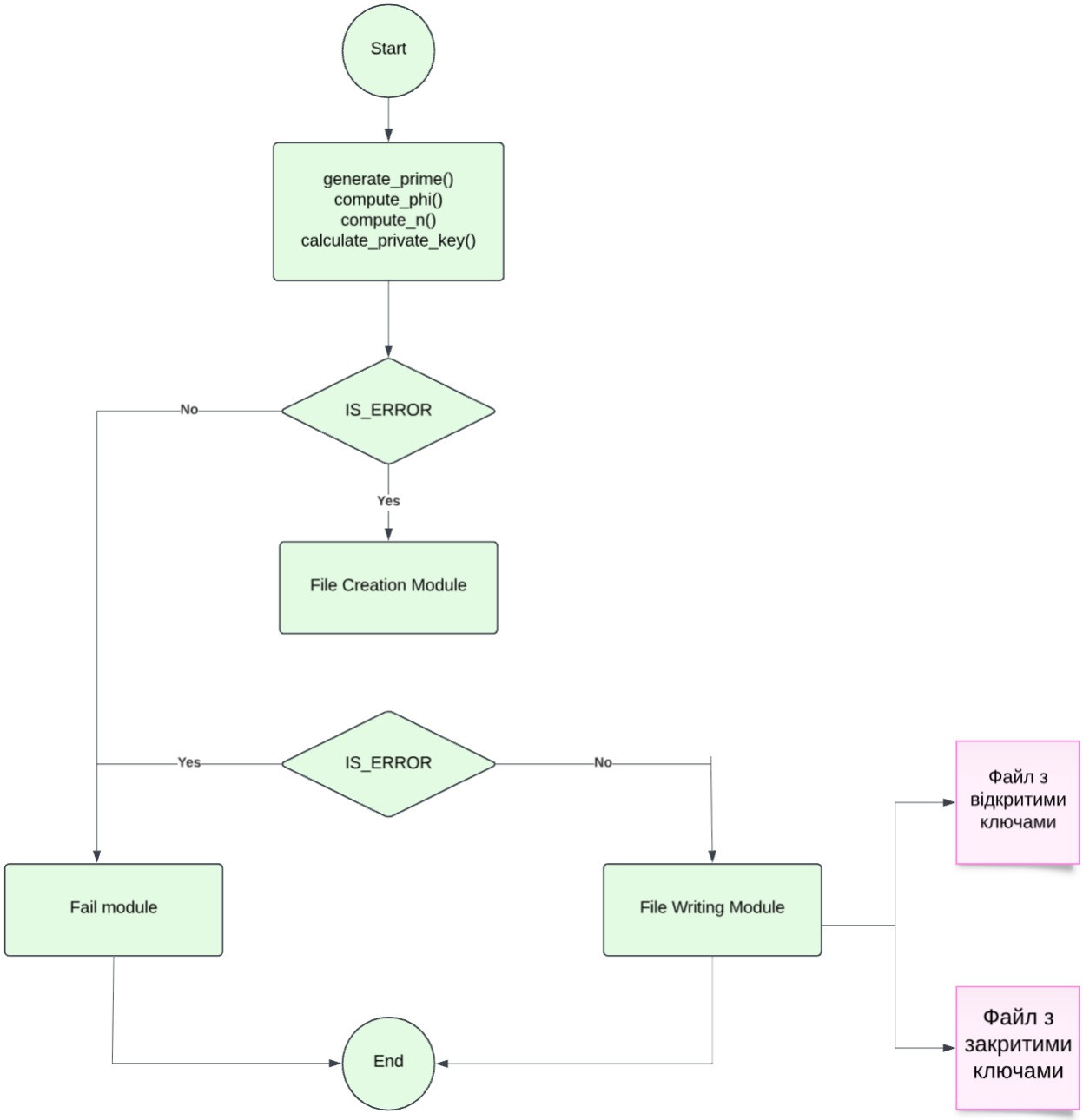


Рисунок 3.2 – Алгоритмічна схема формування ключової пари RSA

Модуль генерації ключів включає в себе низку функцій, які виконуються послідовно для формування надійної та криптографічно стійкої пари RSA-ключів.

Основні з них:

1) **Генерація випадкових простих чисел** – функція `generate_prime()` відповідає за створення випадкових простих чисел, які надалі використовуються як основа для обчислення модуля RSA. Цей процес

базується на ймовірнісних алгоритмах перевірки простоти та генерації випадкових значень [29].

2) **Обчислення функції Ейлера** – за допомогою функції `compute_phi()` розраховується значення φ (φ(n) = (p-1)(q-1)), яке необхідне для визначення відкритого та закритого ключів RSA.

3) **Визначення модуля N** – функція `compute_n()` обчислює модуль N як добуток двох простих чисел p і q. Це значення використовується як основа як для відкритого, так і для закритого ключа.

4) **Обчислення закритого ключа** – функція `calculate_private_key()` реалізує розширений алгоритм Евкліда для знаходження оберненого елемента до відкритої експоненти e за модулем φ(n), що й дозволяє отримати значення закритого ключа d.

Ці функції є критично важливими для формування ключів RSA, оскільки кожна з них виконує окрему математичну операцію, що гарантує правильність, безпеку та унікальність сформованої ключової пари.

Після завершення обчислень модуль записує згенеровані ключі до окремих файлів, використовуючи стандартні засоби мови програмування C для роботи з файловою системою. Запис відкритого та закритого ключів у файли відбувається з обов’язковою перевіркою на наявність помилок, що забезпечує надійність та безпечність збереження критичних даних [30].

У разі виникнення помилок (наприклад, під час відкриття файлу або запису даних), користувач отримує детальне повідомлення з поясненням можливої причини — відсутність прав доступу, недостатній простір на диску або помилковий шлях до файлу. Такий підхід дозволяє ефективно діагностувати проблеми та запобігти втраті даних чи порушенню цілісності ключів.

Модуль генерації ключів реалізує низку ключових функцій, кожна з яких відповідає за окремий етап створення криптографічної пари RSA-ключів.

Функція generate_prime() відповідає за генерацію випадкових простих чисел заданої довжини. Вона працює за наступною схемою:

1. Ініціалізуються необхідні змінні, включаючи генератор випадкових чисел та вказівник на змінну, яка міститиме знайдене просте число.

2. Виконується цикл do-while, у якому:

– генерується випадкове число заданої довжини n;

– знаходиться найближче більше просте число за допомогою ймовірнісного **тесту Міллера–Рабіна**;

– перевіряється довжина знайденого числа у двійковій формі; якщо вона не відповідає заданій — цикл повторюється [31].

3. Після успішного завершення, функція повертає знайдене просте число. Застосування тесту Міллера–Рабіна забезпечує ефективність перевірки простоти та достатній рівень криптографічної надійності згенерованого значення.

Функція compute_phi() реалізує обчислення значення **функції Ейлера** $\varphi(n) = (p-1)(q-1)$, що необхідна для подальшого визначення відкритого та закритого ключів. Для обчислення добутку великих чисел використовується **метод множення "в стовпчик"**, що дозволяє точно обробляти числа великої розрядності.

Функція compute_n() визначає **модуль** n — один із головних компонентів як відкритого, так і закритого ключа RSA. Значення n обчислюється як добуток двох простих чисел p і q, що були згенеровані раніше. Операція виконується із збереженням результату у спеціальну змінну для подальшого використання в обчисленнях.

Функція generate_e() виконує генерацію **публічної експоненти** e, яка повинна бути взаємно простою з $\varphi(n)$. У функції реалізовано цикл, який генерує випадкове число e та перевіряє його взаємну простоту з $\varphi(n)$, використовуючи **алгоритм Евкліда** для знаходження найбільшого спільного дільника (НСД). Цикл триває до тих пір, поки не буде знайдене значення e, для якого $\gcd(e, \varphi(n)) = 1$, або поки не буде досягнуто порогового значення [31-33].

Функція calculate_private_key() виконує обчислення **приватного ключа** d, що є оберненим до e за модулем $\varphi(n)$. Для цього застосовується **розширений алгоритм Евкліда**, який дозволяє ефективно знайти мультиплікативно обернене

число. Отримане значення *d* забезпечує можливість відновлення відкритого тексту з його зашифрованого вигляду.

Загалом, реалізація цих функцій забезпечує надійну та ефективну генерацію ключової пари для алгоритму RSA, з дотриманням сучасних вимог до безпеки криптографічних систем.

Модуль шифрування

Процес шифрування в алгоритмі RSA реалізується за допомогою функції encryption(), яка виконує операцію піднесення до ступеня за модулем. Основна мета цієї функції — перетворити вихідне повідомлення, представлене числовим значенням, у зашифровану форму на основі відкритого ключа.

