МІЖНАРОДНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, програмної інженерії та комп'ютерних наук
Кафедра комп'ютерної інженерії та інноваційних технологій

Пояснювальна записка

до кваліфікаційної роботи
другого (магістерського) рівня

на тему: **ДОСЛІДЖЕННЯ МЕТОДІВ ЗАХИСТУ ЕЛЕКТРОННОГО ДОКУМЕНТООБІГУ**

Виконав: студент 2 курсу, групи КТК – 2.1
спеціальності
125 Кібербезпека
_____ Каюков Л.Л

Керівник_____ Йона Л.Г

Рецензент_____ Соловська І. М.

Одеса – 2023

ДОВІДКА

кафедри КІ та ІТ про виконану магістерську роботу

студента 2 курсу ФКПІ та КН групи КТК – 2.1

Каюкова Леоніда Леонідовича

на тему: Дослідження методів захисту електронного документообігу

Висновок нормоконтролера пояснювальна записка до кваліфікаційної роботи виконана з незначними порушеннями ДСТУ. Оформлено згідно вимог внутрішнього положення НТУ.

Нормоконтролер к.т.н., доц. каф. КІтаІТ _____ В.В. Педяш

(науковий ступінь, вчене звання, посада) (підпис, дата) (і. б. прізвище)

Висновок відповідального за наявність плагіату згідно з сертифікатом ID-1015677899 унікальність роботи підтверджено.

Відповідальна особа к.т.н., доц. каф. КІтаІТ _____ В.В. Педяш

(науковий ступінь, вчене звання, посада) (підпис, дата) (і. б. прізвище)

Попередня експертиза (захист) _____ магістерської роботи _____

(бакалаврської роботи чи магістерської роботи)

студ. Каюкова Л.Л. проведена "12" грудня 2023 р.

(прізвище і.б.)

Висновки Розділи МР відповідають завданню, усі пункти виконані. Проаналізовано наявні принципи безпеки електронного документообігу. Розроблені можливі алгоритми та оцінено доцільність розробки сервісу захисту електронного документообігу. МР відповідає вимогам до КР за даною спеціальністю та може бути рекомендована до захисту в ЕК.

Члени комісії _____ к.т.н., доц. Чіока І.Т.

(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

_____ к.т.н., доц. Педяш В.В.

(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

_____ викл. каф. КІтаІТ Ківець О.В.

(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

МІЖНАРОДНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, програмної інженерії та комп'ютерних наук
Кафедра комп'ютерної інженерії та інноваційних технологій
Освітній ступінь магістр
Галузь знань 12 Інформаційні технології
Спеціальність 125 Кібербезпека

ЗАТВЕРДЖУЮ
Завідуючий кафедрою КІтаІТ
к.т.н., доц.
Л.Г Йона
"25" 09 2023 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ

Каюкову Леоніду Леонідовичу

1. Тема роботи Дослідження методів захисту електронного документообігу
керівник роботи Йона Лариса Григорівна, к.т.н., доц.
затверджені наказом закладу вищої освіти від 25.09.2023 р. № 1951

2. Строк подання студентом роботи 11.12.2023 р.

3. Вихідні дані до роботи науково-методичні розробки авторів, матеріали переддипломної практики, завдання виконуються за допомогою мови програмування c#, та бібліотеки System.Security.Cryptography

4. Зміст розрахунково-пояснювальної записки
Розділ 1: Основи криптографії та безпеки інформації
Розділ 2: Засоби що забезпечуються конфіденційність інформації
Розділ 3: Розробка та аналіз криптографічних алгоритмів використанням.NET

5. Перелік графічного матеріалу (з зазначенням обов'язкових креслень)
Слайд 9 – Outlook з використанням Kleopatra
Слайд 10 – Функції криптографічних перетворень в бібліотеці System.Security.Cryptography

Слайд 11 – Створення приватного, публічного ключа та їх збереження або завантаження
Слайд 12 – Створення та використання функції для зашифруван
Слайд 13 – Створення та використання функції для розшифрування
Слайд 14 – Аналіз алгоритму з різними довжинами ключа та переваги використання бібліотеки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Завдання видав	Завдання прийняв

7. Дата видачі завдання 25.09.2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Основи криптографії та безпеки інформації	25.09.2023– 02.10.2023	Вик
2	Засоби що забезпечують конфіденційність інформації	03.10.2023– 16.10.2023	Вик
3	Розробка та аналіз криптографічних алгоритмів	17.10.2023– 30.10.2023	Вик
4	Висновки та рекомендації	31.10.2023– 06.11.2023	Вик
5	Перелік джерел посилання	07.11.2023 – 13.11.2023	Вик
6	Додаток А Перелік копій демонстраційного матеріалу	14.11.2023 – 30.11.2023	Вик
7	Додаток Б Лістинги модулів програмної системи	01.12.2023 - 05.12.2023	Вик

Студент _____ Л.Л. Каюков
(підпис)

Керівник роботи _____ Л.Г. Йона
(підпис)

ВІДГУК

на випускну кваліфікаційну роботу другого магістерського рівня

зі спеціальності 125 «Кібербезпека»

Каюкова Леоніда Леонідовича

на тему “Дослідження методів захисту електронного документообігу”

Тема магістерської роботи здобувача Каюкова Л. Л. є актуальною та пов’язана з проблемою захисту інформації під час здійснення обміну електронними документами.

У магістерській роботі здобувачем проаналізовано основні засоби безпеки електронного документообігу, проведено огляд запропонованих алгоритмів захисту інформації, надано рекомендації щодо оптимальної довжини ключів для досягнення балансу між безпекою та продуктивністю системи під час електронного документообігу.

Результати дослідження представлені у тезах доповіді «Дослідження методів криптографічного захисту в електронному документообігу» на IX Всеукраїнській науково-практичній конференції студентів, аспірантів та молодих вчених «Гуманітарний і інноваційний ракурс професійної майстерності: пошуки молодих вчених».

Магістерська робота Каюкова Л.Л. відповідає вимогам до випускних кваліфікаційних робіт магістрів та заслуговує оцінки «відмінно».

Здобувач Каюков Л.Л. заслуговує присвоєння кваліфікації магістр з кібербезпеки за заявленою спеціальністю 125 Кібербезпека.

Керівник, к.т.н., доц. кафедри
Комп’ютерної інженерії
та інноваційних технологій Йона Л. Г.

РЕЦЕНЗІЯ

на випускну кваліфікаційну роботу другого магістерського рівня

зі спеціальності 125 «Кібербезпека»

Каюкова Леоніда Леонідовича

на тему “Дослідження методів захисту електронного документообігу”

Магістерська робота виконана на 65 с. текстової частини та 15 слайдах і містить відповідні розділи згідно з завданням на магістерську роботу.

Представлена на рецензію магістерська робота є актуальною та відповідає затвердженій темі.

У магістерській роботі здобувачем проаналізовано основні засоби безпеки електронного документообігу, проведено попередній огляд існуючих систем захисту та виявлено їх основні недоліки. Проведено огляд запропонованих алгоритмів захисту інформації, а також описано засіб захисту електронного документообігу.

Пояснювальна записка виконана охайно й відповідно до вимог ЄСКД.

В практичній реалізації магістерської роботи є деякі недоліки:

1) програмний код використовує роботу з файлами, але він не має належного управління ресурсами та обробки помилок;
2) безпечні соціальні мережі обмежуються лише декількома платформами, хоча існує значно більше аналогічних сервісів;
3) дослідження алгоритмів електронного підпису проведено формально.

Але вказані недоліки не знижують цінності виконаної роботи.

Магістерська робота Каюкова Л.Л. відповідає вимогам до випускних кваліфікаційних робіт магістрів та заслуговує оцінки «відмінно».

Здобувач Каюков Л.Л. заслуговує присвоєння кваліфікації магістр з кібербезпеки за заявленою спеціальністю 125 Кібербезпека .

Рецензент

Завідувачка кафедри комп’ютерних наук

К.т.н., доцент Соловська І.М.

UNICHECK
by Turnitin

Ім'я користувача:
Анна Серединко

ID перевірки:
1015995246

Дата перевірки:
11.12.2023 23:55:37 MSK

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
12.12.2023 00:03:33 MSK

ID користувача:
100001433

Назва документа: Каюков

Кількість сторінок: 115 Кількість слів: 13327 Кількість символів: 114963 Розмір файлу: 5.30 MB ID файлу: 1015677899

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

8.44%
Схожість

Найбільша схожість: 1.43% з джерелом з Бібліотеки (ID файлу: 1015677901)

8.27% Джерела з Інтернету	918	Сторінка 117
1.65% Джерела з Бібліотеки	25	Сторінка 121

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	1
Підозріле форматування	50 сторінок

РЕФЕРАТ

Текстова частина магістерської роботи: 51 с., 66 рис., 4 табл., 2 додатки, 15 джерел.

КРИПТОГРАФІЧНІ АЛГОРИТМИ, КОНФІДЕЦІЙНІСТЬ, ЦІЛІСНІСТЬ, СИМЕТРИЧНЕ ШИФРУВАННЯ, АСИМЕТРИЧНЕ ШИФРУВАННЯ, ХЕШ-ФУНКЦІЇ, ЦИФРОВИЙ ПІДПИС, ДОВЖИНА КЛЮЧА, КРИПТОСТІЙКІСТЬ, ОБЧИСЛЮВАЛЬНЕ НАВАНТАЖЕННЯ

Об'єктом дослідження є методи захисту, зокрема криптографічні методи, використовані для забезпечення безпеки електронного документообігу.

Актуальність теми: проблема захисту електронних документів за допомогою електронного цифрового підпису, різних криптографічних алгоритмів і т.д. надзвичайно поширена в наш час. Оскільки на цей момент настала ера Інтернету та електронних технологій, чим далі ми йдемо, тим актуальнішою стає проблема захисту власних електронних документів і даних.

Метою роботи є дослідження можливості захисту електронного документообігу через глобальну мережу.

Метод дослідження – аналітичний метод з використанням комп'ютерних технологій для розробки програмної реалізації засобом програмування c# та її бібліотеки System.Security.Cryptography.

У магістерській роботі проаналізувані присутні принципи безпеки електронного документообігу та можливості розвитку захисту документообміну засобами мережі Інтернет. Проведений аналіз та попередній огляд наявних систем захисту електронного документообігу. Розроблені можливі алгоритми та оцінено доцільність розробки сервісу захисту електронного документообігу.

ABSTRACT

The text part of the master's thesis: 51 pages, 66 drawings, 4 tables, 2 appendices, 15 sources.

CRYPTOGRAPHIC ALGORITHMS, CONFIDENTIALITY, INTEGRITY, SYMMETRIC ENCRYPTION, ASYMMETRICAL ENCRYPTION, HASH FUNCTIONS, DIGITAL SIGNATURE, KEY LENGTH, CRYPTO STRENGTH, COMPUTATION LOAD

The object of the study is protection methods, in particular cryptographic methods, used to ensure the security of electronic document flow.

Relevance of the topic: the problem of protecting electronic documents using electronic digital signatures, various cryptographic algorithms, etc. extremely common nowadays. Since at this moment the era of the Internet and electronic technologies has arrived, the further we go, the more urgent the problem of protecting one's own electronic documents and data becomes.

The purpose of the work is to study the possibility of protecting electronic document circulation through the global network.

The research method is an analytical method using computer technologies to develop a software implementation using the C# programming tool and its System.Security.Cryptography library.

In the master's thesis, the existing security principles of electronic document circulation and the possibility of developing the protection of document exchange by means of the Internet are analyzed. The analysis and preliminary review of the available electronic document flow protection systems was carried out. Possible algorithms have been developed and the feasibility of developing an electronic document flow protection service has been assessed.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

DES – Data Encryption Standard
AES – Advanced Encryption Standard
RSA – Rivest-Shamir-Adleman
SHA – Secure Hash Algorithm
TLS – Transport Layer Security
SSL - Secure Sockets Layer
ЕЦП – Електронний цифровий підпис
СЕД(ЕД) – Система електронного документообігу
DHKE – Diffie-Hellman Key Exchange
LUKS – Linux Unified Key Setup

ВСТУП

Інформаційна безпека є теоретичною і практичною основою комплексу знань і вмінь, які формують профіль фахівця в галузі інформаційних комп'ютерних систем та технологій і є одним із основних у формуванні знань студентів з питань інформаційної безпеки. В електронному документі інформація, зафіксована з використанням електронних даних, повинна містити обов'язкові реквізити документа, найважливішим з яких є електронний підпис, інакше це підпис в електронній формі. Захист основних атрибутів інформації, а саме конфіденційності, цілісності та доступності, забезпечується за допомогою криптографічних алгоритмів. Якщо це забезпечення конфіденційності, тобто запобігання витоку інформації, то це можна вирішити шляхом шифрування публічних повідомлень. Зашифроване повідомлення в криптографічній формі передається одержувачу через незахищений канал. При цьому для шифрування та розшифрування повідомлень необхідно знати спільний секретний ключ шифрування [4,5].

1

2 ОСНОВИ КРИПТОГРАФІЇ ТА БЕЗПЕКИ ІНФОРМАЦІЇ

1.1 Поняття електронного документа та електронного документообігу

Документи використовуються в різних галузях знань, діяльності людини та суспільного життя. Вони є об’єктами вивчення різних наукових дисциплін, тому поняття документа є неоднозначним і залежить від сфери його використання та призначення. У широкому сенсі документообіг можна визначити як інформаційну діяльність, що здійснюється суб’єктами інформаційних відносин шляхом виконання певних дій з документами. Система управління документами визначається як набір методів, засобів і персоналу, які підтримують управління документами в рамках встановлених правил управління документами [6].

Правила документообігу — сукупність правил інформаційної діяльності суб'єктів інформаційних відносин, визначених законодавством, нормативними актами або договорами. Регламент документообігу визначає ролі та права суб'єктів щодо створення, володіння, користування та розпорядження документами, а також порядок обробки та фіксації інформації на носіях інформації. Як широке поняття, електронний документообіг можна пояснити як інформаційну технологію, яка реалізує життєвий цикл електронних документів. При цьому систему електронного документообігу можна визначити як автоматизовану систему обробки інформації, що реалізує електронний документообіг і поєднується з іншими системами документообігу.

Необхідно підкреслити такі ключові принципи та цілі управління документацією:

- одноразова реєстрація документа, яка дозволяє однозначно ідентифікувати його в будь-якій підсистемі;
- можливість виконання операцій паралельно, що скорочує час руху файлів і підвищує ефективність виконання;
- безперервність руху файлу, що дозволяє ідентифікувати особу, відповідальну за виконання, у кожен момент життєвого циклу файлу;
- єдина файлова інформаційна база даних, яка виключає дублікати файлів;
- ефективно організована система пошуку документів, яка забезпечує пошук мінімальної інформації про сам документ;
- добре розвинена система звітності для різних статусів і властивостей файлів, що дозволяє контролювати рух файлів у процесі файлообміну та приймати управлінські рішення на основі даних у звітах.

1.2 Загрози та способи захисту системи електронного документообігу

Основними загрозами безпеці СЕД є [7]**:**

1 **загроза цілісності інформації** — пошкодження, знищення або спотворення інформації, як навмисне, так і зловмисне;

2 **загроза конфіденційності** - будь-яке порушення конфіденційності, включаючи крадіжку, перехоплення інформації, зміну маршрутів доставлення тощо;

3 **загрози для роботи системи** - загрози, реалізація яких може призвести до збою або припинення СЕД;

4 **дії загрози доступності**, які роблять неможливим або ускладнюють доступ до СЕД, зокрема, створення умов, за яких доступ до послуги чи інформації блокується або можливий на час, що не забезпечить виконання певних цілей.

Безпосередній захист СЕД може бути організований різними способами. Як уже зазначалося, багато власників інформації та документів досі побоюються запроваджувати електронний документообіг саме з міркувань безпеки, втрачаючи при цьому час та гроші. Але зараз ринок цих послуг в Україні досить розвинений, тому кожен може вибрати провайдера, виходячи з власних потреб і фінансових можливостей.

Зрозуміти, що СЕД надійний можна за кількома показниками [7]**:**

1. **забезпечення безпеки ЕД**: перш за все, це резервна копія ЕД. Це також включає зберігання в архіві, який знаходиться в безпечному хмарному середовищі в кількох центрах обробки даних;
2. **закритий доступ до СЕД**: це може бути один зі способів, наприклад пароль, USB-ключ або відбиток пальця. Або він може бути багатоетапним, що складається з кількох кроків: пароль і ключ, або пароль і біометричний запис;
3. **розмежування прав доступу**: цей функціонал дозволяє мати доступ до окремих документів лише певній групі користувачів. Інші користувачі, відповідно, позбавлені цього права;
4. **ступінь конфіденційності**: щоб зберегти конфіденційність, СЕД можуть використовувати криптографічні методи для шифрування даних, які ніхто, крім їх власника та призначених користувачів, не зможе побачити;
5. **забезпечення правдивості інформації**: поки що основним і чи не єдиним пропонованим на ринку рішенням для забезпечення правдивості документа є ЕЦП;
6. **реєстрація дій користувачів СЕД**.

Таблиця 1.1 - Способи захисту електронних документів

Способи захисту електронних документів				
Апаратні засоби	Хардкорне шивфрування: використання модулів для шифрування	Біометрична індетифікація: використання біометричних сканерів для ідентифікації користувачів	Фізичний доступ: захист фізичного доступу до обладнання, серверних приміщень і дата-центрів	Фізичні засоби зберігання: використання сейфів та інших фізичних засобів для зберігання критичних даних
Програмні засоби	Криптографічний захист: використання алгоритмів шифрування для захисту даних	Керування доступом: розмежування прав доступу до документів і ресурсів	Електронний цифровий підпис: використання ЕЦП для підтвердження автентичності та недоторканості документів	Антивірусне програмне забезпечення: використання антивірусних програм для захисту від шкідливих програм

1.3 Криптографічна стійкість, криптографія і її основні поняття

Криптографія надає інструменти для захисту інформації та є складовою діяльності з інформаційної безпеки [3].

Існують різні засоби ініціювання інформації:

1. приховування каналу передачі повідомлень;
2. маскування змісту повідомлення за допомогою стеганографічних методів;
3. ускладнення можливості перехоплення самого повідомлення противником.

На відміну від цих методів, криптографія не «ховає» повідомлення, а перетворює їх у форми, недоступні для розуміння ворога. Це перетворення забезпечується використанням криптографічних систем.

Криптографічна стійкість означає здатність криптографічного алгоритму протистояти можливим атакам. Стійкий алгоритм - це алгоритм, який повинен успішно протистояти можливим атакам:

- вимагає великих обчислювальних ресурсів;
- залишається ефективним при неможливотсі перехоплення відкритих і зашифрованих повідомлень.

Стійкість неможливо підтвердити, її можна спростувати лише шляхом злому. Щоб оцінити стійкість шифру, необхідно оцінити обчислювальну складність алгоритмів, які успішно атакують криптографічну систему. Рівень криптографічної стійкості є мірою криптографічної стійкості алгоритму та пов’язаний з обчислювальною складністю успішної атаки на криптографічну систему.

Як правило, рівень надійності шифрування вимірюється в бітах. N-бітний рівень міцності криптосистеми означає, що для її злому потрібно n обчислювальних операцій. Наприклад, якщо симетричну криптосистему неможливо зламати швидше, ніж значення n-бітного ключа повністю вичерпано, рівень криптографічної міцності називається n. Складність алгоритму зазвичай оцінюється часом виконання, але не менш важливі й інші показники - вимоги до пам'яті та вільний простір на диску. У теорії алгоритмів обчислювальна складність алгоритму — це співвідношення, яке визначає залежність обсягу роботи, яку виконує алгоритм, від розміру вхідних даних.

1.4 Криптосистема DES

Алгоритм шифрування даних DES, опублікований у 1977 році Національним бюро стандартів США, є стандартним блоковим шифром, призначеним для захисту несекретної, але важливої інформації в урядових і комерційних організаціях США від несанкціонованого доступу. У 1980 році заснований на стандартах алгоритм був схвалений Національним інститутом стандартів і технологій США.

