Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра кібербезпеки
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(освітній рівень)

на тему: "Порівняльний аналіз алгоритмів цифрового підпису"

Виконав: студент (ка) IV курсу, групи СБ-41

Спеціальності:
125 «Кібербезпека»
(шифр і назва напряму підготовки, спеціальності)
Іващишин Максим Володимирович
підпис (прізвище та ініціали)

Керівник Загородна Н.В.
підпис (прізвище та ініціали)

Нормоконтроль Дроздова Т.В.
підпис (прізвище та ініціали)

Завідувач кафедри Загородна Н.В.
підпис (прізвище та ініціали)

Рецензент
підпис (прізвище та ініціали)

м. Тернопіль – 2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра кібербезпеки
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Загородна Н.В.
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 125 Кібербезпека
(шифр і назва спеціальності)

Студенту Іващишину Максиму Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Порівняльний аналіз алгоритмів цифрового підпису

Керівник роботи Загородна Наталія Володимирівна, к.т.н., доцент, завідувач кафедри
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «02» 05 2025 року № 4/7-361

2. Термін подання студентом завершеної роботи 16.06.2025

3. Вихідні дані до роботи Літературні джерела по тематиці дослідження

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ
Теоретичні основи цифрового підпису як одного з базових криптографічних інструментів
Математичні моделі алгоритмів цифрового підпису
Порівняння алгоритмів цифрового підпису
Безпека життєдіяльності, основи охорони праці
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Мета та завдання. Завдання безпеки інформації. Теоретичні аспекти цифрового підпису. Алгоритм DSA. Алгоритм ECDSA. Алгоритм EDDSA. Алгоритм RSA. Тестування в ОС Windows. Тестування в ОС Linux. Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи хорони праці	Мариненко С.Ю, к.т.н., доцент кафедри МТ		

7. Дата видачі завдання 29.01.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	20.01 – 24.01	Виконано
2.	Підбір джерел для аналізу алгоритмів цифрового підпису	28.01 – 06.02	Виконано
3.	Опрацювання джерел в галузі дослідження	07.02 – 10.02	Виконано
4.	Проведення аналізу вимог та рішень забезпечення безпеки	12.02 – 16.03	Виконано
5.	Програмна реалізація алгоритмів та модуля аналізу продуктивності	17.03-13.04	Виконано
6.	Досліждення та аналіз отриманих метрик	13.04 – 16.04	Виконано
7.	Оформлення розділу «Теоретичні основи цифрового підпису як одного з базових криптографічних інструментів»	17.04 – 20.04	Виконано
8.	Оформлення розділу «Математичні моделі алгоритмів цифрового підпису»	22.04 – 28.05	Виконано
9.	Оформлення розділу «Порівняння алгоритмів цифрового підпису»	01.05-20.05	Виконано
10.	Виконання завдання до підрозділу «Безпека життєдіяльності, основи охорони праці»	20.05 – 25.05	Виконано
11.	Оформлення кваліфікаційної роботи	26.05 – 10.06	Виконано
12.	Нормоконтроль	13.06 – 16.06	Виконано
13.	Перевірка на плагіат	16.06-18.06	Виконано
14.	Попередній захист кваліфікаційної роботи	19.06	Виконано
15.	Захист кваліфікаційної роботи	25.06.2025	

Студент _____ Іващишин М.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Загородна Н.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Порівняльний аналіз алгоритмів цифрового підпису // Кваліфікаційна робота ОР «Бакалавр» // Іващишин Максим Володимирович // // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра кібербезпеки, група СБ-41 // Тернопіль, 2025 // С. 89, рис. – 20, табл. – 1, кресл. – 0, додат. – 3.

КЛЮЧОВІ СЛОВА: цифровий підпис, алгоритми DSA, ECDSA, EdDSA, RSA, захист інформації.

Дослідження присвячено аналізу та оцінці ефективності сучасних алгоритмів цифрового підпису. Вибір оптимального алгоритму цифрового підпису є критично важливим для забезпечення безпеки інформаційних систем. Отримані результати можуть бути використані для вдосконалення систем кібербезпеки.

У першому розділі проведено аналіз сучасних рішень для забезпечення безпеки інформації, розглянуто теоретичні аспекти цифрового підпису та визначено ключові вимоги до систем, що їх використовують, з акцентом на надійність, стійкість до атак і практичне застосування.

Другий розділ присвячено теоретичним основам і математичним моделям алгоритмів цифрового підпису (DSA, ECDSA, EdDSA, RSA). Описано їхні компоненти, принципи функціонування та особливості, що впливають на безпеку, швидкодію й ефективність у системах захисту інформації.

Третій розділ охоплює практичну реалізацію алгоритмів у програмній системі, розробленій для тестування їхньої продуктивності. Проведено порівняльний аналіз за швидкістю виконання, споживанням ресурсів.

ABSTRACT

Comparative Analysis of Digital Signature Algorithms // Thesis of educational level "Bachelor"// Ivashchyshyn Maksym // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Cybersecurity, group СБ-41 // Ternopil, 2025 // P. 89 fig. – 20, tab. – 1, drawing – 0, added. – 3.

Keywords: digital signature, DSA, ECDSA, EdDSA, RSA algorithms, information security.

This research focuses on the analysis and evaluation of the effectiveness of modern digital signature algorithms. The selection of an optimal digital signature algorithm is critically important for ensuring the security of information systems. The obtained results can be used to improve cybersecurity systems.

The first chapter analyzes modern solutions for information security, examines the theoretical aspects of digital signatures, and defines key requirements for systems that use them, with an emphasis on reliability, attack resistance, and practical application.

The second chapter is dedicated to the theoretical foundations and mathematical models of digital signature algorithms (DSA, ECDSA, EdDSA, RSA). It describes their components, operating principles, and features that impact security, speed, and efficiency in information security systems.

The third chapter covers the practical implementation of the algorithms in a software system developed for testing their performance. A comparative analysis was conducted based on execution speed and resource consumption.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CIA	—	Конфіденційність, цілісність і доступність
ЕЦП	—	Електронний цифровий підпис
DSA	—	Digital Signature Algorithm
ECDSA	—	Elliptic Curve Digital Signature Algorithm
EdDSA	—	Edwards-curve Digital Signature Algorithm
NIST	—	National Institute of Standards and Technology
SHA	—	Secure Hash Algorithm
ПЗ	—	Програмне забезпечення
CPU	—	Центральний процесор
ЦП	—	Цифровий підпис

ВСТУП

У сучасному світі забезпечення конфіденційності цілісності, доступності та даних є досить важливим. Цифрові підписи відіграють ключову роль у досягненні цих цілей, дозволяючи перевіряти походження та незмінність даних. Вони також вирішують завдання автентифікації автора повідомлення, підтверджуючи його створення конкретною особою чи системою та забезпечуючи довіру до взаємодії. Отже, вибір оптимального алгоритму цифрового підпису є надзвичайно важливим для забезпечення безпеки інформаційних систем.

Метою кваліфікаційної роботи є порівняльний аналіз та оцінка ефективності сучасних алгоритмів цифрового підпису. Для досягнення мети було поставлено задачу: проаналізувати технічні та теоретичні вимоги до систем цифрового підпису, дослідити математичні основи алгоритмів цифрового підпису, розробити та протестувати алгоритми цифрового підпис та провести практичне порівняння алгоритмів.

Об'єктом дослідження є сучасні алгоритми цифрового підпису, зокрема RSA, DSA, ECDSA та EdDSA.

Предметом дослідження є критерії ефективності та продуктивності алгоритмів цифрового підпису, що включає оцінку їхніх характеристик за такими метриками, як швидкість виконання операцій підписання та верифікації, а також споживання обчислювальних ресурсів, з метою визначення їх оптимального застосування для вдосконалення систем кібербезпеки.

Отримані результати можуть бути використані для вдосконалення систем кібербезпеки. Це дослідження надає детальну інформацію про продуктивність та ефективність алгоритмів DSA, ECDSA, EdDSA та RSA, що є важливим для вибору оптимального алгоритму в умовах обмежених ресурсів або для систем, що потребують високої швидкодії та енергоефективності.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ЦИФРОВОГО ПІДПИСУ ЯК ОДНОГО З БАЗОВИХ КРИПТОГРАФІЧНИХ ІНСТРУМЕНТІВ

1.1 Аналіз рішень забезпечення безпеки інформації

Захист інформації базується на тріаді CIA відіграє важливу роль у формуванні політики та практик, спрямованих на захист інформації. Ця модель, що включає конфіденційність, цілісність і доступність, забезпечує захист конфіденційних даних і підтримку надійності систем. Зосереджуючись на цих трьох важливих принципах, тріада CIA створює основу для захисту мереж і запобігання зловмисним атакам [1].

Конфіденційність – це забезпечення того, щоб інформація була доступна лише тим лицям, які мають на це право, тобто це властивість інформації бути захищеною від несанкціонованого ознайомлення. Конфіденційність передавання даних забезпечується за допомогою шифрування. В основі забезпечення конфіденційності лежать принципи симетричного та асиметричного шифрування.

Методи симетричного шифрування, як можна зрозуміти із назви використовують лише один крипто-ключ для перетворення даних і для дешифрування. Такий підхід має переваги над асиметричними методами, він значно швидший, потребує меншої потужності в обчисленні. Найбільш поширені та найбільш використовувані алгоритми симетричних систем – AES, DES, 3DES.

Асиметричне шифрування – це методи, які включають в себе кілька ключів для шифрування та дешифрування. Також, цей метод називають криптографією з відкритим ключем. Існує один ключ відомий як відкритий, а інший – як закритий. У цьому методі дані шифруються за допомогою відкритого ключа, який є загальнодоступним, а дані розшифровуються за допомогою закритого ключа, який потрібно зберігати в безпечному стані. До найбільш відомих алгоритмів шифрування в асиметричній криптографії належать, наприклад, Ель-

Гамаля, RSA, ECC та інші. Ці алгоритми є основою сучасної безпеки в інтернеті. Різницю між двома різними системами наведено на рисунку 1.1.

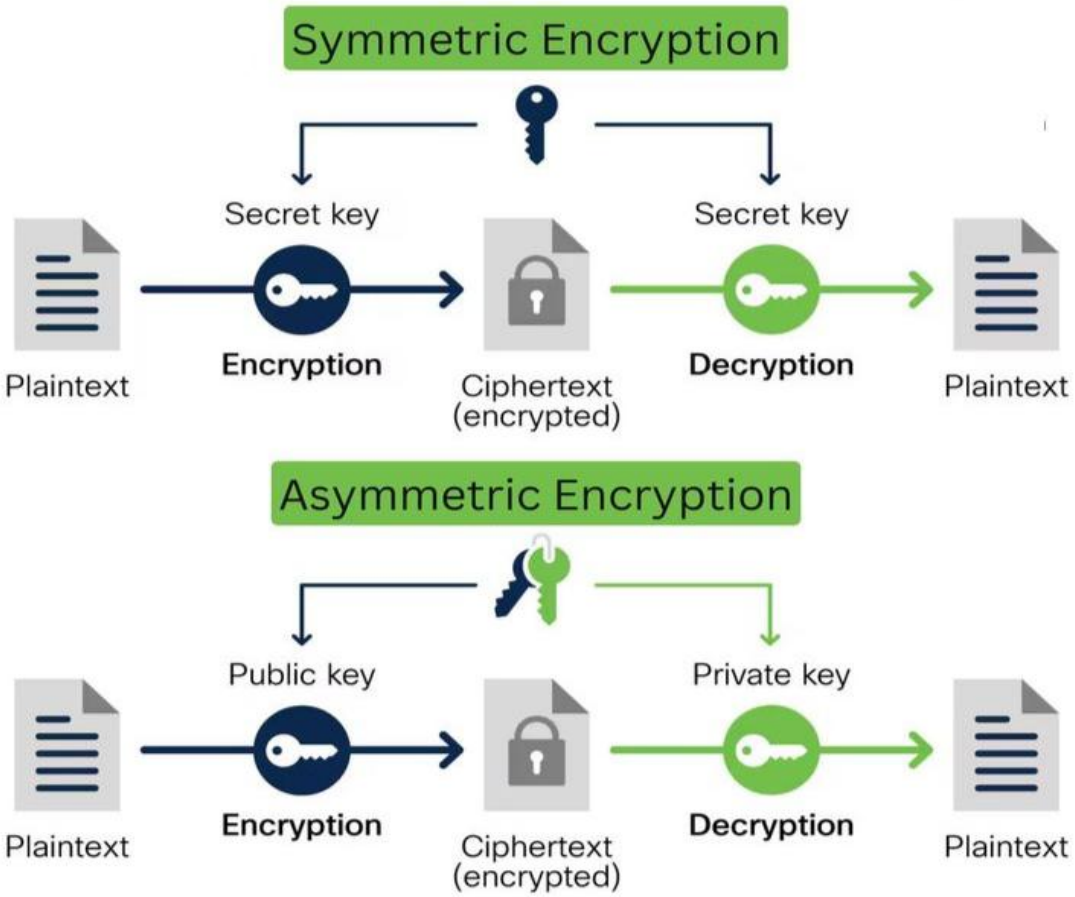

Рисунок 1.1 – Схема шифрування та розшифровування у двох системах

На рисунку 1.1 зображено дві схеми конвеєру даних, симетричне шифрування використовує один ключ для шифрування та дешифрування даних, тоді як асиметричне – пару ключів: відкритий (для шифрування) та закритий (для дешифрування).

Цілісність – це властивість інформації бути захищеною від несанкціонованого змінення, підроблені або знищені. Іншими словами, це забезпечення того, що дані залишаються точними, повними та незмінними від моменту їх створення до моменту їх використання. На відміну від двох інших властивостей цілісність фокусується на незмінності та автентичності даних. Хеш-функція – це одностороння функція, яка хешує інформацію тобто перетворює вхідні дані довільного розміру у ряд фіксованої довжини. Особливістю такого механізму є лавинний ефект: найменша зміна вхідного

масиву призводить до значної зміни хешу. Також стійкість до колізій – практично неможливо знайти два різні вхідні дані, які виробляють однаковий хеш. На рисунку 1.2 зображено схему роботи хешування.

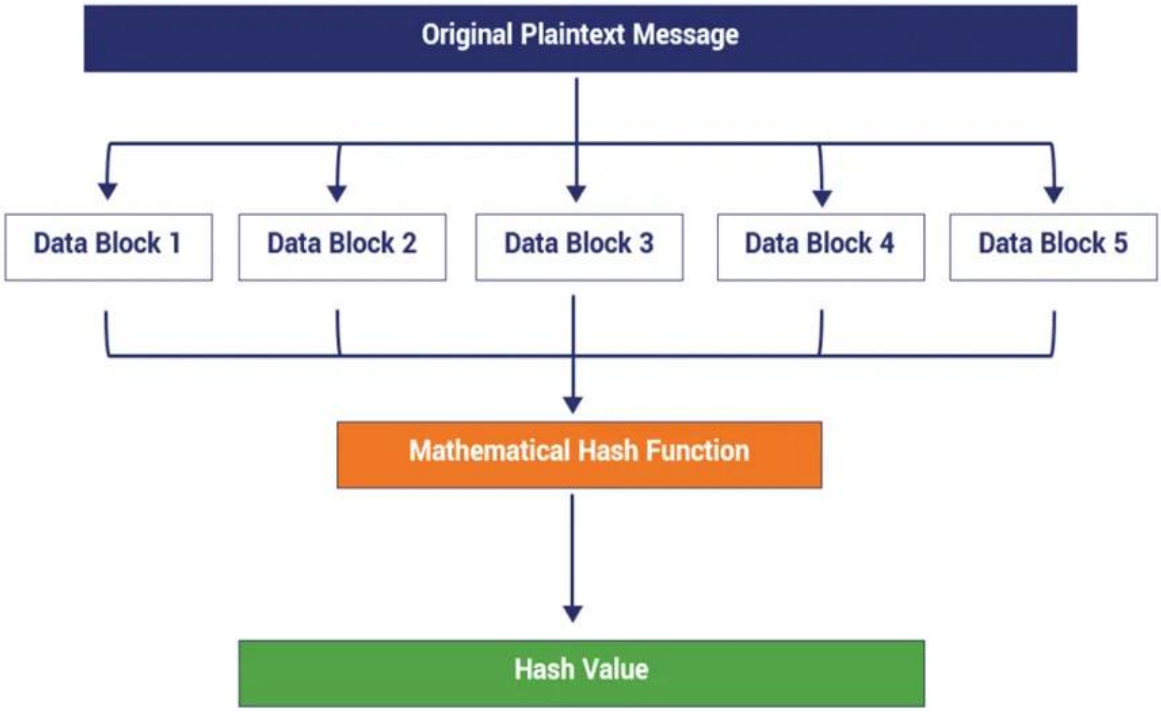

Рисунок 1.2 – Схема перетворення інформації в хеш-значення

На рисунку 1.2 можна побачити, що дані хешуються таким чином: вхідний текст поділяється на блоки, які обробляються математичною хеш-функцією, що перетворює їх у унікальне хеш-значення фіксованої довжини.

Доступність – це властивість системи забезпечувати своєчасний доступ до даних для авторизованих користувачів. Вона є критичною у тріаді CIA, оскільки навіть найбезпечніша система втрачає сенс, якщо нею неможливо скористатися у потрібний момент.

Для забезпечення доступності застосовують резервні системи, розподілені сховища даних, хмарні технології та механізми автоматичного відновлення після збоїв. Основні загрози включають апаратні збої, кібератаки (наприклад, DDoS) та природні катастрофи.

У контексті електронного цифрового підпису доступність має вирішальне значення: будь-який спростій системи може порушити процеси підписання або

перевірки документів, що призводить до фінансових втрат або правових наслідків. Тому такі системи будують на основі географічно розподілених серверів, кластерів і резервних рішень для мінімізації простоїв.

Отже, CIA служить простою моделлю для стратегій безпеки. Організації захищають свою інформацію, оскільки вона залишається конфіденційною. Вони підтримують цілісність своїх даних, а системи залишаються доступними для користувачів. Дотримання принципів тріади важливе для захисту від можливих загроз. Це також зменшує наслідки порушень безпеки.

Проте з плином часу з’явились нові виклики та задачі захисту інформації. Одним із ключових аспектів забезпечення безпеки інформації є автентифікація, яка відіграє не менш важливу роль. Автентифікація дозволяє підтвердити ідентичність суб’єктів (користувачів, авторів чи джерел повідомлень), що беруть участь у процесі обміну інформацією, та забезпечити довіру до взаємодії в інформаційних системах.

Автентифікація користувача (User Authentication) – це найширше поняття, яке стосується перевірки ідентичності особи, що намагається отримати доступ до системи, ресурсу або сервісу. Головна мета – переконатися, що "користувач, який заявляє про себе, дійсно є тим, ким він себе називає". Це може бути вхід до веб-сайту, мобільного застосунку, тощо. Методи включають паролі, біометричні дані, двофакторна або багатофакторна автентифікація.

Автентифікація джерела повідомлення (Message Source Authentication – цей термін зосереджується на перевірці походження конкретного повідомлення або даних. Тобто, ми хочемо переконатися, що повідомлення було відправлене саме тим відправником, від якого воно нібито надійшло. Це критично важливо, щоб уникнути ситуацій, коли зловмисник видає себе за легітимного відправника. Одним із традиційних методів автентифікації джерела повідомлення є використання кодів аутентичності повідомлення (MAC, Message Authentication Codes). MAC-коди базуються на симетричних криптографічних алгоритмах, де відправник і одержувач повідомлення використовують спільний секретний ключ для створення та перевірки коду автентичності. Робота алгоритму зображено на рисунку 1.3.

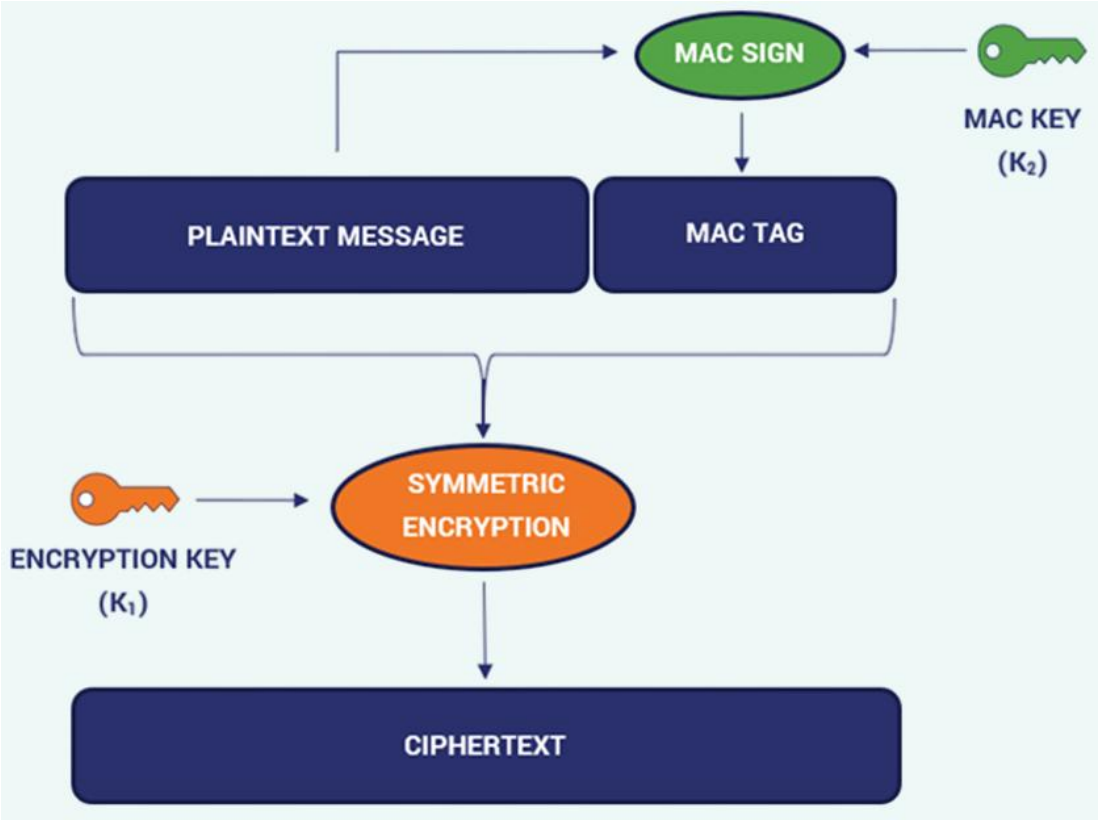

Рисунок 1.3 – Схема шифрування з MAC-тегом

На рисунку 1.3 показано схему роботи MAC-коду. Відправник обчислює унікальний MAC-тег для повідомлення за допомогою секретного ключа та надсилає його разом з повідомленням одержувачу. Одержувач, використовуючи той самий секретний ключ, переобчислює MAC-тег. Якщо теги збігаються, це підтверджує, що повідомлення не було змінене і надійшло від власника ключа. Варто зазначити, що саме по собі шифрування не забезпечує повної автентифікації; для цього потрібні додаткові механізми.