Суть роботи функції полягає у послідовному виконанні таких кроків (рис. 3.3):

1. Вхідне повідомлення, представлене числом msg, приймається на вхід функції.
2. Це значення підноситься до ступеня e, який є відкритим (публічним) експонентом і частиною відкритого ключа.
3. Отриманий результат зводиться по модулю n, що також входить до складу відкритого ключа.
4. Підсумкове значення записується у змінну c, яка і є зашифрованим представленням вхідного повідомлення.

Таким чином, шифрування в RSA базується на модульній експоненціації, що забезпечує односторонність перетворення. Функція encryption() ефективно використовує пару відкритих ключів (e, n) для того, щоб перетворити msg у криптографічно захищене повідомлення c [32].

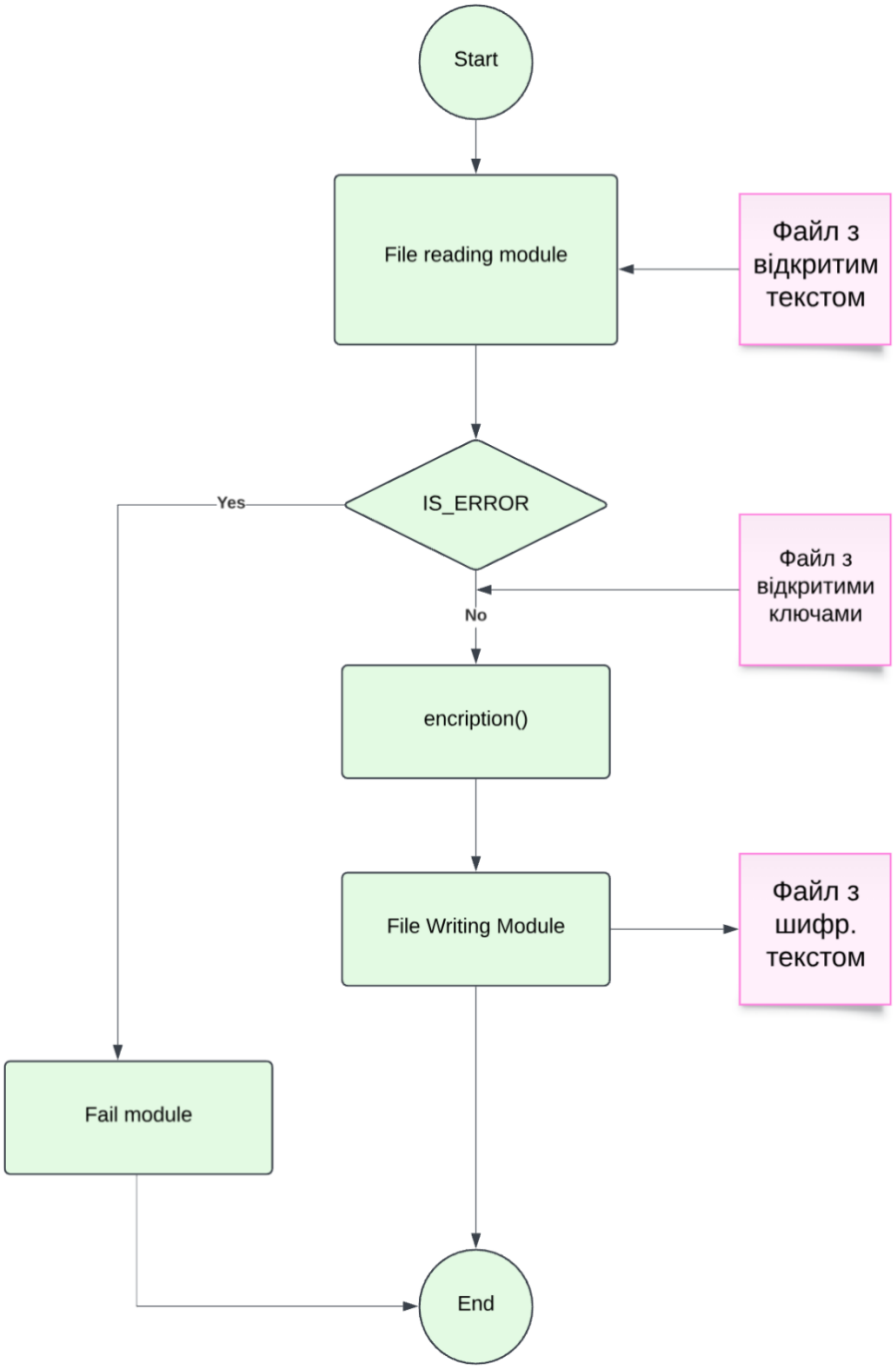


Рисунок 3.3 – Алгоритмічна схема роботи шифрувального модуля

Модуль дешифрування

Функція decryption() виконує процес розшифрування зашифрованого повідомлення з використанням закритого ключа, що був попередньо згенерований у модулі генерації ключів. Вона реалізує математичну операцію модульного

піднесення до ступеня, яка дозволяє відновити вихідні дані з зашифрованого формату відповідно до алгоритму RSA [33].

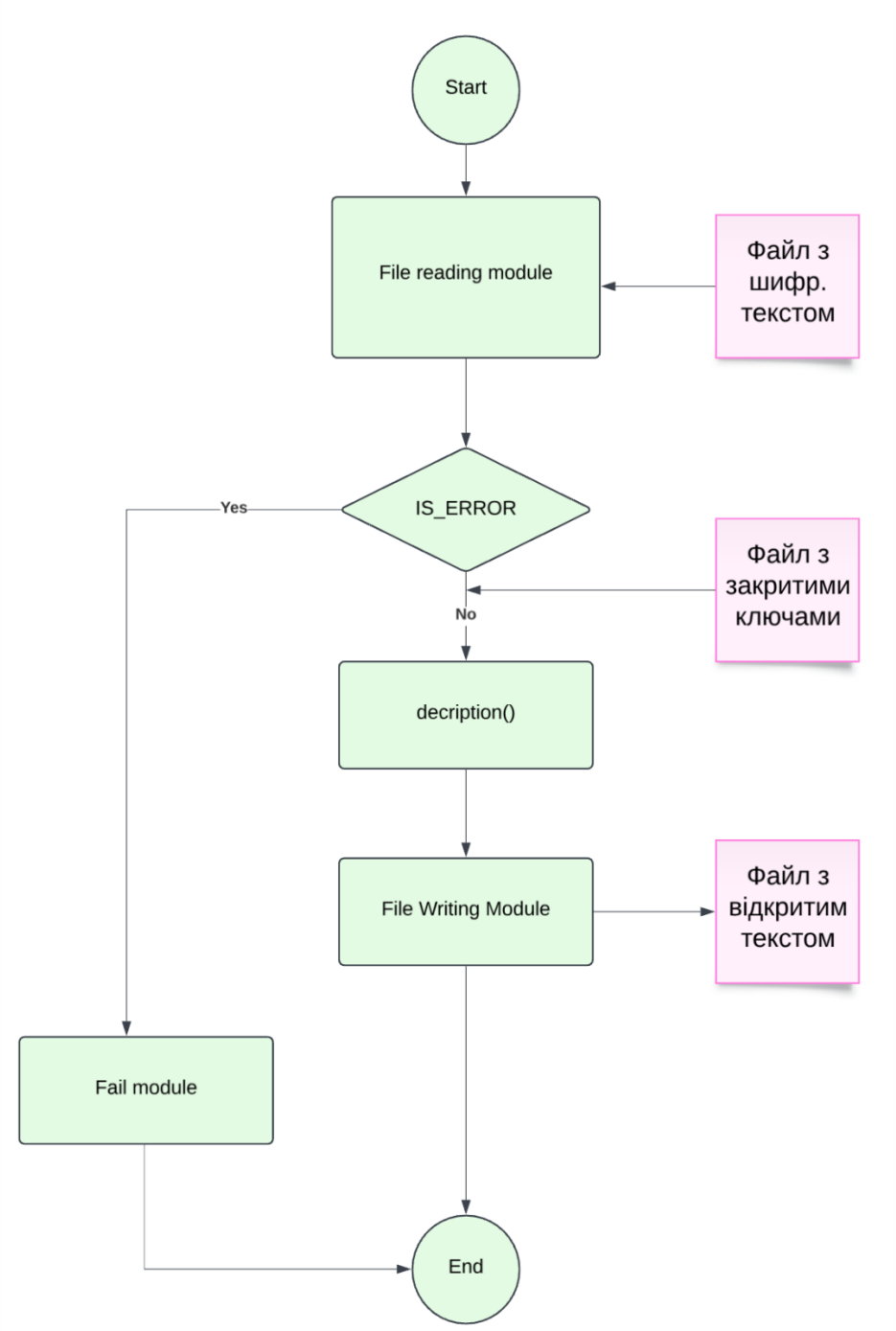


Рисунок 3.3 – Алгоритмічна схема роботи дешифрувального модуля

Опис роботи функції `decryption()` наведено на **рисунку 3.4**. Функція реалізує базову криптографічну операцію **розшифрування повідомлення** за допомогою **закритого ключа** відповідно до принципів алгоритму RSA. Вхідним

параметром є зашифроване повідомлення, представлене числом c, а також компоненти приватного ключа — значення d та модуль n.