DES є важливою частиною комерційних систем захисту інформації та використовується в різних програмах і спеціалізованих комп'ютерах для забезпечення безпеки даних у мережах передачі інформації. Алгоритм DES працює шляхом розбиття послідовності, що передається по каналу зв’язку, на 64 - бітові блоки. Кожен такий блок незалежно шифрується за допомогою 64 - бітового ключа, в якому для шифрування потрібно лише 56 біт. DES використовує різні операції, щоб переставити символи та додати модуль. Однією з переваг цього алгоритму є те, що операції шифрування і дешифрування в ньому взаємопов'язані [3].

1.5 Криптосистема AES

Розширений стандарт шифрування (AES), також відомий як Rijndael, — це алгоритм симетричного блокового шифрування з розміром блоку 128 біт і ключем 128/192/256 біт. Фіналіст конкурсу AES і прийнятий урядом США як стандарт шифрування США. AES було обрано в розрахунку на те, що алгоритм буде широко використовуватися та активно аналізуватися, як і його попередник DES. Після п'яти років підготовки 26 жовтня 2001 року Національний інститут стандартів і технологій випустив попередню специфікацію AES. 26 травня 2002 року AES було оголошено як стандарт шифрування. Станом на 2009 рік AES є одним із найпоширеніших алгоритмів симетричного шифрування. В принципі, алгоритм, запропонований Рейманом і Дейцманом, не збігається з AES. Алгоритм Рейндола підтримує різні розміри блоків і ключів. AES має фіксовану довжину 128 біт, а розмір ключа може бути 128, 192 або 256 біт. Reyndol підтримує розміри блоків і ключів із кроком 32 біти в діапазоні від 128 до 256. Оскільки розмір блоку фіксований, AES працює з масивом 4 × 4 байти, який називається станом [3,8].

Переваги та недоліки AES:

AES безпечний, швидкий і гнучкий. Алгоритми AES працюють швидше, ніж DES. Найбільшою перевагою є різні довжини ключів: чим довше ключ, тим складніше його зламати.

На сьогоднішній день AES є найпопулярнішим алгоритмом шифрування, і він використовується в багатьох додатках, таких як бездротова безпека; безпека процесорів і шифрування файлів; безпека Wi-Fi; шифрування мобільних додатків; протокол SSL/TLS (безпека сайтів); і VPN (virtual private network).

Багато державних установ у Сполучених Штатах використовують алгоритм шифрування AES, щоб захистити свої конфіденційні дані.

Недоліки алгоритму шифрування AES:

1. відомі теоретичні атаки зі складністю, меншою, ніж повний перебір;
2. не може в повній мірі використати можливості 64-бітових платформ;
3. відносна застарілість;
4. відсутність довіри до іноземних апаратних реалізацій AES.

1.6 Криптосистема Калина

На зміну стандарту ДСТУ 28147-2009 (ГОСТ28147-89) прийшов стандарт шифрування ДСТУ 7624:2014 («Калина»). Одним з основних алгоритмів симетричного блокового шифрування, що використовується в Україні, є ДСТУ 7624:2014 («Калина»), який визначає сучасний алгоритм симетричного блокового перетворення для забезпечення конфіденційності та цілісності інформації під час її обробки та встановлює режими його роботи. Новим національним стандартом є симетричний блоковий шифр, який підтримує розмір блоку та довжину ключа шифрування 128, 256 та 512 біт.

Стандарт забезпечує нормальний, високий і надвисокий рівні стабільності, відповідно, і є єдиним у світі стандартом блочного шифрування, який підтримує 512 - бітні симетричні ключі. Алгоритм шифрування даних Kalina використовує криптографічні перетворення, що відповідають сучасним вимогам до рівня криптографічної стійкості та швидкості. Стандарт блочного симетричного шифрування ДСТУ 7624:2014 має десять різних режимів роботи, які широко поширені відповідно до міжнародних стандартів ISO/IEC 10116:2006. телекомунікаційних систем різного призначення може бути забезпечена наявність такої кількості режимів роботи алгоритму. Це спрямовано на забезпечення широкого використання ДСТУ 7624:2014, у тому числі для захисту інформації, що передається комп’ютерними мережами, прозорого шифрування жорстких дисків і знімних носіїв, електронних документів, ключових даних [9].

Основні відмінності «Калини» від «AES» такі:

1. збільшена кількість циклів шифрування;
2. використовувати модуль 264 і додаток 2 для введення ключової інформації;
3. використання чотирьох блоків нелінійного перетворення замість одного;
4. використання випадково згенерованих чотирьох блоків, обраних за критеріями стійкості до диференціального, лінійного криптоаналізу, ступеня нелінійності булевих функцій;
5. принципово нова схема створення з'єднань;
6. досить висока продуктивність;
7. можливість відновлення сеансового ключа за допомогою окремого підключа.

Основними перевагами шифру в порівнянні з іншими міжнародними аналогами є можливість використання блоку даних і ключа шифрування розміром до 512 біт, збільшена кількість циклів шифрування і принципово нова схема створення з'єднань, що забезпечує захист від усіх відомі атаки на схеми їх створення.

1.7 Криптосистема RSA

RSA — це алгоритм шифрування з відкритим ключем, заснований на обчислювальній складності розкладання на множники великого цілого числа. RSA є першим алгоритмом такого роду, придатним для шифрування та цифрових підписів. Цей алгоритм використовується у великій кількості криптографічних програм.

Алгоритм RSA складається з 4 фаз: генерація ключа, шифрування, дешифрування та розподіл ключа. Безпека алгоритму RSA базується на принципі складності цілочисельного розкладання. Цей алгоритм використовує два ключі - відкритий ключ і закритий ключ. Відкритий ключ і відповідний закритий ключ разом утворюють пару ключів. Відкритий ключ не потрібно зберігати в секреті, він використовується для шифрування даних. Якщо повідомлення зашифровано за допомогою відкритого ключа, його можна розшифрувати лише за допомогою відповідного ключа [3].

Основними перевагами RSA є:

1. алгоритм RSA є асиметричним, тобто полягає в розподілі відкритих ключів у мережі. Це дозволяє кільком користувачам обмінюватися інформацією через незахищені канали зв’язку;
2. користувач сам може змінити номер, відкритий і закритий ключі на власний розсуд, а потім повинен поширити відкритий ключ у мережі. Це дозволяє підвищити стійкість до шифрування.

Основні недоліки: низька швидкість бігу, алгоритм RSA повільніший за симетричний алгоритм DES.

Криптосистема RSA, безсумнівно, вважається ефективним інструментом захисту паролем. Хоча він має такі переваги, як безпека передачі ключів і стійкість до криптоаналізу, важливо враховувати недоліки, які обмежують його в певних ситуаціях. Вибір протоколу шифрування повинен враховувати конкретні вимоги до безпеки, швидкості та обчислювальних ресурсів у конкретному прикладному середовищі.

Таблиця 1.2 - Порівняльної характеристики симетричних криптосистем

Крипто система	**Довжина блоку, біт**	**Довжина ключа, біт**	**Кількість етапів шифрування.**	**Кількість підключів (на1 блок)**
DES	64 біт	64(56) біт	16 етапів	16
IDEA	64 біт	128 біт	8 етапів	52
ДСТУ/ГОСТ 29147-2009	64 біт	256	32 етапів	8
AES	128 біт	128, 192 або 256 біт	128біт(10) 192біт(12) 256біт(14)	128біт(11) 192біт(13) 256біт(15)
Калина	128, 256 та 512 біт	128, 256 та 512 біт	10	40

1.8 Хеш-функції SHA

SHA-2 — це загальна назва для односторонніх хеш-функцій SHA-224, SHA-256, SHA-384 і SHA-512. Хеш-функцію також називають функцією згортки, а її результат — хешем, хеш-кодом, хеш-сумою або дайджестом повідомлення. Алгоритм побудований таким чином, що через лавинний ефект найменша зміна повідомлення може призвести до зовсім інших результатів. Після завершення початкове повідомлення ділиться на частини, кожна з яких складається з 16 слів. Алгоритм пропускає кожну частину повідомлення через цикл із 64 або 80 ітерацій. На кожній ітерації 2 слова перетворюються, а функція перетворення встановлюється іншими словами.

Результати обробки кожного блоку додаються, і ця сума є значенням функції. Сімейство хешів поділяється на дві підпослідовності – 32-розрядні (SHA-224 і SHA-256) і 64-розрядні (SHA-384, SHA-512, SHA-512/224, SHA-512/256). 2 алгоритм виконано. Насправді, у всій серії працює той самий робочий алгоритм, але в 32-й підпослідовності робота виконується з використанням 8 32-бітних блоків, а в 64-й підпослідовності для завершення роботи використовуються 8 64-бітних блоків.

Перерахуємо переваги та недоліки хеш-функцій SHA-256, SHA-384, SHA-512.

Однією з головних переваг усіх трьох алгоритмів є їхній високий рівень стійкості до колізій – властивість, що гарантує вкрай низьку ймовірність знаходження двох різних вхідних повідомлень, які генерують однаковий хеш. Крім того, вони мають оптимальні характеристики щодо швидкодії та ресурсомісткості.

Однак недоліками є можливість вразливості до атак типу "довільного обраного тексту" (chosen-prefix collisions) та обмежена можливість використання в області обробки довільно великих даних.

У світі швидко зростаючого обсягу цифрової інформації та комп'ютерних атак важливою стає надійна захист даних. Хеш-функції SHA-256, SHA-384 та SHA-512 виявляються потужними інструментами для досягнення цієї мети. Вибір конкретного алгоритму повинен враховувати специфіку застосування та потреби в безпеці, забезпечуючи оптимальне поєднання швидкодії та стійкості до атак [10].

Таблиця 1.3 – Порівняння характеристик хеш-функцій SHA-256, SHA-384, SHA-512

Хеш-функція	**Довжина блоку, біт**	**Макс. довжина повідомлення**	**Кількість ітерацій**	**Швидкість**
SHA-256	512	2^{64} – 1 біт	64	139 Мб/с
SHA-384	1024	2^{128} – 1 біт	80	154 Мб/с
SHA-512	1024	2^{128} – 1 біт	80	154 Мб/с

1.9 Електронний цифровий підпис

Існує два основних підходи до побудови ЕЦП: асиметричний і симетричний. Асиметрична схема застосування систем ЕЦП передбачає існування мережі абонентів, які пересилають один одному підписані електронні документи. Згенеруйте пару ключів для кожного абонента: секретний ключ і відкритий ключ. Ключ зберігається абонентом в таємниці та використовується ним для формування ЕЦП.

Відкритий ключ відомий усім іншим користувачам і використовується для перевірки ЕЦП одержувачем підписаного електронного документа. Іншими словами, відкритий ключ є вашим обов’язковим інструментом для перевірки автентичності електронного документа та автора підпису. Відкритий ключ не дозволяє обчислити закритий ключ. Для генерації пари ключів в алгоритмі EDS, а також в системах асиметричного шифрування використовуються різні математичні схеми, засновані на застосуванні односторонніх функцій [3].

Програми поділяються на дві групи. Цей розподіл базується на відомій складній обчислювальній задачі:

1. завдання факторизації (розкладання на множники) великих цілих чисел;
2. завдання дискретного логарифмування.

Основні переваги електронно-цифрового підпису (ЕЦП):

1. **криптографічна захищеність:** ЕЦП володіє вищим рівнем захищеності, оскільки його створення ґрунтується на криптографічних алгоритмах, що забезпечують високий ступінь невразливості до несанкціонованого доступу;
2. **ідентифікація та автентифікація:** ЕЦП дозволяє не лише підтверджувати цілісність переданих даних, але й чітко ідентифікувати особу, що здійснює підпис, надаючи неперевершений рівень автентифікації;
3. **ефективність та зручність:** застосування ЕЦП прискорює процеси електронного обміну даними, зменшуючи час на підписання та верифікацію, що робить його важливим інструментом для підприємств та установ;
4. **законна сила:** багато країн закріпили в законодавстві електронний підпис як юридично обов'язковий, що надає йому таку ж саму юридичну силу, як і традиційним рукописним підписам.

Основні недоліки електронно-цифрового підпису (ЕЦП):

1. **відомості про ключі:** збереженість та управління ключами є важливою аспектом забезпечення безпеки ЕЦП; недбале використання може призвести до компрометації системи;
2. **залежність від інфраструктури:** ефективність ЕЦП часто залежить від доступності та стійкості інфраструктури для зберігання ключів та проведення верифікації;
3. **сприйняття користувачами:** деякі користувачі можуть відчувати незручність або непевність при використанні електронних підписів, що може впливати на їхню популярність;
4. **витрати на інфраструктуру:** розгортання та підтримка інфраструктури для впровадження ЕЦП може вимагати значних витрат, що є фактором, який слід враховувати під час впровадження.

1.10 Протокол Diffie-Hellman Key Exchange

DHKE є ключовим елементом криптографії та інформаційної безпеки, пропонуючи ефективний спосіб обміну ключами у відкритих мережах. Цей протокол, розроблений у 1976 році Вітсоном Диффі та Мартіном Геллманом, забезпечує можливість безпечного обміну секретними ключами між сторонами, які навіть не взаємодіють напряму.

Принцип роботи протоколу ґрунтується на складності задачі дискретного логарифмування в математичних групах. Кожна зі сторін генерує власний приватний ключ та обмінюється публічними ключами. За допомогою цих публічних ключів і власного приватного ключа, сторони обчислюють спільний секретний ключ. Важливою перевагою є той факт, що навіть якщо зловмисник перехопить публічні ключі, обчислити приватний ключ залишається вельми трудомісткою задачею [3].

Основні переваги DHKE:

1. **безпека:** забезпечує високий рівень безпеки завдяки складності математичних операцій, пов'язаних із дискретним логарифмуванням.
2. **асиметричність:** не вимагає взаємодії сторін для безпечного обміну ключами, використовуючи асиметричний підхід.
3. **відкритість:** можливість відкритого обміну публічними ключами без ризику компрометації безпеки.

Основні недоліки DHKE: у випадку, коли зловмисник отримує доступ до обмінюваних публічних ключів у реальному часі, протокол може стати уразливим до атаки, згрози розвитку комп'ютерів, Диффі-Геллман Key Exchange може потребувати додаткових заходів захисту.

Диффі-Геллман Key Exchange є важливим інструментом у сучасній криптографії, забезпечуючи безпечний обмін ключами. Хоча протокол має свої переваги, важливо враховувати його обмеження та можливі недоліки. Постійний науковий розвиток у галузі криптографії вимагає постійного вдосконалення протоколів, таких як Диффі-Геллман, для забезпечення ефективного захисту інформації в умовах швидкозмінюваного технологічного середовища.

1.11 Протокол TLS

TLS (Transport Layer Security) [11] – це протокол, реалізований на основі протоколу SSL [12] організацією IETT, яка створила відповідний стандарт. Протокол TLS надає можливість повторного підключення без додаткової автентифікації та збігу сеансового ключа.

Протокол може забезпечити узгодження дійсних криптографічних алгоритмів:

1. генерацію ключів;
2. шифрування;
3. цифровий підпис і аутентифікація;
4. хешування.

Протокол SSL/TLS дозволяє розв'язувати деякі проблеми безпеки. Зазвичай функції протоколу обмежуються шифруванням даних, які передаються через Інтернет. Ви також можете використовувати цей протокол, щоб перевірити, чи дані передаються на відповідний сервер. Однак через технічні та ліцензійні особливості протоколу SSL він вважається не дуже надійним.

Переваги протоколу TLS:

1. використовує стандартний протокол HTTPS, який зазвичай не блокується в громадських місцях;
2. потрібен лише браузер, допоміжні програми не потрібні;
3. потрібно менше адміністративних витрат і технічної підтримки.

Недоліки протоколу TLS:

За допомогою протоколу TLS VPN Unlimited може захистити лише трафік вашого браузера. Клієнт KeepSolid VPN не захистить ваші інші програми чи всю систему.

2 ЗАСОБИ ЩО ЗАБЕЗПЕЧУЮТЬ КОНФІДЕНЦІЙНОСТЬ ІНФОРМАЦІЇ

2.1 Програми для шифрування диска

2.1.1 VeraCrypt

VeraCrypt — це безплатний мультиплатформене програмне забезпечення, яке використовується для миттєвого шифрування дисків і файлів.

VeraCrypt може:

1. створювати безпечні програмні «контейнери», де можна зберігати важливі файли та папки;
2. шифрувати логічні розділи на дисках і цілі диски, включаючи системний диск і, звичайно, флешки.

Особливістю VeraCrypt є те, що шифрування та дешифрування даних виконується на льоту. Кілька простих маніпуляцій на початку робочого дня, опісля VeraCrypt працює у фоновому режимі, не втрачаючи швидкості. У той час як TrueCrypt використовує 1000 ітерацій для створення ключа, який шифрує системний розділ за допомогою алгоритму PBKDF2-RIPEMD-160, VeraCrypt використовує 327 661 ітерацію. Для стандартних зашифрованих розділів диска та файлових контейнерів VeraCrypt використовує 655 331 ітерацію для хеш-функції RIPEMD-160 і 500 000 ітерацій для SHA-2 і Whirlpool. Це значно сповільнює VeraCrypt під час монтування зашифрованих розділів диска, але робить його принаймні в 10 разів більш стійким до повних атак грубої сили [13].

2.1.2 FileVault

FileVault — це система шифрування даних, яка використовує алгоритм XTS-AES-128 із довжиною ключа 256 біт, що забезпечує надзвичайно високий рівень безпеки. Сам ключ шифрування складається з урахуванням пароля користувача за алгоритмом PBKDF2. Надалі вся інформація буде зберігатися у фрагментах по 8 МБ.

Коли ви вперше налаштовуєте захист від втрати пароля, створюється ключ відновлення, який необхідно запам'ятати, оскільки в разі втрати коду дані не можуть бути відновлені. Крім того, ви можете налаштувати скидання пароля за допомогою облікового запису iCloud. Після активації FileVault процес завантаження комп’ютера змінюється з міркувань безпеки. Якщо раніше пароль потрібно було вводити після

завантаження облікового запису, то зараз це доходить до того, що виключається навіть потенційна можливість скидання пароля користувача будь-яким із відомих способів.

Недоліки FileVault:

1. шифрування за допомогою FileVault дуже серйозно впливає на продуктивність Mac;
2. дані неможливо відновити, якщо пароль і ключ відновлення забуті;
3. якщо накопичувач виходить з ладу, дані також будуть втрачені назавжди;
4. зашифровані копії Time Machine не можуть відновити певний файл, лише повну копію.

2.1.3 LUKS

Linux Unified Key Setup (LUKS) — це система шифрування диска, яка зберігає дані в зашифрованому фізичному розділі. Модуль ядра dm-crypt допомагає зашифрувати їх, що дозволяє створити віртуальний блоковий пристрій, з яким взаємодіє користувач. Якщо користувач хоче записати дані на віртуальний пристрій, вони шифруються на льоту та записуються на диск. Якщо він хоче читати з віртуального пристрою, дані, що зберігаються на фізичному диску, розшифровуються та передаються користувачеві через віртуальний диск у відкритому вигляді. Дані залишаються захищеними, навіть якщо диск приєднати до іншого комп'ютера. Завданням створення специфікації LUKS була стандартизація системи керування ключами для розділу диска, доступ до якого здійснюється за допомогою потокового шифрування. Пріоритет надавався безпеці всіх етапів роботи з ключами в умовах використання обладнання, доступного пересічному користувачеві [14].

LUKS вважається еталонною реалізацією моделі TKS2 і, як такий, розроблений, щоб забезпечити наступні переваги моделі:

- підтримка кількох паролів, які використовуються для доступу до одного зашифрованого диска, з можливістю безпечного додавання, зміни та видалення будь-якого ключа за запитом користувача;
- захист від відновлення доступу з видаленим паролем після його видалення;
- прийнятний ступінь стійкості до грубої сили, навіть якщо використовуються слабкі паролі з низькою ентропією.

2.1.4 Bitlocker

Bitlocker — це функція шифрування диска Windows, призначена для захисту ваших даних шляхом шифрування всіх ваших томів. BitLocker усуває загрозу викрадення або розкриття даних із втрачених, викрадених або неправильно виведених з експлуатації пристроїв. BitLocker забезпечує максимальний захист при використанні з довіреним платформним модулем TPM.