Автентифікація автора повідомлення (Message Author Authentication) – цей термін дуже схожий на автентифікацію джерела, часто використовується як синонім. Однак, іноді він може мати дещо глибший сенс, підкреслюючи саме особу, яка створила, а не просто передала повідомлення. Наприклад, якщо повідомлення пройшло через кілька проміжних вузлів, автентифікація джерела може підтвердити, що воно надійшло від останнього вузла перед нами, а автентифікація автора – що його створив певний користувач чи програма на початковому етапі. Цю задачу автентифікації вирішує вже цифровий підпис.

Це робить ЕЦП незамінним інструментом у системах, де необхідно гарантувати, що повідомлення не тільки надходить від надійного джерела, але й створене конкретною особою чи системою. Таким чином, ЕЦП ефективно вирішує проблему автентифікації автора, усуваючи неоднозначності, пов’язані з передачею даних через ненадійні канали.

1.2 Теоретичні аспекти про цифровий підпис

Цифровий підпис – це криптографічний механізм, який використовується для перевірки справжності та цілісності цифрових даних. Ми можемо розглядати його як цифрову версію звичайних рукописних підписів, але з вищим рівнем складності та безпеки. Ми можемо описати цифровий підпис як код, що прикріплений до повідомлення або документа. Після створення, код діє як доказ того, що повідомлення не було підроблено на шляху від відправника до одержувача [2].

Цифровий підпис це не випадковий набір символів, а невід'ємна частина інфраструктури відкритих ключів (PKI), яка забезпечує безпечне використання асиметричної криптографії. У PKI кожен користувач має пару ключів: приватний ключ для створення підпису, що зберігається в секреті, та публічний ключ для його перевірки, який вільно поширюється. Сам собою публічний ключ не може гарантувати, що він належить саме тій особі, за яку вона себе видає. Тут на допомогу приходять Центри сертифікації ключів (ЦСК). Це довірені треті сторони, які видають цифрові сертифікати.

Цифровий сертифікат – це електронний документ, що пов'язує публічний ключ з певною особою або організацією, підтверджуючи її ідентичність. Коли ви отримуєте документ із цифровим підписом, ваша система перевіряє не лише сам підпис, а й чинність сертифіката, який підтверджує належність публічного ключа. Це створює ланцюг довіри: користувач довіряє ЦСК, ЦСК підтверджує ідентичність власника публічного ключа, а цей ключ, своєю чергою, підтверджує дійсність підпису.

Призначення цифрового підпису виходить далеко за межі простої перевірки. Він виконує три ключові функції, які є наріжними каменями безпечного електронного документообігу:

– контроль цілісності даних, як уже неодноразово згадувалося, цифровий підпис гарантує, що повідомлення чи документ не були змінені після того, як їх підписали. Будь-яка, навіть найменша, модифікація даних призведе до того, що перевірка підпису завершиться невдачею, сигналізуючи про компрометацію;

– автентифікація джерела – цифровий підпис однозначно ідентифікує підписувача. Оскільки, лише власник приватного ключа може створити дійсний підпис, одержувач може бути впевнений, що документ надійшов від нього;

– невідмовність, так звана незаперечність – це одна із найважливіших, тому що підпис створюється саме унікальним приватним ключем, який може належати тільки одній особі, підписувач не може пізніше заперечити факт використання та підписання певного документа.

Електронний цифровий підпис дозволяє точно ідентифікувати автора повідомлення, навіть якщо воно пройшло через численні проміжні вузли, забезпечуючи додатково неспростовність і цілісність даних (див. рисунок 1.4).

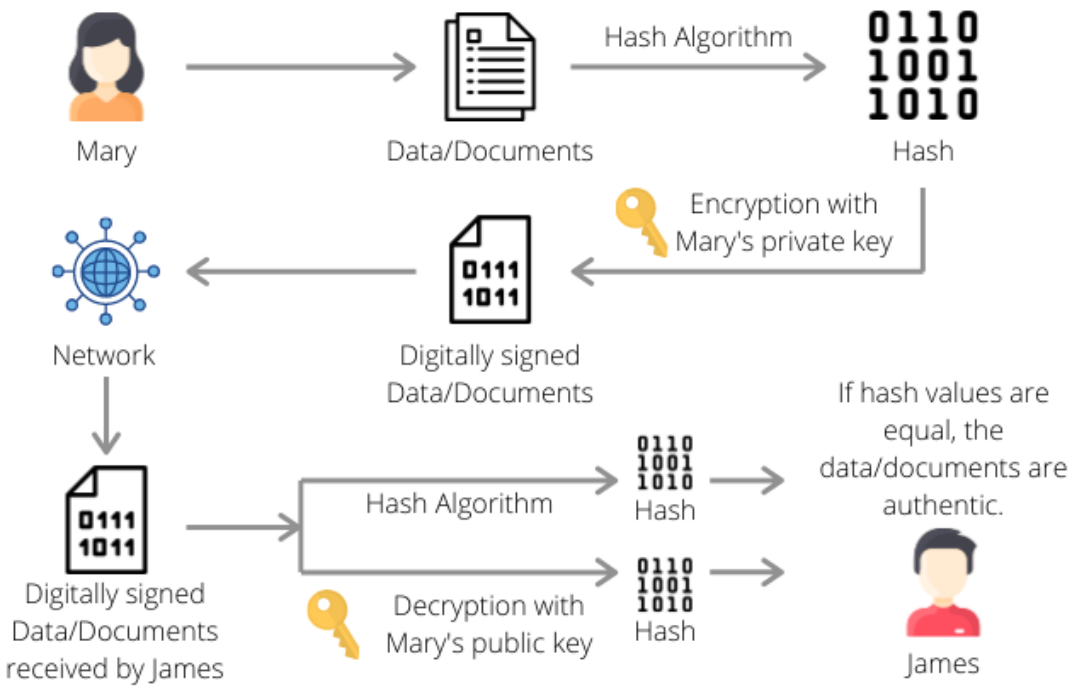

Рисунок 1.4 – Процес підписання документа

Ця схема демонструє роботу цифрового підпису: за допомогою хеш-алгоритму та асиметричного шифрування. Спочатку дані хешуються, отримуючи унікальний хеш, який потім шифрується приватним ключем Мері, утворюючи цифровий підпис. Отримувач (Джеймс) розшифровує підпис загальнодоступним ключем Мері, відновлюючи хеш, далі він тією ж хеш-функцією хешує документ який йому передала Мері і порівнює хеші. Якщо хеші збігаються, це підтверджує автентичність даних та їхню цілісність, оскільки будь-які зміни даних призвели б до іншого хешу. Таким чином, цифровий підпис забезпечує захист від підробки та перевірку джерела інформації.

ЕЦП є аналогом юридичної сили власноручного підпису, але з набагато вищим рівнем доказовості завдяки криптографічним властивостям. Завдяки цим властивостям, цифрові підписи стали незамінним інструментом у багатьох сферах: від електронного урядування та фінансових транзакцій до захисту інтелектуальної власності та забезпечення конфіденційності особистих даних. Вони дозволяють перетворити традиційний паперовий документообіг на ефективний та безпечний електронний формат, знижуючи ризики шахрайства та підвищуючи довіру до цифрової інформації.

1.3 Аналіз вимог до систем цифрового підпису

В цей час, де обмін важливою інформацією відбувається переважно в електронному вигляді, забезпечення її цілісності, автентичності та конфіденційності є досить важливою рисою. Цифрові підписи відіграють ключову роль у досягненні цих цілей, надаючи можливість перевіряти походження та незмінність даних. Системи цифрового підпису базуються на криптографічних алгоритмах, котрі повинні відповідати суворим вимогам, щоб гарантувати надійний захист інформації. Аналіз вимог до систем цифрового підпису передбачає визначення ключових характеристик алгоритмів, таких як контроль цілісності, автентифікація автора, невідмовність, безпека, ефективність та стандартизація.

Контроль цілісності цифрових підписів, підтверджує, що дані, підписані відправником, не були змінені, пошкоджені чи підроблені під час передачі через мережу або зберігання на носіях інформації. Для реалізації цієї властивості використовуються криптографічні хеш-функції, які перетворюють вміст документа у фіксовану послідовність символів – так званий хеш, або "цифровий відбиток". Цей хеш є унікальним для кожного набору даних: навіть найменша зміна в документі (наприклад, додавання пробілу чи зміна однієї літери) призведе до створення абсолютно іншого хешу.

Автентифікація та невідмовність у системах цифрового підпису тісно пов’язані, оскільки обидві властивості спираються на унікальність приватного ключа, відомого лише його власнику, забезпечуючи надійність і юридичну значущість електронних документів. Автентифікація дозволяє достовірно встановити особу відправника, адже цифровий підпис створюється за допомогою приватного ключа, а його перевірка відповідним публічним ключем підтверджує авторство, гарантуючи, що документ підписала саме ця особа. Невідмовність випливає з цього: оскільки лише власник приватного ключа міг створити підпис, він не може заперечити свою причетність до підписання, що робить цифровий підпис незаперечним доказом у правових і комерційних контекстах, надаючи йому юридичну силу.

Безпека є ключовою вимогою до алгоритмів електронного цифрового підпису (ЕЦП). Вони мають бути стійкими до різноманітних атак, зокрема підроблення підпису (спроба зловмисника створити дійсний підпис без приватного ключа) та атак на ключі (перебір, крадіжка). Для забезпечення цілісності використовуються стійкі хеш-функції, а для загальної безпеки – асиметричні криптосистеми, які ускладнюють обчислення для зловмисників.

Вибираючи алгоритми підпису в будь-якій сфері ефективність також важлива. Особливо системах з обмеженими обчислювальними ресурсами, таких як вбудовані системи, мобільні чи IoT-пристрої, алгоритми мають бути оптимізовані для швидкої генерації та перевірки підписів. Ефективність ми розуміємо як здатність чогось виконувати свої функції за оптимальне використання ресурсів та часу. Також це стосується розміру підпису та ключів:

так як чим менший розмір цих параметрів тим більше він сприяє на економію пропускної здатності мережі та пам’яті. Наприклад, алгоритми на основі еліптичних кривих, такі як ECDSA, EdDSA. Команда Бернштейна оптимізувала Ed25519 для сімейства процесорів x86-64 Nehalem/Westmere. Верифікація може бути виконана пакетами по 64 цифрові підписи для ще більшої пропускної здатності [3]. Ці підписи забезпечують менші розміри ключів якщо порівнювати зі традиційними алгоритмами, як-от RSA, при збереженні сильного рівня безпеки. Це їх робить супер привабливими у використанні в умовах коли обмежені ресурси. Швидкість виконання криптографічних операцій є дуже важливим показником для систем, що обробляють великі обсяги даних у реальному часі, для увиразнення, у фінансових транзакціях або аутентифікації користувачів.

Стандартизація відіграє не менш важливу роль у забезпеченні сумісності, надійності та визнання алгоритмів цифрового підпису в різних системах та територіях. Якщо притримуватись міжнародних і національних стандартів це сприяє уніфікації підходів до реалізації та використання цифрових підписів. Міжнародні стандарти, такі як ISO/IEC 14888, визначають загальні вимоги до систем цифрового підпису, включаючи методи генерації та перевірки підписів. Наприклад, ISO/IEC 14888-3:2018 описує механізми цифрового підпису, засновані на дискретному логарифмуванні У США NIST надає рекомендації щодо управління ключами, що є критично важливим для безпеки [4].

В Україні є свій стандарт ДСТУ 4145-2002 "Криптографічний захист інформації. Цифровий підпис на основі еліптичних кривих – національний стандарт, що описує механізм формування та перевірки електронного цифрового підпису, що базується на властивостях груп точок еліптичних кривих над полями $GF(2^m)$ та правилах застосування цих механізмів до повідомлень, що пересилаються каналами зв'язку та/або обробляються у комп'ютеризованих системах загального призначення [5]. Дотримання стандартів підвищує довіру до систем і спрощує їх сертифікацію та впровадження. Огляд сучасних стандартів і рекомендацій показує, що вимоги до алгоритмів цифрового підпису постійно еволюціонують з урахуванням нових загроз і технологічних досягнень.

РОЗДІЛ 2 МАТЕМАТИЧНІ МОДЕЛІ АЛГОРИТМІВ ЦИФРОВОГО ПІДПИСУ

2.1 Фундаментальні концепції цифрового підпису

Генерація ключів є першим і найважливішим кроком у будь-якій схемі цифрового підпису. Цей процес передбачає створення криптографічно пов'язаної пари ключів: приватного (секретного) та публічного (відкритого). Ці ключі генеруються одночасно за допомогою спеціалізованого математичного алгоритму. Приватний ключ зберігається в таємниці підписувачем і використовується для створення підпису, тоді як публічний ключ вільно розповсюджується і використовується для перевірки підпису.

Безпека системи цифрового підпису залежить від того, наскільки важко обчислити приватний ключ з публічного. Це ґрунтується на обчислювально складних математичних задачах, таких як факторизація великих чисел або проблема дискретного логарифму [6]. Ця концепція відома як "односторонні функції з лазівкою" (trapdoor one-way functions). Це означає, що легко обчислити публічний ключ з приватного, але обчислювально неможливо або надзвичайно складно вивести приватний ключ з публічного.

Алгоритм підпису – це центральний та ключовий етап у створенні цифрового підпису. Тут приватний ключ підписувача використовується для формування унікального криптографічного доказу автентичності повідомлення.

Основна ідея алгоритму підпису полягає в тому, що він поєднує приватний ключ підписувача з повідомленням, тобто його хешем для створення цифрового підпису. Хеш-функція тут грає важливу роль: замість того, щоб шифрувати все повідомлення, що було б неефективно для великих файлів, спочатку генерується хеш-значення. Цей хеш виступає як "цифровий відбиток" документа, що гарантує, що навіть мінімальна зміна в повідомленні призведе до зовсім іншого хешу.

Потім цей хеш шифрується за допомогою приватного ключа підписувача, і саме цей зашифрований хеш і є цифровим підписом. Таким чином, хешування є необхідною умовою для ефективного та безпечного підписання.

Конкретні хеш-функції, котрі найчастіше використовуються в цифровому підписі:

– SHA-256 (Secure Hash Algorithm 256-bit) – стандарт, задокументований у NIST FIPS PUB 180-4, генерує хеш довжиною 256 біт (32 байти). Завдяки високій криптографічній стійкості та ефективності, SHA-256 є одним із найпопулярніших виборів для цифрових підписів [7];

– SHA-3 (Secure Hash Algorithm 3), як відомо, є останнім членом сімейства SHA алгоритмів безпечного хешування, опублікованим NIST в 2015 році [8]. Алгоритм побудований за принципом губки з використанням алгоритму Keccak. Цей принцип дозволяє SHA-3 обробляти дані довільної довжини більш гнучко, а також надає додатковий рівень захисту від специфічних атак, зокрема, пов’язаних із структурою Меркл-Дамгарда [9];

– MD5 та SHA-1: хоча ці алгоритми колись активно застосовувалися, наразі вони вважаються застарілими через виявлені вразливості. SHA-1 виробляє 160-бітове хеш-значення. Як і MD5 котрий дає 128-бітове значення, він був початком систем хешування для, проте було виявлено в ньому вразливість до атак за колізіями, що призвело до того, що його криптографічна стійкість вже не відповідає сучасним вимогам [10].

В таблиці 1.1 показано різницю хеш-функцій.

Таблиця 1.1 – Порівняльна таблиця хеш-функцій та їх параметрів

Параметр	SHA-256	SHA-3 (Keccak)	SHA-1	MD5
Довжина хешу	256 біт (32 байти)	До 512 біт (SHA3-256 = 256 біт)	160 біт (20 байт)	128 біт (16 байт)

Продовження таблиці 1.1.

Параметр	SHA-256	SHA-3 (Keccak)	SHA-1	MD5
Структура	Побудований на схемі Меркл-Дамгарда	Побудований за губковою конструкцією	Меркл-Дамгард	Меркл-Дамгард
Дата публікації	2001 (в межах SHA-2)	2015	1995	1992
Криптографічна стійкість	Висока	Дуже висока	Застаріла, відомі колізії	Дуже застаріла, легко зламати
Швидкість обчислення	Швидкий	Повільніший за SHA-2, особливо на ПЗ	Швидкий	Дуже швидкий
Довжина вхідних даних	Обробляє довільну довжину	Надзвичайно гнучкий завдяки губці	Обмежена	Обмежена
Статус використання	Стандарт для цифрових підписів	Новий стандарт, зростає підтримка	Не рекомендовано	Категорично не рекомендується

Найважливіше – обрати хеш-функцію, стійку до колізій, тобто таку, щоб практично неможливо було знайти два різні повідомлення з однаковим хешем. Інакше зловмисник міг би створити підробку з тим самим цифровим підписом.

Після створення хешу документа починається підписування: до хеш-значення застосовується приватний ключ за допомогою одного з відомих алгоритмів цифрового підпису. Це одностороння криптографічна операція, яку може виконати лише власник приватного ключа.

Результатом є унікальна послідовність байтів – цифровий підпис. Він має фіксований розмір і містить усе необхідне для перевірки справжності документа.

Важливо: цифровий підпис не шифрує документ, а підтверджує його автентичність і цілісність. Підпис додається до документа або пересилається разом із ним.

Перевірка цифрового підпису – є третім і заключним етапом схеми цифрового підпису, що дозволяє одержувачу підтвердити як автентичність підпису, так і цілісність підписаного документа. Для здійснення перевірки використовується публічний ключ підписувача, оригінальний документ та отриманий цифровий підпис.

Процес перевірки виконується в два етапи. На першому етапі відбувається власне

– спочатку цифровий підпис розшифровується за допомогою публічного ключа;

– обчислюється хеш-значення отриманого документа за допомогою того ж алгоритму.

На другому етапі відбувається порівняння хешів:

– якщо обчислений хеш збігається з відновленим із цифрового підпису, це свідчить про: відсутність змін у документі після підписання;

– будь-яка невідповідність свідчить про втручання.

Довіра до публічного ключа забезпечується за допомогою Центрів Сертифікації (CA), які видають цифрові сертифікати, що безпосередньо пов’язують публічний ключ із конкретною особою або організацією після ретельної перевірки їх ідентичності [11].

Отже, завдяки ЦП можна впевнено сказати, що документ дійсно був створений певною особою і не змінювався з моменту підписання. Він виконує одразу кілька важливих функцій: перевірку цілісності даних, автентифікацію відправника і створення юридично значимого підтвердження. Це досить важливо, бо більшість операцій вже проводять у цифровому вигляді.

2.2 Алгоритми цифрового підпису та їх математична база

2.2.1 Алгоритм DSA

Алгоритм цифрового підпису (DSA) – це федеральний стандарт обробки інформації, який регулює цифрові підписи. Він використовується для забезпечення дійсності та цілісності повідомлень, програмного забезпечення або цифрових документів. Національний інститут стандартів і технологій (NIST) запропонував DSA в серпні 1991 року для використання в Стандарті цифрового підпису (DSS) [12].

Цифровий підпис за алгоритмом DSA базується на складності дискретного логарифмування, використовуючи пару ключів: приватний для генерації підпису та публічний для його перевірки. Алгоритм застосовує хеш-функцію для створення хеш-коду, який разом із випадковим числом k і приватним ключем утворює підпис із компонентами s і r. Перевірка підпису за допомогою публічного ключа повертає значення, рівне r, якщо підпис дійсний [12].

DSA є застарілим, але ще застосовується в системах із вимогами сумісності, як-от банківські транзакції чи IoT-пристрої. Через вразливість до квантових атак NIST рекомендує перейти на ECDSA або EdDSA. Вибір параметрів, формування та перевірка підпису починаються з визначення трьох загальнодоступних параметрів для групи користувачів:

– Параметр p – це велике просте число довжиною від 512 до 1024 біт з кроком 64 біти, при цьому q повинно ділити (p – 1). Воно визначає розмір групи для модульних обчислень і безпосередньо впливає на безпеку: чим більше p, тим складніше зламати систему. Простота числа p гарантує існування примітивних елементів у групі Z^*_p.

– Параметр q – це N-бітне просте число (зазвичай 160 біт), яке ділить (p – 1). Воно визначає порядок підгрупи в групі Z*p та розмір підпису і ключів. Простота q забезпечує циклічність підгрупи, а всі операції з приватними ключами та підписами виконуються по модулю q, де $2^{N-1} < q < 2^N$. Число q відіграє роль порядку підгрупи в мультиплікативній групі Z^*_p.

– Параметр g (генератор підгрупи) – Останнім визначається параметр g, який обчислюється за формулою:

$$g = h^{\frac{p-1}{q}} \, mod \, p, \tag{2.1}$$

де, h є цілим числом в діапазоні від 1 до (p – 1) з обмеженням, що g повинно бути більше 1.

Параметр g є генератором циклічної підгрупи порядку q в групі Z^*_p. Це означає, що $g^q \equiv 1 \pmod p$ і жоден менший степінь g не дорівнює 1 по модулю p. Генератор g використовується для створення відкритих ключів з приватних та для обчислення підписів. Його роль критично важлива, оскільки безпека всієї системи залежить від складності обчислення дискретного логарифму за основою g.