Основні етапи роботи функції такі:

1. Зчитується зашифроване повідомлення, яке представляється у вигляді цілого числа c.
2. Число c підноситься до ступеня d, що є експонентою приватного ключа.
3. Отриманий результат обчислюється **по модулю** n:

$$m = c^d mod\ n \tag{3.3}$$

4. Результат операції зберігається у змінній m, яка й представляє **оригінальне, розшифроване повідомлення**.

Таким чином, функція `decryption()` реалізує **модульне піднесення до** ступеня — ключову операцію розшифрування в алгоритмі RSA. Завдяки цьому відновлюється початкове повідомлення з його зашифрованої форми, забезпечуючи конфіденційність та цілісність переданих даних [33-34].

3.4 Аналіз продуктивності та порівняння з іншими алгоритмами

Щоб забезпечити швидку, зручну та безпомилкову збірку програмного забезпечення, у рамках проєкту було створено файл Makefile. Це текстовий конфігураційний файл, який містить набір інструкцій для автоматичного компілювання окремих компонентів програми. Реалізація можливості незалежної компіляції кожного модуля окремо значно спрощує процес розробки та пришвидшує відлагодження. Завдяки використанню Makefile, процес збірки стає максимально автоматизованим, що дозволяє розробникам уникати повторюваних дій, оптимально використовувати ресурси та зменшити кількість помилок, пов’язаних із ручним введенням команд [35].

project — nano Makefile — 166×40

```
  UW PICO 5.09                                                File: Makefile

# Параметри компіляції
CC = gcc
CFLAGS = -Wall -Werror -Wextra -Wfloat-equal -Warray-bounds -Wdiv-by-zero -Wdouble-promotion -g
LDFLAGS = -L/opt/homebrew/lib -I/opt/homebrew/include -lgmp -DTIME

# Об'єктні файли
OBJS = crypto.o

# Ціль за замовчуванням
all: rsa_cr key_gen rsa_decr

# Збірка crypto.o
crypto.o: crypto.c
        $(CC) $(CFLAGS) -c crypto.c -o crypto.o

# Збірка rsa_cr
rsa_cr: rsa_cr.c $(OBJS)
        $(CC) $(CFLAGS) rsa_cr.c $(OBJS) -o rsa_cr $(LDFLAGS)

# Збірка генератора ключів
key_gen: key_gen.c $(OBJS)
        $(CC) $(CFLAGS) key_gen.c $(OBJS) -o key_gen $(LDFLAGS)

# Збірка дешифрувальника
rsa_decr: rsa_decr.c $(OBJS)
        $(CC) $(CFLAGS) rsa_decr.c $(OBJS) -o rsa_decr $(LDFLAGS)

# Очистка
clean:
        rm -f *.o rsa_cr key_gen rsa_decr
```

project — nano crypto.c — 166×40

```
  UW PICO 5.09                                                File: crypto.c

#include <stdio.h>

void crypto_placeholder() {
    printf("Файл crypto.c: працює загальна крипто-функція.\n");
}
```

project — nano rsa_cr.c — 166×40

UW PICO 5.09 File: rsa_cr.c

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Використання: %s public.key input.txt output.txt\n", argv[0]);
        return 1;
    }

    FILE *pub = fopen(argv[1], "r");
    if (!pub) {
        perror("Не вдалося відкрити файл публічного ключа");
        return 1;
    }

    mpz_t e, n;
    mpz_inits(e, n, NULL);
    gmp_fscanf(pub, "%Zd\n%Zd\n", e, n);
    fclose(pub);

    FILE *in = fopen(argv[2], "r");
    if (!in) {
        perror("Не вдалося відкрити вхідний файл");
        return 1;
    }

    fseek(in, 0, SEEK_END);
    long size = ftell(in);
    rewind(in);

    char *buffer = malloc(size + 1);
    fread(buffer, 1, size, in);
    buffer[size] = '\0';
```

project — nano key_gen.c — 166×40

UW PICO 5.09 File: key_gen.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <gmp.h>

// Генерація великого простого числа
gmp_randstate_t state;

void generate_prime(mpz_t prime, int bits) {
    mpz_rrandomb(prime, state, bits);
    mpz_nextprime(prime, prime);
}

// Обчислення НСД (алгоритм Евкліда)
void compute_gcd(mpz_t result, mpz_t a, mpz_t b) {
    mpz_gcd(result, a, b);
}

// Обернене число (розширений алгоритм Евкліда)
void compute_inverse(mpz_t result, mpz_t e, mpz_t phi) {
    if (mpz_invert(result, e, phi) == 0) {
        fprintf(stderr, "Помилка: немає оберненого числа для e!\n");
        exit(1);
    }
}

int main(int argc, char *argv[]) {
    int bits = 2048;
    if (argc > 1) {
        bits = atoi(argv[1]);
    }

    // Ініціалізація
    mpz_t p, q, n, phi, e, d, gcd;
    mpz_inits(p, q, n, phi, e, d, gcd, NULL);
```

^G Get Help ^O WriteOut ^R Read File ^Y Prev Pg
^X Exit ^J Justify ^W Where is ^V Next Pg

project — nano rsa_decr.c — 166×40

```
  UW PICO 5.09                                                File: rsa_decr.c

#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Використання: %s private.key encrypted.txt output.txt\n", argv[0]);
        return 1;
    }

    FILE *priv = fopen(argv[1], "r");
    if (!priv) {
        perror("Не вдалося відкрити файл приватного ключа");
        return 1;
    }

    mpz_t d, n;
    mpz_inits(d, n, NULL);
    gmp_fscanf(priv, "%Zd\n%Zd\n", d, n);
    fclose(priv);

    FILE *in = fopen(argv[2], "r");
    if (!in) {
        perror("Не вдалося відкрити зашифрований файл");
        return 1;
    }

    mpz_t c, m;
    mpz_inits(c, m, NULL);
    gmp_fscanf(in, "%Zd\n", c);
    fclose(in);

    clock_t start = clock();
```

Рисунок 3.4 – Створений Makefile та його модулі

```
kilim@MacBook-Pro-Kilim project % make
gcc -Wall -Werror -Wextra -Wfloat-equal -Warray-bounds -Wdiv-by-zero -Wdouble-promotion -g -c crypto.c -o crypto.o
gcc -Wall -Werror -Wextra -Wfloat-equal -Warray-bounds -Wdiv-by-zero -Wdouble-promotion -g rsa_cr.c crypto.o -o rsa_cr -L/opt/homebrew/lib -I/opt/homebrew/include -lg
mp -DTIME
gcc -Wall -Werror -Wextra -Wfloat-equal -Warray-bounds -Wdiv-by-zero -Wdouble-promotion -g key_gen.c crypto.o -o key_gen -L/opt/homebrew/lib -I/opt/homebrew/include -
lgmp -DTIME
gcc -Wall -Werror -Wextra -Wfloat-equal -Warray-bounds -Wdiv-by-zero -Wdouble-promotion -g rsa_decr.c crypto.o -o rsa_decr -L/opt/homebrew/lib -I/opt/homebrew/include
 -lgmp -DTIME
kilim@MacBook-Pro-Kilim project % nano Makefile
kilim@MacBook-Pro-Kilim project % ls
Makefile        crypto.o        key_gen.c       rsa_cr          rsa_cr.dSYM     rsa_decr.c
crypto.c        key_gen         key_gen.dSYM    rsa_cr.c        rsa_decr        rsa_decr.dSYM
kilim@MacBook-Pro-Kilim project % nano crypto.c
kilim@MacBook-Pro-Kilim project % nano rsa_cr.c
kilim@MacBook-Pro-Kilim project % nano key_gen
kilim@MacBook-Pro-Kilim project % nano key_gen.c
kilim@MacBook-Pro-Kilim project % nano rsa_decr
```

Рисунок 3.5 – Процес компіляції Makefile

Для експериментальної перевірки роботи шифрувального модуля було використано авторський вірш із семи рядків, записаний українською мовою. Текст було збережено у файл з назвою "text", який використовувався як вхідні дані для подальшого шифрування (рис. 3.6) [35].