TPM — це апаратний компонент, встановлений на багатьох пристроях, який працює з BitLocker для захисту даних користувача та захисту даних, коли система перебуває в автономному режимі. Переносячи операції шифрування на апаратне забезпечення, зашифровані жорсткі диски покращують продуктивність BitLocker і зменшують використання процесора та споживання енергії. Зашифровані жорсткі диски швидко шифрують дані, дозволяючи корпоративним пристроям розгортати BitLocker з мінімальним впливом на продуктивність.

Зашифровані жорсткі диски – це новий клас жорстких дисків, які само шифруються на апаратному рівні та забезпечують повне апаратне шифрування диска. Починаючи з Windows 8 і Windows Server 2012, зашифровані жорсткі диски можна використовувати без будь-яких додаткових змін [15].

Зашифровані жорсткі диски забезпечують:

1. **покращена продуктивність**: інтегрований у контролер диска пристрій шифрування дозволяє працювати на повній швидкості передачі даних без шкоди для продуктивності;
2. **надійна безпек:** шифрування завжди «ввімкнено», і ключ шифрування ніколи не залишає ваш жорсткий диск. Перевірка автентифікації користувача виконується диском перед його розблокуванням;
3. **простоте використання**: шифрування прозоре для користувачів; користувачам не потрібно вмикати шифрування. Зашифровані жорсткі диски легко стираються за допомогою вбудованих ключів шифрування. Нема потреби повторно шифрувати дані на диску;
4. **зменшена вартість володіння**: BitLocker використовує наявну інфраструктуру для зберігання інформації про відновлення, тому для керування ключами шифрування не потрібна нова інфраструктура. Процес шифрування не вимагає ресурсів центрального процесора, тому ваш пристрій працює ефективніше.

Зашифровані жорсткі диски спочатку підтримуються операційними системами за допомогою таких механізмів:

1. **ідентифікація**: операційна система визначає, що диск є певним типом зашифрованого жорсткого диска;

2. **активація**: утиліти керування дисками операційної системи активують і створюють томи та відповідно відображають їх у діапазонах;

3. **API підтримка:** керування зашифрованими жорсткими дисками незалежно від BitLocker Drive Encryption.

Виконати процедуру активації BitLocker на зазначеному носії даних шляхом виконання дії "Увімкнути BitLocker" на цьому носії (рис. 2.1). Передбачається, що дана операція спрямована на встановлення шифрування пристрою з використанням програмного забезпечення BitLocker, що забезпечує безпеку даних та обмеження доступу до них шляхом застосування шифрування на рівні пристрою.

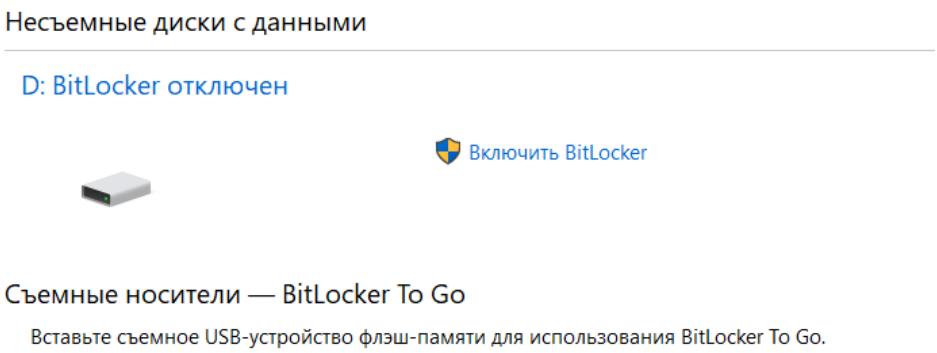


Рисунок 2.1 – Шифрування диску D

Рекомендовано обрати опцію "Використовувати пароль для розблокування диска" та надати відповідний пароль відповідно до вимог безпеки (рис. 2.2). Ця процедура передбачає використання парольного механізму як одного з методів аутентифікації для розблокування зазначеного диска (рис. 2.6).

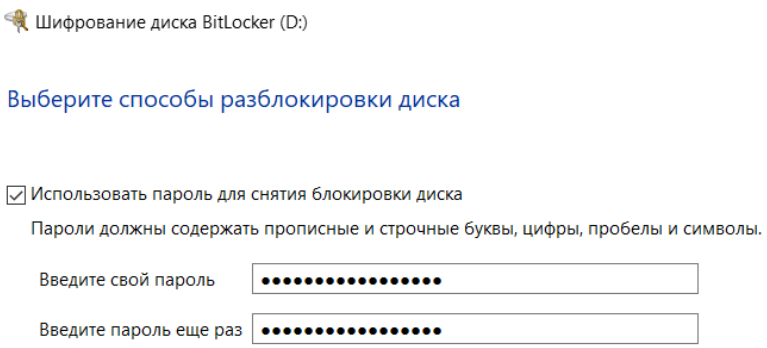


Рисунок 2.2 – Встановлення паролю

Обрати опцію "Як створити резервну копію ключа відновлення" (рис. 2.3). Ключ відновлення у цьому контексті є унікальним 48-значним числовим паролем, який має свою особливу природу. У випадку втрати основного пароля, користувач має можливість використовувати ключ відновлення для отримання доступу до засобів зберігання даних (рис. 2.7). Також слід відзначити, що операційна система Windows вимагатиме ключа відновлення BitLocker, якщо буде виявлено підозрілу або небезпечну активність, яка може вказувати на намагання несанкціонованого доступу до конфіденційної інформації.

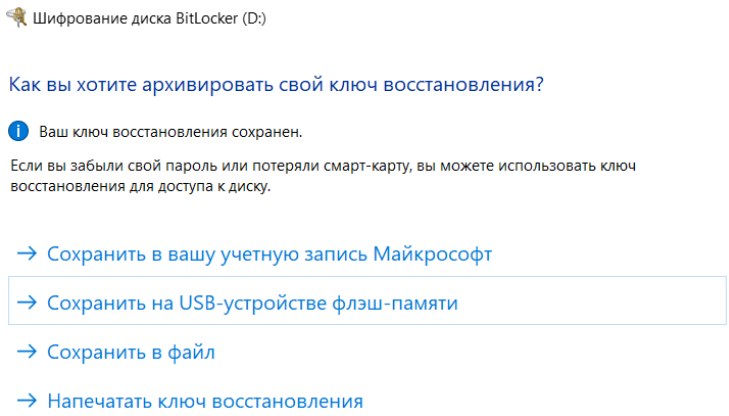


Рисунок 2.3 – Створення USB-key

Вибору обсягу дискового простору для зашифрування (рис. 2.4). Зазначена процедура передбачає можливість вибору обсягу даних на носії, який підлягатиме процесу шифрування.

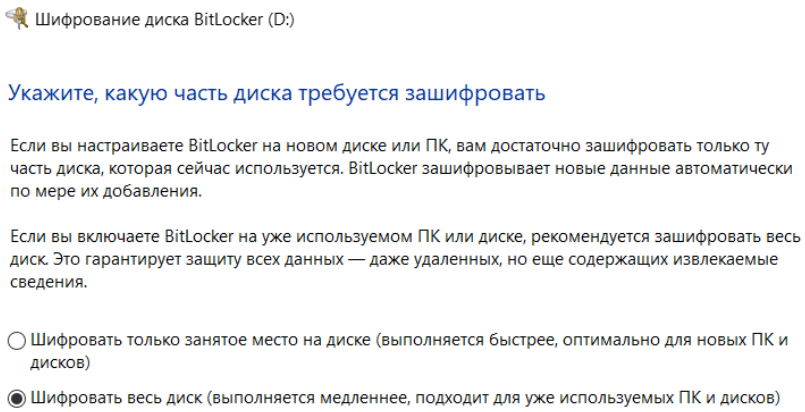


Рисунок 2.4 – Шифровка всього диску

Було створено резервний ключ відновлення та збережено його на зовнішньому носії інформації у формі USB-пристрою (рис. 2.5).

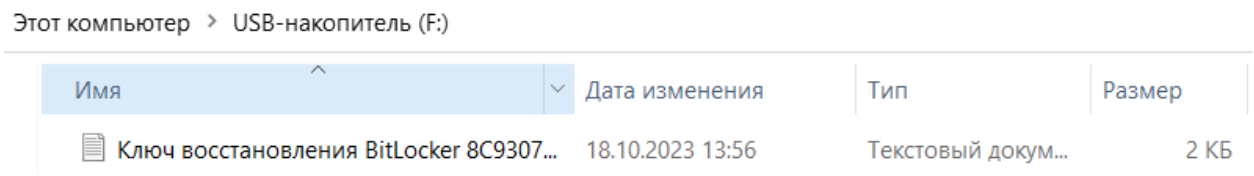


Рисунок 2.5 – Резервний ключ відновлення

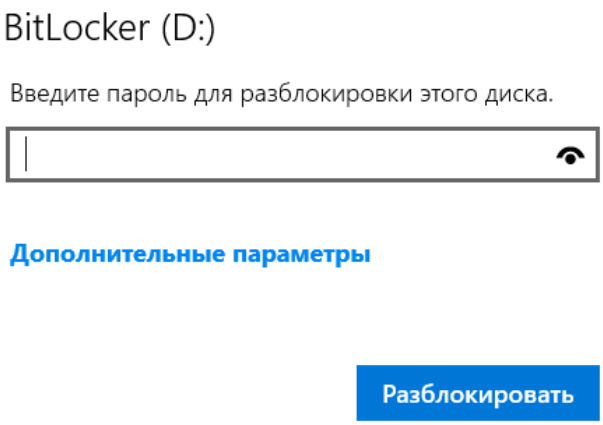


Рисунок 2.6 – Використання паролю для розблокування диску

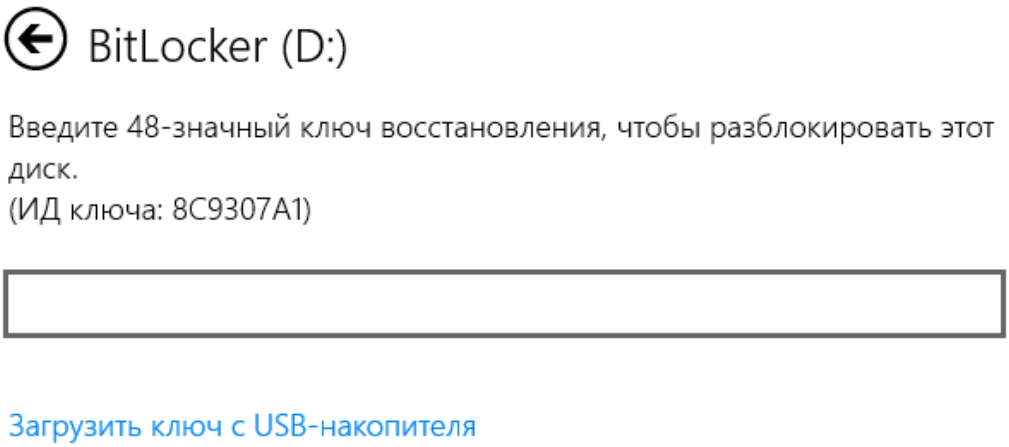


Рисунок 2.7 – Використання USB-key для розблокування диску

2.2 Засоби конфіденційного зв'язку

2.2.1 Telegram

Що стосується безпеки та конфіденційності, Telegram набагато кращий за такі популярні програми обміну повідомленнями, як WhatsApp і Facebook Messenger. Telegram є відкритим кодом, і весь його код доступний онлайн для перевірки експертами з безпеки. Однак за замовчуванням Telegram пропонує лише шифрування сервера. Telegram синхронізує ваші дані між усіма вашими пристроями та завантажує свої сервери за умовчанням. Усі дані на серверах Telegram зашифровані. Це означає, що ви захищені від постачальників послуг Інтернету, підслуховування Wi-Fi та інших третіх осіб. З іншого боку, якщо ви бажаєте наскрізне шифрування, де ніхто, включаючи Telegram, не може отримати доступ до ваших даних, вам потрібно перейти в інший режим, який називається секретним чатом.

Режим секретного чату працює для окремих розмов, а не для груп. Після входу в режим секретного чату Telegram пропонує наскрізне шифрування. Це означає, що лише ви та одержувач можете читати ваші повідомлення, оскільки вони зберігаються лише на вашому пристрої, а не на серверах Telegram. Повідомлення, що містять Telegram, неможливо отримати або розшифрувати. Повідомлення в секретних чатах не можна пересилати, і одержувач повідомить про це, якщо ви зробите знімок екрана. Видалення повідомлення видаляє його для обох користувачів.

2.2.2 Signal

Signal розроблений не для збору конфіденційної інформації. Сторонні особи не можуть отримати доступ до ваших сигнальних повідомлень або дзвінків, оскільки вони завжди зашифровані, приватні та захищені наскрізним шифруванням. Тобто лише призначені одержувачі можуть їх прочитати або почути.

Конфіденційність не є необов’язковим режимом – це просто спосіб роботи Signal. Ви завжди можете бути впевнені, що ваші сервери працюють належним чином і що ви спілкуєтеся з потрібними людьми за допомогою свого номера безпеки. Повний вихідний код для клієнтів і сервера Signal доступний на GitHub. Це дозволяє зацікавленим сторонам перевірити безпеку та правильність коду.

Ось як переконатися, що ваші повідомлення є конфіденційними:

1. Signal повідомлення буде записаний у полі введення повідомлення;
2. після введення повідомлення значок надсилання стане синім, а замок буде закрито;
3. в Signal голосові дзвінки та відеодзвінки мають особливий вигляд і текст Signal на екрані.

Як виглядає незахищена розмова:

1. Signal використовувався для підтримки незахищених SMS/MMS, але тепер цю опцію вилучено;
2. усі незахищені SMS/MMS будуть позначені піктограмою відкритого замка в хмарі повідомлень;
3. додаткова інформація не відображається під час взаємодії між користувачами Signal.

2.2.3 Outlook з використанням Kleopatra

Outlook — це універсальний інструмент і універсальний органайзер для роботи з електронним дизайном, функціями календаря, планувальником, блокнотом та менеджером контактів. Крім того, Outlook може надати роботу документи Microsoft Office для автоматичного створення гаманця робота. Outlook може використовувати окрему програму, тому мені, як клієнту Microsoft Exchange Server, довелося додати нові функції до організації, спільною для користувачів: екран загальної пошти, папки завдань, календар, зустрічі, планування та бронювання, час та документальне погодження загальних зборів акціонерів.

Microsoft Outlook і Microsoft Exchange Server — це платформи для організації документообігу, які не входять до стандартної комплектації, але можуть бути запрограмовані з додатковими функціями для документообігу. Використовуйте розділ «Довідка», щоб знайти інформацію про деякі надбудови для Microsoft Outlook.

Kleopatra — це багатофункціональна графічна програма C++ / QT/ KDE, яка дозволяє підписувати та шифрувати файли, а також забезпечує створення, зберігання та керування сертифікатами та ключами шифрування. Kleopatra розробляється як частина середовища робочого столу KDE, підтримує управління сертифікатами X.509 і OpenPGP в ключах шифрування GnuPG, основні функції програми реалізовані на криптографічній бібліотеці libkleo за допомогою функціональності GnuPG Made Easy. GPG / GnuPG (GNU Privacy Guard) —

консольна утиліта для шифрування інформації та створення електронних цифрових підписів за допомогою різних алгоритмів (RSA, DSA, AES тощо). Утиліта була створена як безплатна альтернатива пропрієтарному PGP і повністю сумісна зі стандартом IETF OpenPGP.

Крок 1: Вибір опції "Файл" та подальший вибір "Створити пару ключів" відкриває можливість генерації ключової пари, що складається з одного закритого ключа (приватного ключа), який доступний лише власнику, і одного публічного ключа, який може бути розповсюджений іншим користувачам для шифрування повідомлень, надсиланих вам.

Крок 2: Далі обирається тип ключів OpenPGP для створення особистої пари ключів. В наступному віконці вводяться особисті ідентифікатори, такі як ім'я та електронна пошта (рис. 2.8). Також надається можливість встановити строк дії сертифіката та вибір способу шифровки (рис. 2.9).

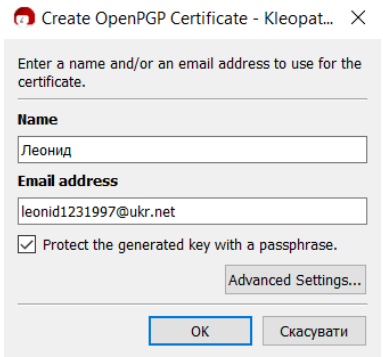


Рисунок 2.8 – Введення ідентифікатора

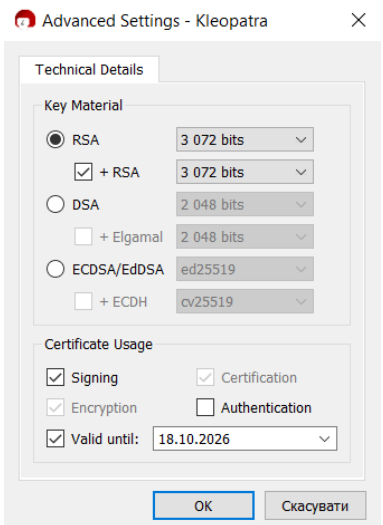


Рисунок 2.9 – Додаткові параметри

Крок 3: Після успішного завершення всіх цих кроків, на екрані відображається повідомлення "Новий OpenPGP сертифікат був успішно створений"(рис. 2.10).

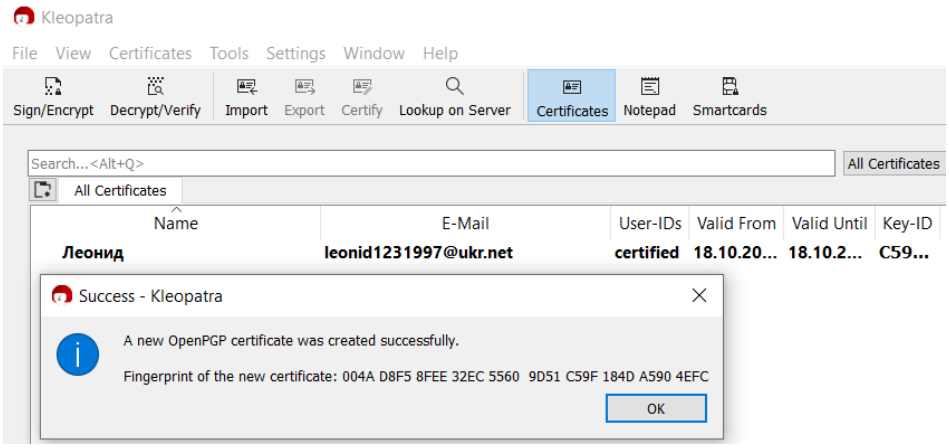


Рисунок 2.10 – Успішне створення сертифікату

На цьому етапі можна виконати резервне копіювання ключів, наприклад, зберегти їх на зовнішньому носії, такому як флешпам'ять. Також можна надіслати відкритий ключ іншим користувачам електронною поштою.

Завершення останнього кроку означає, що електронні листи, навіть якщо сторонні особи мають доступ до вашої електронної адреси та паролю, не можуть бути прочитані з інших комп'ютерів. Доступ до таких листів можливий лише з комп'ютера, на якому зберігається відповідний сертифікат (рис. 2.11, рис. 2.12).

Ця функціональність може бути корисною в ситуаціях, коли конфіденційні листи надсилаються до організацій, де збереження конфіденційності має високий пріоритет.