Далі йде етап створення ключів – встановлення приватного ключа та створення публічного:

– Приватний ключ x обирається випадково або псевдовипадково в межах від 1 до q – 1. Це секретне число, відоме лише власнику ключової пари, і використовується для створення цифрових підписів. Його випадковість є критично важливою, адже передбачуваний ключ може бути скомпрометований. Розмір x обмежується значенням q забезпечує ефективність обчислень;

– відкритий ключ обчислюється на основі приватного ключа за формулою:

$$y = g^x \, mod \, p, \tag{2.2}$$

обчислення y за відомим x є простою операцією модульного піднесення до степеня. Однак, маючи відкритий ключ y, вважається обчислювально неможливим визначити x, що являє собою дискретний логарифм y за основою g по модулю p.

Далі йде процес підписання:

– обчислюється хеш повідомлення m, використовуючи хеш-функцію, в результаті отримуємо H(m);

– обирається k – випадкове або псевдовипадкове ціле число з умовою 0 < k < q, яке має бути унікальним для кожного підписання;

– обчислюється r за формулою 2.3, як результат піднесення генератора g до степеня k по модулю p, після чого результат береться по модулю q. Це значення представляє "геометричну" частину підпису, яка залежить від випадкового параметра k та публічних параметрів системи.

$$r = (g^k \, mod \, p) \times mod \, q \tag{2.3}$$

– Обчислюється s за формулою 2.4, яка виглядає досить складно. Добуток оберненого до k елемента по модулю q на суму хеш-коду повідомлення та добутку приватного ключа на r. Цей параметр "зв'язує" підпис з конкретним повідомленням та приватним ключем підписувача.

$$s = k^{-1} \times (H(m) + x \times r) mod \, q, \tag{2.4}$$

де, k^{-1} означає мультиплікативний обернений елемент до k по модулю q, тобто число таке, що $k \times k^{-1} \equiv 1 \pmod q$.

– Обчислюється хеш повідомлення m, використовуючи хеш-функцію, в результаті отримуємо H(m);

– в результаті в нас є цифровий підпис який складається з пари (r,s). Ці два значення і є те, що передається.

На рисунку 2.1 зображено використання та обчислення компонентів DSA.

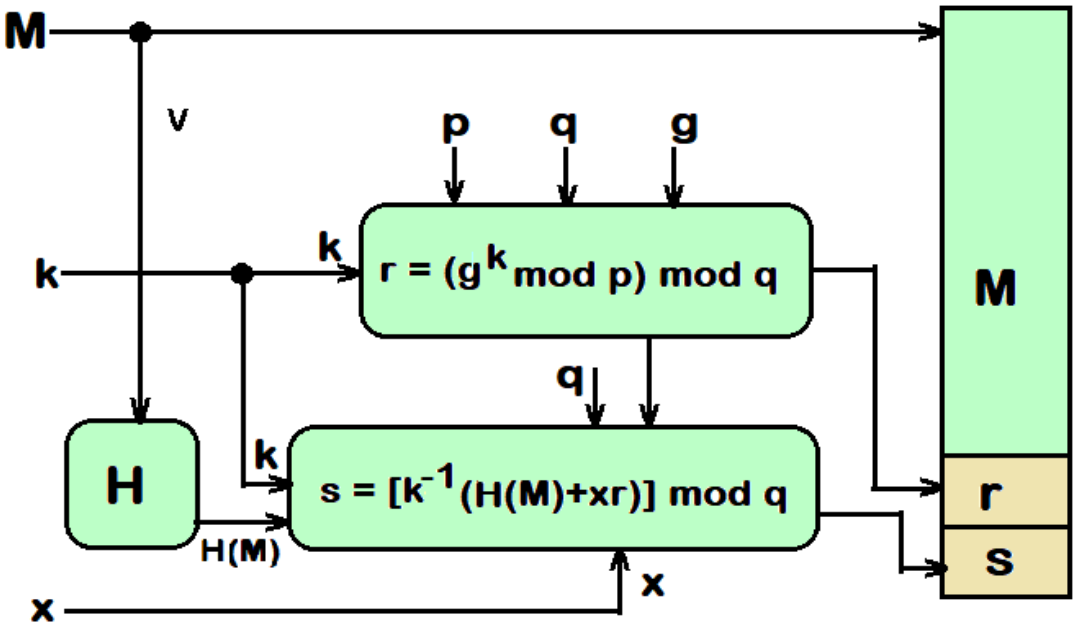

Рисунок 2.1 – Схема підписання повідомлення

І останній етап це верифікація:

– як і в процесі підписування обчислюється хеш повідомлення m, використовуючи хеш-функцію, в результаті отримуємо H(m), але в цьому процесів вже звіряються хеші для контролю цілісності;

– коли отримуються всі параметри, а саме: m, (r,s), y, (p, q, g), можна проводити процедуру верифікації;

– обчислюється w за формулою:

$$w = s^{-1} \, mod \, q, \tag{2.5}$$

де, w – це мультиплікативний обернений до s по модулю q. Він використовується для "розкриття" інформації, закодованої в підписі.

– Далі обчислюється частини експоненти u_1 та u_2 за формулами 2.6 та 2.7.

$$u_1 = (H(m) \times w) \, mod \, q \tag{2.6}$$

$$u_2 = (r \times w) \, mod \, q \tag{2.7}$$

– обчислюється V за формулою 2.8:

$$v = \left((g^{u_1} \times y^{u_2})\ mod\ p\ \right) mod\ q \tag{2.8}$$

– фінальний тест перевірки, якщо $v = r$, то підпис валідний.

Якщо умова виконується, то це означає, що підпис був створений власником приватного ключа, також він відповідає відкритому ключу y, а повідомлення m не було змінено.

На рисунку 2.2 зображено використання компонентів для верифікації.

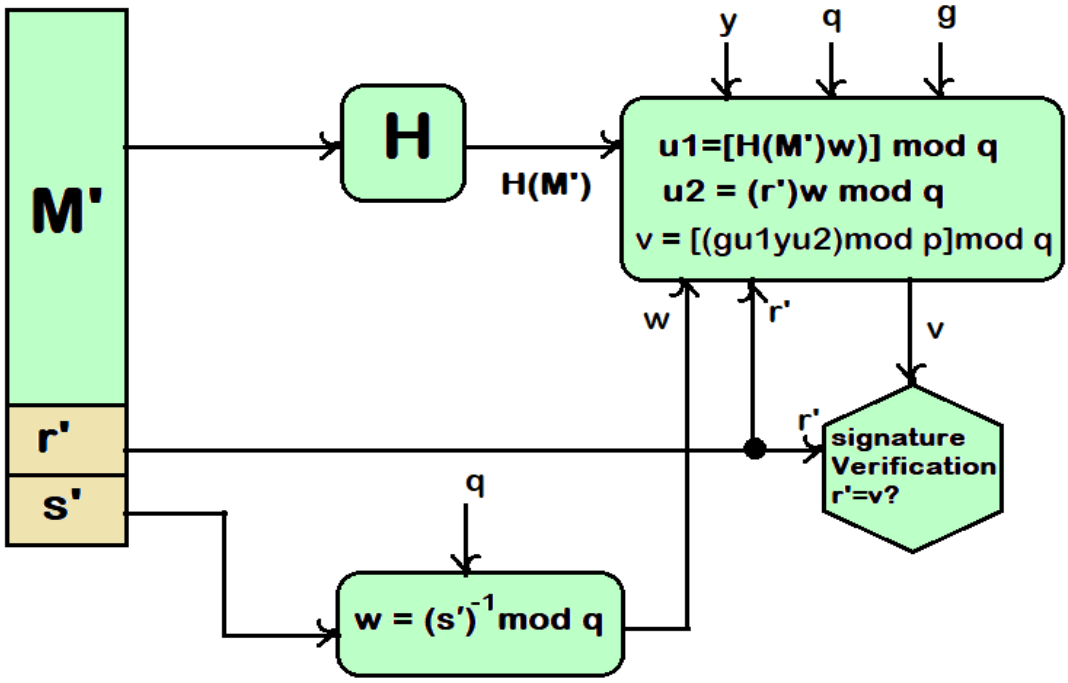

Рисунок 2.2 – Схема верифікації DSA

Отже, до основних переваг алгоритму цифрового підпису DSA відносяться:

– Компактність підпису – розмір підпису DSA значно менший порівняно з RSA при однаковому рівні безпеки;

– Ефективність, операції верифікації виконуються швидше завдяки меншому розміру чисел [13];

– Стандартизація: DSA є федеральним стандартом США (FIPS 186) та широко використовується в криптографічних системах [13];

– Гнучкість параметрів, тобто можливість вибору різних розмірів p та q залежно від потрібного рівня безпеки.

Однак важливо пам'ятати про критичність правильної генерації випадкового параметра k – його компрометація призводить до повного зламу системи.

2.2.2 Алгоритм ECDSA

Алгоритм ECDSA забезпечує автентичність і цілісність цифрових повідомлень, використовуючи криптографію еліптичних кривих (ECC) — варіант DSA з вищим рівнем безпеки при менших ключах порівняно з RSA. Його безпека базується на складності проблеми дискретного логарифму на еліптичних кривих (ECDLP), що робить обчислення k із Q і G практично неможливим [14].

ECDSA забезпечує еквівалентну або вищу стійкість порівняно з RSA (наприклад, 256-бітний ключ ECDSA дорівнює 3072-бітному RSA), зменшуючи обчислювальне навантаження, час операцій і вимоги до зберігання. Це робить його придатним для ресурсообмежених пристроїв, як-от мобільні гаджети та IoT [15].

ECDSA застосовується в протоколах SSL/TLS для захисту веб-комунікацій (HTTPS) та автентифікації серверів і клієнтів, а також у блокчейнах, зокрема Bitcoin і Ethereum, де приватні ключі підписують транзакції, гарантуючи контроль власника над активами [14].Алгоритм ECDSA базується на математиці еліптичних кривих, визначених над скінченними полями. Еліптична крива E над полем F_p (де p – просте число) зазвичай описується рівнянням Веєрштрасса:

$$y^2 = x^3\ a \times x + b\ (mod\ p), \tag{2.9}$$

де a, b $\in$ F_p і $4a^3 + 27b^2 \not\equiv 0$ (mod p). Ця умова гарантує, що крива не має особливих точок. До множини точок на кривій також додається "точка на нескінченності", яка позначається як O.

Перейдемо до етапів виконання алгоритму, так як цей алгоритм є покращенням DSA формули та етапи досить схожі. Початковим етапом є процес генерації ключів, як і у всіх алгоритмах ЦП:

– вибираються параметри еліптичної кривої: просте число p, коефіцієнти a,b, що визначають криву (див. формулу 2.9);

– вибирається базова точка G (генераторна точка) на кривій, яка генерує підгрупу простого порядку n;

– приватний ключ d_A (або як p у деяких джерелах) вибирається як випадкове ціле число в інтервалі [1, n – 1];

– публічний ключ Q_A обчислюється шляхом скалярного множення приватної точки на базову точку:

$$Q_A = d_A \times G \tag{2.10}$$

Другим етапом йде вже процес генерації підпису:

– для повідмлення m, як і в усіх алгоритмах ЦП обчислюється хеш H(m) за допомогою хеш-функції;

– генерується криптографічно безпечне випадкове число k в інтервалі [1,n−1]. Це число повинно бути унікальним для кожного підпису та залишатися секретним;

– за формулою 2.11 обчислюється випадкова точка R на кривій:

$$R = k \times G \tag{2.11}$$

– витягується x – координата точки R, яка стає першою частиною підпису:

$$\mathrm{r} = \mathrm{R.x} \pmod{\mathrm{n}} \tag{2.12}$$

Якщо r=0, процес повторюється з новим k.

– обчислюється друга частина підпису s за формулою:

$$s = k^{-1}\ (H(m) + d_A\ \times r)\ (mod\ n), \qquad (2.13)$$

де k^{-1} – модульний обернений до k за модулем n. Якщо s=0, процес повторюється з новим k.

– В результаті формується пара компонентів ЦП (r, s).

На рисунку 2.3 зображено етапи підписання.

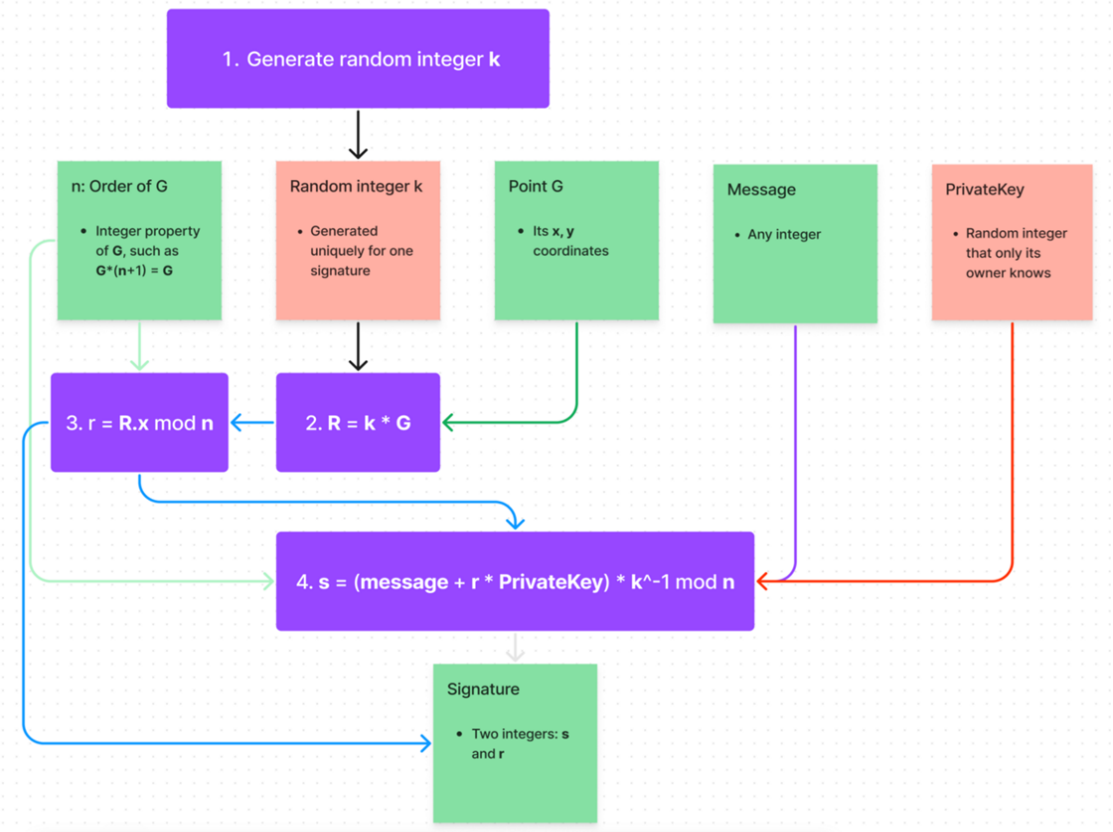

Рисунок 2.3 – Схема процесу підписання ECDSA

Перейдемо до заключного третього етапу, а саме процес верифікації. Одержувач отримує пару підпису (r,s), повідомлення m та n. Також одержувач маючи публічний ключ підписувача виконує наступні кроки:

– перевіряється, що r та s знаходяться в інтервалі [1, n−1];

– обчислюється хеш повідомлення H(m) за тією ж хеш-функцією, що використовувалася для підпису;

– обчислюється модульний обернений до s за формулою:

$$w = s^{-1}\ (mod\ n) \qquad (2.14)$$

– обчислюються два допоміжні значення u_1 та u_2 формулами відповідно:

$$u_1 = H(m) \times w \ (modn) \tag{2.15}$$

$$u_2 = r \times w \ (mod \ n) \tag{2.16}$$

– обчислюється точка на еліптичній кривій R′ за формулою 2.17:

$$R' = u_1 \times G + u_2 \times Q_A \tag{2.17}$$

– витягується x – координата точки R′ за формулою 2.18:

$$v = R'.x \ (mod \ n) \tag{2.18}$$

– підпис вважається дійсним тоді, коли задовільняється v = r.

На рисунку 2.4 зображено наглядну верифікацію.

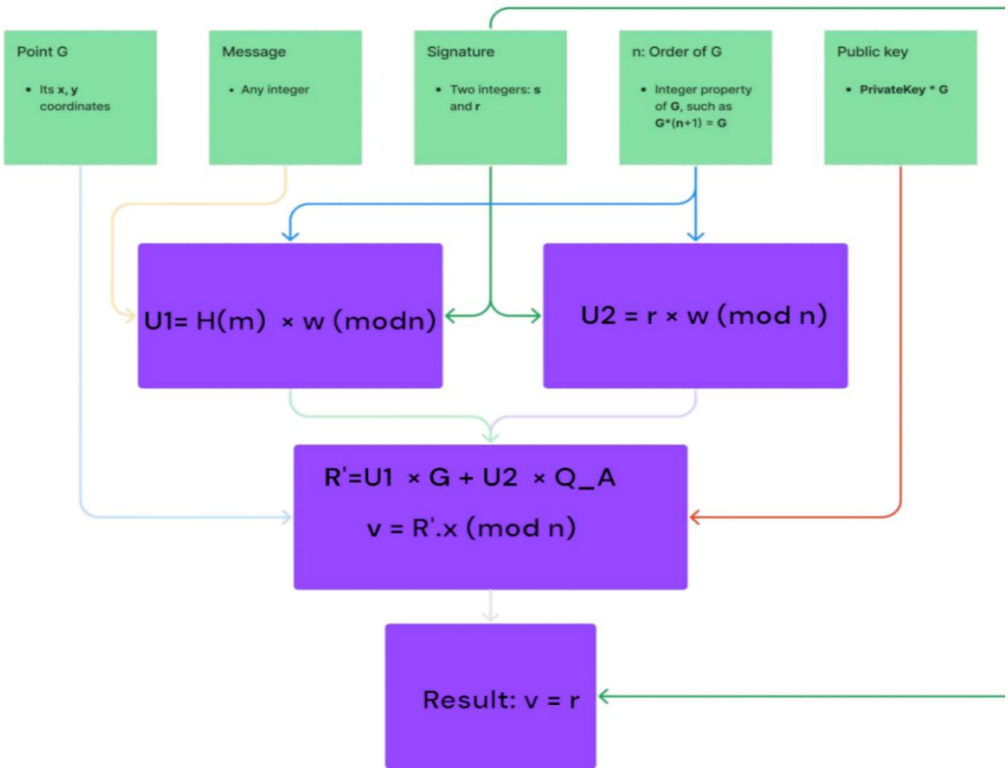

Рисунок 2.4 – Схема перевірки підпису ECDSA

ECDSA забезпечує високу безпеку з меншими ключами (наприклад, 256 біт еквівалентні 3072 біт RSA чи 1024 біт DSA), зменшуючи обчислювальне навантаження, прискорюючи підписання та верифікацію, а також знижуючи вимоги до зберігання [15]. Його стійкість базується на складності дискретного логарифму на еліптичних кривих (ECDLP), що робить атаки, як-от відновлення ключа чи підробка, нездійсненними за відомими методами [16].

Завдяки ефективності та компактності ключів ECDSA ідеальний для пристроїв із обмеженими ресурсами, таких як смарт-карти, мобільні телефони та IoT. Він широко застосовується в SSL/TLS, блокчейнах (Bitcoin, Ethereum), S/MIME та підписанні коду, будучи стандартизованим ANSI та NIST (FIPS 186) для критичних систем.

2.2.3 Алгоритм EdDSA

У криптографії з відкритим ключем алгоритм цифрового підпису на основі кривих Едвардса (EdDSA) – це схема цифрового підпису, що використовує варіант підпису Шнорра, заснований на скручених кривих Едвардса. Він розроблений, щоб бути швидшим за існуючі схеми цифрового підпису, не жертвуючи при цьому безпекою. Вона була розроблена командою в складі Даніеля Бернштейна, Нільса Дуйфа, Тані Ланге, Пітера Швабе та Бо-Ін Янга. Еталонна реалізація є загальнодоступним програмним забезпеченням [17].

Ключовою особливістю EdDSA, що відрізняє його від попередників, таких як DSA та ECDSA, є детермінована генерація одноразового числа k або (nonce) [17]. На відміну від DSA та ECDSA, які вимагають використання криптографічно стійкого випадкового числа для кожного підпису, EdDSA генерує це значення детерміновано на основі хешу повідомлення та приватного ключа [18]. Це усуває критичну вразливість, пов'язану з поганою випадковістю або повторним використанням nonce, яка могла призвести до витоку приватного ключа в DSA та ECDSA.

Безпека EdDSA, як і інших алгоритмів на еліптичних кривих, ґрунтується на обчислювальній складності проблеми дискретного логарифму на еліптичних

кривих (ECDLP). Алгоритм був стандартизований Інженерною робочою групою Інтернету (IETF) у RFC 8032 та Національним інститутом стандартів і технологій (NIST) як частина FIPS 186-5 [17].

Однією з основних сфер застосування EdDSA є захищені комунікації та програмне забезпечення. Він широко використовується в таких інструментах, як OpenSSH, GnuPG та signify tool від OpenBSD, для автентифікації та забезпечення цілісності даних. Стандартизація використання Ed25519 (конкретна реалізація EdDSA) у протоколі SSH підкреслює його надійність для віддаленого доступу та управління [17].

Крім того, EdDSA активно впроваджується в блокчейн-технологіях та криптовалютах, особливо в нових системах, таких як Solana, Polkadot та Cardano. У цих децентралізованих мережах EdDSA забезпечує безпечне підписання транзакцій, підтверджуючи їхню автентичність та цілісність без розкриття приватних ключів [18].

На відміну від ECDSA, який зазвичай використовує криві Веєрштрасса, EdDSA оперує на скручених кривих Едвардса (twisted Edwards curves), що визначаються рівнянням:

$$ax^2 + y^2 = 1 + dx^2 \times y^2 \ (mod\ p), \qquad (2.19)$$

де, a та d є параметрами кривої, а p – велике просте число. Ці криві пропонують переваги у продуктивності та стійкості до атак по сторонніх каналах завдяки їхнім "повним" формулам додавання, які дійсні для всіх точок на кривій без винятків.