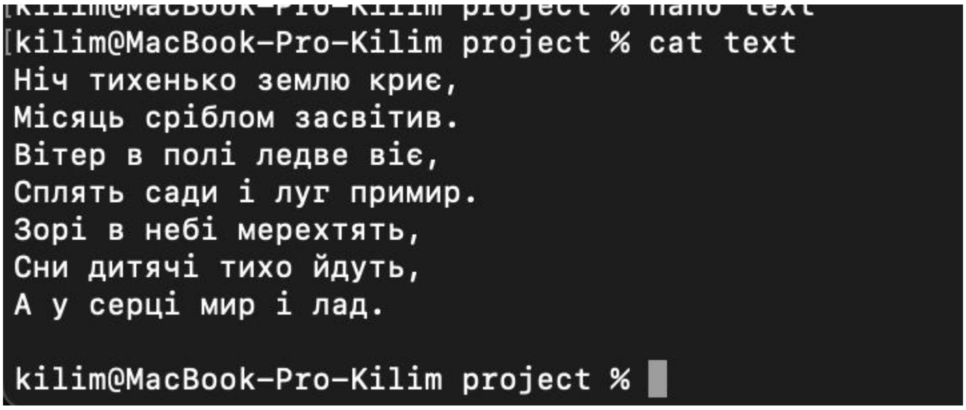


Рисунок 3.6 – Авторський відкритий текст, підготовлений для шифрування

Для створення ключової пари було запущено модуль key_gen, який відповідає за генерацію відкритого та закритого ключів RSA. У командному рядку було передано параметр 2048, що вказує на бажану довжину ключа в бітах. Якщо користувач не задає цей аргумент, програма автоматично використовує значення 2048 біт як типове, оскільки воно є широко прийнятим стандартом для криптографічних систем на основі RSA. На рисунку 3.7 показано результат запуску генерації ключів із зазначеною довжиною [35].

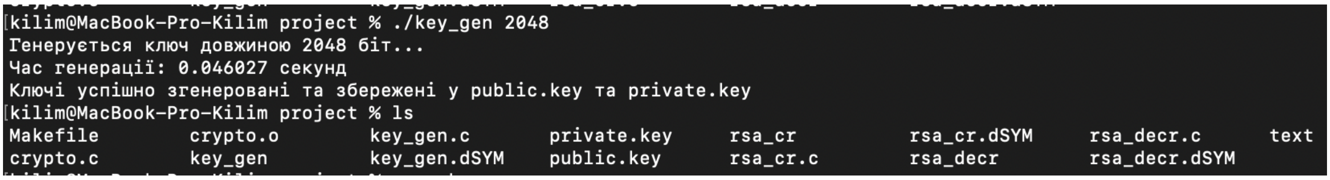


Рисунок 3.7 – Генерація RSA-ключів довжиною 2048 біт

У результаті виконання програми створюються два файли: private.key та public.key, які містять відповідно закритий і відкритий ключі. На рисунку 3.8 представлено їхній вміст.

Рисунок 3.8 – Вміст private.key та public.key

Виконаємо запуск модуля rsa_cr, який відповідає за шифрування даних. Як аргументи командного рядка передаються файл з відкритим ключем — public.key, а також файл з відкритим текстом — text. Процес запуску показано на рисунку 3.9.

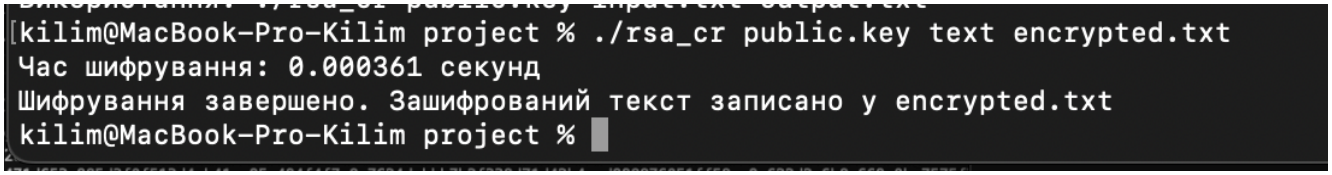


Рисунок 3.9 – Запуск модуля шифрування з передачею вхідних параметрів командного рядка

У результаті виконання шифрувального модуля в поточній директорії створюється файл text.cr, який містить зашифровану версію вхідного тексту. На рисунку 3.10 наведено вміст цього файлу [35].

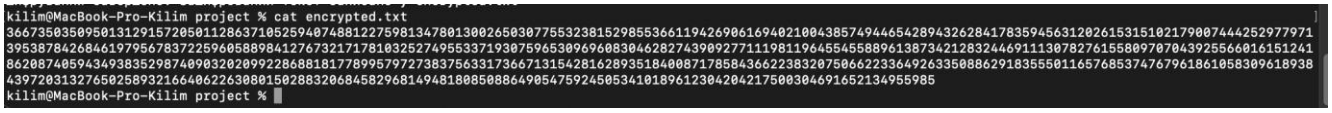


Рисунок 3.10 – Вміст файлу з зашифрованим текстом

Для виконання розшифрування запускається модуль rsa_decr, якому передаються два аргументи: файл private.key, що містить закритий ключ, та файл text.cr із зашифрованим повідомленням. Приклад запуску програми представлено на рисунку 3.11.

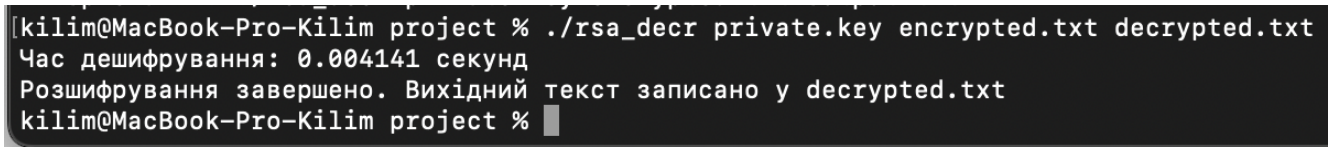


Рисунок 3.11 – Виконання дешифрування RSA з використанням приватного ключа

У результаті виконання команди в поточній директорії створюється файл text.cr.dect, який містить розшифрований текст. На рисунку 3.11 представлено вміст даного файлу для перевірки коректності процесу дешифрування.

```
kilim@MacBook-Pro-Kilim project % cat decrypted.txt
Ніч тихенько землю криє,
Місяць сріблом засвітив.
Вітер в полі ледве віє,
Сплять сади і луг примир.
Зорі в небі мерехтять,
Сни дитячі тихо йдуть,
А у серці мир і лад.

kilim@MacBook-Pro-Kilim project %
```

Рисунок 3.11 – Вміст файлу з розшифрованим текстом

З метою оцінки ефективності реалізованого програмного забезпечення було проведено експериментальне порівняння продуктивності з аналогічною програмою, створеною за допомогою готової бібліотеки RSA для Python. Дослідження охоплювало окремий аналіз часу генерації ключів, процесу шифрування, дешифрування, а також загального часу виконання. У якості відкритого тексту для шифрування та розшифрування використовувався авторський вірш із семи рядків, що був попередньо збережений у вхідний файл [35].

Довжина RSA-ключів змінювалася в діапазоні від 256 до 8192 біт. Для забезпечення об’єктивності результатів, кожна з програм виконувалася по 10 разів для кожного значення довжини ключа, після чого обчислювалося середнє значення часу. На рисунку 3.12 представлено графік залежності часу генерації ключів від їхньої довжини: вісь X позначає довжину ключа (в бітах), вісь Y — середній час генерації ключів (у секундах).