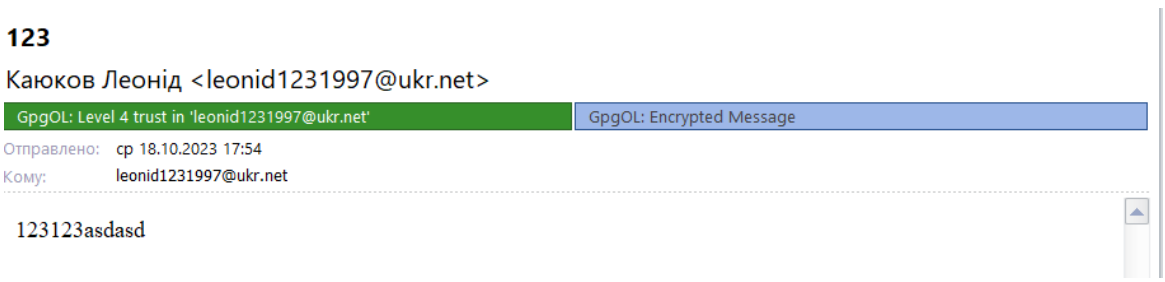


Рисунок 2.11 – Комп'ютер з відповідним сертифікатом

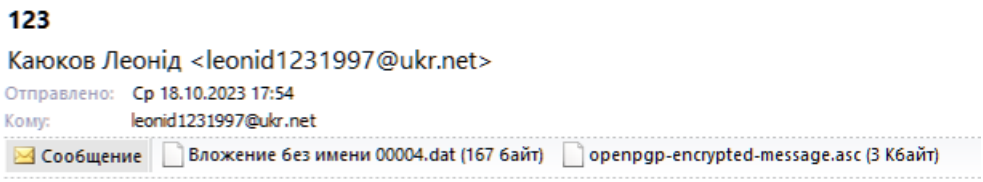


Рисунок 2.12 – Комп'ютер без сертифікату

3 РОЗРОБКА ТА АНАЛІЗ КРИПТОГРАФІЧНИХ АЛГОРИТМІВ З ВИКОРИСТАННЯМ .NET

3.1 Основні компоненти коду

3.1.1 Бібліотеки .NET

Опис бібліотек яких було використано в мові програмування C#:

1. **System:** це простір імен базових класів .NET Framework. Містить фундаментальні класи та типи для основних операцій програмування, таких як робота з рядками, конвертація типів, операції введення/виведення та інші загальні функції;
2. **System.IO:** містить класи для взаємодії з файловою системою. Надає функціонал для читання та запису в файли, роботи з каталогами та інші операції введення/виведення;
3. **System.Numerics:** містить типи та класи для виконання операцій з великими числами (як цілими, так і дійсними), векторами та матрицями. Ця бібліотека корисна для роботи з великими числовими значеннями та математичними операціями;
4. **System.Security:** містить класи для роботи з загальними операціями забезпечення безпеки, такі як аутентифікація, контроль доступу та інші;
5. **System.Security.Cryptography:** надає функції для виконання криптографічних операцій, таких як шифрування, розшифрування та хешування даних [1,2];
6. **System.Collections.Generic:** містить загальні класи та інтерфейси для роботи з колекціями даних в .NET, зокрема для використання більш безпечних та ефективних типізованих колекцій;
7. **System.Text:** надає класи для роботи з рядками та текстовими операціями, включаючи кодування, декодування та роботу з символьними послідовностями;
8. **System.Linq:** це простір імен, який містить розширення мови запитів LINQ (Language-Integrated Query). Надає методи розширень для колекцій та інших типів даних для виконання запитів та операцій фільтрації, сортування та проекції даних.

Кожна з цих бібліотек має свою специфічну функціональність і призначена для вирішення певних завдань у програмуванні на платформі .NET, забезпечуючи розмаїття інструментів для різноманітних операцій у різних галузях розробки програм.

3.1.2 Головне і вторинне меню

Використовуючи цикл while, який працює в безкінечному режимі while (true). Цикл виводить головне меню програми у консольному інтерфейсі, де користувач може вибрати різні опції. Програма очікує введення користувача та реагує на вибір, виконуючи відповідні дії залежно від обраної опції.

Кожен пункт меню має свій номер і пов'язані з ним дії. Наприклад:

1. симетричне шифрування: AES;
2. асиметричне шифрування: RSA;
3. створення та перевірка ЕЦП для файлу;
4. хеш-функції: SHA-256, SHA-384, та SHA-512;
5. аутентифікація та обмін ключами: Diffie-Hellman Key Exchange;
6. вихід з програми.

Після вибору користувачем пункту меню програма викликає відповідні функції (KaMenu, Ka2Menu, Ka2_1Menu, Ka2_2Menu, Ka2_3Menu), які відповідають за виконання функціоналу, пов'язаного з кожним пунктом меню (рис. 3.1).

Коли користувач вибирає "0. Вихід з програми", виводиться повідомлення про завершення програми "Дякую за використання програми. До побачення!" і виконання циклу завершується за допомогою return, що припиняє безкінечний цикл і програма завершує свою роботу.

```
while (true)
{
    Console.WriteLine("Головне меню:");
    Console.WriteLine("1. Симетричне шифрування: AES");
    Console.WriteLine("2. Асиметричне шифрування: RSA");
    Console.WriteLine("3. Створення та перевірка ЕЦП для файлу");
    Console.WriteLine("4. Хеш-функції: SHA-256, SHA-384, та SHA-512");
    Console.WriteLine("5. Аутентифікація та обмін ключами: Diffie-Hellman Key Exchange");
    Console.WriteLine("0. Вихід з програми");
    Console.Write("Введіть номер опції: ");
    string userInput = Console.ReadLine();
    switch (userInput)
    {
        case "1":
            menuStack.Push("1");
            KaMenu(menuStack);
            break;
        case "2":
            menuStack.Push("2");
            Ka2Menu(menuStack);
            break;
        case "3":
            menuStack.Push("3");
            Ka2_1Menu(menuStack);
            break;
        case "4":
            menuStack.Push("4");
            Ka2_2Menu(menuStack);
            break;
        case "5":
            menuStack.Push("5");
            Ka2_3Menu(menuStack);
            break;
        case "0":
            Console.WriteLine("Дякую за використання програми. До побачення!");
            return;
        default:
            Console.WriteLine("Невірний вибір. Будь ласка, виберіть дійсний номер опції.");
            break;
    }
}
```

Рисунок 3.1 – Головне меню

Після вибору відповідного пункту в головному меню відкривається вторинне меню в якому і відбувається основні процеси коду.

Вторинне меню AES:

1. зашифрувати і зберегти файл;
2. розшифрувати і зберегти файл;
3. перевірити цілісність даних після розшифрування;
4. повернутися до головного меню;
5. вийти з програми.

Вторинне меню RSA:

1. зашифрувати файл за допомогою алгоритму RSA;
2. розшифрувати файл за допомогою приватного ключа;
3. перевірити цілісність файлу;
4. повернутися до головного меню;
5. вийти з програми.

Вторинне меню ЕЦП:

1. створення об'єкта, що містить дані для підпису;
2. добавляємо підпис до файлу;
3. перевірка підпису в файлі;
4. видалення підпису з файла;
5. повернутися до головного меню ;
6. вийти з програми.

Вторинне меню Хеш-функції:

1. хеш-функції: SHA-256;
2. хеш-функції: SHA-384;
3. хеш-функції: SHA-512;
4. повернутися до головного меню;
5. вийти з програми.

Вторинне меню Diffie-Hellman Key Exchange:

1. аутентифікація та обмін ключами: Diffie-Hellman Key Exchange
2. повернутися до головного меню;
3. вийти з програми.

3.2 Програмна реалізація алгоритму AES засобами .NET

Використовуючи функцію **EncryptFile** (рис. 3.2), яка зашифровує дані файлу **fileBytes** за допомогою заданого ключа та вектора ініціалізації. Результат цього шифрування зберігається у змінній **encryptedBytes**. Створюється повний шлях до файлу на робочому столі. Використовується метод Path.Combine для створення шляху до файлу з базовим шляхом робочого столу **desktopPath** та ім'ям файлу "coder_document.txt". Використовується метод **File.WriteAllBytes**, щоб зберегти зашифровані дані у файл на робочому столі за вказаним шляхом **desktopFilePath** [1,2]. Вміст файлу, який був зашифрований та збережений у **encryptedBytes**, тепер записується у новий файл на робочому столі з назвою "coder_document.txt" (рис. 3.3). Результати зашифровки продемонстровані в (рис. 3.4 і рис. 3.5).

```
// Метод для шифрування файлу
static byte[] EncryptFile(byte[] plainBytes, byte[] key, byte[] IV)
{
    using (Aes aesAlg = Aes.Create())
    {
        aesAlg.Key = key;
        aesAlg.IV = IV;
        // Створення об'єкта, який виконує процес шифрування
        ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
        using (MemoryStream msEncrypt = new MemoryStream())
        {
            using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write))
            {
                // Запис зашифрованих даних в потік
                csEncrypt.Write(plainBytes, 0, plainBytes.Length);
            }
            // Повернення зашифрованих даних у вигляді масиву байтів
            return msEncrypt.ToArray();
        }
    }
}
```

Рисунок 3.2 – Функція EncryptFile для зашифрування

```
case "1":
    // Опція 1: Зашифрувати файл і зберегти
    Console.WriteLine("Ви вибрали опцію 1.");
    encryptedBytes = EncryptFile(fileBytes, key, IV);
    // Створити повний шлях до файлу на робочому столі
    string desktopFilePath = Path.Combine(desktopPath, "coder_document.txt");
    // Зберегти зашифрований файл на робочому столі
    File.WriteAllBytes(desktopFilePath, encryptedBytes);
    Console.WriteLine("Файл успішно зашифровано і збережено на робочому столі.");
```

Рисунок 3.3 – Зашифрування файлу з функцією EncryptFile

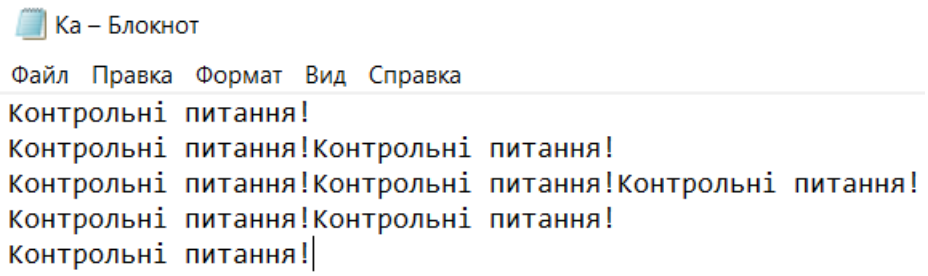


Рисунок 3.4 – Незашифрований файл

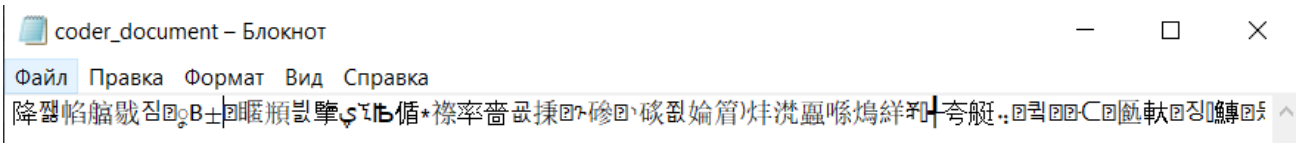


Рисунок 3.5 – Зашифрований файл

Далі створюється рядок **filePath2**, який містить повний шлях до файлу, який буде розшифровуватися. Файл, який зберігається у **filePath**2, зчитується у байтовий масив за допомогою **File.ReadAllBytes**. Після цього викликається функція DecryptFile, що розшифровує цей байтовий масив **fileBytes** за допомогою ключа **key** та вектора ініціалізації **IV** (рис. 3.6).

Результат розшифрування зберігається у змінній **decryptedBytes**. Рядок **desktopFilePath2**, містить повний шлях до розшифрованого файлу, який буде збережений. Використовуючи метод **File.WriteAllBytes**, щоб зберегти розшифровані дані у файл на робочому столі за вказаним шляхом **desktopFilePath2** [1,2]. Вміст файлу, який був розшифрований та збережений у **decryptedBytes**, тепер записується у новий файл на робочому столі з назвою "decoder_document.txt" (рис. 3.7). Результати розшифровки продемонстровані в (рис. 3.8).

```
// Метод для розшифрування файлу
static byte[] DecryptFile(byte[] encryptedBytes, byte[] key, byte[] IV)
{
    using (Aes aesAlg = Aes.Create())
    {
        aesAlg.Key = key;
        aesAlg.IV = IV;
        // Створення об'єкта, який виконує процес розшифрування
        ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
        using (MemoryStream msDecrypt = new MemoryStream(encryptedBytes))
        {
            using (CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))
            {
                using (MemoryStream msPlainText = new MemoryStream())
                {
                    csDecrypt.CopyTo(msPlainText);
                    // Повернути розшифрований текст як масив байтів
                    return msPlainText.ToArray();
                }
            }
        }
    }
}
```

Рисунок 3.6 – Функція DecryptFile для розшифрування

```
case "2":
    // Опція 2: Розшифрувати файл і зберегти
    Console.WriteLine("Ви вибрали опцію 2.");
    // Файл, який буде розшифровуватися
    string filePath2 = Path.Combine(desktopPath, "coder_document.txt");
    // Розшифрувати файл
    decryptedBytes = DecryptFile(File.ReadAllBytes(filePath2), key, IV);
    // Створити повний шлях до файлу на робочому столі
    string desktopFilePath2 = Path.Combine(desktopPath, "decoder_document.txt");
    // Зберегти розшифрований файл на робочому столі
    File.WriteAllBytes(desktopFilePath2, decryptedBytes);
    Console.WriteLine("Файл успішно розшифровано і збережено на робочому столі.");
```

Рисунок 3.7 – Розшифрування файлу з функцією DecryptFile

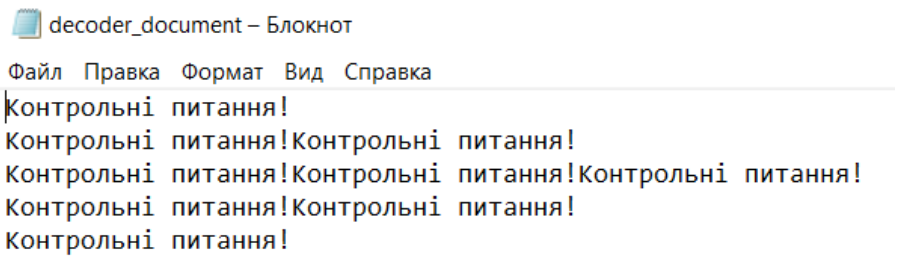


Рисунок 3.8 – Розшифрований файл

Далі здійснюється виклик функції **VerifyIntegrity**, яка припускає перевірку цілісності розшифрованих даних **decryptedBytes** (рис. 3.9). Це може бути важливо

для забезпечення, що розшифровані дані не були ушкоджені або змінені під час передачі чи зберігання. Функція, ймовірно, приймає розшифровані дані та шлях до вихідного зашифрованого файлу **filePath**. В залежності від результату перевірки цілісності **isIntegrityVerified**, програма виводить відповідне повідомлення у консоль. Якщо цілісність підтверджена, виводиться повідомлення, що "Цілісність документа після розшифрування підтверджена. Інформація не була змінена.". В іншому випадку, якщо цілісність не підтверджена, виводиться повідомлення про те, що "Увага! Інформація була змінена після розшифрування." (рис. 3.10). Результати перевірки цілісності файлу продемонстровані в (рис. 3.11).

```
// Метод для перевірки цілісності розшифрованого тексту
static bool VerifyIntegrity(byte[] decryptedBytes, string filePath)
{
    byte[] originalFileBytes = File.ReadAllBytes(filePath);

    // Порівняти розшифрований текст і оригінальний файл за допомогою SequenceEqual
    return StructuralComparisons.StructuralEqualityComparer.Equals(decryptedBytes, originalFileBytes);
}
```

Рисунок 3.9 – Функція VerifyIntegrity для перевірки цілісності

```
case "3":
    // Опція 3: Перевірити цілісність розшифрованого тексту
    Console.WriteLine("Ви вибрали опцію 3.");
    bool isIntegrityVerified = VerifyIntegrity(decryptedBytes, filePath);

    if (isIntegrityVerified)
    {
        Console.WriteLine("Цілісність документа після розшифрування підтверджена. Інформація не була змінена.");
    }
    else
    {
        Console.WriteLine("Увага! Інформація була змінена після розшифрування.");
    }
```

Рисунок 3.10 – Перевірка цілісності файлу

```
Меню:
1. Зашифрувати і зберегти файл
2. Розшифрувати і зберегти файл
3. Перевірити цілісність даних після розшифрування
~. Повернутися до головного меню
0. Вийти з програми
Введіть номер опції: 3
_____________________________________________________
Ви вибрали опцію 3.
Цілісність документа після розшифрування підтверджена. Інформація не була змінена.
_____________________________________________________
```

Рисунок 3.11 – Результати перевірки цілісності

3.3 Програмна реалізація алгоритму RSA засобами .NET

Оголосивши дві змінні типу **RSAParameters**, які призначені для збереження публічного **publicKey** і приватного **privateKey** ключів **RSA**. Відбувається перевірка наявності приватного ключа у файлі "private_key.txt" на робочому столі. Якщо файл існує, приватний ключ завантажується за допомогою функції **LoadKeyFromFile** (рис. 3.12).

У випадку, якщо файл не існує, генерується нова пара ключів (приватний і публічний ключі **RSA**) за допомогою об'єкту **RSACryptoServiceProvider**. Публічний ключ publicKey отримується шляхом експорту публічних параметрів, тоді як приватний ключ **privateKey** отримується експортом приватних параметрів. Приватний ключ також зберігається у файл "private_key.txt" за допомогою функції **SaveKeyToFile** (рис. 3.13 і рис. 3.14). Створений приватний ключ зображений на (рис. 3.15) [1,2].

```
// Завантаження ключа із файлу
static RSAParameters LoadKeyFromFile(string filePath)
{
    RSAParameters key = new RSAParameters();
    using (StreamReader sr = new StreamReader(filePath))
    {
        key.Modulus = Convert.FromBase64String(sr.ReadLine());
        key.Exponent = Convert.FromBase64String(sr.ReadLine());
        key.D = Convert.FromBase64String(sr.ReadLine());
        key.P = Convert.FromBase64String(sr.ReadLine());
        key.Q = Convert.FromBase64String(sr.ReadLine());
        key.DP = Convert.FromBase64String(sr.ReadLine());
        key.DQ = Convert.FromBase64String(sr.ReadLine());
        key.InverseQ = Convert.FromBase64String(sr.ReadLine());
    }
    return key;
}
```

Рисунок 3.12 – Завантаження ключа із файлу

```
// Збереження ключа у файл
static void SaveKeyToFile(string filePath, RSAParameters key)
{
    using (StreamWriter sw = new StreamWriter(filePath))
    {
        sw.WriteLine(Convert.ToBase64String(key.Modulus));
        sw.WriteLine(Convert.ToBase64String(key.Exponent));
        sw.WriteLine(Convert.ToBase64String(key.D));
        sw.WriteLine(Convert.ToBase64String(key.P));
        sw.WriteLine(Convert.ToBase64String(key.Q));
        sw.WriteLine(Convert.ToBase64String(key.DP));
        sw.WriteLine(Convert.ToBase64String(key.DQ));
        sw.WriteLine(Convert.ToBase64String(key.InverseQ));
    }
}
```

Рисунок 3.13 – Збереження ключа у файл

```
RSAParameters publicKey;
RSAParameters privateKey;
// Завантаження приватного ключа, якщо він вже існує
string privateKeyFilePath = Path.Combine(desktopPath, "private_key.txt");
if (File.Exists(privateKeyFilePath))
{
    privateKey = LoadKeyFromFile(privateKeyFilePath);
    publicKey = new RSAParameters(); // Ініціалізуємо пустий об'єкт RSAParameters для уникнення помилок
}
else
{
    // Генерація пари ключів (приватний і публічний ключі RSA)
    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider())
    {
        // Публічний ключ RSA
        publicKey = rsa.ExportParameters(false);

        // Приватний ключ RSA
        privateKey = rsa.ExportParameters(true);
        // Збереження приватного ключа у файл
        SaveKeyToFile(privateKeyFilePath, privateKey);
    }
}
```

Рисунок 3.14 – Створення приватного та публічного ключа

Рисунок 3.15 – Приватний ключ збережений в файлі

Використовуючи функцію **EncryptFile**, яка відповідає за зашифрування файлу (рис. 3.16) [1,2]. Параметри функції:

1. **originalFilePath**: Шлях до оригінального файлу, який потрібно зашифрувати;
2. **encryptedFilePath**: Шлях, куди буде збережено зашифрований файл;
3. **publicKey**: Публічний ключ, що використовується для зашифрування.