Далі розглянемо перший етап, а саме генерація ключів:

– вибираються параметри еліптичної кривої: просте число q (поле), коефіцієнти a, d, що визначають скручену криву Едвардса;

– вибирається базова точка B (генераторна точка) на кривій, яка генерує підгрупу простого порядку l.

– Приватний ключ k_{priv} вибирається як випадковий b-бітний рядок;

– обчислюється хеш приватного ключа $H(k_{priv}) = (h_0, h_1, \dots h_{2b-1})$;

– з хешу $H(k_{priv})$ формується скаляр за формулою:

$$s = 2^n + \sum_{c \le i \le n} 2^i \times h_i, \tag{2.20}$$

де, n та c – параметри кривої.

– Публічний ключ A, який є точкою на кривій обчислюється шляхом скалярного множення на базову точку за формулою:

$$A = s \times B \tag{2.21}$$

Другий етап – генерація підпису, для підпису повідомлення m підписувач виконує наступні кроки:

– обчислюється хеш повідомлення $H_0(m)$ за допомогою криптографічної хеш-функції (наприклад, SHA-512 для Ed25519);

– генерується детерміноване одноразове число (nonce) r шляхом хешування об'єднання хешу приватної частини ключа та хешу повідомлення:

$$r = H(H(k_{priv})_{b..2b-1}, H_0(m)) \tag{2.22}$$

– обчислюється випадкова точка R на кривій за формулою:

$$R = r \times B \tag{2.23}$$

– обчислюється хеш-виклик e беручи хеш від значень у формулі:

$$e = H(R, A, m) \tag{2.24}$$

– обчислюється друга частина підпису S за формулою:

$$S = (r + e \times s) \ (mod\ l) \tag{2.25}$$

На рисунку 2.5 зображено візуальну схему подій при підписуванні.

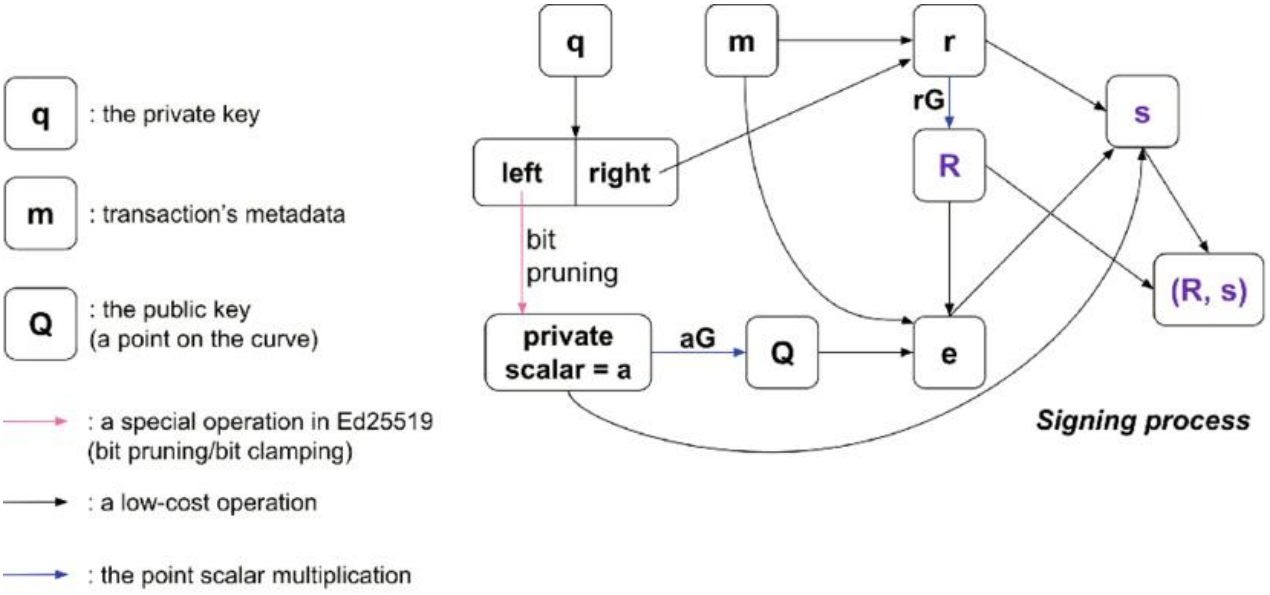

Рисунок 2.5 – Схема підписання EdDSA

І заключний етап, одержувач, щоб перевірити підпис (R,S) для повідомлення M за допомогою публічного ключа підписувача A, виконує наступні кроки:

– обчислюється хеш виклик відповідно формулі 2.24, за допомогою хеш-функції, яку використовували для підпису;

– перевіряється рівняння 2.27:

$$S \times B = R + e \times A \tag{2.26}$$

– підпис вважається валідним, якщо обидві сторони рівняння збігаються.

Принцип роботи верифікації зображено на рисунку 2.6.

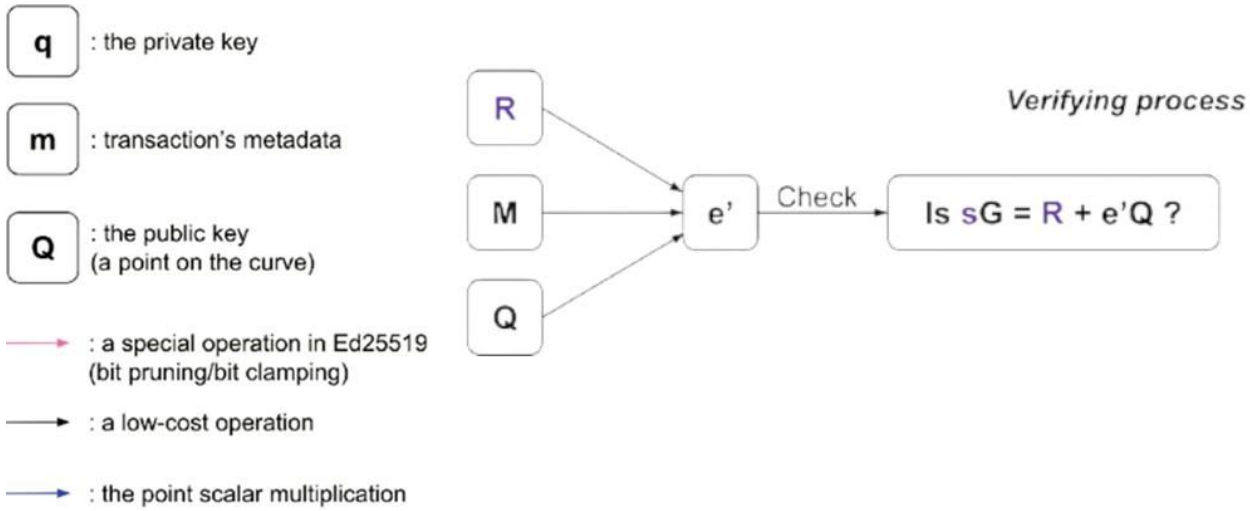

Рисунок 2.7 – Схема верифікації алгоритму EdDSA

EdDSA вирізняється надійністю завдяки детермінованій генерації nonce на основі хешу повідомлення та приватного ключа, що усуває вразливості, пов'язані з поганою випадковістю, як в ECDSA. Алгоритм забезпечує високу продуктивність на всіх етапах, використовує компактні ключі та підписи, ідеальні для IoT і мобільних пристроїв. Повні формули спрощують реалізацію, зменшуючи помилки, а дизайн захищає від атак по сторонніх каналах, роблячи EdDSA надійним і зручним рішенням для цифрового підпису.

2.2.4 Алгоритм RSA

Алгоритм RSA (Rivest-Shamir-Adleman) є наріжним каменем сучасної асиметричної криптографії, що відіграє центральну роль у забезпеченні безпеки цифрових комунікацій. Цей алгоритм, названий на честь його винахідників Рона Рівеста, Аді Шаміра та Леонарда Адлемана, був опублікований у 1977 році і швидко став першим та найбільш широко використовуваним методом криптографії з відкритим ключем [20].

Порівнюючи із минулими алгоритмами, цей має перевагу у вигляді шифрування інформації. Тобто RSA надає дві функції, а саме: шифрування повідомлень та накладання цифрового підпису.

Безпека RSA базується на складності факторизації великих чисел: легко перемножити два прості числа, але важко розкласти їх добуток. Чим більший ключ, тим складніше зламати шифр, однак зростання обчислювальних потужностей загрожує безпеці. 1024-бітні ключі RSA можуть стати вразливими, а квантові комп'ютери з алгоритмом Шора створюють серйозну загрозу, вимагаючи переходу до пост-квантової криптографії для довгострокової безпеки. Експерти вже висловлюють занепокоєння, що 1024-бітні ключі RSA можуть бути зламані в найближчому майбутньому [21]. Це вказує на постійну "гонку озброєнь" між криптографами та зловмисниками.

Генерація ключів RSA є першим і найважливішим кроком у впровадженні алгоритму, оскільки від неї залежить безпека всієї криптосистеми:

– Вибір простих великих чисел p та q, Ці прості числа повинні обиратися випадковим чином, бути великими і мати значну різницю між собою, щоб ускладнити їх факторизацію;

– обчислюється n за формулою 2.27 та функція Ейлера (φ(n)) за формулою 2.28:

$$n = p \times q \tag{2.27}$$

$$\varphi(n) = (p-1) \times (q-1) \tag{2.28}$$

– обирається ціле число e, щоб воно задовольняло умови 1 < e < φ(n) та gdc(e, φ(n)) = 1;

– визначається d, як модульним мультиплікативним оберненим e за модулем Ф(n) (див. рівність 2.29), тобто обчислюється за допомогою розширеного алгоритму Евкліда. d і є приватним ключем;

$$d \equiv e^{-1} \, (mod \, \varphi(\mathrm{n})) \tag{2.29}$$

– в результаті маємо дві пари публічний ключ (n, e) та приватний (n, d).

Етап два – процес створення підпису:

– як і в минулих алгоритмах, повідомлення m хешується і виходить H(m);

– створюється цифровий підпис за формулою 2.30:

$$s = H(m)^d \ mod \ n \tag{2.30}$$

– відправник надсилає повідомлення із підписом та публічним ключем.

Етап три – процес верифікації підпису:

– обчислення хешу повідомлення за домовленою функцією;

– так зване "дешифрування" – отриманий цифровий підпис перевіряється за допомогою публічного ключа відправника (e, n), формула 2.31:

$$H(m)' = s^e \ (mod \ n) \tag{2.31}$$

– далі порівняння – якщо обчислений хеш збігається з дешифрованим хешем з підпису, підпис вважається дійсним.

RSA пропонує високу безпеку, але це відбувається за рахунок продуктивності та більших розмірів ключів. Порівняно з ECC-алгоритмами, RSA вимагає значно довших ключів для еквівалентного рівня безпеки. Це не лише питання "швидкості", а й "ресурсів" – менші ключі означають менше пам'яті та обчислювальних циклів. Цей компроміс є центральним для вибору криптографічного алгоритму в реальних системах.

РОЗДІЛ 3 ПОРІВНЯННЯ АЛГОРИТМІВ ЦИФРОВОГО ПІДПИСУ

3.1 Вибір інструментів та середовища розробки алгоритмів

Для проведення практичного порівняльного аналізу алгоритмів цифрового підпису необхідно обґрунтовано підійти до вибору інструментів та середовища розробки. Правильний вибір технологічного стеку значно впливає на якість дослідження, достовірність результатів та можливість відтворення експерименту.

У якості основної мови програмування для реалізації алгоритмів цифрового підпису було обрано Python. Цей вибір обумовлений кількома ключовими факторами. Python характеризується високою читабельністю коду та простотою синтаксису, що дозволяє зосередитися на алгоритмічних аспектах дослідження, а не на технічних деталях реалізації. Мова має розвинену екосистему бібліотек для криптографії та математичних обчислень, що робить її ідеальним вибором для академічних досліджень у галузі інформаційної безпеки.

Python широко використовується в наукових дослідженнях завдяки своїм можливостям для швидкого прототипування та тестування алгоритмів. Інтерпретована природа мови дозволяє оперативно вносити зміни в код та проводити експерименти без необхідності компіляції. Крім того, Python має потужні засоби для візуалізації даних та створення звітів, що є важливим для аналізу результатів порівняння алгоритмів.

Для реалізації криптографічних операцій було вибрано бібліотеку cryptography – одну з найпопулярніших та найбезпечніших криптографічних бібліотек для Python. Ця бібліотека включає як рецепти високого рівня, так і інтерфейси низького рівня до загальних криптографічних алгоритмів, таких як симетричні чіпсети, дайджести повідомлень та ключові функції деривації. Наприклад, для шифрування чогось із симетричним рецептом шифрування високого рівня криптографії [22].

Бібліотека cryptography надає високорівневі API для основних криптографічних примітивів, включаючи алгоритми цифрового підпису. Вона

підтримує широкий спектр алгоритмів, зокрема RSA, DSA, ECDSA та EdDSA, що дозволяє провести комплексне порівняння різних підходів до створення цифрових підписів.

Visual Studio Code – сучасний та потужний редактор коду від Microsoft. VSCode забезпечує багатий функціонал для розробки на Python, включаючи автодоповнення коду, налагодження та лінтинг [23].

Особливо корисними для криптографічного дослідження є можливості інтерактивного налагодження Visual Studio Code. Редактор дозволяє встановлювати точки зупину, покроково виконувати код та досліджувати значення змінних на кожному етапі виконання алгоритму. Це значно спрощує процес розробки та тестування складних криптографічних операцій.

Visual Studio Code також підтримує Jupyter Notebooks через відповідне розширення, що дозволяє поєднувати код, документацію та результати візуалізації в єдиному документі. Така функціональність особливо корисна для документування процесу дослідження та представлення результатів.

3.2 Практична реалізація алгоритмів цифрового підпису та збір даних продуктивності

3.2.1 Опис реалізації програмної системи ЦП

Програмна система для цифрового підпису (ЦП), розроблена в рамках цього дослідження, є модульною платформою, призначеною для порівняльного аналізу алгоритмів цифрового підпису, таких як RSA, DSA, ECDSA та EdDSA. Основна мета системи – забезпечити гнучкість, розширюваність і зручність у використанні, дозволяючи легко інтегрувати нові алгоритми та проводити їх тестування на продуктивність і безпеку. Архітектура спроектована з урахуванням найкращих практик об’єктно-орієнтованого програмування, зокрема принципів SOLID, а також патернів проектування, таких як Factory Pattern і Dependency Injection. Це забезпечує чіткий розподіл відповідальностей між компонентами та полегшує подальший розвиток системи.

Система включає ключові модулі, що забезпечують її функціональність: основний файл digital_signature_system.py реалізує ядро з алгоритмами підпису та їхнім управлінням, модуль performance_analyzer.py аналізує продуктивність, а main.py інтегрує компоненти для практичних демонстрацій. У цьому розділі детально розглядається структура системи, її компоненти та принципи.

Модуль алгоритмів підпису в digital_signature_system.py є основою, об'єднуючи реалізації RSA, DSA, ECDSA та EdDSA через спільний інтерфейс ISignatureAlgorithm (див. лістинг 3.1), Цей інтерфейс визначає базові методи, необхідні для роботи з будь-яким алгоритмом підпису: генерацію ключів, підписання документа та перевірку підпису. Використання інтерфейсу дозволяє системі працювати з алгоритмами уніфіковано, що відповідає принципу Interface Segregation Principle (ISP).

Лістинг 3.1 – Код інтерфейсу цифрового підпису

```
class ISignatureAlgorithm(ABC):
    """Інтерфейс для алгоритмів цифрового підпису (Interface Segregation Principle)"""

    @abstractmethod
    def generate_keys(self) -> Tuple[Any, Any]:
        """Генерує пару ключів (приватний, публічний)"""
        pass

    @abstractmethod
    def sign_document(self, document: bytes, private_key: Any) -> bytes:
        """Підписує документ приватним ключем"""
        pass

    @abstractmethod
    def verify_signature(self, document: bytes, signature: bytes, public_key: Any) -> bool:
        """Перевіряє підпис документа публічним ключем"""
        pass

    @abstractmethod
    def get_algorithm_info(self) -> Dict[str, Any]:
        """Повертає інформацію про алгоритм"""
        pass
```

Далі, для кращого розуміння конкретної реалізації, на буде описано функції класу програмні реалізації одного із досліджуваних алгоритмів, а саме ECDSASignature.

Перший метод це generate_keys() функція генерує пару ключів для ECDSA: приватний і відповідний йому публічний ключ. Приватний ключ створюється на основі заданої еліптичної кривої, а публічний ключ обчислюється з приватного Код функції наведено у лістингу 3.2.

Лістинг 3.2 – Код функції для генерації ключів

```
    def generate_keys(self) -> Tuple[EllipticCurvePrivateKey,
EllipticCurvePublicKey]:
        private_key = ec.generate_private_key(self.curve)
        public_key = private_key.public_key()
        return private_key, public_key
```

Метод створює цифровий підпис для вхідного документа, використовуючи приватний ключ та алгоритм ECDSA з хеш-функцією SHA-256 (див. лістинг 3.3). Документ передається у вигляді байтового масиву.

Лістинг 3.3 – Код функції sign_document

```
    def sign_document(self, document: bytes, private_key:
EllipticCurvePrivateKey) -> bytes:
        signature = private_key.sign(document,
ec.ECDSA(hashes.SHA256()))
        return signature
```

Метод перевіряє валідність цифрового підпису для заданого документа за допомогою публічного ключа. Функція приймає параметри (документ, підпис, відкритий ключ. Використовується алгоритм ECDSA з хеш-функцією SHA-256. Якщо підпис валідний, повертається True, інакше – False. Лістинг 3.4 показує код методу.

Лістинг 3.4 – Код функції verify_signature

```
    def verify_signature(self, document: bytes, signature: bytes,
public_key: EllipticCurvePublicKey) -> bool:
        try:
            public_key.verify(signature, document,
ec.ECDSA(hashes.SHA256()))
```

Продовження Лістингу 3.4.

```
            return True
        except Exception:
            return False
```

Модуль управління підписами, реалізований конкретно у файлі digital_signature_system.py, включає класи SignatureManager і DocumentSigner. SignatureManager координує створення підписувачів для різних алгоритмів через єдиний інтерфейс, використовуючи фабрику SignatureAlgorithmFactory, логіка якої наведена на лістингу 3.5.

Лістинг 3.5 – Код класу SignatureAlgorithmFactory

```
    class SignatureAlgorithmFactory:
        """Фабрика для створення алгоритмів підпису (Factory
Pattern)"""

        @staticmethod
        def create_algorithm(algorithm: SignatureAlgorithm,
**kwargs) -> ISignatureAlgorithm:
            """Створює екземпляр алгоритму підпису"""
            if algorithm == SignatureAlgorithm.RSA:
                return
RSASignature(key_size=kwargs.get('key_size', 2048))
            elif algorithm == SignatureAlgorithm.DSA:
                return
DSASignature(key_size=kwargs.get('key_size', 2048))
            elif algorithm == SignatureAlgorithm.ECDSA:
                curve = kwargs.get('curve', ec.SECP256R1())
                return ECDSASignature(curve=curve)
            elif algorithm == SignatureAlgorithm.EDDSA:
                return EDDSASignature()
            else:
                raise ValueError(f"Непідтримуваний алгоритм:
{algorithm}")
```

На лістингу 3.5 зображено клас із патерном "Фабрика", що реалізує метод create_algorithm() для уніфікованого створення об’єктів алгоритмів підпису за типом із енумерації SignatureAlgorithm і додатковими параметрами.

Клас DocumentSigner виступає обгорткою над алгоритмом, забезпечуючи генерацію ключів, підписання документів і верифікацію. При підписанні він читає файл, передає дані алгоритму і повертає об’єкт SignatureResult , а при

верифікації – об’єкт VerificationResult із інформацією про валідність код класу зображено у Додатку А.

DocumentSigner, ключовий у системі, приймає алгоритм через інтерфейс ISignatureAlgorithm, зберігаючи посилання та ключі, ініціалізовані методом generate_keys(). Методи sign_document() і verify_document_signature() створюють і перевіряють підписи, повертаючи об’єкти SignatureResult і VerificationResult із відповідними даними. Допоміжні методи _read_document(), _get_algorithm_enum() і _get_key_size() забезпечують внутрішню логіку, гарантуючи чітке розділення обов’язків і делегування криптографічних операцій через абстрактний інтерфейс.

3.2.2 Опис реалізації модуля аналізу продуктивності

Модуль аналізу продуктивності, реалізований у файлі performance_analyzer.py, призначений для оцінки ефективності алгоритмів підпису. Він вимірює такі метрики, як час підписання, час перевірки та використання CPU. Основним класом цього модуля є PerformanceAnalyzer, який координує роботу колекторів метрик (TimeCollector, CPUCollector), кожен із яких реалізує інтерфейс IPerformanceCollector (див. лістинг 3.6).

Лістинг 3.6 – Код класу IPerformanceCollector

```
class IPerformanceCollector(ABC):
    """Інтерфейс для збирачів метрик продуктивності"""

    @abstractmethod
    def start_collection(self) -> None:
        """Розпочинає збір метрик"""
        pass

    @abstractmethod
    def stop_collection(self) -> PerformanceMetric:
        """Зупиняє збір і повертає метрику"""
        pass
```

Клас TimeCollector (див. лістинг 3.7) є реалізацією інтерфейсу IPerformanceCollector та спеціалізується на збиранні метрик часу виконання

операцій. Конструктор класу приймає тип метрики (MetricType) та назву алгоритму, ініціалізуючи внутрішні змінні для відстеження процесу вимірювання. Змінна start_time спочатку встановлюється в None, що вказує на те, що вимірювання ще не було розпочато.

Процес збирання метрики часу реалізований через два ключові методи: start_collection() та stop_collection(). Метод start_collection() фіксує початковий час за допомогою високоточного лічильника time.perf_counter(), який забезпечує найбільш точне вимірювання інтервалів часу в Python. Метод stop_collection() завершує процес вимірювання, обчислюючи різницю між кінцевим та початковим часом, при цьому включає валідацію того, що вимірювання було коректно розпочато. Результат вимірювання повертається у вигляді об'єкта PerformanceMetric, який містить структуровану інформацію про зібрану метрику.