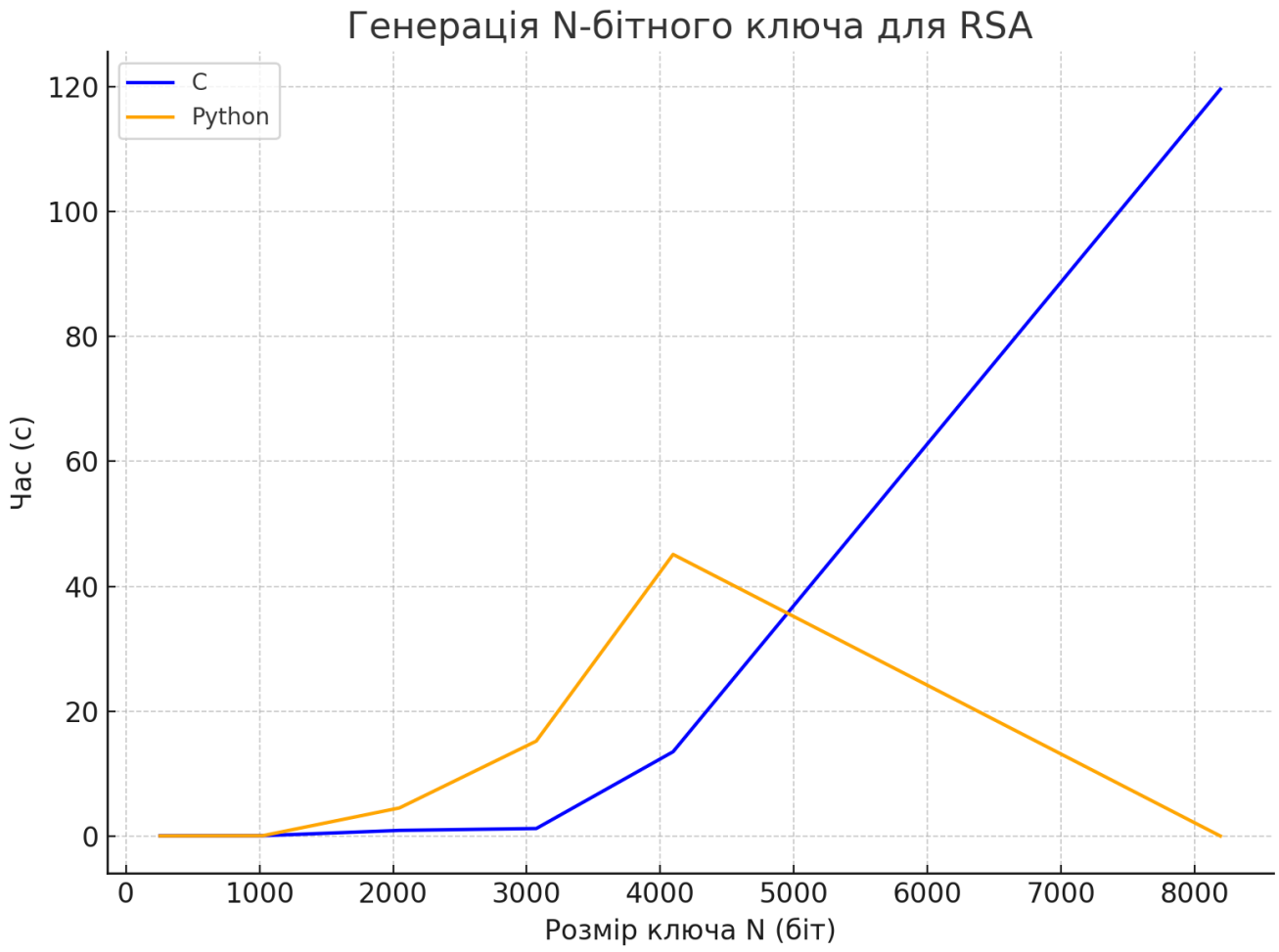


Рисунок 3.12 – Порівняння часу створення RSA-ключів у мовах C та Python

На графіку видно чітку залежність між довжиною RSA-ключа та часом, необхідним для його генерації: зі зростанням розміру ключа, відповідно збільшується і час обчислень. Це логічно пояснюється тим, що створення ключів великої бітності потребує складніших математичних операцій і значно більше обчислювальних ресурсів [36].

Для всіх протестованих довжин ключів — 256, 512, 1024, 2048, 4096 та 8192 біт — реалізація алгоритму генерації ключів мовою C продемонструвала стабільно кращу продуктивність порівняно з аналогом на Python. Це є свідченням ефективності та високого рівня оптимізації програмного коду, написаного мовою низького рівня.

Слід зауважити, що реалізація RSA в Python не змогла завершити генерацію ключа довжиною 8192 біти навіть за 10 хвилин, після чого виконання було

примусово припинене. Така ситуація може бути обумовлена як високими обчислювальними вимогами при роботі з надвеликими числами, так і апаратними обмеженнями комп'ютерної системи, на якій проводився експеримент.

Порівняння середнього часу шифрування відкритого тексту наведено на рисунку 3.13.

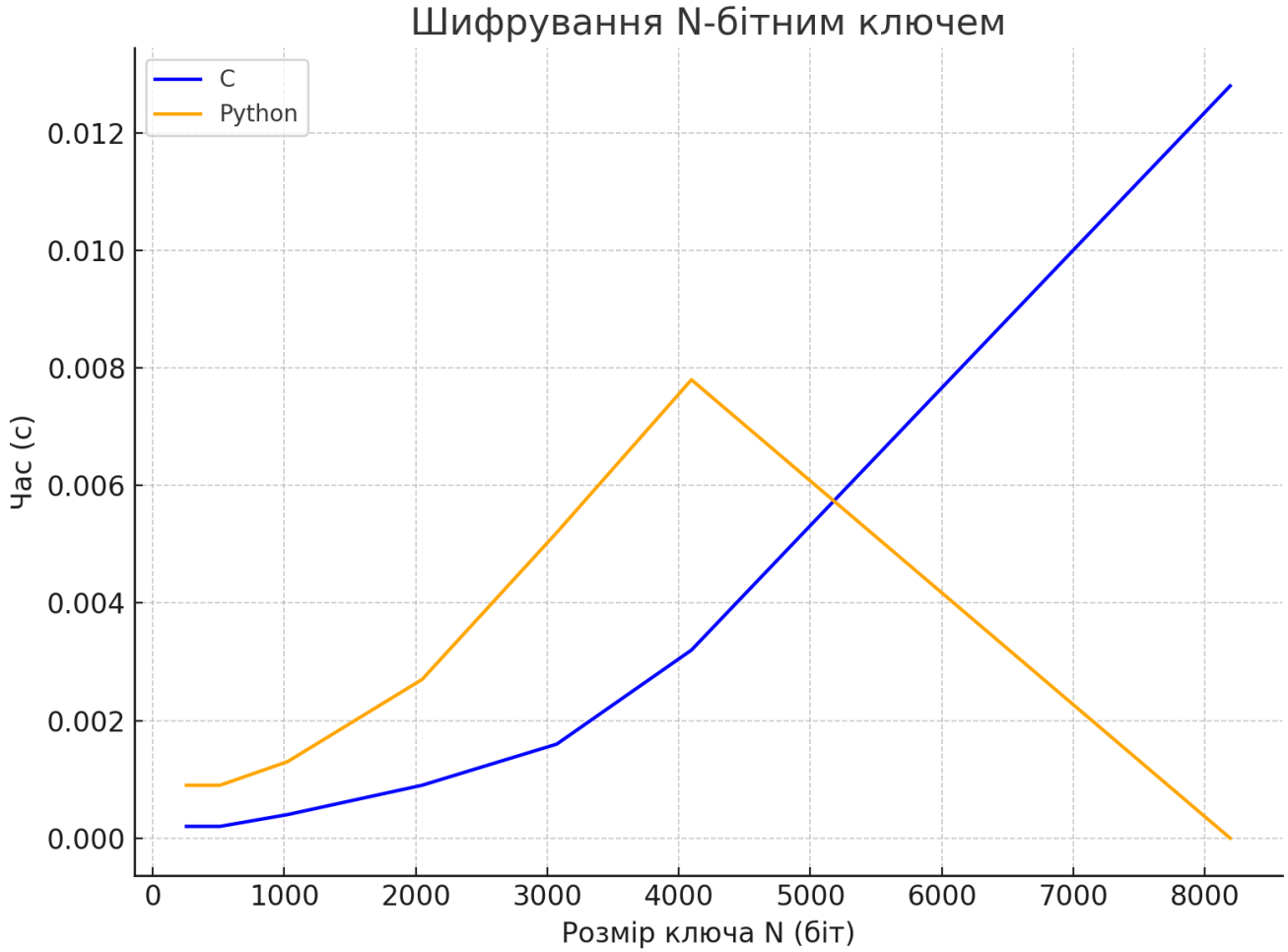


Рисунок 3.13 – Порівняння продуктивності шифрування у C та Python

Порівняльний аналіз часу, необхідного для виконання дешифрування, наведено на рисунку 3.14. На основі отриманих графіків можна зробити висновок, що реалізація RSA-алгоритму мовою C забезпечує суттєво вищу продуктивність та менший час обробки, ніж аналогічна реалізація мовою Python. Це підтверджує переваги використання низькорівневих мов програмування для виконання ресурсоємних криптографічних операцій [37].

Рисунок 3.14 – Залежність часу дешифрування RSA від довжини ключа

У результаті проведеного дослідження було встановлено, що реалізація алгоритму RSA мовою програмування C демонструє суттєві переваги порівняно з готовим RSA-модулем, написаним на Python. Основні висновки з проведеного експерименту можна сформулювати наступним чином:

а) Продуктивність і швидкодія. Реалізація RSA на C показала значно кращі результати за швидкістю виконання всіх етапів — генерації ключів, шифрування та дешифрування даних. Це свідчить про високу ефективність алгоритмічної реалізації та глибоку оптимізацію коду, характерну для мов низького рівня.

б) Робота з великими ключами. Модуль на C продемонстрував стабільну здатність працювати з ключами великої довжини (до 8192 біт), не виходячи за межі прийнятного часу виконання.

Натомість Python-модуль не зміг завершити генерацію ключа такої довжини у встановлений часовий інтервал, що вказує на меншу придатність для високонавантажених криптографічних задач [37].

ВИСНОВКИ

Розглянуто теоретичні основи криптографічного захисту інформації, що є фундаментальною складовою сучасної системи інформаційної безпеки. Було визначено, що криптографія забезпечує реалізацію таких ключових властивостей, як конфіденційність, цілісність, автентичність та незаперечність даних, які є критично важливими в умовах зростання кіберзагроз.