Функція **EncryptFile** виконує процес зашифрування файлу за допомогою публічного ключа **publicKey**. Після завершення шифрування зашифрований файл зберігається за вказаним шляхом **encryptedFilePath**. Після завершення процесу зашифрування виводиться повідомлення у консоль про те, що файл був успішно зашифрований і збережений на робочому столі (рис. 3.17). Результати зашифровки продемонстровані в (рис. 3.18).

```
// Зашифрування файлу за допомогою алгоритму RSA
static void EncryptFile(string inputFile, string outputFile, RSAParameters publicKey)
{
    using (RSACryptoServiceProvider rsaEncrypt = new RSACryptoServiceProvider())
    {
        rsaEncrypt.ImportParameters(publicKey);
        using (FileStream inputStream = File.OpenRead(inputFile))
        using (FileStream outputStream = File.Create(outputFile))
        {
            int keySize = rsaEncrypt.KeySize / 8;
            int blockSize = keySize - 11; // Розмір блока для RSA PKCS#1 v1.5
            int bufferSize = blockSize;
            byte[] buffer = new byte[bufferSize];
            int bytesRead;
            while ((bytesRead = inputStream.Read(buffer, 0, bufferSize)) > 0)
            {
                byte[] encryptedBytes = rsaEncrypt.Encrypt(buffer.Take(bytesRead).ToArray(), false);
                outputStream.Write(encryptedBytes, 0, encryptedBytes.Length);
            }
        }
    }
}
```

Рисунок 3.16 – Функція EncryptFile для зашифрування

```
case "1":
    // Опція 1: Зашифрувати файл і зберегти
    Console.WriteLine("Ви вибрали опцію 1.");
    // Зашифрування файлу
    EncryptFile(originalFilePath, encryptedFilePath, publicKey);
    Console.WriteLine("Файл успішно зашифровано і збережено на робочому столі.");
```

Рисунок 3.17 – Зашифрування файлу з функцією EncryptFile

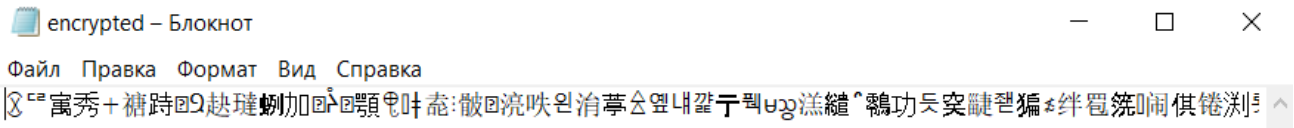


Рисунок 3.18 – Результати зашифровки файлу

Далі перевіряється результат розшифрування файлу за допомогою функції **DecryptFile** (рис. 3.19) [1,2]. Параметри цієї функції:

1. **encryptedFilePath**: Шлях до зашифрованого файлу, який потрібно розшифрувати;
2. **decryptedFilePath**: Шлях, куди буде збережено розшифрований файл;
3. **privateKey**: Приватний ключ, який використовується для розшифрування.

Якщо розшифрування успішне повертається **true**, виводиться повідомлення у консоль про те, що "Файл успішно розшифровано і збережено на робочому столі". У разі, якщо розшифрування невдале повертається **false**, виводиться повідомлення, що "Файл не вдалося розшифрувати і зберегти на робочому столі. Можливо, дані пошкоджено!" (рис. 3.20). Результати розшифровки продемонстровані в (рис. 3.21).

```
// Розшифрування файлу за допомогою приватного ключа
static bool DecryptFile(string inputFile, string outputFile, RSAParameters privateKey)
{
    try
    {
        using (RSACryptoServiceProvider rsaDecrypt = new RSACryptoServiceProvider())
        {
            rsaDecrypt.ImportParameters(privateKey);
            using (FileStream inputStream = File.OpenRead(inputFile))
            using (FileStream outputStream = File.Create(outputFile))
            {
                int keySize = rsaDecrypt.KeySize / 8;
                int blockSize = keySize; // Розмір блока для RSA PKCS#1 v1.5
                int bufferSize = blockSize;
                byte[] buffer = new byte[bufferSize];
                int bytesRead;
                while ((bytesRead = inputStream.Read(buffer, 0, bufferSize)) > 0)
                {
                    byte[] decryptedBytes = rsaDecrypt.Decrypt(buffer.Take(bytesRead).ToArray(), false);
                    outputStream.Write(decryptedBytes, 0, decryptedBytes.Length);
                }
            }
        }
        return true; // Повертаємо true, якщо розшифрування пройшло успішно
    }
    catch (CryptographicException ex)
    {
        Console.WriteLine("Помилка розшифрування: " + ex.Message);
        return false; // Повертаємо false у випадку помилки розшифрування
    }
}
```

Рисунок 3.19 – Функція DecryptFile для розшифрування

```
case "2":
    // Опція 2: Розшифрувати файл і зберегти
    Console.WriteLine("Ви вибрали опцію 2.");
    // Розшифрування файлу
    if (DecryptFile(encryptedFilePath, decryptedFilePath, privateKey))
    {
        // Розшифрування успішне
        Console.WriteLine("Файл успішно розшифровано і збережено на робочому столі.");
        Console.WriteLine("_____");
    }
    else
    {
        // Розшифрування не вдалося (дані були пошкоджені)
        Console.WriteLine("Файл не вдалося розшифрувати і зберегти на робочому столі. Можливо, дані пошкоджено!");
        Console.WriteLine("_____");
    }
    break;
```

Рисунок 3.20 – Розшифрування файлу з функцією DecryptFile

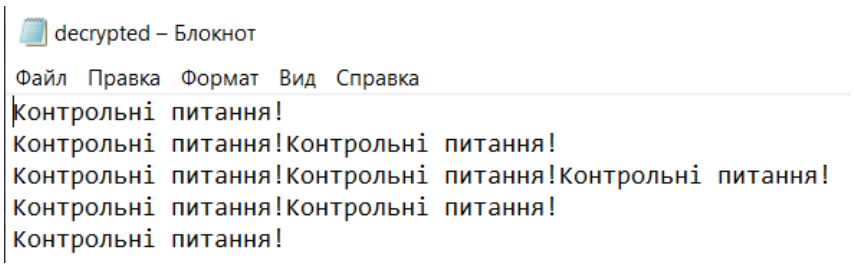


Рисунок 3.21 – Результати розшифровки файлу.

Потім викликається функція **VerifyIntegrity**, яка призначена для перевірки цілісності даних перед розшифруванням (рис. 3.22). Параметри функції:

1. **originalFilePath**: шлях до оригінального файлу, який був зашифрований.
2. **decryptedFilePath**: шлях до розшифрованого файлу.

В залежності від результату перевірки цілісності **isIntegrityVerified**, програма виводить відповідне повідомлення у консоль. Якщо цілісність підтверджена, виводиться повідомлення: "Файл був зашифрований і розшифрований успішно, цілісність даних підтверджена." В іншому випадку, якщо перевірка цілісності не пройшла, виводиться повідомлення: "Перевірка цілісності не пройшла. Дані були змінені." (рис. 3.23). Результати перевірки цілісності файлу продемонстровані в (рис. 3.24).

```
// Перевірка цілісності файлу
static bool VerifyIntegrity(string originalFilePath, string decryptedFilePath)
{
    // Зчитуємо байти обох файлів
    byte[] originalFileBytes = File.ReadAllBytes(originalFilePath);
    byte[] decryptedFileBytes = File.ReadAllBytes(decryptedFilePath);
    // Перевіряємо, чи файли мають однаковий розмір
    if (originalFileBytes.Length != decryptedFileBytes.Length)
    {
        return false;
    }
    // Порівнюємо файли байт-по-байту
    for (int i = 0; i < originalFileBytes.Length; i++)
    {
        if (originalFileBytes[i] != decryptedFileBytes[i])
        {
            return false;
        }
    }
    return true;
}
```

Рисунок 3.22 – Функція VerifyIntegrity для перевірки цілісності файлу

```
case "3":
    // Опція 3: Перевірити цілісність розшифрованого тексту
    Console.WriteLine("Ви вибрали опцію 3.");
    // Перевірка цілісності даних перед розшифруванням
    bool isIntegrityVerified = VerifyIntegrity(originalFilePath, decryptedFilePath);
    if (isIntegrityVerified)
    {
        Console.WriteLine("Файл був зашифрований і розшифрований успішно, і цілісність даних підтверджена.");
    }
    else
    {
        Console.WriteLine("Перевірка цілісності не пройшла. Дані були змінені.");
    }
    Console.WriteLine("_____");
    break;
```

Рисунок 3.23 – Перевірка цілісності файлу використовуючи функцію VerifyIntegrity

```
Меню:
1. Зашифрувати файл за допомогою алгоритму RSA.
2. Розшифрувати файл за допомогою приватного ключа.
3. Перевірити цілісність файлу.
~. Повернутися до головного меню
0. Вийти з програми
Введіть номер опції: 3

Ви вибрали опцію 3.
Файл був зашифрований і розшифрований успішно, і цілісність даних підтверджена.
```

Рисунок 3.24 – Результати перевірки цілісності даних в файлі

3.4 Програмна реалізація ЕЦП засобами .NET

Використовуючи рядок **fullName**, який містить повне ім'я "Kayukov Leonid Leonidovych", а також рядок **date**, який містить поточну дату та час у форматі "yyyy-MM-dd HH:mm:ss". Створюється об'єкт **dataToSignBuilder** типу **StringBuilder**, який використовується для формування даних для підпису. Цей об'єкт використовується для збирання рядків інформації. Додаються рядки, що містять повне ім'я та поточну дату у вигляді "Full Name: [fullName]" та "Date: [date]" (рис. 3.25) [1,2].

```
case "1":
    // Виконати дії для опції 1
    Console.WriteLine("Ви вибрали опцію 1.");
    // Створення об'єкта, що містить дані для підпису
    string fullName = "Kayukov Leonid Leonidovych";
    string date = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
    dataToSignBuilder.AppendLine("Full Name: " + fullName);
    dataToSignBuilder.AppendLine("Date: " + date);
    Console.WriteLine(dataToSignBuilder);
    Console.WriteLine("_____");
    break;
```

Рисунок 3.25 – Створення підпису

Відбувається запит імені файлу з робочого столу користувача та шлях до нього. Потім зчитується введене користувачем ім'я файлу та формується повний шлях до цього файлу на робочому столі. Перевіряється, чи існує файл за вказаним шляхом. Якщо файл не існує, виводиться повідомлення "Файл не існує.". Файл зчитується у вигляді масиву байтів **fileData**. Далі у користувача запитується нова назва файлу, в якому буде збережений підпис. Вміст основного файлу зберігається в новоствореному - за новим шляхом **desktopFilePath**.

Зчитаний вміст файлу конвертується до рядка, додається до попередньо підготовленого об'єкту **dataToSignBuilder**,а потім конвертується назад в масив байтів для підпису **dataToSign**. Використовуючи функцію **SignData** створено підпис на основі підготовлених даних **dataToSign** за допомогою об'єкту rsa (рис. 3.26). Остаточний підпис зберігається в файлі разом із збереженим файлом на робочому столі. Потім виводиться повідомлення про успішне збереження підпису у файлі (рис. 3.27). Результат добавлення підпису до файлу продемонстрований в (рис. 3.28).

```
// Метод для створення ЕЦП для даних
public static byte[] SignData(byte[] data, RSACryptoServiceProvider rsa)
{
    // Створення підпису
    return rsa.SignData(data, CryptoConfig.MapNameToOID("SHA256"));
}
```

Рисунок 3.26 – Функція SignData для підпису даних в файлі

```
case "2":
    // Виконати дії для опції 2
    Console.WriteLine("Ви вибрали опцію 2.");
    // Зчитування файлу
    Console.Write("Введіть назву файла з робочого столу: ");
    string faleName1 = Console.ReadLine();
    string filePath = Path.Combine(desktopPath, faleName1);
    if (!File.Exists(filePath))
    {
        Console.WriteLine("Файл не існує.");
        continue; // або інша логіка, яка вам потрібна
    }
    byte[] fileData = File.ReadAllBytes(filePath);
    // Робимо копію файла на робочому столі
    Console.Write("Введіть нову назву для файла в якому буде збережений підпис: ");
    string faleName2 = Console.ReadLine();
    desktopFilePath = Path.Combine(desktopPath, faleName2);
    File.WriteAllBytes(desktopFilePath, fileData);
    // Зчитування файлу
    byte[] fileCopyData = File.ReadAllBytes(desktopFilePath);
    // Додаємо дані з файлу до об'єкта для підпису
    dataToSignBuilder.Append(Encoding.UTF8.GetString(fileCopyData));
    dataToSign = Encoding.UTF8.GetBytes(dataToSignBuilder.ToString());
    // Підписуємо дані
    signature = SignData(dataToSign, rsa);
    // Зберігаємо підпис у файлі на робочому столі, не перезаписуючи його
    AppendSignatureToFile(desktopFilePath, signature);
    Console.WriteLine("Підпис збережено в файлі на робочому столі.");
    Console.WriteLine("_____________________________________________________");
    break;
```

Рисунок 3.27 – Добавлення підпису до файлу

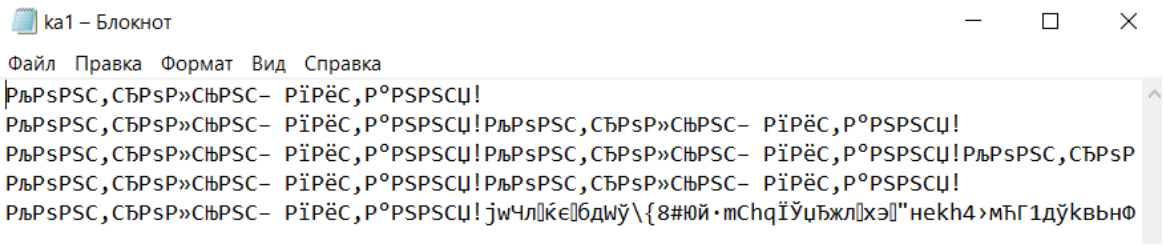


Рисунок 3.28 – Результати добавлення підпису до файлу

Далі викликається функція **HasSignatureInFile** (рис. 3.29), яка призначена для перевірки наявності підпису у файлі за вказаним шляхом **desktopFilePath**,з використанням значення підпису signature. Залежно від результату перевірки (знайдений підпис чи ні), програма виводить відповідне повідомлення у консоль. Якщо підпис знайдено в файлі, виводиться повідомлення "Підпис знайдено в файлі."

В іншому випадку, якщо підпис не знайдено, виводиться повідомлення "Підпис не знайдено в файлі."(рис. 3.30). Результати перевірки наявності підпису в файлі продемонстровано в (рис. 3.31).

```
// Метод для перевірки наявності підпису в файлі
public static bool HasSignatureInFile(string desktopFilePath, byte[] signature)
{
    // Зчитуємо вміст файлу у вигляді байтів
    byte[] fileData = File.ReadAllBytes(desktopFilePath);

    // Перевірка наявності підпису в байтовому масиві
    return FindSignatureIndex(fileData, signature) >= 0;
}
```

Рисунок 3.29 – Функція HasSignatureInFile для перевірки наявності підпису

```
case "3":
    // Виконати дії для опції 3
    Console.WriteLine("Ви вибрали опцію 3.");
    // Перевіряємо наявність підпису в файлі
    bool hasSignature = HasSignatureInFile(desktopFilePath, signature);
    if (hasSignature)
    {
        Console.WriteLine("Підпис знайдено в файлі.");
    }
    else
    {
        Console.WriteLine("Підпис не знайдено в файлі.");
    }
    Console.WriteLine("_____");
    break;
```

Рисунок 3.30 – Перевірки наявності підпису в файлі

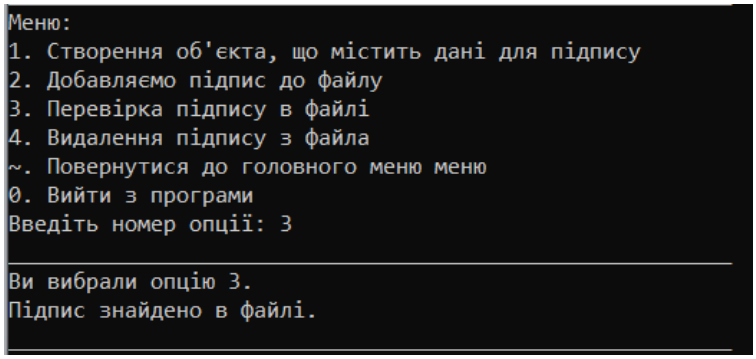


Рисунок 3.31 – Результати перевірки наявності підпису в файлі

Викликаємо функцію **RemoveSignatureFromFile**, яка призначена для видалення підпису з файлу, який знаходиться за шляхом **desktopFilePath**, з використанням значення підпису signature (рис. 3.32).

Функція **RemoveSignatureFromFile** виконує операцію щодо видалення певної частини файлу, що містить сам підпис (рис. 3.33). Результати видалення підпису з файлу продемонстровані в (рис. 3.34).

```
// Метод для видалення підпису з файлу
public static void RemoveSignatureFromFile(string desktopFilePath, byte[] signature)
{
    // Зчитуємо вміст файлу у вигляді байтів
    byte[] fileData = File.ReadAllBytes(desktopFilePath);
    // Пошук позиції підпису у файлі
    int signatureIndex = FindSignatureIndex(fileData, signature);
    if (signatureIndex >= 0)
    {
        // Видаляємо підпис, розширюючи або обрізаючи байтовий масив за необхідністю
        byte[] newData = new byte[fileData.Length - signature.Length];
        Array.Copy(fileData, 0, newData, 0, signatureIndex);
        Array.Copy(fileData, signatureIndex + signature.Length, newData, signatureIndex, fileData.Length - (signatureIndex + signature.Length));

        // Зберігаємо оновлений вміст файлу
        File.WriteAllBytes(desktopFilePath, newData);
        Console.WriteLine("Підпис видалено з файлу.");
    }
    else
    {
        Console.WriteLine("Підпис не знайдено в файлі.");
    }
}
```

Рисунок 3.33 – Функція RemoveSignatureFromFile для видалення підпису

```
case "4":
    // Виконати дії для опції 4
    Console.WriteLine("Ви вибрали опцію 4.");
    // Видаляємо підпис з файлу
    RemoveSignatureFromFile(desktopFilePath, signature);
    Console.WriteLine("_____");
    break;
```

Рисунок 3.34 – Видалення підпису з файлу

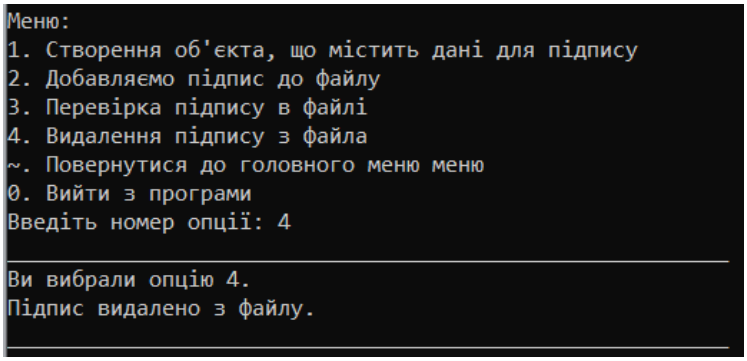


Рисунок 3.34 – Результати видалення підпису з файлу

3.5 Програмна реалізація Хеш-функцій засобами .NET

Спочатку відбувається обчислення хешів рядка inputString за допомогою алгоритмів хешування **SHA-256, SHA-384 і SHA-512**. Потім викликається функція **CalculateSHA256Hash** (рис. 3.35), яка обчислює хеш рядка inputString за допомогою алгоритму хешування **SHA-256**.