Лістинг 3.7 – Код класу TimeCollector

```
class TimeCollector(IPerformanceCollector):
    """Збирач метрик часу виконання"""

    def __init__(self, metric_type: MetricType, algorithm:
str):
        self.metric_type = metric_type
        self.algorithm = algorithm
        self.start_time = None

    def start_collection(self) -> None:
        self.start_time = time.perf_counter()

    def stop_collection(self) -> PerformanceMetric:
        if self.start_time is None:
            raise ValueError("Збір не було розпочато")

        end_time = time.perf_counter()
        execution_time = end_time - self.start_time

        return PerformanceMetric(
            metric_type=self.metric_type,
            value=execution_time * 1000,
            unit="ms",
            timestamp=datetime.now(),
            algorithm=self.algorithm
        )
```

КласCPUCollector реалізує складну багатопоточну систему моніторингу використання CPU під час криптографічних операцій, використовуючи інтервал

(за замовчуванням 0.1 секунди) для збирання даних через psutil.cpu_percent(). Метод start_collection() запускає асинхронний потік із функцією _monitor_cpu(), яка періодично фіксує навантаження, забезпечуючи точний моніторинг. Метод stop_collection() завершує процес, обчислюючи середнє, максимальне та мінімальне значення CPU за допомогою модуля statistics, повертаючи результат у форматі PerformanceMetric із одиницею "%" для сумісності з системою. Асинхронність із TimeCollector, який використовує time.perf_counter() для точного вимірювання часу, підвищує достовірність даних.

Клас PerformanceAnalyzer є центральним компонентом аналізу продуктивності, вимірюючи швидкість підписання, верифікації та генерації ключів із заданою кількістю ітерацій для статистичної точності. Метод run_comprehensive_test() інтегрується з SignatureManager, тестуючи всі алгоритми (RSA, DSA, ECDSA, EdDSA) і формуючи об’єкти PerformanceReport із середніми, мінімальними, максимальними значеннями, медіаною та стандартним відхиленням. Клас підтримує експорт результатів у JSON, CSV (див. рисунок 3.1) для статистики та окремих CSV для ітерацій, а також виведення звітів. Повний код наведено в Додатку Б через його обсяг і складність.

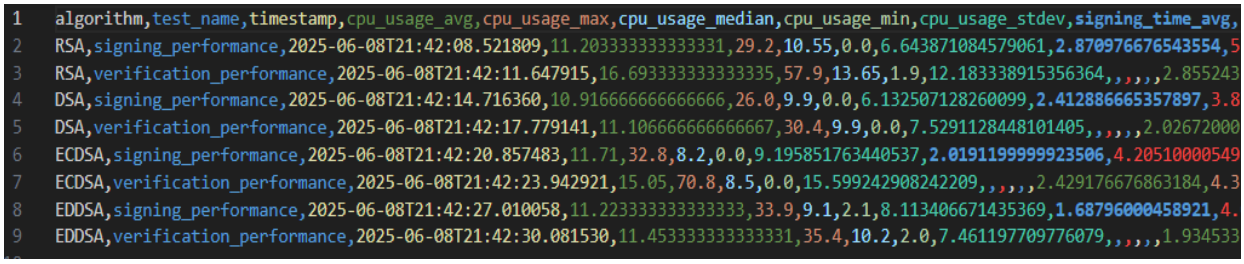

Рисунок 3.1 – Вміст CSV файлу

На рисунку 3.1 зображено статистичну інформацію включаючи середні значення, для кожного алгоритму: використання центрального процесора, час підписування та час верифікації.

3.2.3 Опис реалізації основного модуля

Головний модуль, реалізований у файлі main.py, слугує для демонстрації можливостей системи. Він включає функції для створення тестових документів, підписання їх різними алгоритмами та аналізу продуктивності, функція demo_basic_functionality показує базову роботу системи: створення документа, підписання його кожним алгоритмом і перевірку підписів.

Метод demo_basic_functionality() є демонстраційним модулем, який показує базові можливості розробленої системи цифрових підписів. Функція створює екземпляр класу SignatureManager та ініціалізує всі чотири підтримувані алгоритми підпису (RSA, DSA, ECDSA, EdDSA), демонструючи уніфікований підхід до роботи з різними криптографічними методами. Для тестування використовується файл "document_1MB.txt", що дозволяє оцінити роботу системи з документами середнього розміру.

Основний цикл функції послідовно тестує кожен алгоритм, виводячи інформацію про його параметри через метод get_algorithm_info(), створюючи цифровий підпис документа та відображаючи розмір отриманого підпису. Важливою особливістю є створення окремих файлів для кожного алгоритму з назвами типу "{algorithm}_signed_output.txt" (див. рисунок 3.2), які містять як оригінальний текст документа, так і його цифровий підпис у форматі base64. Це забезпечує наочність результатів та можливість подальшого аналізу створених підписів.

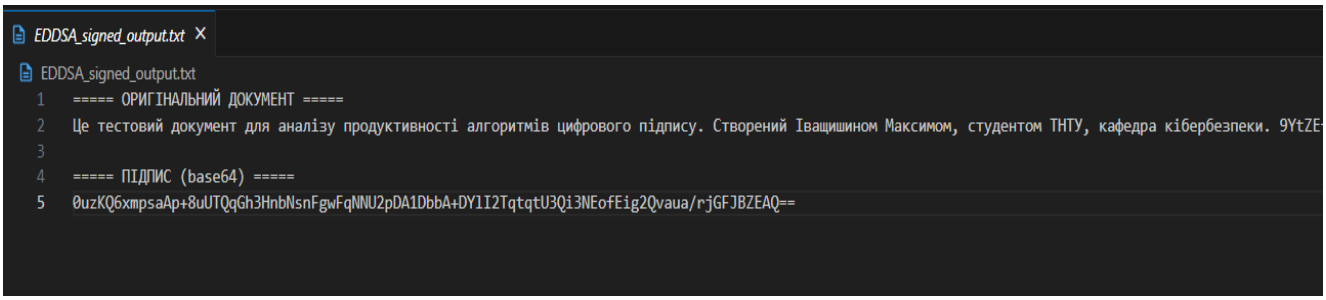

Рисунок 3.2 – Вигляд підписаного документа

Останній етап тестування кожного алгоритму включає верифікацію підпису через метод verify_document_with_algorithm(), результати якої відображаються у зручному форматі з символами ✓ та ✗ для позначення валідності.

Функція demo_performance_analysis демонструє комплексний аналіз продуктивності алгоритмів цифрового підпису (RSA, DSA, ECDSA, EdDSA), ініціалізуючи SignatureManager та PerformanceAnalyzer. Вона проводить тест на документі довільного розміру із 30 ітераціями для кожної операції, експортуючи результати у JSON і CSV файли. Код функції наведено у Додатку В.

Розроблена система цифрового підпису є гнучким і розширюваним інструментом для порівняння алгоритмів RSA, DSA, ECDSA та EdDSA завдяки модульній архітектурі та принципам SOLID. Уніфіковані інтерфейси й фабрики полегшують додавання нових алгоритмів, а модуль аналізу продуктивності дозволяє оцінити їхню ефективність, роблячи систему цінним ресурсом для досліджень і розробок у кібербезпеці.

3.3 Тестування системи та порівняння отриманих результатів

Розділ присвячений детальному аналізу результатів тестування алгоритмів цифрового підпису (RSA, DSA, ECDSA та EdDSA) на базі двох операційних систем: Windows 11 та Linux Ubuntu. Тестування проводилося з метою оцінки продуктивності алгоритмів за такими метриками: середній час підписання документа, середній час верифікації підпису, а також середнє використання центрального процесора (CPU) під час цих операцій. Експерименти виконувалися на трьох файлах різного розміру – 1KB, 1MB і 10MB – для виявлення залежності продуктивності від обсягу даних. Кожен алгоритм тестувався в 30 ітераціях, а отримані дані усереднювалися для підвищення точності результатів. Дані з тестів на Windows 11 візуалізовані за допомогою бібліотеки Matplotlib на основі CSV-файлів, що містять результати вимірювань. Спочатку розглянемо результати тестування на Windows 11, а потім порівняємо їх із даними з Linux Ubuntu.

Тестування на Windows 11 проводилося в контрольованих умовах із однаковою конфігурацією апаратного забезпечення на процесорі – Intel Core i5-8300H та програмного середовища для всіх алгоритмів. Графіки середнього часу підписання та верифікації, які продемонстровано на рисунку 3.3.

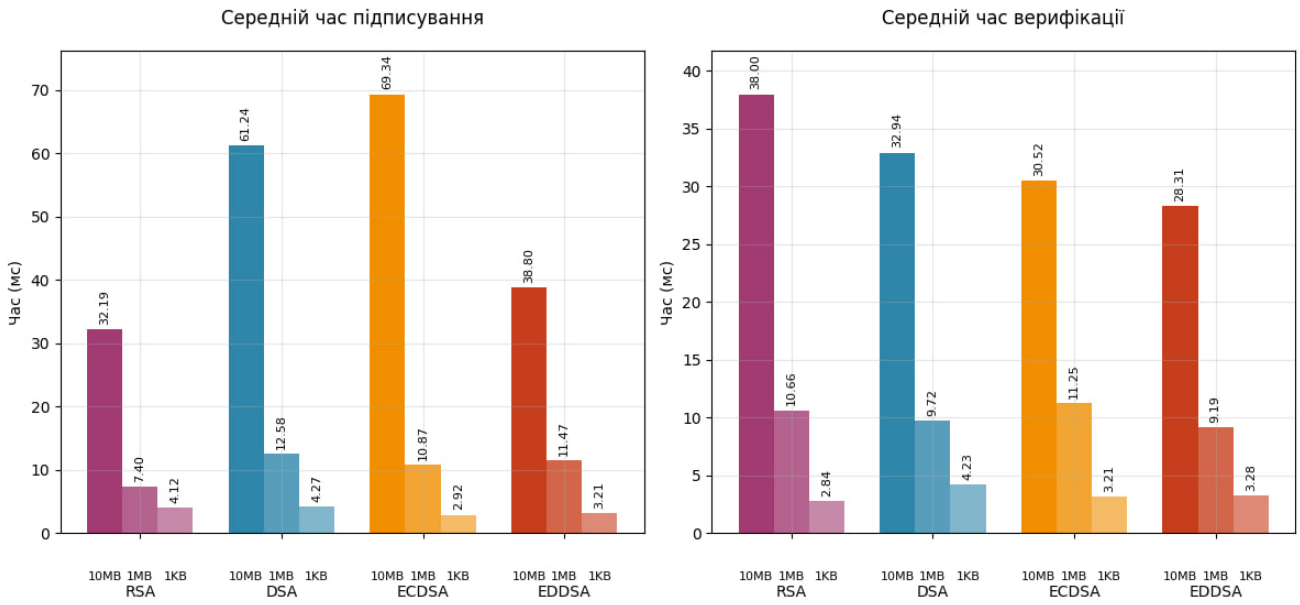

Рисунок 3.3 – Порівняння часу підписування та верифікації алгоритмів на Windows 11

На графіках чітко видно, що для всіх алгоритмів спостерігається закономірне зростання часу підписування та верифікації зі збільшенням розміру файлу. Найбільший час виконання демонструють алгоритми ECDSA та DSA, особливо на файлах розміром 10MB, де середній час підписування перевищує 60 мс. Це пояснюється особливостями їхньої криптографічної побудови використовують великі цілі числа та операції з ними, що є ресурсомісткими для процесора. Натомість RSA, який також працює з великими числами як і DSA показує досить хороші результати на файлах більшого розміру. EdDSA та ECDSA, які базуються на еліптичних кривих, показують кращу масштабованість – для малих файлів (1KB) їхній час підписування та верифікації становить лише 2.92 та 3.21 мілісекунд, що майже вдвічі швидше за класичні алгоритми.

Схожа тенденція спостерігається і для часу верифікації: для файлів 10MB усі алгоритми працюють приблизно однаково (28–38 мс), для 1MB – 9–12 мс, для

1KB – 3–4 мс. Знову ж таки, ECDSA та EDDSA мають невелику перевагу на малих файлах, але на великих розмірах різниця між алгоритмами практично зникає, хоча дивно, що RSA при верифікації найменшого файлу показав себе найкраще. Це свідчить про те, що сучасні реалізації алгоритмів цифрового підпису на Windows 11 добре оптимізовані, і при роботі з великими файлами їхня продуктивність вирівнюється.

Почнемо з підписування. Для файлу розміром 10MB найбільше навантаження на процесор створює DSA – 34,13%, що є найвищим показником серед усіх алгоритмів. RSA також споживає значний ресурс – 26,03%. Натомість ECDSA (19,81%) та EDDSA (18,27%) демонструють помітно менше навантаження, що свідчить про їхню ефективність при роботі з великими файлами. Зменшення розміру файлу до 1MB і 1KB призводить до зниження використання CPU для всіх алгоритмів, але співвідношення між ними зберігається: DSA залишається найресурсоємнішим (6,62% для 1MB і 6,13% для 1KB), а EDDSA – найекономнішим (6,10% для 1MB і 4,76% для 1KB). Для малих файлів (1KB) різниця між алгоритмами стає менш вираженою, але EDDSA та ECDSA все одно мають найнижчі показники (див. рисунок 3.4).

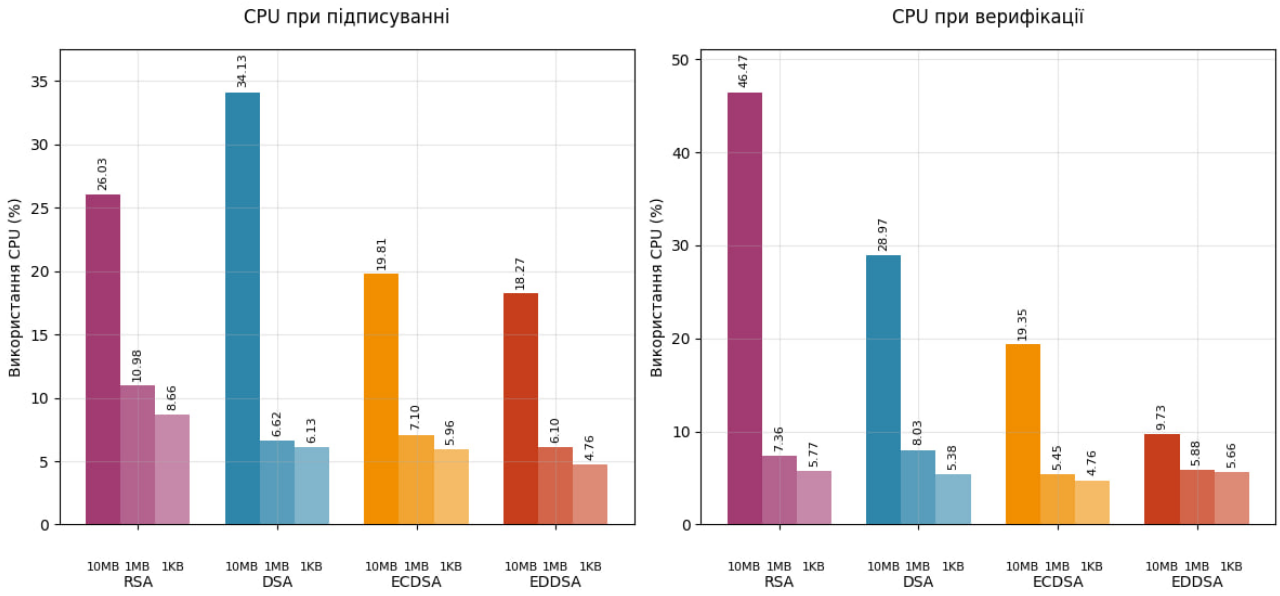

Рисунок 3.4 – Порівняння використання CPU при підписуванні та верифікації на Windows 11

Щодо верифікації, тут картина ще більш контрастна. Для файлу 10MB RSA споживає рекордні 46,47% CPU, що майже вдвічі більше, ніж у DSA (28,97%) і значно перевищує показники ECDSA (19,35%) та EDDSA (9,73%). EDDSA тут знову демонструє найкращу ефективність, споживаючи у 4-5 разів менше процесорного часу, ніж RSA. Для файлів 1MB і 1KB навантаження на CPU зменшується для всіх алгоритмів, але EDDSA і ECDSA залишаються найменш ресурсоємними (наприклад, для 1KB: EDDSA – 5,66%, ECDSA – 4,76%, тоді як RSA – 5,77%, DSA – 5,88%).

Загалом, можна зробити кілька важливих висновків. По-перше, алгоритми на основі еліптичних кривих (ECDSA, EDDSA) значно ефективніші за класичні RSA та DSA з точки зору використання процесора, особливо при роботі з великими файлами. По-друге, EDDSA стабільно показує найнижче навантаження на CPU як при підписуванні, так і при верифікації, що робить його оптимальним вибором для систем із обмеженими ресурсами або для задач, де важлива енергоефективність. По-третє, для малих файлів різниця між алгоритмами згладжується, але EDDSA та ECDSA все одно залишаються лідерами за економністю. Нарешті, RSA і особливо DSA можуть створювати суттєве навантаження на процесор при обробці великих обсягів даних, що варто враховувати при виборі алгоритму для практичних застосувань.

Далі перейдемо до результатів іншої операційної системи Linux Ubuntu. Апаратне забезпечення у вигляді процесора – Intel Core i7-9850H. На графіках, отриманих у середовищі Ubuntu, спостерігається на графіку часу підписування видно, що для великих файлів (10MB) усі алгоритми демонструють дуже схожі результати – близько 42 мс. Це свідчить про те, що при великих обсягах даних основний вплив має саме розмір вхідного файлу, а не вибір алгоритму. Для файлів 1MB час підписування зменшується до 10–12 мс, а для 1KB – до 1.7–3 мс. Найменший час підписування на малих файлах показують ECDSA та EDDSA, що підтверджує їхню ефективність для невеликих обсягів даних. RSA та DSA на малих файлах працюють трохи повільніше, але різниця не є критичною. Результати порівнянь зображено на рисунку 3.5.

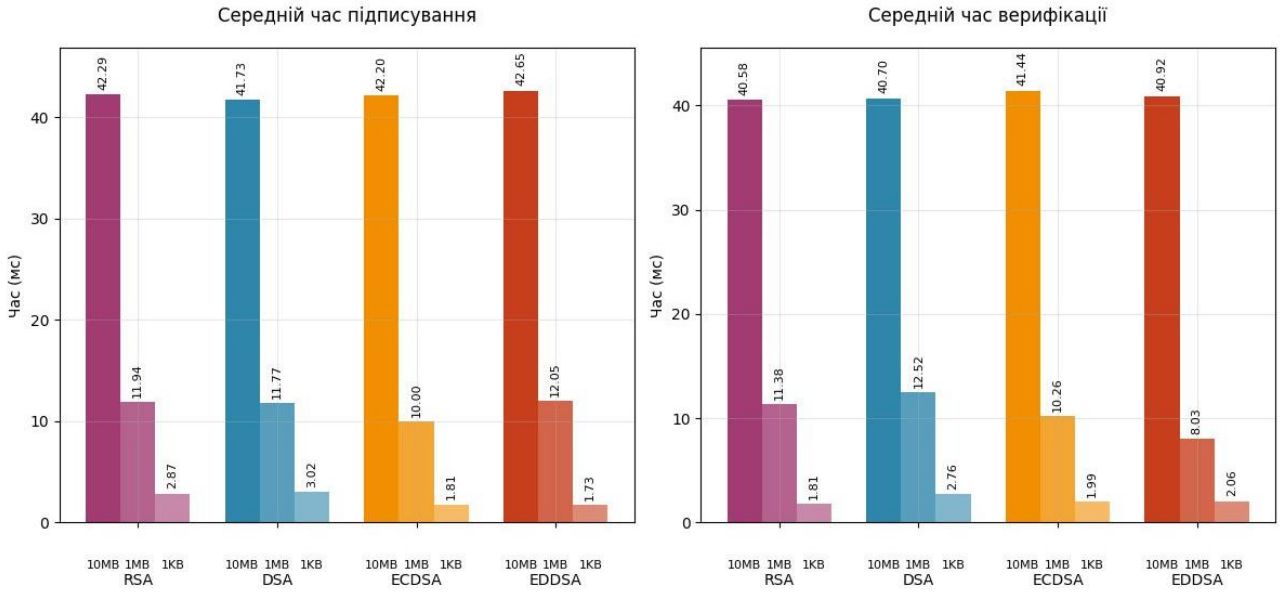

Рисунок 3.5 – Порівняння часу підписування та верифікації алгоритмів на системі Ubuntu

Схожа тенденція спостерігається і для часу верифікації: для файлів 10MB усі алгоритми працюють приблизно однаково (40–41 мс), для 1MB – 8–12 мс, для 1KB – 1.8–2.7 мс. Знову ж таки, ECDSA та EDDSA мають невелику перевагу на малих файлах перед DSA, але RSA показав найкращий результат. Це свідчить про те, що сучасні реалізації алгоритмів цифрового підпису на Linux добре оптимізовані, і при роботі з великими файлами їхня продуктивність вирівнюється.

Якщо розглядати підписування, то найбільше навантаження на CPU створює RSA, особливо для великих файлів: для 10MB цей показник становить 12,11%, тоді як для DSA – 7,81%, для ECDSA – 9,60%, а для EDDSA – 7,72%. Зменшення розміру файлу до 1MB і 1KB закономірно знижує навантаження для всіх алгоритмів, але співвідношення між ними зберігається: EDDSA та DSA залишаються найменш ресурсоємними, а RSA – найвимогливішим до ресурсів процесора. На малих файлах (1KB) різниця між алгоритмами стає мінімальною, але EDDSA та ECDSA все одно демонструють найнижчі показники.. Середні значення замірів після 30 ітерацій зображено на рисунку 3.6.