Проведено аналіз основних понять криптографії, таких як шифрування, дешифрування, криптоаналіз, ключі та хеш-функції. Здійснено класифікацію сучасних алгоритмів шифрування на блочні, поточні та хеш-функції, з детальним описом їхніх характеристик та механізмів роботи. Особливу увагу приділено огляду та порівнянню найбільш відомих криптографічних алгоритмів, таких як RSA, AES, ECC, ElGamal, DES та SHA-2, а також їх практичним сферам застосування.

Порівняльний аналіз симетричних та асиметричних методів шифрування дозволив виявити їхні переваги, недоліки та доцільність використання у різних прикладних сценаріях. Було встановлено, що оптимальним підходом у більшості систем захисту є використання гібридної криптографії, яка поєднує високу швидкодію симетричних алгоритмів із безпечним обміном ключами, забезпеченим асиметричними методами.

Досліджено криптографічний алгоритм RSA, його історичне становлення, принципи функціонування та математичне підґрунтя. Розглянуто основні етапи генерації ключів, включаючи вибір простих чисел та визначення публічного і приватного ключів.

Описано найпоширеніші типи атак, такі як факторизаційні атаки, атаки через побічні канали та інші, а також наведено сучасні методи захисту від криптоаналізу.

Проаналізовано практичне застосування RSA у сучасних криптографічних протоколах, зокрема SSL/TLS, SSH, PGP, а також його роль у створенні електронного цифрового підпису. Зроблено висновок, що RSA залишається одним

із найнадійніших алгоритмів для забезпечення конфіденційності, автентичності та цілісності даних у цифрових системах.

Розглянуто практичні аспекти реалізації криптографічного алгоритму RSA, з акцентом на чинники, що безпосередньо впливають на його криптостійкість та ефективність. Проведено обґрунтований вибір програмних засобів для реалізації, зокрема мови програмування C, яка забезпечує низькорівневий контроль та високу швидкодію.

Розроблений програмний модуль детально описано з поетапним поясненням роботи всіх його частин. У якості демонстраційного прикладу було використано рядки авторського вірша, які успішно зашифровано та дешифровано, що підтвердило працездатність розробленої системи.

Проведено порівняльний аналіз між створеним рішенням та готовою бібліотекою RSA, реалізованою мовою Python, з використанням однакових вхідних даних.

У результаті експериментів встановлено, що реалізація алгоритму на C є значно ефективнішою, особливо у задачах, де важлива висока продуктивність та обробка ключів великої довжини.

Таким чином, можна зробити висновок, що мова C є доцільним вибором для реалізації криптографічних алгоритмів у системах, що вимагають максимальної швидкодії, стабільності та контролю над ресурсами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Homomorphic Encryption. URL: https://chain.link/education-hub/homomorphicencryption

2. Все про технологію доведення з нульовим розголошенням і її вплив на блокчейн. URL: https://academy.binance.com/uk/articles/what-is-zero-knowledge-proof-and-how-does-itimpact-blockchain

3. Доведення з нульовим розголошенням (ZKP): як це працює і чому це важливо. URL: https://medium.com/@kyivskiicryptan/BE-1581b67634e9

4. Дерево Меркла. – URL: https://academy.binance.com/uk/glossary/merkle-tree

5. What Is a MixNet and How Does It Work? – URL: https://www.makeuseof.com/what-isa-mixnet

6. Burmaka Ivan, et al. Proof of Stake for Blockchain Based Distributed Intrusion Detecting System. In: International scientific-practical conference. Springer, Cham, 2020. P. 237–247.

Matsui M. Linear cryptanalysis method for DES cipher. // Proceeding of Eurocrypt-93, LNCS 765. Springer,- 1994,- P.386-397.

7. Rivest R., Shamir A., Adleman L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems // Communications of the ACM. – 1978. – Vol. 21(2). – P. 120–126.

8. Boneh D. Twenty Years of Attacks on the RSA Cryptosystem // Notices of the AMS. – 1999. – Vol. 46(2). – P. 203–213.

9. Shor P. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer // SIAM J. Comput. – 1997. – Vol. 26(5). – P. 1484–1509.

10. Stinson D. R. Cryptography: Theory and Practice. – 4th ed. – Chapman and Hall/CRC, 2018. – 728 p.

11. Paar C., Pelzl J. Understanding Cryptography: A Textbook for Students and Practitioners. – Springer, 2010.

12. Katz J., Lindell Y. Introduction to Modern Cryptography. – 2nd ed. – CRC Press, 2014.

13. Menezes A. J., van Oorschot P. C., Vanstone S. A. Handbook of Applied Cryptography. – CRC Press, 1996.

14. Mitrokotsa A., Vaudenay S. Side-Channel Attacks and Countermeasures. – In: Springer Briefs in Computer Science, 2012.

15. Smart N. P. Cryptography: An Introduction. – 3rd ed. – McGraw-Hill, 2008.

16. Bernstein D. J. Introduction to post-quantum cryptography. – Springer, 2009.

17. Chen L. et al. Report on Post-Quantum Cryptography. – NISTIR 8105. – National Institute of Standards and Technology, 2016.

18. Koblitz N., Menezes A. The brave new world of bodacious assumptions in cryptography // Notices of the AMS. – 2015.

19. Dhem J. F. et al. A Practical Implementation of the Timing Attack // Proceedings of CARDIS. – 1998.

20. Kocher P. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems // CRYPTO’96. – LNCS 1109. – P. 104–113.

21. OpenSSL Project. RSA Cryptography Overview. URL: https://www.openssl.org/docs/manmaster/man7/RSA.html

22. NIST. FIPS PUB 186-4: Digital Signature Standard (DSS). – National Institute of Standards and Technology, 2013.

23. ISO/IEC 9796-2:2010. Information technology – Security techniques – Digital signature schemes giving message recovery.

24. Lenstra A. K., Verheul E. R. Selecting Cryptographic Key Sizes // Journal of Cryptology. – 2001.

25. RSA Laboratories. PKCS #1: RSA Cryptography Standard (Version 2.2). – November 2012.

26. European Union Agency for Cybersecurity (ENISA). Algorithms, Key Size and Parameters Report – 2021.

27. Buchmann J. Introduction to Cryptography. – 2nd ed. – Springer, 2004. – 338 p.

28. Aggarwal D., Brennen G. K., Lee T., Santha M., Tomamichel M. Quantum Attacks on Public-Key Cryptosystems: Theory and Practice // Communications of the ACM. – 2017. – Vol. 60(2). – P. 62–71.

29. Wang Y., Hu R. A Survey on RSA Cryptosystem // Proceedings of the International Conference on E-Business and Telecommunications. – Springer, 2015.

30. Galbraith S. D. Mathematics of Public Key Cryptography. – Cambridge University Press, 2012.

31. Brier E., Clavier C., Olivier F. Correlation Power Analysis with a Leakage Model // CHES 2004. – LNCS 3156. – Springer, 2004.

32. Nguyen P. Q., Stern J. The Two Faces of Lattices in Cryptology // Cryptography and Lattices. – Springer, 2001.

33. Bos J. W., Kaihara M., Kleinjung T. Factoring RSA-768 with the Number Field Sieve // Asiacrypt 2010. – LNCS 6477. – Springer.

34. Avanzi R. Side-Channel Attacks: Ten Years After Its Publication, Kocher’s Timing Attack Still Threatens // IE Security & Privacy. – 2005. – Vol. 3(2).

35. Heninger N., Durumeric Z., Wustrow E., Halderman J. A. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices // USENIX Security Symposium. – 2012.

36. Bernstein D. J., Lange T. Post-Quantum Cryptography: State of the Art. – 2020.

37. Barker E. Recommendation for Key Management – Part 1: General // NIST SP 800-57, Rev. 5. – 2020.

ДЕМОНСТРАЦІЙНИЙ МАТЕРІАЛ (Презентація)

Державний університет інформаційно-комунікаційних технологій

Кафедра інформаційних систем та технологій

Кваліфікаційна робота на тему:

«АНАЛІЗ МЕТОДІВ КРИПТОГРАФІЧНОГО ЗАХИСТУ ДАНИХ»

на здобуття освітнього ступеня бакалавра
зі спеціальності 124 Системний аналіз
освітньо-професійної програми Системний аналіз

Виконав: КАРПІЧ Владислав САД-41
Науковий керівник роботи: МАСТАКОВ Олександр

КИЇВ - 2025

Актуальність дослідження. У сучасному цифровому світі інформація є одним із найцінніших ресурсів, а її захист – одним із ключових викликів кібербезпеки. З розвитком технологій, обсяг переданих і збережених даних стрімко зростає, що підвищує ризики несанкціонованого доступу, витоків та атак на інформаційні системи.