Результат зберігається у змінній **sha256Hash**. Аналогічно, викликається функція **CalculateSHA384Hash** (рис. 3.36), **CalculateSHA512Hash** (рис. 3.37) щоб отримати хеш рядка inputString відповідно до алгоритму **SHA-384** і **SHA-512** [1,2]. Результат зберігається у змінній **sha384Hash, sha512Hash** (рис. 3.38).

```
//Використання SHA-256
static string CalculateSHA256Hash(string input)
{
    using (SHA256 sha256 = SHA256.Create())
    {
        // Конвертує введені дані в байтовий масив
        byte[] inputBytes = Encoding.UTF8.GetBytes(input);
        // Обчислює SHA-256 хеш
        byte[] hashBytes = sha256.ComputeHash(inputBytes);
        // Перетворює байти хешу в рядок в шістнадцятковому форматі (hex)
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < hashBytes.Length; i++)
        {
            sb.Append(hashBytes[i].ToString("x2"));
        }
        // Повертає отриманий SHA-256 хеш у вигляді рядка
        return sb.ToString();
    }
}
```

Рисунок 3.35 – Функція CalculateSHA256Hash для обчислення хеш рядка

```
//Використання SHA-384
static string CalculateSHA384Hash(string input)
{
    using (SHA384 sha384 = SHA384.Create())
    {
        // Конвертує введені дані в байтовий масив
        byte[] inputBytes = Encoding.UTF8.GetBytes(input);
        // Обчислює SHA-384 хеш
        byte[] hashBytes = sha384.ComputeHash(inputBytes);
        // Перетворює байти хешу в рядок в шістнадцятковому форматі (hex)
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < hashBytes.Length; i++)
        {
            sb.Append(hashBytes[i].ToString("x2"));
        }
        // Повертає отриманий SHA-384 хеш у вигляді рядка
        return sb.ToString();
    }
}
```

Рисунок 3.36 – Функція CalculateSHA384Hash для обчислення хеш рядка

```
//Використання SHA-512
static string CalculateSHA512Hash(string input)
{
    using (SHA512 sha512 = SHA512.Create())
    {
        // Конвертує введені дані в байтовий масив
        byte[] inputBytes = Encoding.UTF8.GetBytes(input);
        // Обчислює SHA-512 хеш
        byte[] hashBytes = sha512.ComputeHash(inputBytes);
        // Перетворює байти хешу в рядок в шістнадцятковому форматі (hex)
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < hashBytes.Length; i++)
        {
            sb.Append(hashBytes[i].ToString("x2"));
        }
        // Повертає отриманий SHA-512 хеш у вигляді рядка
        return sb.ToString();
    }
}
```

Рисунок 3.37 – Функція CalculateSHA512Hash для обчислення хеш рядка

Формується шлях до файлу "SHA256Hash.txt" на робочому столі. Перевіряється наявність файлу за вказаним шляхом filePath1. Якщо файл вже існує, програма запитує користувача про бажання перезаписати його. Якщо відповідь не "Y" (англ. "Yes"), операція перезапису відміняється та виконується continue, що переходить до наступної ітерації циклу.

Далі зберігається хеш **sha256Hash** у файлі за шляхом **filePath1** використовуючи функцію **SaveHashToFile** (рис. 3.38). Після збереження виводиться повідомлення про успішне збереження хешу. Потім проводиться перевірка цілісності даних, збережених у файлі за шляхом **filePath1** використовуючи функцію **VerifyDataIntegrity** (рис. 3.39). Якщо цілісність підтверджена, виводиться

відповідне повідомлення,а у випадку порушення цілісності - виводиться відповідне повідомлення (рис. 3.40). Результати продемонстровані в (рис. 3.41 і рис 3.42).

```
//Зберегти в файлі
static void SaveHashToFile(string filePath, string data)
{
    using (StreamWriter writer = new StreamWriter(filePath))
    {
        // Збереження даних
        writer.WriteLine(data);

        // Обчислення та збереження хеш-значення
        string hash = CalculateSHA256Hash(data);
        writer.WriteLine($"Hash: {hash}");
    }
}
```

Рисунок 3.38 – Функція SaveHashToFile для збереження sha256Hash у файлі

```
//Перевірка цілісності даних
static bool VerifyDataIntegrity(string filePath)
{
    try
    {
        using (StreamReader reader = new StreamReader(filePath))
        {
            // Зчитування даних
            string data = reader.ReadLine();
            // Зчитування збереженого хеш-значення
            string savedHashLine = reader.ReadLine();
            string savedHash = savedHashLine.Split(' ')[1]; // Отримання хеш-значення з рядка
            // Обчислення хеш-значення для перевірки
            string calculatedHash = CalculateSHA256Hash(data);
            // Порівняння хеш-значень
            return string.Equals(calculatedHash, savedHash, StringComparison.OrdinalIgnoreCase);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Помилка: {ex.Message}");
        return false;
    }
}
```

Рисунок 3.39 – Функція VerifyDataIntegrity для перевірки цілісності даних

```
case "1":
    // Виконати дії для опції 1
    Console.WriteLine("Ви вибрали опцію 1.");
    string faleName1 = "SHA256Hash.txt";
    string filePath1 = Path.Combine(desktopPath, faleName1);
    //Перевірка наявності файлу перед збереженням
    if (File.Exists(filePath1))
    {
        Console.WriteLine("Файл вже існує. Ви бажаєте перезаписати його? (Y/N)");
        string overwriteChoice = Console.ReadLine();
        if (overwriteChoice.ToLowerInvariant() != "y")
        {
            Console.WriteLine("Операція скасована.");
            continue;
        }
    }
    // Зберегти SHA-256 хеш у файл
    SaveHashToFile(filePath1, sha256Hash);
    Console.WriteLine("Hashes 256 saved to files.");
    // Перевірка цілісності даних при читанні
    bool isIntegrityVerified1 = VerifyDataIntegrity(filePath1);

    if (isIntegrityVerified1)
    {
        Console.WriteLine("Цілісність даних підтверджена.");
    }
    else
    {
        Console.WriteLine("Цілісність даних порушена.");
    }
    Console.WriteLine("_____________________________________________________");
    break;
```

Рисунок 3.40 – Операція збереження SHA-256 у файлі та перевірка цілісності даних

```
Ви у розділі - Хеш-функції: SHA-256, SHA-384, та SHA-512
Меню:
1. Хеш-функції: SHA-256
2. Хеш-функції: SHA-384
3. Хеш-функції: SHA-512
~. Повернутися до головного меню
0. Вийти з програми
Введіть номер опції: 1

_____________________________________________________
Ви вибрали опцію 1.
Hashes 256 saved to files.
Цілісність даних підтверджена.

_____________________________________________________
```

Рисунок 3.41 – Підтвердження цілісності даних в файлі

SHA256Hash – Блокнот

Файл Правка Формат Вид Справка

```
dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f
Hash: 2d86b9f8b1e3c767f8c886e8056b9b14ec91e4213f4898f82c3f092bb96f8ee3
```

Рисунок 3.42 – Збережений SHA-256 у файлі

Далі відповідно виконуються **SHA-384 та SHA-512**, так як вони мають схожий код будуть наведені тільки результати їх збереження в файлі (рис. 3.43 і рис. 3.44).

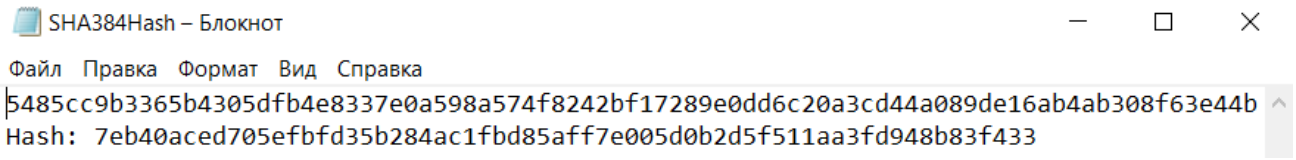


Рисунок 3.43 – Збережений SHA-384 у файлі

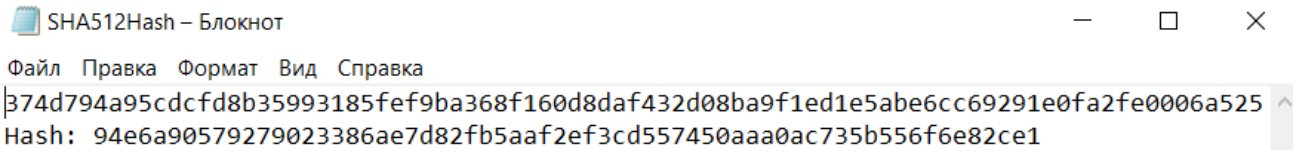


Рисунок 3.44 – Збережений SHA-512 у файлі

3.6 Програмна реалізація Diffie-Hellman Key Exchange засобами .NET

Спочатку створюється екземпляр **RSACryptoServiceProvider** з довжиною ключа 2048 біт. Експортуємо параметри **RSA**, якщо довжина модуля менша за очікувану, виводиться повідомлення про недостатню довжину модуля **RSA**. Виконуємо конвертацію параметрів RSA в велике просте число prime, ініціалізуємо загальноприйнятий генератор для Діффі-Геллмана - generator. Після чого генеруються приватні ключі для "Особи 1" та "Особи 2", конвертуються у значення типу BigInteger використовуючи функцію **GeneratePrivateKey** (рис. 3.45).

```
// Генерація випадкового приватного ключа
static SecureString GeneratePrivateKey(BigInteger prime)
{
    using (RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider())
    {
        byte[] randomBytes = new byte[32]; // змініть розмір відповідно до потреб
        rng.GetBytes(randomBytes);
        // Отримання випадкового приватного ключа та конвертація його в SecureString
        return ConvertToSecureString(BigInteger.Remainder(BigInteger.Abs(new BigInteger(randomBytes)), prime - 2) + 1);
    }
}
```

Рисунок 3.45 – Генерація приватних ключів

Обчислюються відкриті ключі "Особи 1" та "Особи 2" використовуючи функцію **CalculatePublicKey** (рис. 3.46).

```
// Обчислення відкритого ключа
static BigInteger CalculatePublicKey(BigInteger generator, BigInteger prime, BigInteger privateKey)
{
    // Відкритий ключ (BigInteger), отриманий шляхом взяття генератора до ступеня приватного ключа по модулю простого числа.
    return BigInteger.ModPow(generator, privateKey, prime);
}
```

Рисунок 3.46 – Обчислення відкритих ключів

Здійснюється обмін публічними ключами та обчислюються спільні ключі для кожної особистості, використовуючи функцію **CalculateSharedKey** (рис. 3.47).

```
// Обчислення спільного ключа
static BigInteger CalculateSharedKey(BigInteger publicKey, BigInteger prime, BigInteger privateKey)
{
    // Спільний ключ (BigInteger), отриманий шляхом взяття публічного ключа до ступеня приватного ключа по модулю простого числа.
    return BigInteger.ModPow(publicKey, privateKey, prime);
}
```

Рисунок 3.47 – Обчислення спільного ключа

Перевіряється рівність обчислених спільних ключів. Якщо вони рівні, результат зберігається у файлі "shared_key.txt", використовуючи функцію **SaveKeyToFile** (рис. 3.48). Якщо спільні ключі не рівні - виводиться відповідне повідомлення. Результати продемонстровані в (рис. 3.49).

```
// Збереження ключа у файл
static void SaveKeyToFile(string filePath, BigInteger key)
{
    using (StreamWriter writer = new StreamWriter(filePath))
    {
        writer.Write(key.ToString());
    }
}
```

Рисунок 3.48 – Збереження спільного ключа в файлі

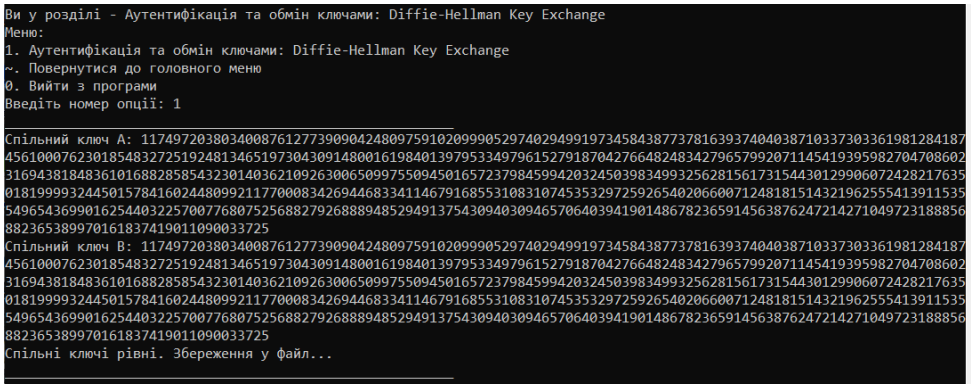


Рисунок 3.49 – Результати знаходження спільного ключа

3.7 Аналіз алгоритму RSA з ключами різної довжини

Алгоритм RSA базується на складності факторизації великих цілих чисел. Змінюючи біти в ключах RSA з 1024 біт на 2048, 3072 і 4096, це вплине на безпеку та продуктивність алгоритму. Зазвичай, збільшення довжини ключа призводить до підвищення криптостійкості алгоритму, але це також призводить до збільшення обчислювального навантаження. Ось загальні впливи, які можна очікувати при зміні довжини ключа в алгоритмі RSA:

1. **збільшення безпеки**: збільшення довжини ключа забезпечує більшу безпеку, оскільки ускладнює факторизацію чисел і ускладнює атаки злому. З 1024 біт на 2048, 3072 і 4096, безпека значно зросте через ускладнення процесу факторизації;
2. **збільшення обчислювальних витрат**: збільшення довжини ключа призводить до збільшення обчислювальних витрат. шифрування, дешифрування, підписи та перевірка підписів можуть займати більше часу через більшу довжину ключа;
3. **збільшення ресурсів пам'яті**: більші ключі також потребують більше пам'яті для збереження та обробки;
4. **збільшення захищеності від атак**: із збільшенням довжини ключа, алгоритм стає менш вразливим до обчислювань.

У цілому, зміна довжини ключа з 1024 біт на 2048, 3072 і 4096 забезпечить значне покращення безпеки системи, але збільшить обчислювальне навантаження. Вибір оптимальної довжини ключа повинен бути здійснений з урахуванням балансу між безпекою та продуктивністю системи залежно від конкретних потреб та обмежень. Для оцінки навантаження при зміні з 1024 біт на 2048, 3072 і 4096 будемо використовувати язик програмування c#, бібліотеку **System.Security.Cryptography та** клас **Stopwatch** для вимірювання часу виконання певних ділянок коду, в нашому випадку зашифровка і розшифровка. Результат при використанні 1024 біт в коді зображено на (рис. 3.50).

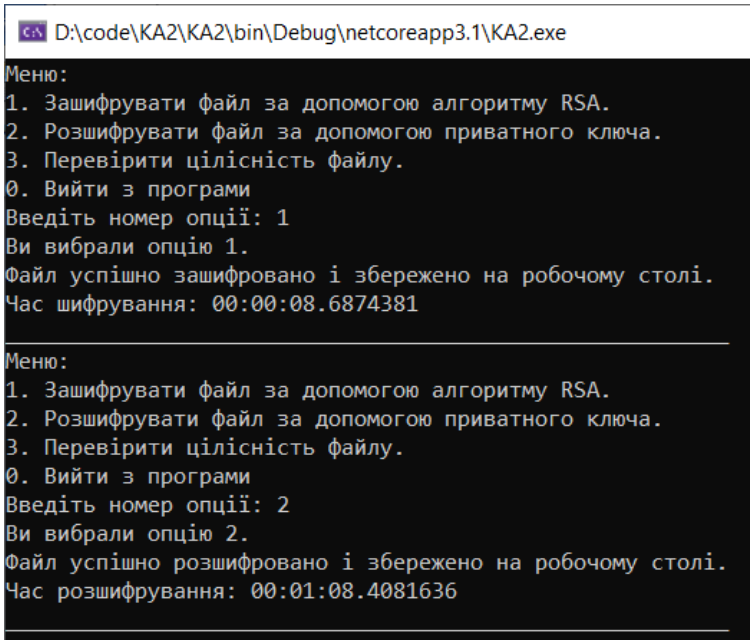


Рисунок 3.50 – Швидкість зашифрування і розшифрування при 1024 біт

Результат при використанні 2048 біт в коді зображено на (рис. 3.51).

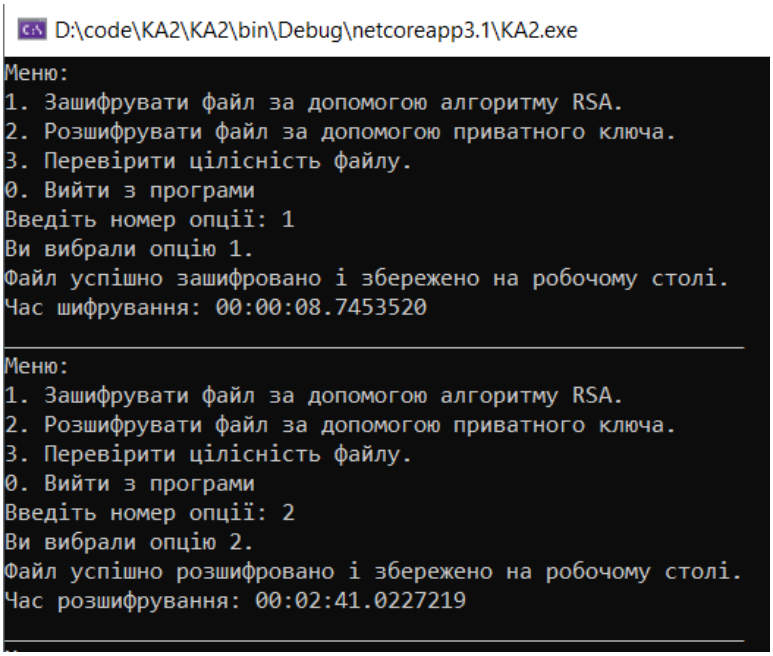


Рисунок 3.51 – Швидкість зашифрування і розшифрування при 2048 біт

Результат при використанні 3072 біт в коді зображено на (рис. 3.52).

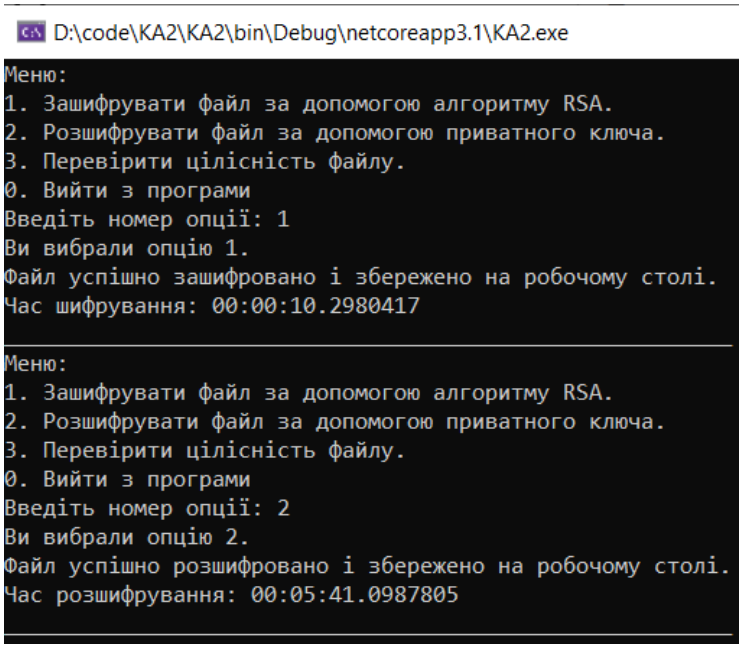


Рисунок 3.52 – Швидкість зашифрування і розшифрування при 3072 біт

Результат при використанні 4096 біт в коді зображено на (рис. 3.53).