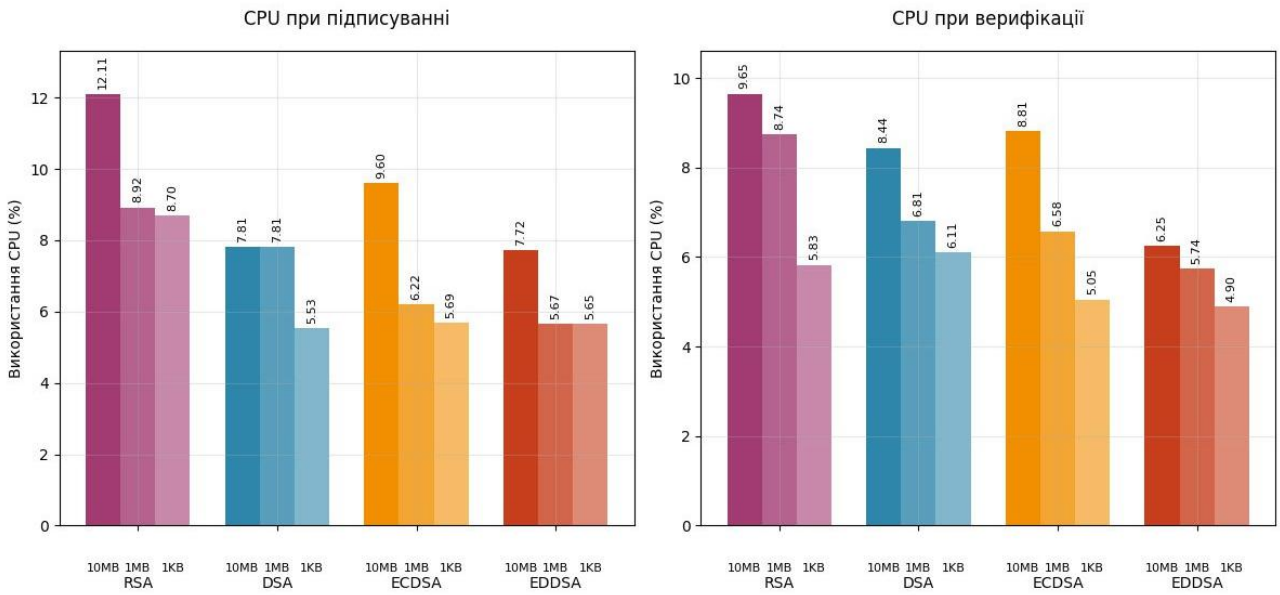

Рисунок 3.6 – Порівняння використання CPU при підписуванні та верифікації на системі Ubuntu

Під час верифікації ситуація дещо змінюється. RSA знову лідирує за використанням CPU на великих файлах (9,65% для 10MB), але різниця з іншими алгоритмами вже не така суттєва. DSA використовує 8,44%, ECDSA – 8,81%, а EDDSA – 6,25%. Зменшення розміру файлу до 1MB і 1KB призводить до ще більшого вирівнювання показників: для 1KB EDDSA споживає лише 4,90%, ECDSA – 5,05%, тоді як DSA і RSA – 6,11% та 5,83% відповідно. Це свідчить про те, що для невеликих обсягів даних сучасні алгоритми на еліптичних кривих (ECDSA, EDDSA) залишаються найефективнішими, а класичні RSA і DSA поступово втрачають свою перевагу навіть у плані швидкодії.

Порівняння результатів тестування на Windows 11 та Linux Ubuntu показало, що для обох систем характерна залежність часу підписування й верифікації від розміру файлу, а алгоритми на еліптичних кривих (ECDSA, EdDSA) стабільно демонструють кращу масштабованість і менше навантаження на процесор, особливо при роботі з невеликими файлами. Це робить їх оптимальним вибором для сучасних систем, де важлива швидкодія та енергоефективність.

Водночас, на Ubuntu різниця у використанні CPU між алгоритмами більш помітна, тоді як на Windows вона згладжується, що пов’язано з особливостями оптимізації операційних систем і бібліотеки cryptography.

РОЗДІЛ 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Розрахунок штучного освітлення

Штучне освітлення буває двох типів: загальне та комбіноване. Загальне – це освітлення, яке реалізовано шляхом розміщення світильників у верхній частині приміщень з рівномірним розподілом над робочими зонами. Такий підхід забезпечує достатню освітленість для виконання основних виробничих операцій. Комбіноване – це освітлення загальне і місцеве. Місцеве освітлення – це світильники, які встановлено безпосередньо на робочих місцях, що дозволяє збільшити освітленість у зонах виконання високоточних операцій та забезпечити можливість зміни напрямку світлового потоку відповідно до технологічних потреб.

Робоче освітлення слід передбачати для усіх приміщень будинків, а також ділянок відкритих просторів, призначених для роботи, проходу людей та руху транспорту. Для приміщень, які мають зони з різними умовами природного освітлення та різними режимами роботи, має бути передбачено окреме керування освітленням таких зон [24].

Джерелами світла є газорозрядні лампи і лампи розжарювання

Лампи розжарювання бувають: вакуумні, газонаповнені біспіральні, біспіральні з крептоно-ксеноновим наповненням, дзеркальні з дифузно-відбиваючим шаром, галогенні лампи з йодним циклом, які в колбі містять пари йоду, що підвищує температуру розжарення спіралі і запобігає розпилюванню вольфрамом нитки, збільшуючи термін служби.

Газорозрядні лампи – це лампи, в яких випромінювання оптичного діапазону спектра виникає в результаті електричного розряду в атмосфері інертних газів і парів металів, а також внаслідок явища люмінесценції. Основний недолік полягає в тому, що при розгляданні певних деталей виникає стробоскопічний ефект спотворення зорового сприйняття об'єктів, які рухаються, обертаються чи змінюються в пульсуючому світлі, що виникає при

збігові кратності частотних характеристик руху об'єктів і зміни світлового потоку в часі освітлювальних установок, що живляться змінним струмом.

Газорозрядні лампи бувають високого і низького тиску. Газорозрядні лампи низького тиску, що називають ще люмінесцентними, широко застосовують для освітлення як на виробництві, так і в побуті. Проте вони не можуть використовуватись при низьких температурах, оскільки погано запалюються і характеризуються малою одиничною потужністю Газорозрядні лампи високого тиску застосовують в умовах, коли необхідна високо світлова віддача при компактності джерел світла і стійкості до умов зовнішнього середовища. Серед них найчастіше використовують металогенні, дугові ртутні та натрієві.

Метод коефіцієнта використання – найбільш поширений для розрахунку загального рівномірного освітлення горизонтальних робочих поверхонь. У цьому методі враховуються пряма компонента світлового потоку від світильників і відбите світло від стін та стелі. Він дозволяє визначити оптимальну кількість і потужність ламп для заданої середньої освітленості приміщення.

За цим методом використання, середня освітленість Е визначається як:

$$E = \frac{\Phi \cdot N \cdot \eta \cdot k_z \cdot z}{S} \qquad (4.1)$$

де Ф – світловий потік однієї лампи (лм);

N – загальна кількість;

η – коефіцієнт використання світлового потоку;

k_z – коефіцієнт запасу;

z – коефіцієнт мінімальної освітленості;

S – площа освітлюваної поверхні (м²).

Формулу перетворюють для визначення необхідної кількості світильників.

При відомому нормативному рівні освітленості E_n маємо:

$$N = \frac{E_n \cdot S}{\Phi \cdot n \cdot k_z \cdot z} \quad (4.2)$$

Для практичного розрахунку часто спочатку розраховують величину світловий потік вибраної лампи, задаючи необхідну E_n та S, а потім отримане значення порівнюють з параметрами стандартних ламп.

Контроль освітленості здійснюється щороку з використанням люксметрів типу Ю-16, Ю-17, Ю-116, Ю-117, які працюють на принципі фотоелектричного ефекту. Прилади відрізняються діапазонами вимірювання та конструктивним виконанням, але забезпечують необхідну точність вимірювань згідно з вимогами ДСТУ 3801-98.

Заміна ламп проводиться при зниженні світлового потоку на 25% від початкового значення. Встановлено однотипні газорозрядні лампи з ідентичними спектральними та світловим потоком у межах кожної групи світильників для забезпечення рівномірності освітлення.

4.2 Заходи щодо евакуації людей із виробничих приміщень цеху

Ефективна евакуація є критично важливою для збереження життя та здоров’я працівників у разі виникнення надзвичайних ситуацій, таких як пожежі, вибухи, аварії. На виробничих об'єктах, зокрема в цехах, ризики виникнення цих ситуацій є високими через наявність тех-обладнань, горючих матеріалів. Забезпечення безпечної та своєчасної евакуації є невід'ємною частиною системи безпеки життєдіяльності, що вимагає системного підходу.

Заходи з евакуації регулюються низкою державних будівельних норм (ДБН), національних стандартів (ДСТУ), правил пожежної безпеки (НАПБ) та методичних рекомендацій, які встановлюють загальні та специфічні вимоги до проектування, експлуатації та організації евакуаційних процесів. Евакуація є основним способом захисту населення у разі загрози або виникнення

надзвичайних ситуацій природного, техногенного, соціального або воєнного характеру, внаслідок яких виникає безпосередня загроза життю та заподіяння шкоди здоров'ю людей [25]. Метою евакуації визначено організоване виведення чи вивезення населення із зони надзвичайної ситуації, якщо виникає загроза його життю або здоров'ю, якщо виникає загроза [25].

Шлях евакуації визначається як шлях до евакуаційного виходу з будь-якої точки приміщення, будівлі або споруди [26]. Евакуаційні шляхи та виходи є основними елементами системи безпеки, які безпосередньо дають змогу вижити в небезпечних ситуаціях.

Виходи вважаються евакуаційними, якщо вони ведуть із приміщень:

– першого поверху, безпосередньо назовні або через коридор, вестибюль, сходову клітку;

– будь-якого поверху, окрім першого, до коридору, що веде на сходову клітку або безпосередньо у сходову клітку. При цьому сходові клітки повинні мати вихід назовні безпосередньо, що відокремлений перегородками з дверима;

– до сусіднього приміщення на тому ж поверсі, яке забезпечене вже згаданими виходами, за винятком випадків, зазначених у будівельних нормах і правилах (СНіП).

На рисунку 4.1 зображено схему виходу з приміщень першого поверху.

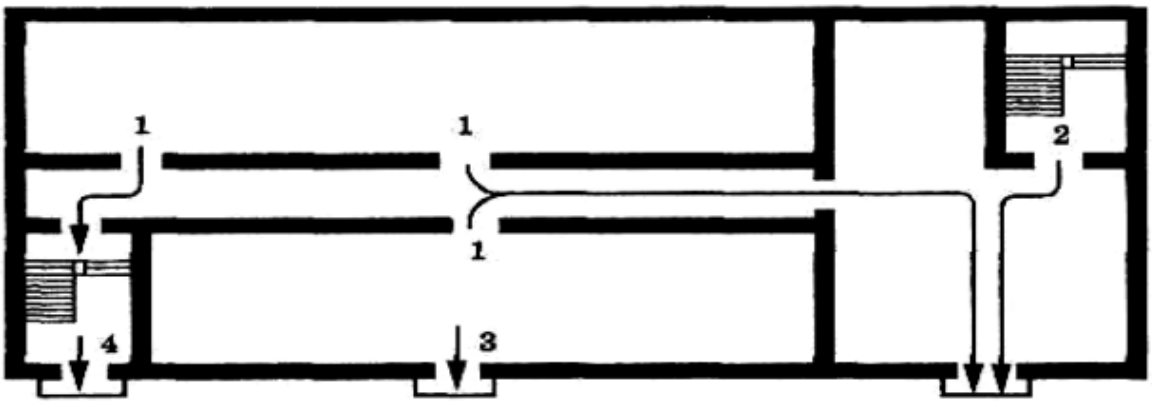

Рисунок 4.1 – Евакуаційні виходи з приміщень першого поверху

де, 1– вихід з приміщення в коридор, який веде назовні; 2 – вихід із сходової клітки крізь вестибюль назовні; 3 – вихід з приміщення назовні; 4 – вихід із сходової клітки назовні.

На рисунку 4.2 зображено схему виходу з приміщень вищих поверхів.

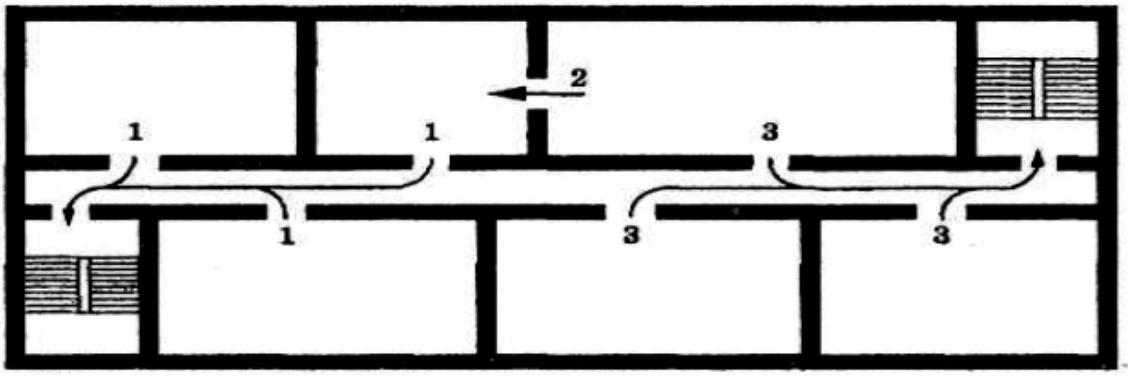

Рисунок 4.2 – Евакуаційні виходи з приміщень другого та вищих поверхів

де 1– вихід з приміщення в коридор, який веде вихід назовні; 2 – вихід у сусіднє приміщення; 3 – вихід з приміщення в коридор, який веде до сходової клітки, що має вихід через вестибюль.

Велике значення має також й освітлення сходових кліток, воно виконується природним через вікна у зовнішніх стінах. Тому улаштування так званих "темних" сходових кліток, що не мають природного освітлення, є небажаним і допускається у випадках при улаштуванні їх протидимного захисту.

Допускається у ролі другого передбачати вихід на зовнішні евакуаційні сходи (див. рисунок 4.3).

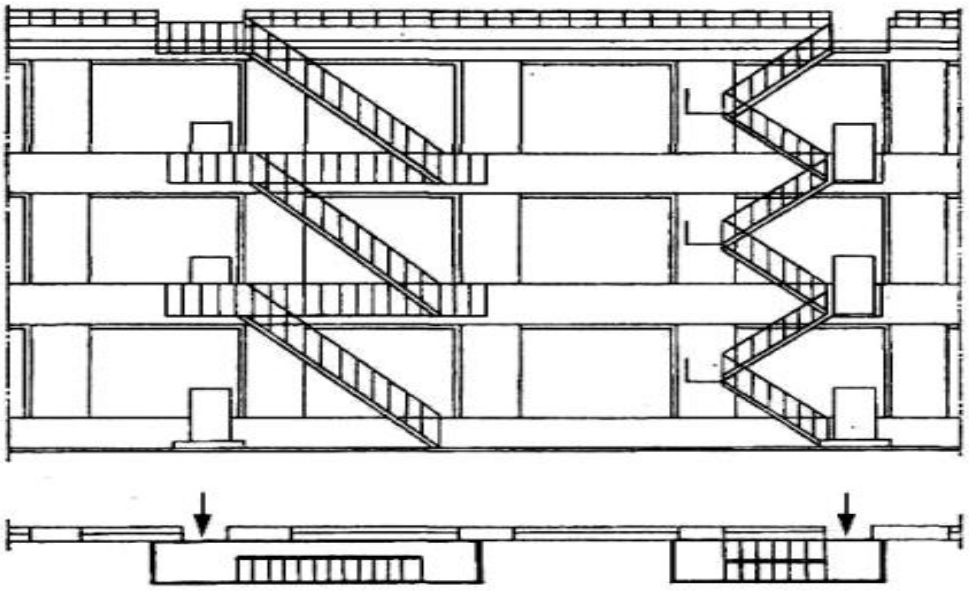

Рисунок 4.3 – Зовнішні евакуаційні сходи

До шляхів евакуації та евакуаційних виходів висуваються чіткі вимоги щодо їх геометричних параметрів. Ширина шляхів евакуації повинна бути не менше 1,0 м, а висота – не менше 2,0 м. Для проходів до одиночних робочих місць у межах одного приміщення допускається зменшення ширини до 0,7 м [26]. Ці норми розроблені з урахуванням психофізіологічних особливостей людини в умовах стресу.

Кількість евакуаційних виходів з будинку повинна бути не менша за кількість евакуаційних виходів з будь-якого його поверху. Евакуаційні виходи повинні розташовуватися розосереджено, щоб уникнути їх одночасного блокування та забезпечити альтернативні шляхи відходу. Мінімальну відстань L, м, між суміжними евакуаційними виходами з приміщення слід визначати за емпіричною формулою: $L = 1{,}5\sqrt{П}$, де П – периметр приміщення, м. Ця вимога є критично важливою для забезпечення безпеки, оскільки вона мінімізує ризик того, що всі виходи будуть недоступними в разі локальної пожежі або іншої надзвичайної події [27].

Утримання шляхів евакуації повинно відповідати суворим правилам пожежної безпеки. Категорично забороняється:

– захаращувати сходові марші, площадки, коридори та інші шляхи евакуації будь-якими матеріалами, обладнанням чи меблями;

– влаштовувати у сходових клітках приміщення будь-якого призначення (кіоски, склади);

– розташовувати в ліфтових холах приміщення різного призначення;

– зменшувати нормативну площу фрамуг закладати їх;

–прокладати в сходових клітках електропроводи та кабелі незалежно від їх напруги, крім електропроводки для освітлення звичайних сходових кліток;

– двері евакуаційних виходів з коридорів поверху, сходових кліток, вестибюлів та інші двері на шляхах евакуації не повинні мати запорів, що перешкоджають їх вільному відчиненню у разі пожежі [28].

Системи оповіщення про пожежу та управління евакуацією (СО) є активним компонентом протипожежного захисту, призначеним для своєчасного інформування людей у будівлі про пожежу з метою забезпечення їхньої

безпечної та організованої евакуації. Ефективність СО безпосередньо впливає на швидкість початку евакуації та її успішність. СО класифікуються за типом оповіщення на світлові (візуальні), звукові, мовні та комбіновані системи (див. рисунок 4.4).

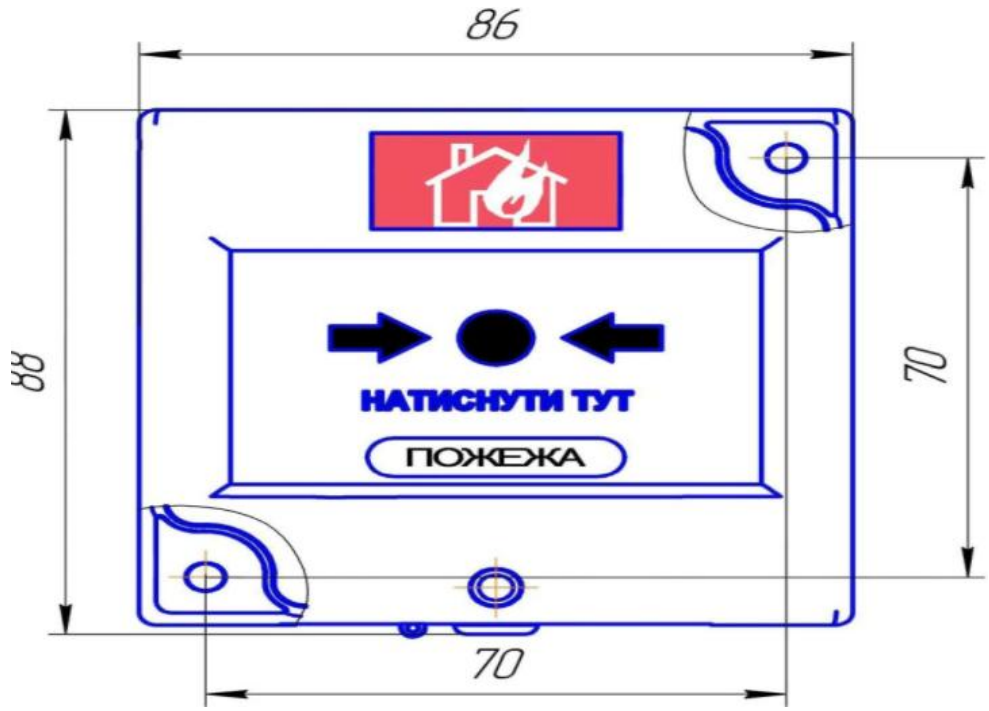

Рисунок 4.4 – Приклад кнопки пожежної сигналізації

СО можуть активуватися автоматично від систем пожежної сигналізації (СПС) або вручну персоналом з пожежного поста. Оповіщення повинно забезпечуватися у всіх приміщеннях, а за необхідності – і на прилеглій території. Голосові повідомлення повинні бути короткими, чіткими та зрозумілими, а за потреби – багатомовними.

Прийняті рішення щодо заходів евакуації у виробничих приміщеннях цеху ґрунтуються на ретельному аналізі чинних нормативно-правових актів України та інженерних розрахунках. Розрахунок часу евакуації може підтвердити, що існуючі евакуаційні шляхи та виходи, за умови їх належного утримання, здатні забезпечити своєчасне виведення персоналу з небезпечної зони.. Системи оповіщення та протидимного захисту, що відповідають ДБН В.2.5-56:2014, забезпечують своєчасне інформування та створення безпечних умов для евакуації.

ВИСНОВКИ

Представлена кваліфікаційна робота присвячена критично важливому аспекту сучасної кібербезпеки – порівняльному аналізу алгоритмів цифрового підпису (ЦП), що є ключовим для забезпечення надійності та захисту інформаційних систем. Актуальність дослідження зумовлена постійно зростаючою потребою у забезпеченні цілісності, автентичності та невідмовності цифрових даних в умовах масового електронного документообігу та обміну інформацією. Метою роботи було не лише глибоке вивчення теоретичних засад ЦП, а й проведення практичного тестування для оцінки ефективності чотирьох провідних алгоритмів: RSA, DSA, ECDSA та EdDSA.