Метою роботи є аналіз методів криптографічного захисту даних, детальне дослідження алгоритму RSA, його практичного застосування та реалізація програмного рішення для шифрування та дешифрування інформації.

Об'єктом дослідження методи криптографічного захисту інформації та їх застосування у сучасних інформаційних системах.

Предмет дослідження – алгоритми криптографічного захисту, їх математичні основи, механізми реалізації та стійкість до атак.

Для досягнення цієї мети в роботі необхідно виконати наступні **завдання:**

- дослідити основні підходи до шифрування, класифікувати алгоритми на симетричні та асиметричні, розглянути їхні переваги та недоліки;
- вивчити математичні основи криптографічних алгоритмів, зокрема RSA, AES, ECC, та оцінити їхню криптостійкість у сучасних умовах;
- проаналізувати можливі вектори атак на криптографічні алгоритми, включаючи факторизацію чисел, атаки по часу виконання (timing attacks) та квантові загрози;
- оцінити ефективність алгоритму RSA у порівнянні з іншими методами шифрування за критеріями продуктивності та криптостійкості;
- розробити та протестувати програмну реалізацію алгоритму RSA для шифрування та дешифрування даних, проаналізувати її продуктивність та безпеку;
- сформувати рекомендації щодо вибору криптографічних методів для захисту інформації в сучасних інформаційних системах.

Криптографія як ключовий елемент інформаційної безпеки

Криптографія – це наука та практика забезпечення конфіденційності, цілісності, автентичності та достовірності інформації шляхом її перетворення у форму, незрозумілу для сторонніх осіб, за допомогою спеціальних математичних методів та алгоритмів.

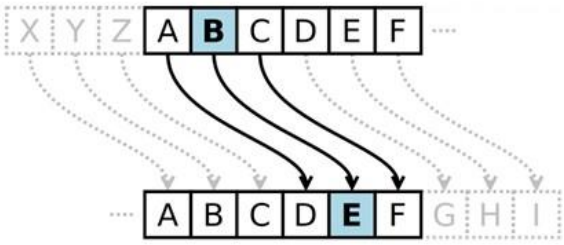


Рисунок 1 – Схематичне зображення шифру Цезаря

1	т	е	к	с	т
2	о	в	е	п	о
3	в	і	д	о	м
4	л	е	н	н	я

Рисунок 2 Шифр Скитала

Рисунок 3 Квадрат Віженера

Таблиця 1 – Порівняльна таблиця методів шифрування

Критерій	Симетричне шифрування	Асиметричне шифрування	Гібридне шифрування
Кількість ключів	Один спільний ключ	Пара: відкритий і закритий ключі	Обидва типи ключів
Швидкодія	Висока	Нижча через складність обчислень	Висока (основна частина – симетричне шифрування)
Безпека обміну ключами	Вразлива	Надійна	Надійна
Складність реалізації	Нижча	Вища	Найвища
Застосування	Шифрування файлів, VPN	Цифрові підписи, обмін ключами	Інтернет-протоколи (TLS, HTTPS, PGP)

Огляд сучасних алгоритмів шифрування та їх класифікація

Алгоритм	Тип шифрування	Довжина ключа (типова)	Стійкість до атак	Швидкодія	Основні сфери застосування
RSA	Асиметричне	2048–4096 біт	Висока (при достатній довжині ключа)	Низька	Цифрові підписи, передача ключів, сертифікати
AES	Симетричне	128/192/256 біт	Висока	Висока	VPN, Wi-Fi, захист файлів, бази даних
ECC	Асиметричне	160–521 біт	Дуже висока при коротших ключах	Вища за RSA при однаковій безпеці	IoT, мобільні пристрої, блокчейн
ElGamal	Асиметричне	≥1024 біт	Висока	Низька	PGP, електронна пошта, цифровий підпис
DES	Симетричне	56 біт	Низька (застарілий)	Середня	Історичне значення, деякі апаратні пристрої
SHA-2	Хеш-функція	256/512 біт	Висока (SHA-2), SHA-1 – зламана	Дуже висока	Контроль цілісності, цифрові підписи, паролі

5

Історичний розвиток та принцип роботи RSA

Алгоритм RSA

Створено у 1977 р. Р. Рівестом, А. Шаміром і Л. Адлеманом (MIT).

•Перший **практичний алгоритм асиметричної криптографії**.

•Оснований на складності **факторизації великих простих чисел**.

•Кожен користувач має **відкритий** і **закритий** ключ.

•Застосування:

- Електронна пошта (PGP, S/MIME)
- SSL/TLS, X.509 сертифікати
- Е-підписи, банківські системи, блокчейн

•Переваги: безпечна передача даних, цифровий підпис.

•Недоліки:

- Повільне шифрування
- Великі розміри ключів (2048+ біт)
- Вразливість до **квантових обчислень** (алгоритм Шора)

•У майбутньому RSA можуть замінити **постквантові алгоритми**.

Атаки на RSA та методи захисту від криптоаналізу

Уразливості та загрози для RSA

•Факторизація модуля n

→ Квадратичне та числове решето (особливо небезпечно для ключів < 1024 біт)

•Слабкі ключі / помилки налаштування

→ Низьке e (наприклад, e=3), схожі p і q, повторне використання n

•Атаки по сторонньому каналу

→ Аналіз часу (Timing Attack), енергоспоживання, випромінення, помилки при обчисленнях

•Квантові обчислення

→ Алгоритм Шора зможе зламати RSA, коли з'являться потужні квантові комп'ютери

•Реакція спільноти

→ NIST розробляє **постквантові алгоритми** для заміни RSA в майбутньому

Використання RSA у криптографічних протоколах та цифровому підписі

SSL/TLS

•Аутентифікація сервера

•Безпечний обмін симетричним ключем

•RSA використовується на етапі встановлення з'єднання

•У TLS 1.3 частково замінений на еліптичні криві (X25519)

SSH

•Безпечне підключення до віддалених систем

•RSA використовується для генерації ключів та встановлення симетричного шифрування

PGP

•Захист листування та файлів

•RSA шифрує симетричний ключ та створює цифровий підпис

•Поєднується з AES для шифрування даних

Електронний цифровий підпис (ЕЦП)

•Підпис створюється шифруванням хешу приватним ключем

•Верифікація — через публічний ключ

•Забезпечує: автентичність, цілісність, невідмовність

Розробка програмної реалізації та тестування ефективності

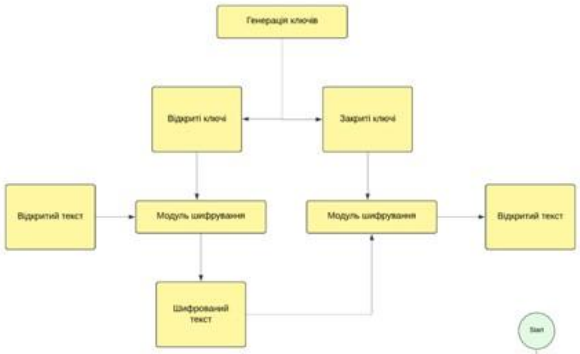


Рисунок 4 – Структурна схема програмного модуля шифрування RSA

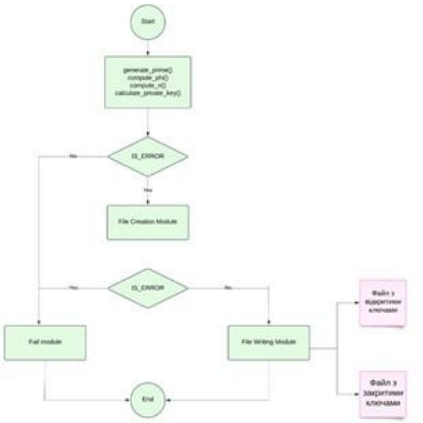


Рисунок 5 – Алгоритмічна схема формування ключової пари RSA

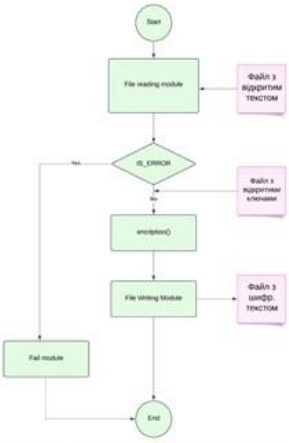


Рисунок 6 – Алгоритмічна схема роботи шифрувального модуля

Аналіз продуктивності та порівняння з іншими алгоритмами

project — nano Makefile — 166×40

```
  UW PICO 5.09                                File: Makefile

# Параметри компіляції
CC = gcc
CFLAGS = -Wall -Werror -Wextra -Wfloat-equal -Warray-bounds -Wdiv-by-zero -Wdouble-promotion -g
LDFLAGS = -L/opt/homebrew/lib -I/opt/homebrew/include -lgmp -DTIME

# Об'єктні файли
OBJS = crypto.o

# Ціль за замовчуванням
all: rsa_cr key_gen rsa_decr

# Збірка crypto.o
crypto.o: crypto.c
        $(CC) $(CFLAGS) -c crypto.c -o crypto.o

# Збірка rsa_cr
rsa_cr: rsa_cr.c $(OBJS)
        $(CC) $(CFLAGS) rsa_cr.c $(OBJS) -o rsa_cr $(LDFLAGS)

# Збірка генератора ключів
key_gen: key_gen.c $(OBJS)
        $(CC) $(CFLAGS) key_gen.c $(OBJS) -o key_gen $(LDFLAGS)

# Збірка дешифрувальника
rsa_decr: rsa_decr.c $(OBJS)
        $(CC) $(CFLAGS) rsa_decr.c $(OBJS) -o rsa_decr $(LDFLAGS)

# Очистка
clean:
        rm -f *.o rsa_cr key_gen rsa_decr
```