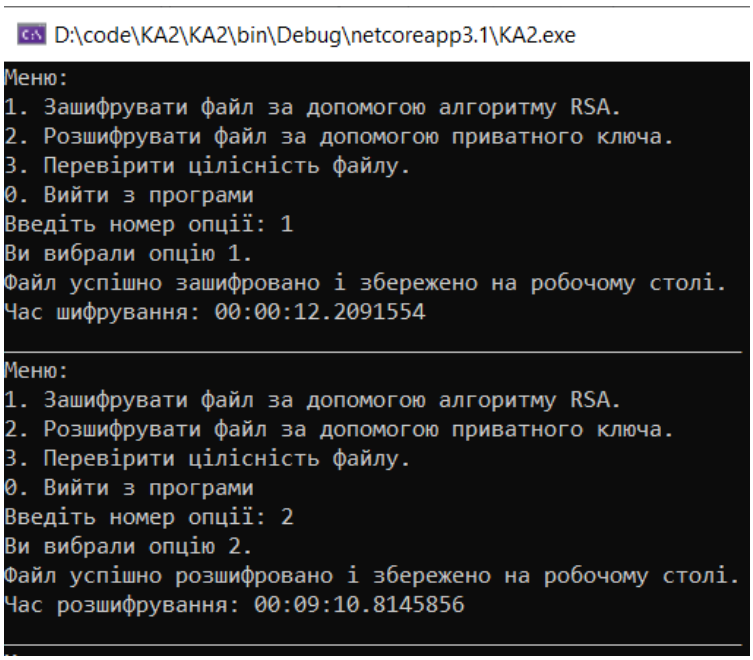


Рисунок 3.53 – Швидкість зашифрування і розшифрування при 4096 біт

Результати можна проаналізувати відносно кожного наступного розміру ключа:

1. від 1024 біт до 2048 біт: збільшення розміру ключа від 1024 біт до 2048 біт призвело до збільшення часу розшифрування (з 1 хвилини 8 секунд до 2 хвилин 41 секунди) і незначного збільшення часу шифрування (з 8.6 секунд до 8.7 секунд);
2. від 2048 біт до 3072 біт: знову збільшення розміру ключа до 3072 біт призвело до помітного збільшення часу розшифрування (з 2 хвилин 41 секунди до 5 хвилин 41 секунди) та збільшення часу шифрування (з 8.7 секунд до 10.3 секунд);
3. від 3072 біт до 4096 біт: збільшення розміру ключа до 4096 біт призвело до подальшого збільшення часу розшифрування (з 5 хвилин 41 секунди до 9 хвилин 10 секунд) та збільшення часу шифрування (з 10.3 секунд до 12.2 секунд).

Таблиця 1 – Швидкість зашифрування і розшифрування з різними бітами

Процес	**Довжина ключа, біт**			
	1024	**2048**	**3072**	**4096**
Зашифрування	8.6 с.	8.7 с.	10.3 с.	12.2 с.
Розшифрування	1 хв. 8 с.	2 хв. 41 с.	5 хв. 41 с.	9 хв. 10 с.

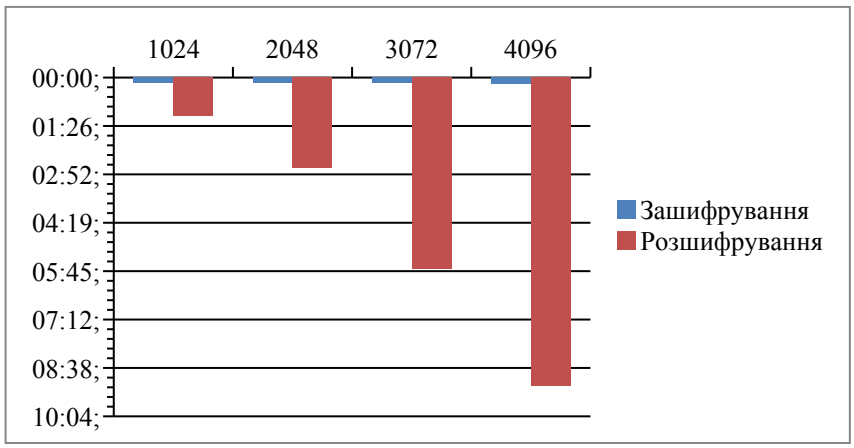


Рисунок 3.54 – Діаграма швидкості зашифрування і розшифрування алгоритмом RSA з ключем різної довжини в бітах

Отже, при збільшенні розміру ключа ми отримуємо більшу безпеку, але збільшується і час обробки даних. Можна помітити, що з кожним збільшенням розміру ключа час шифрування та розшифрування зростає, і це зростання не є лінійним. Якщо розглядати ефективність, 2048 біт ключ може бути найбільш оптимальним компромісом між безпекою та продуктивністю. 1024 біт можуть бути

недостатніми для вимогливих застосувань, тоді як 4096 біт можуть бути занадто витратними у випадках, де вимагається велика продуктивність.

Бібліотека **System.Security.Cryptography** в **C#** містить готові реалізації різних криптографічних алгоритмів, таких як **RSA, AES, DES, SHA, HMAC**, та інших. Використання цієї бібліотеки має кілька переваг порівняно зі створенням власних реалізацій алгоритмів.

1. **готові реалізації**: бібліотека вже містить перевірені й оптимізовані реалізації алгоритмів, що забезпечує якість та безпеку криптографічних операцій. Власні реалізації можуть вимагати значних знань і часу для забезпечення адекватної безпеки;
2. **стандарти безпеки**: реалізації в бібліотеці **System.Security.Cryptography** відповідають стандартам безпеки та можуть бути сумісними з іншими реалізаціями, наприклад, з іншими мовами програмування або криптографічними бібліотеками;
3. **оновлення та підтримка**: криптографічні алгоритми постійно еволюціонують для відповіді на нові загрози, бібліотека має шанс на регулярні оновлення та виправлення вразливостей, тоді як власні реалізації можуть вимагати постійного відстежування та оновлення;

У багатьох випадках, особливо для виробничого коду, використання готових реалізацій з бібліотеки ***System.Security.Cryptography*** є раціональним рішенням, оскільки вони забезпечують якість, безпеку та можливість швидкої інтеграції без необхідності повного розуміння деталей алгоритму.

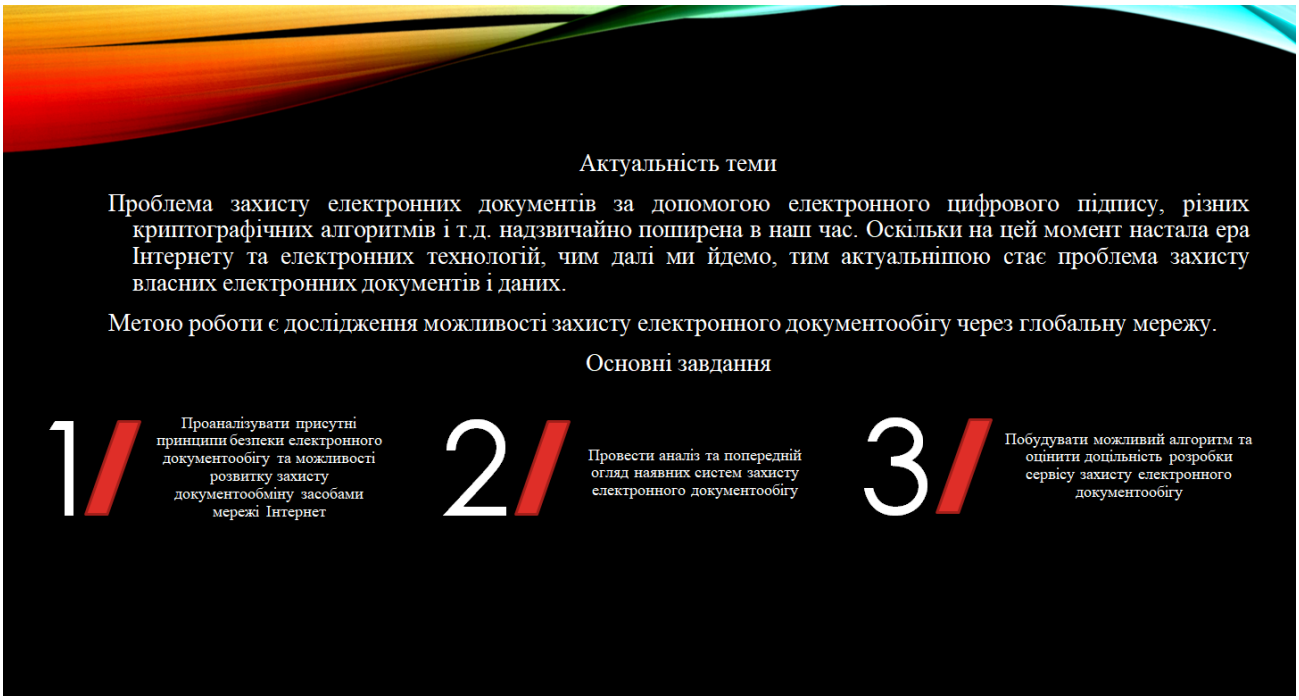
ВИСНОВКИ

Здійснено аналіз основних засобів безпеки електронного документообігу, проведено попередній огляд існуючих систем захисту та виявлено їх основні недоліки. Проведено огляд запропонованих алгоритмів, а також описано засіб захисту електронного документообігу. Отже, дослідження надало усвідомлення про важливість вибору криптографічних методів, а також надало рекомендації стосовно оптимального розміру ключів для досягнення балансу між безпекою та продуктивністю системи в контексті електронного документообігу. Крім того, дослідження підкреслило вигоди використання бібліотек таких як **System.Security.Cryptography** які містять часткові реалізації криптографічних алгоритмів, сприяючи швидкій інтеграції та забезпечення даних в електронному середовищі. Реалізовані алгоритми які використовують бібліотеку **System.Security.Cryptography,** відповідають стандартам, що робить їх відмінним вибором для забезпечення конфіденційності, цілісності та доступності даних в електронному документообігу. Проте, необхідно усвідомлювати, що важливо обирати алгоритми захисту інформації в залежності від їх призначення. Отже, для процесу шифрування/розшифрування краще обирати симетричні алгоритми. Асиметричні алгоритми бажано використовувати для шифрування тільки невеликих повідомлень. Для процесу автентифікації документів та управління ключами треба обирати асиметричні криптоалгоритми.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. System.Security.Cryptography.
URL: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography
2. .NET-cryptography model.
URL: https://learn.microsoft.com/en-us/dotnet/standard/security/cryptography-model
3. Технології захисту інформації.
URL: https://ela.kpi.ua/bitstream/123456789/23896/1/TZI_book.pdf
4. Йона Л.Г. Криптографічний захист електронного документообігу [Текст] / Л. Г. Йона, О. О. Йона, В. С. Терешко // Цифрові технології.–2013. – № 13. – С. 142–146. – Режим доступу: http://nbuv.gov.ua/UJRN/ct_2013_13_20
5. Йона Л.Г. Аналіз діючих протоколів криптографічного захисту електронних транзакцій[Текст] / / Л. Г. Йона, О. О. Кюне // Цифрові технології. - 2017. - Вип. 22. - С. 96-102. - Режим доступу: http://nbuv.gov.ua/UJRN/ct_2017_22_13
6. Про електронні документи та електронний документообіг: Закон України від 22 травня 2003 р. № 851-IV: [Електронний ресурс] – Режим доступу: http://www.rada.gov.ua
7. URL: https://dspace.nlu.edu.ua/bitstream/123456789/6710/1/Maznichenko.pdf
8. Biryukov, Alex and Khovratovich, Dmitry. Related-key Cryptanalysis of the Full AES-192 and AES-256. — Advances in Cryptology – ASIACRYPT 2009, 2009. — Vol. 5912. — DOI:10.1007/978-3-642-10366-7_1.
9. ДСТУ 7624:2014 «Інформаційні технології. Криптографічний захист інформації. Алгоритм симетричного блокового перетворення». – Режим доступу: http://online.budstandart.com/ua/catalog/docpage?id_doc=65314
10. Dobraunig, C., Eichlseder, Maria., Mendel, F. (2016). Analysis of SHA-512/224 and SHA-512/256. IACR Cryptology ePrint Archive, 374. – Access mode: https://eprint.iacr.org/2016/374.pdf
11. RFC 5246 – The Transport Layer Security (TLS) Protocol Version 1.2 [Електронний ресурс] – Режим доступу: https://www.rfceditor.org/rfc/rfc5246
12. RFC 6101 – The Secure Sockets Layer (SSL) Protocol Version 3.0 [Електронний ресурс] – Режим доступу: https://www.rfc-editor.org/rfc/rfc6101
13. URL: https://veracrypt.fr/en/Documentation.html
14. URL: https://gitlab.com/cryptsetup/cryptsetup/blob/master/README.md
15. Implementing BitLocker Drive Encryption for Forensic Analysis: стаття, авт. J. Kornblum, Digital Investigation. 2009. С. 75–84.

ДОДАТОК А
Перелік копій демонстраційного матеріалу



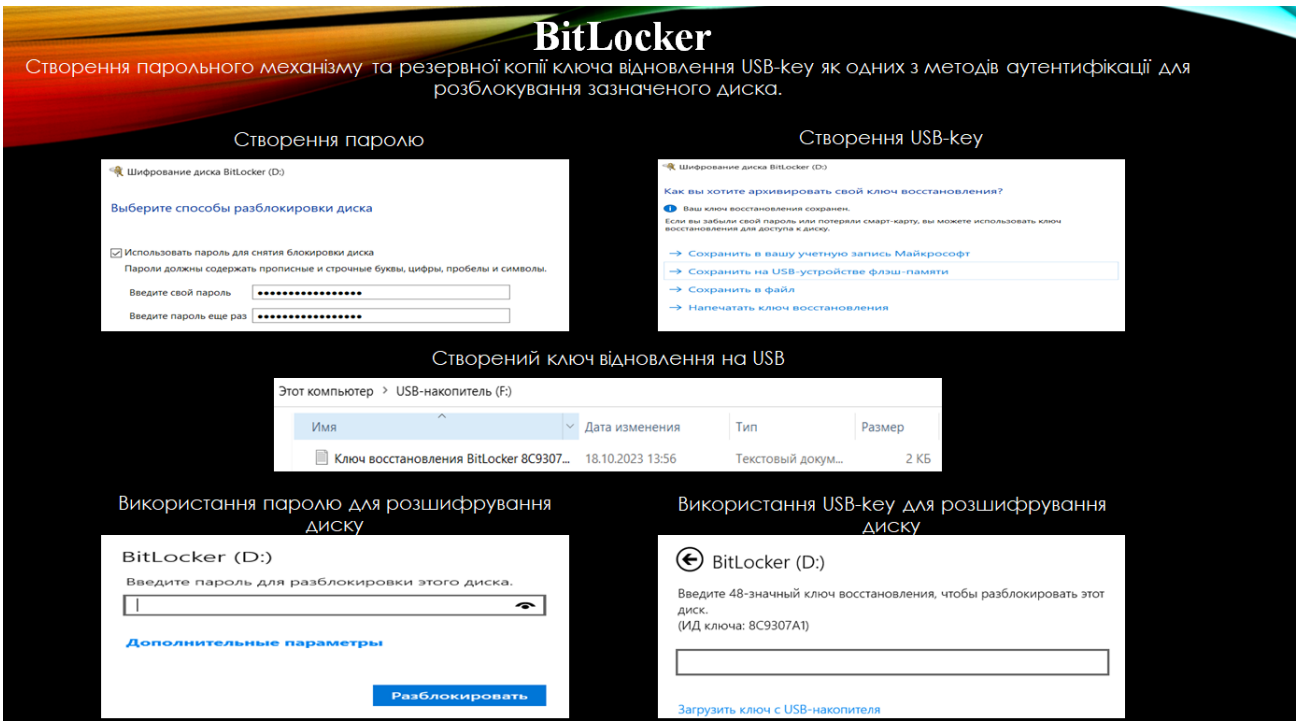
Слайд 1 – Актуальність, мета, основні завдання

Слайд 2 – Загрози безпеки ЕД

Слайд 3 – Способи захисту ЕД

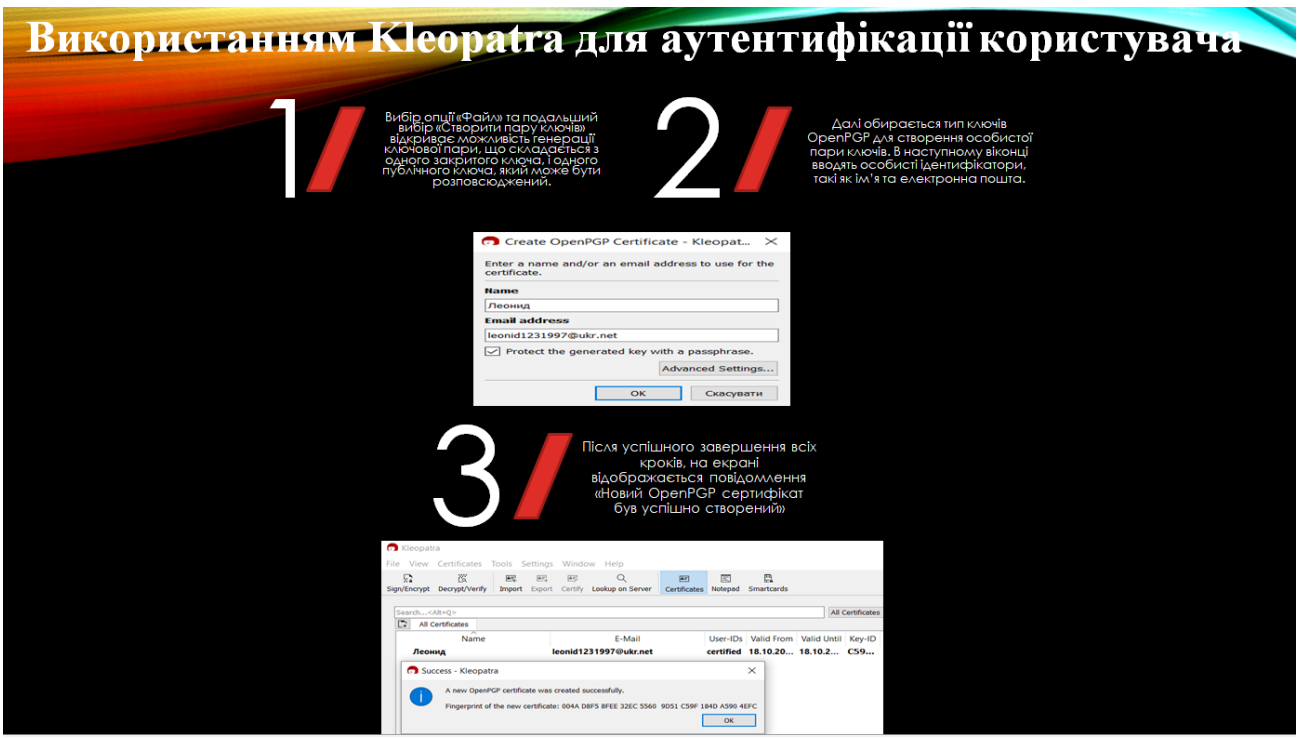
Слайд 4 – Криптографія, криптографічна стійкість