Теоретичний аналіз включав розгляд сучасних рішень для забезпечення безпеки інформації на засадах тріади CIA (Конфіденційність, Цілісність, Доступність) та ключових вимог до систем ЦП, таких як безпека (стійкість до атак). Для кожного алгоритму було детально описано фундаментальні концепції ЦП, включаючи генерацію ключів, процес підписання та процедуру верифікації. Було надано глибокий опис алгоритмів DSA, ECDSA, EdDSA та RSA, висвітлено їхні компоненти, принципи функціонування та особливості, що впливають на безпеку, швидкодію та ефективність у системах захисту інформації.

Таким чином, дослідження підтвердило, що алгоритми на основі еліптичних кривих, особливо EdDSA та ECDSA, є більш ефективними та енергоощадними порівняно з RSA та DSA, що робить їх кращим вибором для сучасних систем. EdDSA виділяється завдяки своїй надійності та винятковій продуктивності, пропонуючи оптимальне поєднання безпеки, швидкості та компактності. Загалом, варто сказати, що EdDSA є одним із найкращих алгоритмів цифрового підпису для реального використання.

Отримані результати можуть бути використані для вдосконалення систем кібербезпеки та прийняття обґрунтованих рішень щодо вибору оптимальних алгоритмів цифрового підпису для конкретних практичних застосувань. Це дозволить підвищити рівень захисту інформації, забезпечити її цілісність та автентичність, а також оптимізувати використання обчислювальних ресурсів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. GeeksforGeeks. (2018, 15 January). What is CIA Triad? - GeeksforGeeks. https://www.geeksforgeeks.org/the-cia-triad-in-cryptography/

2. Academy, B. (2019, 19 August). What is a digital signature? | binance academy. Binance Academy. https://academy.binance.com/uk-UA/articles/what-is-a-digital-signature

3. Contributors to Wikimedia projects. (2013, 12 October). EdDSA - wikipedia. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/EdDSA

4. Iso/iec 14888-3:2018. (n. d.). ISO. https://www.iso.org/standard/76382.html

5. Учасники проектів Вікімедіа. (2010, 29 жовтень). ДСТУ 4145-2002 — вікіпедія. Вікіпедія. https://uk.wikipedia.org/wiki/ДСТУ_4145-2002

6. Cryptographic hash functions | IBM Quantum Learning. (n. d.). IBM Quantum Learning. https://learning.quantum.ibm.com/course/practical-introduction-to-quantum-safe-cryptography/cryptographic-hash-functions

7. National Institute of Standards and Technology. (2015). Secure Hash Standard (FIPS 180-4). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.180-4

8. GeeksforGeeks. (2024, 20 March). SHA-256 and SHA-3 - GeeksforGeeks. https://www.geeksforgeeks.org/sha-256-and-sha-3/

9. A Deep Dive into SHA-256 vs SHA3-256. (n. d.). MojoAuth. https://compare-hashing-algorithms.mojoauth.com/sha-256-vs-sha3-256/

10. freeCodeCamp. (2020, 26 March). MD5 vs SHA-1 vs SHA-2 - which is the most secure encryption hash and how to check them. freeCodeCamp.org. https://www.freecodecamp.org/news/md5-vs-sha-1-vs-sha-2-which-is-the-most-secure-encryption-hash-and-how-to-check-them/

11. How to verify digital signature? A step by step guide. (n. d.). Certinal | Digital Signature Solution | Digital Document Signing. https://www.certinal.com/blog/how-to-verify-digital-signature

12. GeeksforGeeks. (2024, 17 April). Digital signature algorithm (DSA) - geeksforgeeks. https://www.geeksforgeeks.org/digital-signature-algorithm-dsa/

13. Jena, B. K. (2021, 29 July). Digital Signature Algorithm (DSA) in Cryptography: A Complete Guide | Simplilearn. Simplilearn.com. https://www.simplilearn.com/tutorials/cryptography-tutorial/digital-signature-algorithm

14. ECDSA explained. (n. d.). isecjobs.com. https://isecjobs.com/insights/ecdsa-explained/

15. What is elliptic curve digital signature algorithm (ECDSA)? (n. d.). 1Kosmos. https://www.1kosmos.com/security-glossary/elliptic-curve-digital-signature-algorithm-ecdsa/

16. What Is Elliptic Curve Digital Signature Algorithm? - ECDSA. (n. d.). Blockchain Security, Smart Contract Audits, Developer Education - Cyfrin. https://www.cyfrin.io/blog/elliptic-curve-digital-signature-algorithm-and-signatures

17. Contributors to Wikimedia projects. (2013b, 12 October). EdDSA - Wikipedia. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/EdDSA#cite_note-RFC8032-1

18. Comparison of Cryptographic Algorithms. (n. d.). GpgFrontend. https://www.gpgfrontend.bktus.com/extra/algorithms-comparison/

19. Use Cases of Elliptic Curve Cryptography - Sefik Ilkin Serengil. (n. d.). Sefik Ilkin Serengil. https://sefiks.com/2023/10/12/use-cases-of-elliptic-curve-cryptography/

20. GeeksforGeeks. (2017, 22 April). RSA Algorithm in Cryptography - GeeksforGeeks. https://www.geeksforgeeks.org/rsa-algorithm-cryptography/

21. RSA Algorithm in Cryptography: Rivest Shamir Adleman Explained | Splunk. (n. d.). Splunk. https://www.splunk.com/en_us/blog/learn/rsa-algorithm-cryptography.html

22. cryptography. (n. d.). PyPI. https://pypi.org/project/cryptography/

23. Microsoft. (2021, 3 November). Python in Visual Studio Code. Visual Studio Code - Code Editing. Redefined. https://code.visualstudio.com/docs/languages/python

24. Касьянов, М., Гунченко, О., Вільсон, О., & Журавська, Н. (2016). Основи охорони праці. Дослідження та оцінка виробничого освітлення: методичні вказівки до проведення лабораторної роботи. КНУБА

25. Андрієнко, М. В., Фомін, А. І., Слуцька, О. М., Слюсар, А. А., Калиненко, Л. В., & Чайковський, Ю. М. (2022). Посібник з реалізації заходів евакуації населення, матеріальних і культурних цінностей в умовах загрози та виникнення надзвичайних ситуацій і збройних конфліктів: практичний посібник. ІДУ НД ЦЗ

26. Шляхи і правила евакуації. (б. д.). Довідник спеціаліста з охорони праці.

27. ДБН В.1.1-7:2016 "Пожежна безпека об’єктів будівництва. Загальні вимоги" №ДБН В.1.1-7:2016. (б. д.). Портал Єдиної державної електронної системи у сфері будівництва. https://e-construction.gov.ua/laws_detail/3080743763845318619

28. Про правила пожежної безпеки в Україні та пожежну безпеку об’єктів будівництва. (б. д.). Головна. https://varash-rada.gov.ua/ofitsijna-informatsiya/6291-pro-pravyla-pozhezhnoi-bezpeky-v-ukraini-ta-pozhezhnu-bezpeku-obiektiv-budivnytstva

29. ZAGORODNA, N., STADNYK, M., LYPA, B., GAVRYLOV, M., & KOZAK, R. (2022). Network Attack Detection Using Machine Learning Methods. Challenges to national defence in contemporary geopolitical situation, 2022(1), 55-61.

30. ТИМОЩУК, Д., & ЯЦКІВ, В. (2024). USING HYPERVISORS TO CREATE A CYBER POLYGON. MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES, (3), 52-56.

31. Kovalchuk, O., Karpinski, M., Banakh, S., Kasianchuk, M., Shevchuk, R., & Zagorodna, N. (2023). Prediction machine learning models on propensity convicts to criminal recidivism. Information, 14(3), art. no. 161, 1-15. doi: 10.3390/info14030161.