project — nano rsa_cr.c — 166×40

```
  UW PICO 5.09                                File: rsa_cr.c

#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Використання: %s public.key input.txt output.txt\n", argv[0]);
        return 1;
    }

    FILE *pub = fopen(argv[1], "r");
    if (!pub) {
        perror("Не вдалося відкрити файл публічного ключа");
        return 1;
    }

    mpz_t e, n;
    mpz_inits(e, n, NULL);
    gmp_fscanf(pub, "%Zd\n%Zd\n", e, n);
    fclose(pub);

    FILE *in = fopen(argv[2], "r");
    if (!in) {
        perror("Не вдалося відкрити вхідний файл");
        return 1;
    }

    fseek(in, 0, SEEK_END);
    long size = ftell(in);
    rewind(in);

    char *buffer = malloc(size + 1);
    fread(buffer, 1, size, in);
    buffer[size] = '\0';
```

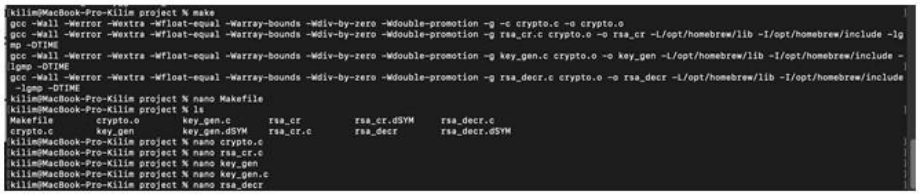
project — nano crypto.c — 166×40

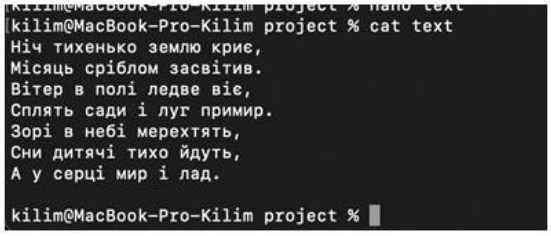
```
  UW PICO 5.09                                File: crypto.c

#include <stdio.h>

void crypto_placeholder() {
    printf("Файл crypto.c: працює загальна крипто-функція.\n");
}
```

project — nano key_gen.c — 166×40

```
  UW PICO 5.09                                File: key_gen.c

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <gmp.h>

// Генерація великого простого числа
gmp_randstate_t state;

void generate_prime(mpz_t prime, int bits) {
    mpz_rrandomb(prime, state, bits);
    mpz_nextprime(prime, prime);
}

// Обчислення НСД (алгоритм Евкліда)
void compute_gcd(mpz_t result, mpz_t a, mpz_t b) {
    mpz_gcd(result, a, b);
}

// Обернене число (розширений алгоритм Евкліда)
void compute_inverse(mpz_t result, mpz_t e, mpz_t phi) {
    if (mpz_invert(result, e, phi) == 0) {
        fprintf(stderr, "Помилка: немає оберненого числа для e!\n");
        exit(1);
    }
}

int main(int argc, char *argv[]) {
    int bits = 2048;
    if (argc > 1) {
        bits = atoi(argv[1]);
    }

    // Ініціалізація
    mpz_t p, q, n, phi, e, d, gcd;
    mpz_inits(p, q, n, phi, e, d, gcd, NULL);

^G Get Help    ^O WriteOut    ^R Read File    ^Y Prev Pg
^X Exit        ^J Justify     ^W Where is     ^V Next Pg
```

Аналіз продуктивності та порівняння з іншими алгоритмами





Аналіз продуктивності та порівняння з іншими алгоритмами

```
kilim@MacBook-Pro-Kilim project % ./rsa_cr public.key text encrypted.txt
Час шифрування: 0.000361 секунд
Шифрування завершено. Зашифрований текст записано у encrypted.txt
kilim@MacBook-Pro-Kilim project %
```

```
kilim@MacBook-Pro-Kilim project % ./rsa_decr private.key encrypted.txt decrypted.txt
Час дешифрування: 0.004141 секунд
Розшифрування завершено. Вихідний текст записано у decrypted.txt
kilim@MacBook-Pro-Kilim project %
```

```
kilim@MacBook-Pro-Kilim project % cat decrypted.txt
Ніч тихенько землю криє,
Місяць сріблом засвітив.
Вітер в полі ледве віє,
Сплять сади і луг примир.
Зорі в небі мерехтять,
Сни дитячі тихо йдуть,
А у серці мир і лад.

kilim@MacBook-Pro-Kilim project %
```

Аналіз продуктивності та порівняння з іншими алгоритмами

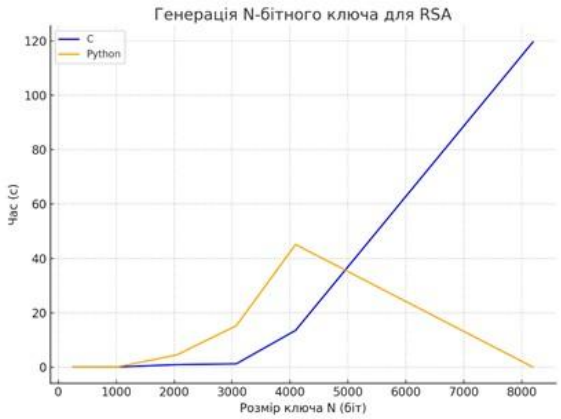


Рисунок 7 – Порівняння часу створення RSA-ключів у мовах C та Python

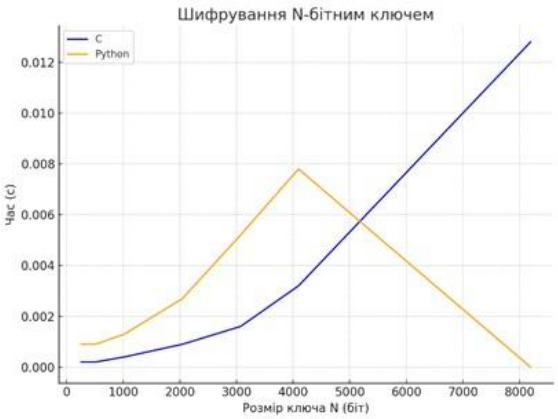


Рисунок 8 – Порівняння продуктивності шифрування у C та Python

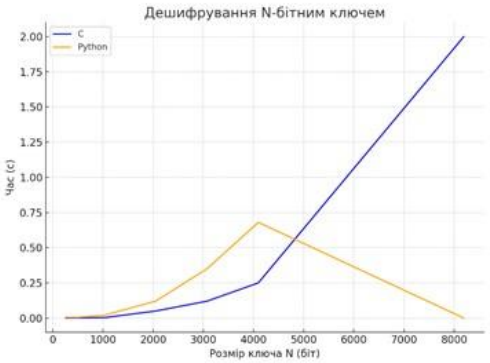


Рисунок 9 – Залежність часу дешифрування RSA від довжини ключа

ВИСНОВКИ

- Криптографія є ключовим елементом інформаційної безпеки, забезпечуючи конфіденційність, цілісність, автентичність і незаперечність даних.
- Проведено класифікацію сучасних алгоритмів шифрування (симетричних, асиметричних, хеш-функцій) та аналіз їх переваг і недоліків.
- Алгоритм RSA досліджено з точки зору принципу дії, історії створення та математичної основи; описано процедуру генерації ключів та типи криптоаналітичних атак.
- Показано, що гібридна криптографія є оптимальним підходом, який поєднує переваги симетричного шифрування з безпечним розповсюдженням ключів.
- Проаналізовано застосування RSA в реальних системах: SSL/TLS, SSH, PGP, цифрові підписи, де алгоритм відіграє критично важливу роль.
- Реалізовано власний програмний модуль RSA мовою C, що продемонстрував високу продуктивність і ефективність у порівнянні з реалізацією на Python.
- Мова програмування C обґрунтовано вибрана як оптимальний інструмент для розробки криптографічних рішень у системах, де важливі швидкодія, стабільність та контроль над ресурсами.

Дякую за увагу!
Доповідь закінчено