32. Деркач М. В., Мишко О. Є. Використання алгоритму шифрування AES-256-CBC для зберігання даних автентифікації автономного помічника. Наукові вісті Далівського університету. 2023. №24.

```
    """
    Система цифрових підписів з підтримкою DSA, ECDSA, EDDSA та
RSA алгоритмів.
    """

    from abc import ABC, abstractmethod
    from typing import Tuple, Dict, Any
    import hashlib
    import os
    from dataclasses import dataclass
    from enum import Enum

    from cryptography.hazmat.primitives import hashes,
serialization
    from cryptography.hazmat.primitives.asymmetric import rsa,
dsa, ec
    from cryptography.hazmat.primitives.asymmetric.rsa import
RSAPrivateKey, RSAPublicKey
    from cryptography.hazmat.primitives.asymmetric.dsa import
DSAPrivateKey, DSAPublicKey
    from cryptography.hazmat.primitives.asymmetric.ec import
EllipticCurvePrivateKey, EllipticCurvePublicKey
    from cryptography.hazmat.primitives.asymmetric.ed25519 import
Ed25519PrivateKey, Ed25519PublicKey
    from cryptography.hazmat.primitives.asymmetric import padding

    class SignatureAlgorithm(Enum):
        """Енумерація підтримуваних алгоритмів підпису"""
        RSA = "RSA"
        DSA = "DSA"
        ECDSA = "ECDSA"
        EDDSA = "EDDSA"

    @dataclass
    class SignatureResult:
        """Результат операції підпису"""
        signature: bytes
        algorithm: SignatureAlgorithm
        key_size: int
        signature_size: int

    @dataclass
    class VerificationResult:
        """Результат перевірки підпису"""
        is_valid: bool
        algorithm: SignatureAlgorithm

    class ISignatureAlgorithm(ABC):
        """Інтерфейс для алгоритмів цифрового підпису (Interface
Segregation Principle)"""

        @abstractmethod
```

```
        def generate_keys(self) -> Tuple[Any, Any]:
            """Генерує пару ключів (приватний, публічний)"""
            pass

        @abstractmethod
        def sign_document(self, document: bytes, private_key: Any)
-> bytes:
            """Підписує документ приватним ключем"""
            pass

        @abstractmethod
        def verify_signature(self, document: bytes, signature:
bytes, public_key: Any) -> bool:
            """Перевіряє підпис документа публічним ключем"""
            pass

        @abstractmethod
        def get_algorithm_info(self) -> Dict[str, Any]:
            """Повертає інформацію про алгоритм"""
            pass

    class RSASignature(ISignatureAlgorithm):
        """Реалізація RSA підпису"""

        def __init__(self, key_size: int = 2048):
            self.key_size = key_size

        def generate_keys(self) -> Tuple[RSAPrivateKey,
RSAPublicKey]:
            private_key = rsa.generate_private_key(
                public_exponent=65537,
                key_size=self.key_size
            )
            public_key = private_key.public_key()
            return private_key, public_key

        def sign_document(self, document: bytes, private_key:
RSAPrivateKey) -> bytes:
            signature = private_key.sign(
                document,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
            return signature

        def verify_signature(self, document: bytes, signature:
bytes, public_key: RSAPublicKey) -> bool:
            try:
                public_key.verify(
                    signature,
                    document,
                    padding.PSS(
```

```
                        mgf=padding.MGF1(hashes.SHA256()),
                        salt_length=padding.PSS.MAX_LENGTH
                    ),
                    hashes.SHA256()
                )
                return True
            except Exception:
                return False

        def get_algorithm_info(self) -> Dict[str, Any]:
            return {
                "name": "RSA",
                "key_size": self.key_size,
                "hash_algorithm": "SHA-256",
                "padding": "PSS"
            }

    class DSASignature(ISignatureAlgorithm):
        """Реалізація DSA підпису"""

        def __init__(self, key_size: int = 2048):
            self.key_size = key_size

        def generate_keys(self) -> Tuple[DSAPrivateKey,
DSAPublicKey]:
            private_key =
dsa.generate_private_key(key_size=self.key_size)
            public_key = private_key.public_key()
            return private_key, public_key

        def sign_document(self, document: bytes, private_key:
DSAPrivateKey) -> bytes:
            signature = private_key.sign(document,
hashes.SHA256())
            return signature

        def verify_signature(self, document: bytes, signature:
bytes, public_key: DSAPublicKey) -> bool:
            try:
                public_key.verify(signature, document,
hashes.SHA256())
                return True
            except Exception:
                return False

        def get_algorithm_info(self) -> Dict[str, Any]:
            return {
                "name": "DSA",
                "key_size": self.key_size,
                "hash_algorithm": "SHA-256"
            }

    class ECDSASignature(ISignatureAlgorithm):
        """Реалізація ECDSA підпису"""
```

```
        def __init__(self, curve=ec.SECP256R1()):
            self.curve = curve

        def generate_keys(self) -> Tuple[EllipticCurvePrivateKey,
EllipticCurvePublicKey]:
            private_key = ec.generate_private_key(self.curve)
            public_key = private_key.public_key()
            return private_key, public_key

        def sign_document(self, document: bytes, private_key:
EllipticCurvePrivateKey) -> bytes:
            signature = private_key.sign(document,
ec.ECDSA(hashes.SHA256()))
            return signature

        def verify_signature(self, document: bytes, signature:
bytes, public_key: EllipticCurvePublicKey) -> bool:
            try:
                public_key.verify(signature, document,
ec.ECDSA(hashes.SHA256()))
                return True
            except Exception:
                return False

        def get_algorithm_info(self) -> Dict[str, Any]:
            return {
                "name": "ECDSA",
                "curve": self.curve.name,
                "hash_algorithm": "SHA-256"
            }
    class EDDSASignature(ISignatureAlgorithm):
        """Реалізація Ed25519 (EDDSA) підпису"""

        def __init__(self):
            self._private_key = None
            self._public_key = None
            self.generate_keys()  # Генеруємо ключі відразу при
створенні

        def generate_keys(self) -> Tuple[Ed25519PrivateKey,
Ed25519PublicKey]:
            """Генерує пару ключів"""
            self._private_key = Ed25519PrivateKey.generate()
            self._public_key = self._private_key.public_key()
            return self._private_key, self._public_key

        def sign_document(self, document: str | bytes,
private_key: Ed25519PrivateKey = None) -> bytes:
            """Підписує документ"""
            # Читаємо файл якщо отримали шлях
            if isinstance(document, str):
                with open(document, 'rb') as f:
                    document = f.read()

            # Використовуємо наявний або переданий ключ
```

```
            key = private_key or self._private_key
            if not key:
                raise ValueError("Не вказано приватний ключ")

            return key.sign(document)

        def verify_signature(self, document: str | bytes,
signature: bytes,
                             public_key: Ed25519PublicKey = None) -
> bool:
            """Перевіряє підпис"""
            # Читаємо файл якщо отримали шлях
            if isinstance(document, str):
                with open(document, 'rb') as f:
                    document = f.read()

            # Використовуємо наявний або переданий ключ
            key = public_key or self._public_key
            if not key:
                raise ValueError("Не вказано публічний ключ")

            try:
                key.verify(signature, document)
                return True
            except Exception:
                return False

        def get_algorithm_info(self) -> Dict[str, Any]:
            return {
                "name": "Ed25519 (EDDSA)",
                "key_size": 256,
                "curve": "Curve25519",
                "security_level": "128-bit"
            }

    class SignatureAlgorithmFactory:
        """Фабрика для створення алгоритмів підпису (Factory
Pattern)"""

        @staticmethod
        def create_algorithm(algorithm: SignatureAlgorithm,
**kwargs) -> ISignatureAlgorithm:
            """Створює екземпляр алгоритму підпису"""
            if algorithm == SignatureAlgorithm.RSA:
                return
RSASignature(key_size=kwargs.get('key_size', 2048))
            elif algorithm == SignatureAlgorithm.DSA:
                return
DSASignature(key_size=kwargs.get('key_size', 2048))
            elif algorithm == SignatureAlgorithm.ECDSA:
                curve = kwargs.get('curve', ec.SECP256R1())
                return ECDSASignature(curve=curve)
            elif algorithm == SignatureAlgorithm.EDDSA:
                return EDDSASignature()
            else:
```

```
                raise ValueError(f"Непідтримуваний алгоритм:
{algorithm}")

    class DocumentSigner:
        """Основний клас для підпису документів (Single
Responsibility Principle)"""

        def __init__(self, algorithm: ISignatureAlgorithm):
            self.algorithm = algorithm
            self.private_key = None
            self.public_key = None

        def generate_keys(self) -> None:
            """Генерує нову пару ключів"""
            self.private_key, self.public_key =
self.algorithm.generate_keys()

        def sign_document(self, document_path: str) ->
SignatureResult:
            """Підписує документ з файлу"""
            if not self.private_key:
                raise ValueError("Приватний ключ не згенерований")

            document_data = self._read_document(document_path)
            signature =
self.algorithm.sign_document(document_data, self.private_key)

            return SignatureResult(
                signature=signature,
                algorithm=self._get_algorithm_enum(),
                key_size=self._get_key_size(),
                signature_size=len(signature)
            )

        def verify_document_signature(self, document_path: str,
signature: bytes) -> VerificationResult:
            """Перевіряє підпис документа"""
            if not self.public_key:
                raise ValueError("Публічний ключ не згенерований")

            document_data = self._read_document(document_path)
            is_valid =
self.algorithm.verify_signature(document_data, signature,
self.public_key)

            return VerificationResult(
                is_valid=is_valid,
                algorithm=self._get_algorithm_enum()
            )

        def get_algorithm_info(self) -> Dict[str, Any]:
            """Повертає інформацію про алгоритм"""
            return self.algorithm.get_algorithm_info()

        def _read_document(self, document_path: str) -> bytes:
```

```
            """Читає документ з файлу"""
            try:
                with open(document_path, 'rb') as file:
                    return file.read()
            except FileNotFoundError:
                raise FileNotFoundError(f"Документ не знайдено:
{document_path}")

        def _get_algorithm_enum(self) -> SignatureAlgorithm:
            """Повертає енум алгоритму на основі класу"""
            class_name = self.algorithm.__class__.__name__
            if "RSA" in class_name:
                return SignatureAlgorithm.RSA
            elif "DSA" in class_name and "EC" not in class_name
and "ED" not in class_name:
                return SignatureAlgorithm.DSA
            elif "ECDSA" in class_name:
                return SignatureAlgorithm.ECDSA
            elif "EDDSA" in class_name or "Ed25519" in class_name:
                return SignatureAlgorithm.EDDSA
            else:
                raise ValueError(f"Невідомий алгоритм:
{class_name}")


        def _get_key_size(self) -> int:
            """Повертає розмір ключа"""
            info = self.algorithm.get_algorithm_info()
            return info.get('key_size', 0)

    class SignatureManager:
        """Менеджер для роботи з різними алгоритмами підпису
(Open/Closed Principle)"""

        def __init__(self):
            self.signers = {}

        def add_signer(self, algorithm: SignatureAlgorithm,
**kwargs) -> None:
            """Додає новий підписувач"""
            signature_algorithm =
SignatureAlgorithmFactory.create_algorithm(algorithm, **kwargs)
            signer = DocumentSigner(signature_algorithm)
            signer.generate_keys()
            self.signers[algorithm] = signer

        def sign_document_with_algorithm(self, document_path: str,
algorithm: SignatureAlgorithm) -> SignatureResult:
            """Підписує документ вказаним алгоритмом"""
            if algorithm not in self.signers:
                raise ValueError(f"Алгоритм {algorithm} не
додано")

            return
self.signers[algorithm].sign_document(document_path)
```

```
        def verify_document_with_algorithm(self, document_path:
str, signature: bytes,
                                           algorithm:
SignatureAlgorithm) -> VerificationResult:
            """Перевіряє підпис документа вказаним алгоритмом"""
            if algorithm not in self.signers:
                raise ValueError(f"Алгоритм {algorithm} не
додано")

            return
self.signers[algorithm].verify_document_signature(document_path,
signature)

        def get_all_algorithms_info(self) ->
Dict[SignatureAlgorithm, Dict[str, Any]]:
            """Повертає інформацію про всі додані алгоритми"""
            return {alg: signer.get_algorithm_info() for alg,
signer in self.signers.items()}


    def main():
        """Демонстрація використання системи цифрових підписів"""

        # Створення тестового документа
        test_document = "document.txt"

        # Ініціалізація менеджера підписів
        manager = SignatureManager()

        # Додавання всіх алгоритмів
        algorithms_to_test = [
            SignatureAlgorithm.RSA,
            SignatureAlgorithm.DSA,
            SignatureAlgorithm.ECDSA,
            SignatureAlgorithm.EDDSA
        ]

        for algorithm in algorithms_to_test:
            manager.add_signer(algorithm)

        # Тестування кожного алгоритму
        for algorithm in algorithms_to_test:
            print(f"\n=== Тестування {algorithm.value} ===")

            # Отримання інформації про алгоритм
            info = manager.signers[algorithm].get_algorithm_info()
            print(f"Інформація про алгоритм: {info}")

            # Підпис документа
            signature_result =
manager.sign_document_with_algorithm(test_document, algorithm)
            print(f"Підпис створено. Розмір підпису:
{signature_result.signature_size} байт")
```

```
            # Перевірка підпису
            verification_result =
manager.verify_document_with_algorithm(
                test_document, signature_result.signature,
algorithm
            )
            print(f"Результат перевірки: {'✓ Валідний' if
verification_result.is_valid else '✗ Невалідний'}")

        # Очищення тестового файлу
        os.remove(test_document)
        print(f"\nТестування завершено!")

    if __name__ == "__main__":
        main()
```

```
    """
    Модуль для вимірювання продуктивності алгоритмів цифрового
підпису.
    Вимірює час підпису, час перевірки, використання пам'яті та
CPU.
    """

    import time
    import psutil
    import threading
    import statistics
    from abc import ABC, abstractmethod
    from typing import Dict, List, Any, Callable, Optional, Tuple
    from dataclasses import dataclass, field
    from datetime import datetime
    import json
    import csv
    import os
    from contextlib import contextmanager

    # Імпортуємо з нашого основного модуля
    from digital_signature_system import (
        SignatureManager, SignatureAlgorithm,
        SignatureAlgorithmFactory, RSASignature, DSASignature,
        ECDSASignature, EDDSASignature
    )

    from enum import Enum

    class MetricType(Enum):
        """Типи метрик для вимірювання"""
        SIGNING_TIME = "signing_time"
        VERIFICATION_TIME = "verification_time"
        MEMORY_USAGE = "memory_usage"
        CPU_USAGE = "cpu_usage"
        KEY_GENERATION_TIME = "key_generation_time"

    @dataclass
    class PerformanceMetric:
        """Метрика продуктивності"""
        metric_type: MetricType
        value: float
        unit: str
        timestamp: datetime
        algorithm: str
        additional_info: Dict[str, Any] =
field(default_factory=dict)

    @dataclass
    class PerformanceReport:
        """Звіт про продуктивність"""
        algorithm: str
        test_name: str
        metrics: List[PerformanceMetric]
```

```
        statistics: Dict[str, float]
        test_parameters: Dict[str, Any]
        timestamp: datetime

    class IPerformanceCollector(ABC):
        """Інтерфейс для збирачів метрик продуктивності"""

        @abstractmethod
        def start_collection(self) -> None:
            """Розпочинає збір метрик"""
            pass

        @abstractmethod
        def stop_collection(self) -> PerformanceMetric:
            """Зупиняє збір і повертає метрику"""
            pass

    class TimeCollector(IPerformanceCollector):
        """Збирач метрик часу виконання"""

        def __init__(self, metric_type: MetricType, algorithm:
str):
            self.metric_type = metric_type
            self.algorithm = algorithm
            self.start_time = None

        def start_collection(self) -> None:
            self.start_time = time.perf_counter()

        def stop_collection(self) -> PerformanceMetric:
            if self.start_time is None:
                raise ValueError("Збір не було розпочато")

            end_time = time.perf_counter()
            execution_time = end_time - self.start_time

            return PerformanceMetric(
                metric_type=self.metric_type,
                value=execution_time * 1000,
                unit="ms",
                timestamp=datetime.now(),
                algorithm=self.algorithm
            )

    class MemoryCollector(IPerformanceCollector):
        """Збирач метрик використання пам'яті"""

        def __init__(self, algorithm: str):
            self.algorithm = algorithm
            self.process = psutil.Process()
            self.initial_memory = None
            self.peak_memory = None
            self.monitoring = False
            self.monitor_thread = None
```

```
        def start_collection(self) -> None:
            self.initial_memory = self.process.memory_info().rss
            self.peak_memory = self.initial_memory
            self.monitoring = True
            self.monitor_thread =
threading.Thread(target=self._monitor_memory)
            self.monitor_thread.start()

        def stop_collection(self) -> PerformanceMetric:
            self.monitoring = False
            if self.monitor_thread:
                self.monitor_thread.join()

            memory_diff = self.peak_memory - self.initial_memory

            return PerformanceMetric(
                metric_type=MetricType.MEMORY_USAGE,
                value=memory_diff / 1024 / 1024,
                unit="MB",
                timestamp=datetime.now(),
                algorithm=self.algorithm,
                additional_info={
                    "initial_memory_mb": self.initial_memory /
1024 / 1024,
                    "peak_memory_mb": self.peak_memory / 1024 /
1024
                }
            )

        def _monitor_memory(self) -> None:
            """Моніторить використання пам'яті в окремому
потоці"""
            while self.monitoring:
                current_memory = self.process.memory_info().rss
                if current_memory > self.peak_memory:
                    self.peak_memory = current_memory
                time.sleep(0.001)

    class CPUCollector(IPerformanceCollector):
        """Збирач метрик використання CPU"""

        def __init__(self, algorithm: str, interval: float = 0.1):
            self.algorithm = algorithm
            self.interval = interval
            self.cpu_readings = []
            self.monitoring = False
            self.monitor_thread = None

        def start_collection(self) -> None:
            self.cpu_readings = []
            self.monitoring = True
            # Перший виклик для ініціалізації
            psutil.cpu_percent(interval=None)
            self.monitor_thread =
threading.Thread(target=self._monitor_cpu)
```

```
            self.monitor_thread.start()

        def stop_collection(self) -> PerformanceMetric:
            self.monitoring = False
            if self.monitor_thread:
                self.monitor_thread.join()

            avg_cpu = statistics.mean(self.cpu_readings) if
self.cpu_readings else 0

            return PerformanceMetric(
                metric_type=MetricType.CPU_USAGE,
                value=avg_cpu,
                unit="%",
                timestamp=datetime.now(),
                algorithm=self.algorithm,
                additional_info={
                    "max_cpu": max(self.cpu_readings) if
self.cpu_readings else 0,
                    "min_cpu": min(self.cpu_readings) if
self.cpu_readings else 0,
                    "readings_count": len(self.cpu_readings)
                }
            )

        def _monitor_cpu(self) -> None:
            """Моніторить використання CPU в окремому потоці"""
            while self.monitoring:
                cpu_percent =
psutil.cpu_percent(interval=self.interval)
                self.cpu_readings.append(cpu_percent)

    class PerformanceAnalyzer:
        """Основний клас для аналізу продуктивності алгоритмів
підпису"""

        def __init__(self):
            self.results: List[PerformanceReport] = []



        def measure_signing_performance(self, signer,
document_path: str,
                                        iterations: int = 10) ->
PerformanceReport:
            """Вимірює продуктивність підпису документа"""
            algorithm_name = signer._get_algorithm_enum().value
            all_metrics = []



            print(f"Вимірювання продуктивності підпису для
{algorithm_name}...")

            for i in range(iterations):
```

```
                # Створюємо колектори для кожної ітерації
                time_collector =
TimeCollector(MetricType.SIGNING_TIME, algorithm_name)
                memory_collector = MemoryCollector(algorithm_name)
                cpu_collector = CPUCollector(algorithm_name)

                # Запускаємо збір метрик
                time_collector.start_collection()
                memory_collector.start_collection()
                cpu_collector.start_collection()

                # Виконуємо операцію підпису
                result = signer.sign_document(document_path)

                # Зупиняємо збір і отримуємо метрики
                time_metric = time_collector.stop_collection()
                memory_metric = memory_collector.stop_collection()
                cpu_metric = cpu_collector.stop_collection()

                all_metrics.extend([time_metric, memory_metric,
cpu_metric])

                print(f"  Ітерація {i+1}/{iterations} завершена")

            return self._create_report(
                algorithm_name,
                "signing_performance",
                all_metrics,
                {"iterations": iterations, "document_path":
document_path}
            )

        def measure_verification_performance(self, signer,
document_path: str,
                                             signature: bytes,
iterations: int = 10) -> PerformanceReport:
            """Вимірює продуктивність перевірки підпису"""
            algorithm_name = signer._get_algorithm_enum().value
            all_metrics = []


            print(f"Вимірювання продуктивності перевірки для
{algorithm_name}...")

            for i in range(iterations):
                # Створюємо колектори для кожної ітерації
                time_collector =
TimeCollector(MetricType.VERIFICATION_TIME, algorithm_name)
                memory_collector = MemoryCollector(algorithm_name)
                cpu_collector = CPUCollector(algorithm_name)

                # Запускаємо збір метрик
                time_collector.start_collection()
                memory_collector.start_collection()
                cpu_collector.start_collection()
```

```
                # Виконуємо операцію перевірки
                result =
signer.verify_document_signature(document_path, signature)

                # Зупиняємо збір і отримуємо метрики
                time_metric = time_collector.stop_collection()
                memory_metric = memory_collector.stop_collection()
                cpu_metric = cpu_collector.stop_collection()

                all_metrics.extend([time_metric, memory_metric,
cpu_metric])

                print(f"  Ітерація {i+1}/{iterations} завершена")

            return self._create_report(
                algorithm_name,
                "verification_performance",
                all_metrics,
                {"iterations": iterations, "document_path":
document_path}
            )

        def measure_key_generation_performance(self,
algorithm_factory_func: Callable,
                                               iterations: int = 10)
-> PerformanceReport:
            """Вимірює продуктивність генерації ключів"""
            # Отримуємо назву алгоритму з першого виклику
            test_algorithm = algorithm_factory_func()
            algorithm_name =
test_algorithm.__class__.__name__.replace("Signature", "")

            all_metrics = []


            for i in range(iterations):
                # Створюємо колектори для кожної ітерації
                time_collector =
TimeCollector(MetricType.KEY_GENERATION_TIME, algorithm_name)
                memory_collector = MemoryCollector(algorithm_name)
                cpu_collector = CPUCollector(algorithm_name)

                # Запускаємо збір метрик
                time_collector.start_collection()
                memory_collector.start_collection()
                cpu_collector.start_collection()

                # Виконуємо операцію генерації ключів
                algorithm = algorithm_factory_func()
                private_key, public_key =
algorithm.generate_keys()

                # Зупиняємо збір і отримуємо метрики
                time_metric = time_collector.stop_collection()
```

```
                memory_metric = memory_collector.stop_collection()
                cpu_metric = cpu_collector.stop_collection()

                all_metrics.extend([time_metric, memory_metric,
cpu_metric])

                print(f"  Ітерація {i+1}/{iterations} завершена")

            return self._create_report(
                algorithm_name,
                "key_generation_performance",
                all_metrics,
                {"iterations": iterations}
            )

        def run_comprehensive_test(self, signature_manager,
document_path: str,
                                   iterations: int = 10) ->
List[PerformanceReport]:
            """Запускає комплексний тест продуктивності для всіх
алгоритмів"""
            all_reports = []

            for algorithm, signer in
signature_manager.signers.items():
                print(f"\n{'='*50}")
                print(f"Тестування алгоритму: {algorithm.value}")
                print(f"{'='*50}")

                # Генеруємо підпис для тестування перевірки
                test_signature =
signer.sign_document(document_path)

                # Тестуємо підпис
                signing_report = self.measure_signing_performance(
                    signer, document_path, iterations
                )
                all_reports.append(signing_report)

                # Тестуємо перевірку
                verification_report =
self.measure_verification_performance(
                    signer, document_path,
test_signature.signature, iterations
                )
                all_reports.append(verification_report)

            self.results.extend(all_reports)
            return all_reports

        def _create_report(self, algorithm: str, test_name: str,
                           metrics: List[PerformanceMetric],
                           test_parameters: Dict[str, Any]) ->
PerformanceReport:
            """Створює звіт про продуктивність з статистикою"""
```

```
            # Групуємо метрики за типом
            metrics_by_type = {}
            for metric in metrics:
                if metric.metric_type not in metrics_by_type:
                    metrics_by_type[metric.metric_type] = []

metrics_by_type[metric.metric_type].append(metric.value)

            # Обчислюємо статистику
            statistics_data = {}
            for metric_type, values in metrics_by_type.items():
                if values:
                    statistics_data[f"{metric_type.value}_avg"] =
statistics.mean(values)
                    statistics_data[f"{metric_type.value}_min"] =
min(values)
                    statistics_data[f"{metric_type.value}_max"] =
max(values)
                    statistics_data[f"{metric_type.value}_median"]
= statistics.median(values)
                    if len(values) > 1:

statistics_data[f"{metric_type.value}_stdev"] =
statistics.stdev(values)

            return PerformanceReport(
                algorithm=algorithm,
                test_name=test_name,
                metrics=metrics,
                statistics=statistics_data,
                test_parameters=test_parameters,
                timestamp=datetime.now()
            )

        def export_results_to_json(self, filename: str) -> None:
            """Експортує результати в JSON файл"""
            data = []
            for report in self.results:
                report_data = {
                    "algorithm": report.algorithm,
                    "test_name": report.test_name,
                    "statistics": report.statistics,
                    "test_parameters": report.test_parameters,
                    "timestamp": report.timestamp.isoformat(),
                    "metrics_count": len(report.metrics)
                }
                data.append(report_data)

            with open(filename, 'w', encoding='utf-8') as f:
                json.dump(data, f, indent=2, ensure_ascii=False)

            print(f"Результати експортовано в {filename}")

        def export_results_to_csv(self, filename: str) -> None:
            """Експортує статистику в CSV файл"""
```

```
            if not self.results:
                print("Немає результатів для експорту")
                return

            with open(filename, 'w', newline='', encoding='utf-8')
as f:
                # Збираємо всі можливі поля статистики
                all_fields = set()
                for report in self.results:
                    all_fields.update(report.statistics.keys())

                fieldnames = ['algorithm', 'test_name',
'timestamp'] + sorted(all_fields)
                writer = csv.DictWriter(f, fieldnames=fieldnames)

                writer.writeheader()
                for report in self.results:
                    row = {
                        'algorithm': report.algorithm,
                        'test_name': report.test_name,
                        'timestamp': report.timestamp.isoformat()
                    }
                    row.update(report.statistics)
                    writer.writerow(row)

            print(f"Статистика експортована в {filename}")

        def print_summary(self) -> None:
            """Виводить сумарний звіт про продуктивність"""
            if not self.results:
                print("Немає результатів для відображення")
                return

            print(f"\n{'='*60}")
            print("ЗВІТ ПРО ПРОДУКТИВНІСТЬ АЛГОРИТМІВ ЦИФРОВОГО
ПІДПИСУ")
            print(f"{'='*60}")

            # Групуємо результати за алгоритмами
            by_algorithm = {}
            for report in self.results:
                if report.algorithm not in by_algorithm:
                    by_algorithm[report.algorithm] = []
                by_algorithm[report.algorithm].append(report)

            for algorithm, reports in by_algorithm.items():
                print(f"\n{algorithm}:")
                print("-" * 30)

                for report in reports:
                    print(f"  {report.test_name}:")
                    for key, value in report.statistics.items():
                        if isinstance(value, float):
                            print(f"    {key}: {value:.3f}")
                        else:
```

```
                            print(f"    {key}: {value}")

        def export_iteration_results_to_csv(self, filename_prefix:
str) -> None:
            """Експортує результати кожної ітерації в окремі CSV
файли"""
            if not self.results:
                print("Немає результатів для експорту")
                return

            for report in self.results:
                # Створюємо ім'я файлу на основі алгоритму та типу
тесту
                filename =
f"{filename_prefix}_{report.algorithm}_{report.test_name}_iteratio
ns.csv"

                # Групуємо метрики за типом
                metrics_by_iteration = {}
                for metric in report.metrics:
                    iteration = len(metrics_by_iteration)
                    if iteration not in metrics_by_iteration:
                        metrics_by_iteration[iteration] = {
                            'algorithm': report.algorithm,
                            'test_name': report.test_name,
                            'timestamp':
metric.timestamp.isoformat()
                        }

metrics_by_iteration[iteration][f"{metric.metric_type.value}"] =
metric.value

                # Записуємо в CSV
                if metrics_by_iteration:
                    with open(filename, 'w', newline='',
encoding='utf-8') as f:
                        fieldnames = ['algorithm', 'test_name',
'timestamp'] + \
                                     list(set(key for d in
metrics_by_iteration.values()
                                              for key in d.keys()
                                              if key not in
['algorithm', 'test_name', 'timestamp']))

                        writer = csv.DictWriter(f,
fieldnames=fieldnames)
                        writer.writeheader()
                        for iteration_data in
metrics_by_iteration.values():
                            writer.writerow(iteration_data)

                    print(f"Дані ітерацій експортовано в
{filename}")

    if __name__ == "__main__":
```

```
        print("Цей модуль призначений для імпорту. Запустіть
main.py для демонстрації.")
```

Додаток В Лістинг файлу main.py

```
    """
    Демонстрація системи цифрових підписів з аналізом
продуктивності
    """

    import os
    from digital_signature_system import SignatureManager,
SignatureAlgorithm
    from performance_analyzer import PerformanceAnalyzer

    def create_test_document(filename: str, size_kb: int = 10) ->
str:
        """Створює тестовий документ заданого розміру"""
        content = "Це тестовий документ для аналізу продуктивності
алгоритмів цифрового підпису. Створений Іващишином Максимом,
студентом ТНТУ, кафедра кібербезпеки " * (size_kb * 10)

        with open(filename, 'w', encoding='utf-8') as f:
            f.write(content)

        print(f"Створено тестовий документ '{filename}' розміром
{os.path.getsize(filename)} байт")
        return filename

    def demo_basic_functionality():
        """Демонстрація базової функціональності системи"""
        print("="*60)
        print("ДЕМОНСТРАЦІЯ БАЗОВОЇ ФУНКЦІОНАЛЬНОСТІ")
        print("="*60)

        # Створюємо тестовий документ
        test_doc = "document_10MB.txt"

        # Ініціалізуємо систему
        manager = SignatureManager()

        # Додаємо всі алгоритми
        algorithms = [
            SignatureAlgorithm.RSA,
            SignatureAlgorithm.DSA,
            SignatureAlgorithm.ECDSA,
            SignatureAlgorithm.EDDSA
        ]

        for algorithm in algorithms:
            manager.add_signer(algorithm)

        # Тестуємо кожен алгоритм
        for algorithm in algorithms:
            print(f"\n--- Тест {algorithm.value} ---")

            # Отримуємо інформацію про алгоритм
            info = manager.signers[algorithm].get_algorithm_info()
```

```
            print(f"Алгоритм: {info}")

            # Підписуємо документ
            signature_result =
manager.sign_document_with_algorithm(test_doc, algorithm)
            print(f"✓ Підпис створено, розмір:
{signature_result.signature_size} байт")

            signed_filename =
f"{algorithm.value}_signed_output.txt"
            with open(test_doc, 'r', encoding='utf-8') as
original_file:
                original_content = original_file.read()

            with open(signed_filename, 'w', encoding='utf-8') as
signed_file:
                signed_file.write("===== ОРИГІНАЛЬНИЙ ДОКУМЕНТ
=====\n")
                signed_file.write(original_content)
                signed_file.write("\n\n===== ПІДПИС (base64)
=====\n")
                import base64

signed_file.write(base64.b64encode(signature_result.signature).dec
ode('utf-8'))

            # Перевіряємо підпис
            verification_result =
manager.verify_document_with_algorithm(
                test_doc, signature_result.signature, algorithm
            )
            status = "✓ Валідний" if verification_result.is_valid
else "✗ Невалідний"
            print(f"Перевірка підпису: {status}")

        print(f"\nБазове тестування завершено!\n")

    def demo_performance_analysis():
        """Демонстрація аналізу продуктивності"""
        print("="*60)
        print("ДЕМОНСТРАЦІЯ АНАЛІЗУ ПРОДУКТИВНОСТІ")
        print("="*60)

        # Створюємо тестовий документ
        test_doc = "document_10MB.txt"

        # Ініціалізуємо системи
        manager = SignatureManager()
        analyzer = PerformanceAnalyzer()

        # Додаємо алгоритми
        algorithms = [
            SignatureAlgorithm.RSA,
            SignatureAlgorithm.DSA,
            SignatureAlgorithm.ECDSA,
```

```
            SignatureAlgorithm.EDDSA
        ]

        for algorithm in algorithms:
            manager.add_signer(algorithm)

        # Запускаємо комплексний тест продуктивності
        print(f"\nЗапуск комплексного аналізу продуктивності...")
        print(f"Кількість ітерацій: 3 (для швидкого тесту)")

        reports = analyzer.run_comprehensive_test(manager,
test_doc, iterations=30)

        # Виводимо результати
        analyzer.print_summary()

        # Експортуємо результати

analyzer.export_results_to_json("performance_results.json")
        analyzer.export_results_to_csv("performance_results.csv")

        analyzer.export_iteration_results_to_csv("iterations")

        print(f"\nРезультати збережено у файли:")
        print(f"- performance_results.json")
        print(f"- performance_results.csv")


    def demo_individual_tests():
        """Демонстрація окремих тестів продуктивності"""
        print("="*60)
        print("ДЕМОНСТРАЦІЯ ОКРЕМИХ ТЕСТІВ")
        print("="*60)

        # Створюємо тестовий документ
        test_doc = "document_10MB.txt"

        # Ініціалізуємо систему
        manager = SignatureManager()
        analyzer = PerformanceAnalyzer()

        # Додаємо один алгоритм для детального тестування
        manager.add_signer(SignatureAlgorithm.RSA)
        rsa_signer = manager.signers[SignatureAlgorithm.RSA]

        print(f"\nДетальне тестування RSA алгоритму:")

        # Тест підпису
        print(f"\n1. Тестування продуктивності підпису...")
        signing_report =
analyzer.measure_signing_performance(rsa_signer, test_doc,
iterations=30)

        # Тест перевірки
        print(f"\n2. Тестування продуктивності перевірки...")
```

```
        test_signature = rsa_signer.sign_document(test_doc)
        verification_report =
analyzer.measure_verification_performance(
            rsa_signer, test_doc, test_signature.signature,
iterations=30
        )

        # Тест генерації ключів
        print(f"\n3. Тестування продуктивності генерації
ключів...")
        from digital_signature_system import RSASignature
        key_gen_report =
analyzer.measure_key_generation_performance(
            lambda: RSASignature(key_size=2048), iterations=30
        )

        # Виводимо детальні результати
        print(f"\n--- ДЕТАЛЬНІ РЕЗУЛЬТАТИ ---")
        for report in [signing_report, verification_report,
key_gen_report]:
            print(f"\nТест: {report.test_name}")
            for key, value in report.statistics.items():
                if isinstance(value, float):
                    print(f"  {key}: {value:.3f}")
                else:
                    print(f"  {key}: {value}")


    def main():
        """Головна функція демонстрації"""
        print("СИСТЕМА ЦИФРОВИХ ПІДПИСІВ З АНАЛІЗОМ
ПРОДУКТИВНОСТІ")
        print("="*60)

        try:
            # 1. Демонстрація базової функціональності
            demo_basic_functionality()

            # 2. Демонстрація аналізу продуктивності
            demo_performance_analysis()

            # 3. Демонстрація окремих тестів
           # demo_individual_tests()

            print("\n" + "="*60)
            print("ВСІ ТЕСТИ ЗАВЕРШЕНО УСПІШНО!")
            print("="*60)

            # Показуємо створені файли
            files_created = []
            for filename in ["performance_results.json",
"performance_results.csv"]:
                if os.path.exists(filename):
                    files_created.append(filename)
```

```
        if files_created:
            print(f"\nСтворені файли з результатами:")
            for filename in files_created:
                size = os.path.getsize(filename)
                print(f"- {filename} ({size} байт)")

    except Exception as e:
        print(f"\nПомилка під час виконання: {e}")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    main()
```