Слайд 5 – Програми для шифрування диска

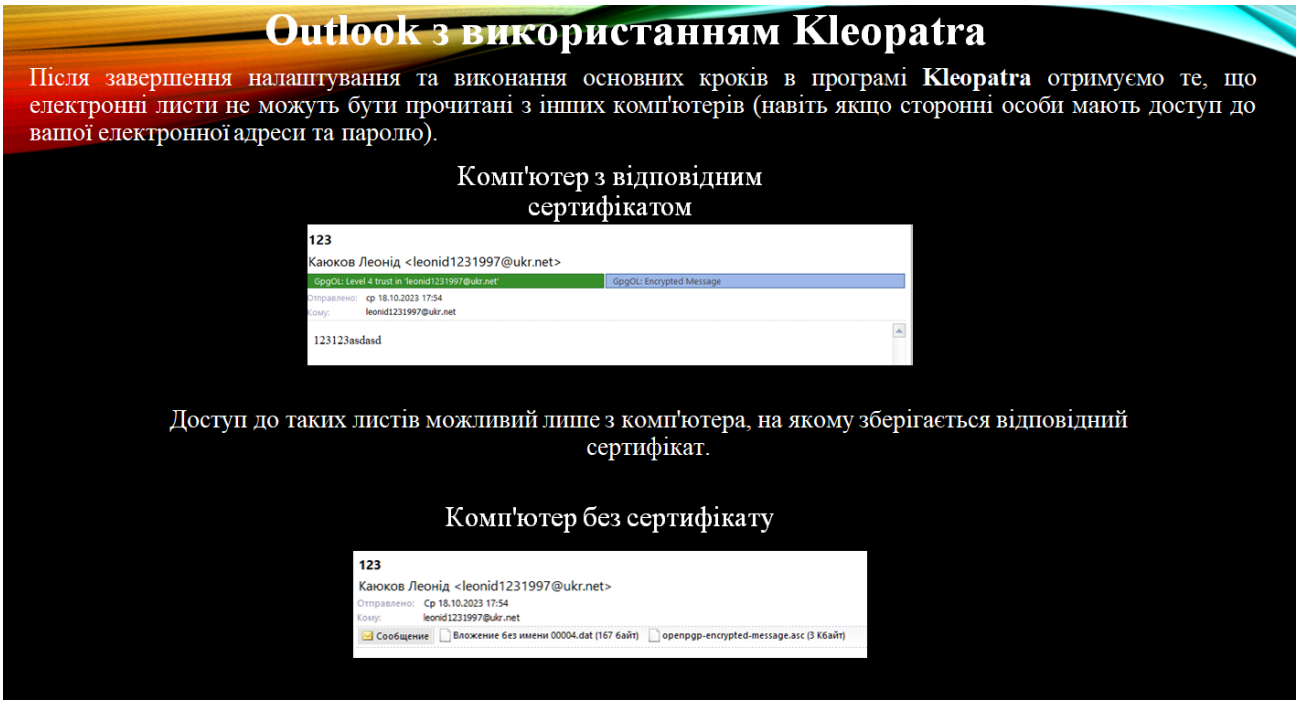


Слайд 6 – Використання BitLocker

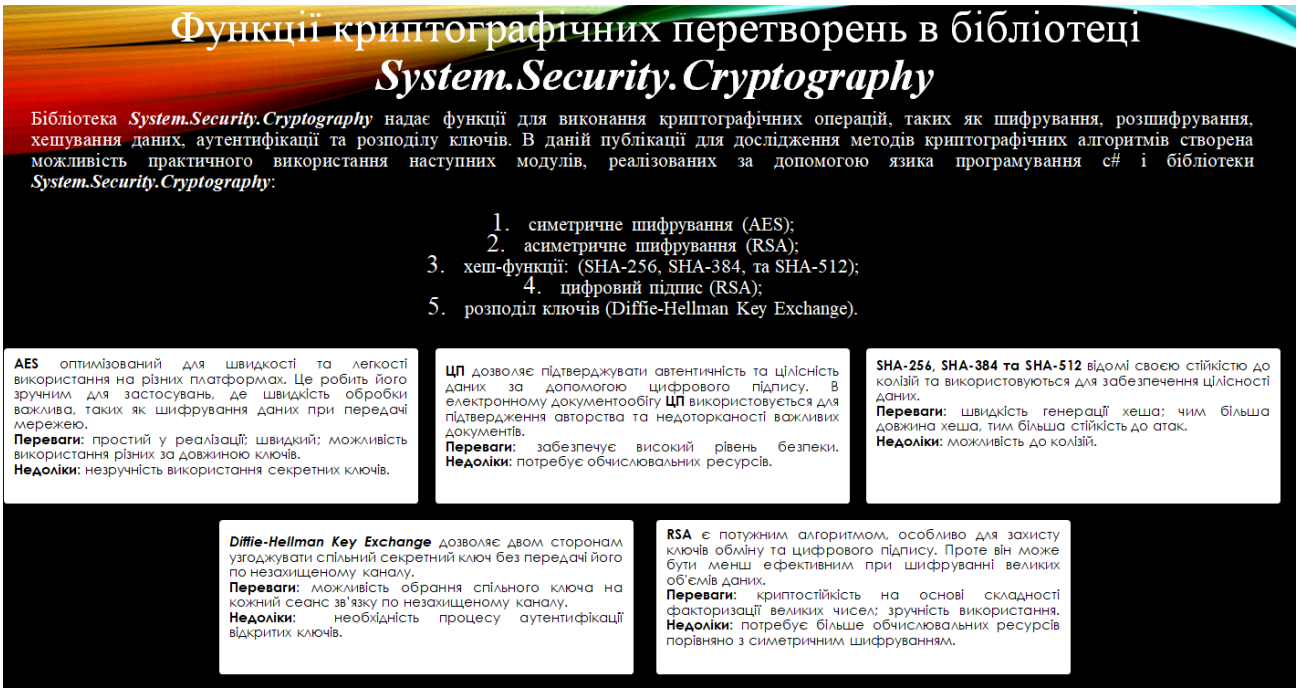
Слайд 7 – Засоби конфіденційного зв'язку



Слайд 8 – Використанння Kleopatra для аутентифікації користувача



Слайд 9 – Outlook з використанням Kleopatra



Слайд 10 – Функції криптографічних перетворень в бібліотеці System.Security.Cryptography

Програмна реалізація алгоритму RSA мережевими засобами

Оголосивши дві змінні типу **RSAParameters**, які призначені для збереження публічного **publicKey** і приватного **privateKey** ключів **RSA**. Відбувається перевірка наявності приватного ключа у файлі "private_key.txt" на робочому столі. Якщо файл існує, приватний ключ завантажується за допомогою функції **LoadKeyFromFile**.

У випадку, якщо файл не існує, генерується нова пара ключів (приватний і публічний ключі **RSA**) за допомогою об'єкту **RSACryptoServiceProvider**. Публічний ключ publicKey отримується шляхом експорту публічних параметрів, тоді як приватний ключ **privateKey** отримується експортом приватних параметрів. Приватний ключ також зберігається у файл "private_key.txt" за допомогою функції **SaveKeyToFile**.

Завантаження існуючого ключа

```
// Завантаження ключа із файлу
static RSAParameters LoadKeyFromFile(string filePath)
{
    RSAParameters key = new RSAParameters();
    using (StreamReader sr = new StreamReader(filePath))
    {
        key.Modulus = Convert.FromBase64String(sr.ReadLine());
        key.Exponent = Convert.FromBase64String(sr.ReadLine());
        key.D = Convert.FromBase64String(sr.ReadLine());
        key.P = Convert.FromBase64String(sr.ReadLine());
        key.Q = Convert.FromBase64String(sr.ReadLine());
        key.DP = Convert.FromBase64String(sr.ReadLine());
        key.DQ = Convert.FromBase64String(sr.ReadLine());
        key.InverseQ = Convert.FromBase64String(sr.ReadLine());
    }
    return key;
}
```

Збереження ключа

```
// Збереження ключа у файл
static void SaveKeyToFile(string filePath, RSAParameters key)
{
    using (StreamWriter sw = new StreamWriter(filePath))
    {
        sw.WriteLine(Convert.ToBase64String(key.Modulus));
        sw.WriteLine(Convert.ToBase64String(key.Exponent));
        sw.WriteLine(Convert.ToBase64String(key.D));
        sw.WriteLine(Convert.ToBase64String(key.P));
        sw.WriteLine(Convert.ToBase64String(key.Q));
        sw.WriteLine(Convert.ToBase64String(key.DP));
        sw.WriteLine(Convert.ToBase64String(key.DQ));
        sw.WriteLine(Convert.ToBase64String(key.InverseQ));
    }
}
```

Створення приватного та публічного ключа

```
RSAParameters publicKey;
RSAParameters privateKey;
// Завантаження приватного ключа, якщо він вже існує
string privateKeyFilePath = Path.Combine(desktopPath, "private_key.txt");
if (File.Exists(privateKeyFilePath))
{
    privateKey = LoadKeyFromFile(privateKeyFilePath);
    publicKey = new RSAParameters(); // Ініціалізуємо пустий об'єкт RSAParameters для уникнення помилок
}
else
{
    // Генерація пари ключів (приватний і публічний ключі RSA)
    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider())
    {
        // Публічний ключ RSA
        publicKey = rsa.ExportParameters(false);

        // Приватний ключ RSA
        privateKey = rsa.ExportParameters(true);
        // Збереження приватного ключа у файл
        SaveKeyToFile(privateKeyFilePath, privateKey);
    }
}
```

Слайд 11 – Створення приватного, публічного ключа та їх збереження або завантаження

Програмна реалізація алгоритму RSA засобами .NET

Використовуючи функцію **EncryptFile**, яка відповідає за зашифрування файлу.

Параметри функції

1. **originalFilePath**: Шлях до оригінального файлу, який потрібно зашифрувати;
2. **encryptedFilePath**: Шлях, куди буде збережено зашифрований файл;
3. **publicKey**: Публічний ключ, що використовується для зашифрування.

Функція **EncryptFile** виконує процес зашифрування файлу за допомогою публічного ключа **publicKey**. Після завершення шифрування зашифрований файл зберігається за вказаним шляхом **encryptedFilePath**. Після завершення процесу зашифрування виводиться повідомлення у консоль про те, що файл був успішно зашифрований і збережений на робочому столі.

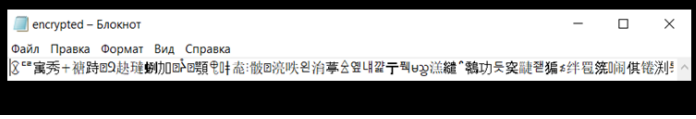
Функція EncryptFile для зашифрування

```
// Зашифрування файлу за допомогою алгоритму RSA
static void EncryptFile(string inputFile, string outputFile, RSAParameters publicKey)
{
    using (RSACryptoServiceProvider rsaEncrypt = new RSACryptoServiceProvider())
    {
        rsaEncrypt.ImportParameters(publicKey);
        using (FileStream inputStream = File.OpenRead(inputFile))
        using (FileStream outputStream = File.Create(outputFile))
        {
            int keySize = rsaEncrypt.KeySize / 8;
            int blockSize = keySize - 11; // Розмір блока для RSA PKCS#1 v1.5
            int bufferSize = blockSize;
            byte[] buffer = new byte[bufferSize];
            int bytesRead;
            while ((bytesRead = inputStream.Read(buffer, 0, bufferSize)) > 0)
            {
                byte[] encryptedBytes = rsaEncrypt.Encrypt(buffer.Take(bytesRead).ToArray(), false);
                outputStream.Write(encryptedBytes, 0, encryptedBytes.Length);
            }
        }
    }
}
```

Зашифрування файлу з функцією EncryptFile

```
case "1":
    // Опція 1: Зашифрувати файл і зберегти
    Console.WriteLine("Ви вибрали опцію 1.");
    // Зашифрування файлу
    EncryptFile(originalFilePath, encryptedFilePath, publicKey);
    Console.WriteLine("Файл успішно зашифровано і збережено на робочому столі.");
```

Результати зашифрування файлу



Слайд 12 – Створення та використання функції для зашифрування

Програмна реалізація алгоритму RSA мережевими засобами

Перевірка результату розшифрування файлу за допомогою функції **DecryptFile.**

Параметри функції

encryptedFilePath: шлях до зашифрованого файлу, який потрібно розшифрувати;

decryptedFilePath: шлях, куди буде збережено розшифрований файл;

privateKey: приватний ключ, який використовується для розшифрування.

Якщо розшифрування успішне повертається **true**, виводиться повідомлення у консоль про те, що "Файл успішно розшифровано і збережено на робочому столі". У разі, якщо розшифрування невдале повертається **false**, виводиться повідомлення, що "Файл не вдалося розшифрувати і зберегти на робочому столі. Можливо, дані пошкоджено!".

Функція DecryptFile для розшифрування

```
// Розшифрування файлу за допомогою приватного ключа
static bool DecryptFile(string inputFile, string outputFile, RSAParameters privateKey)
{
    try
    {
        using (RSACryptoServiceProvider rsaDecrypt = new RSACryptoServiceProvider())
        {
            rsaDecrypt.ImportParameters(privateKey);
            using (FileStream inputStream = File.OpenRead(inputFile))
            using (FileStream outputStream = File.Create(outputFile))
            {
                int keySize = rsaDecrypt.KeySize / 8;
                int blockSize = keySize; // Розмір блока для RSA PKCS#1 v1.5
                int bufferSize = blockSize;
                byte[] buffer = new byte[bufferSize];
                int bytesRead;
                while ((bytesRead = inputStream.Read(buffer, 0, bufferSize)) > 0)
                {
                    byte[] decryptedBytes = rsaDecrypt.Decrypt(buffer.Take(bytesRead).ToArray(), false);
                    outputStream.Write(decryptedBytes, 0, decryptedBytes.Length);
                }
            }
        }
        return true; // Повертаємо true, якщо розшифрування пройшло успішно
    }
    catch (CryptographicException ex)
    {
        Console.WriteLine("Помилка розшифрування: " + ex.Message);
        return false; // Повертаємо false у випадку помилки розшифрування
    }
}
```

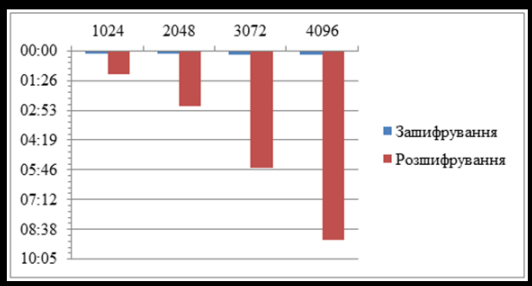
Розшифрування файлу з функцією DecryptFile

```
case "2":
    // Опція 2: Розшифрувати файл і зберегти
    Console.WriteLine("Ви вибрали опцію 2.");
    // Розшифрування файлу
    if (DecryptFile(encryptedFilePath, decryptedFilePath, privateKey))
    {
        // Розшифрування успішне
        Console.WriteLine("Файл успішно розшифровано і збережено на робочому столі.");
        Console.WriteLine("_____");
    }
    else
    {
        // Розшифрування не вдалося (дані були пошкоджені)
        Console.WriteLine("Файл не вдалося розшифрувати і зберегти на робочому столі. Можливо, дані пошкоджено!");
        Console.WriteLine("_____");
    }
    break;
```

Слайд 13 – Створення та використання функції для розшифрування

Аналіз алгоритму RSA з ключами різної довжини

Процес	Довжина ключа, біт			
	1024	2048	3072	4096
Зашифрування	8.6 с.	8.7 с.	10.3 с.	12.2 с.
Розшифрування	1 хв. 8 с.	2 хв. 41 с.	5 хв. 41 с.	9 хв. 10 с.



Бібліотека **System.Security.Cryptography** в **C#** містить готові реалізації різних криптографічних алгоритмів, таких як **RSA, AES, DES, SHA, HMAC**, та інших. Використання цієї бібліотеки має кілька переваг порівняно зі створенням власних реалізацій алгоритмів.

Готові реалізації

Бібліотека вже містить перевірені й оптимізовані реалізації алгоритмів, що забезпечує якість та безпеку ваших криптографічних операцій. Власні реалізації можуть вимагати значних знань і часу для забезпечення адекватної безпеки.

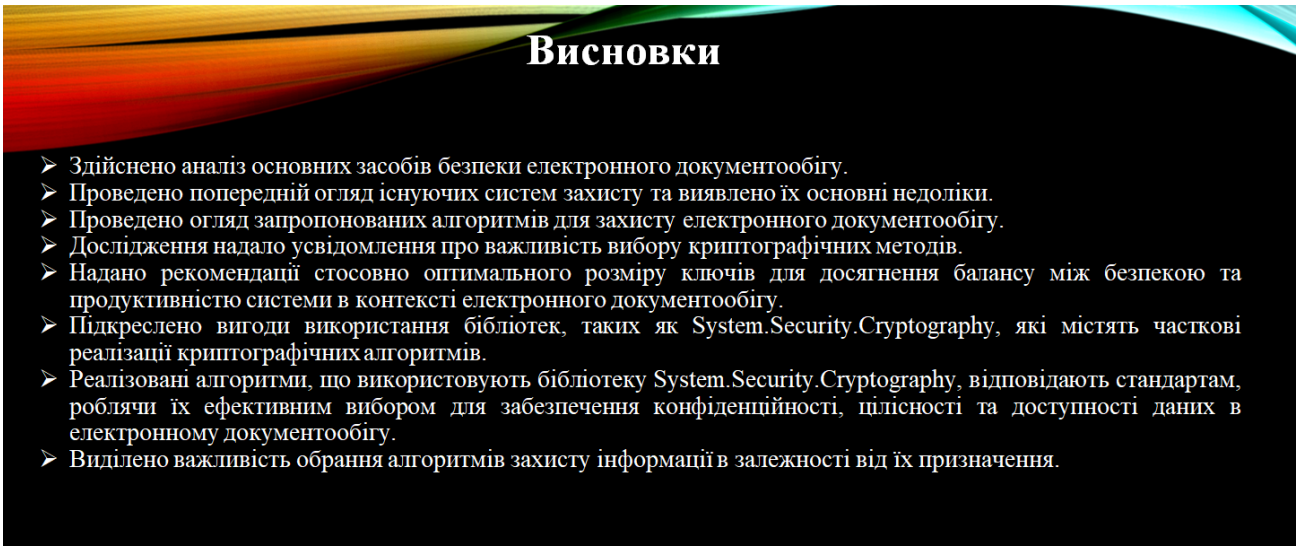
Стандарти безпеки

Реалізації в бібліотеці **System.Security.Cryptography** відповідають стандартам безпеки та можуть бути сумісними з іншими реалізаціями, наприклад, з іншими мовами програмування або криптографічними бібліотеками.

Оновлення та підтримка

Криптографічні алгоритми постійно еволюціонують для відповіді на нові загрози. Бібліотека має шанс на регулярні оновлення та виправлення вразливостей, тоді як власні реалізації можуть вимагати постійного відстежування та оновлення.

Слайд 14 – Аналіз алгоритму з різними довжинами ключа та переваги використання бібліотеки



Слайд 15 – Висновки

ДОДАТОК Б
Лістинги модулів програмної системи

```
using System;
using System.IO;
using System.Numerics;
using System.Security;
using System.Security.Cryptography;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.Linq;

class Program
{
   static void Main()
   {
      Console.OutputEncoding = System.Text.Encoding.Default;
        Stack<string> menuStack = new Stack<string>(); // Стек для відстеження стану меню

      while (true)
      {
         Console.WriteLine("Головне меню:");
         Console.WriteLine("1. Симетричне шифрування: AES");
         Console.WriteLine("2. Асиметричне шифрування: RSA");
         Console.WriteLine("3. Створення та перевірка ЕЦП для файлу");
         Console.WriteLine("4. Хеш-функції: SHA-256, SHA-384, та SHA-512");
         Console.WriteLine("5. Аутентифікація та обмін ключами: Diffie-Hellman Key Exchange");
         Console.WriteLine("0. Вихід з програми");
         Console.Write("Введіть номер опції: ");

         string userInput = Console.ReadLine();

         switch (userInput)
```

```
        {
            case "1":
                menuStack.Push("1");
                KaMenu(menuStack);
                break;

            case "2":
                menuStack.Push("2");
                Ka2Menu(menuStack);
                break;

            case "3":
                menuStack.Push("3");
                Ka2_1Menu(menuStack);
                break;

            case "4":
                menuStack.Push("4");
                Ka2_2Menu(menuStack);
                break;

            case "5":
                menuStack.Push("5");
                Ka2_3Menu(menuStack);
                break;

            case "0":
                        Console.WriteLine("Дякую за використання програми. До
побачення!");
                return;

            default:
                Console.WriteLine("Невірний вибір. Будь ласка, виберіть дійсний
номер опції.");
                break;
        }
```

```
            }
        }

        // Код AES
        static void KaMenu(Stack<string> menuStack)
        {
Console.WriteLine("_____
_____");
            Console.WriteLine("Ви у розділі - Симетричне шифрування: AES");
            // Задаємо ключ для шифрування і вектор ініціалізації (IV)
            byte[] key = new byte[16]; // 128-бітний ключ (16 байтів)
            byte[] IV = new byte[16];  // 128-бітний IV (16 байтів)

            // Отримуємо шлях до робочого столу
                                                    string desktopPath =
Environment.GetFolderPath(Environment.SpecialFolder.Desktop);

            // Ім'я файлу, який використовується
            string fileName = "Ka.txt";

            // Файл, який буде зашифраваний
            string filePath = Path.Combine(desktopPath, fileName);

            // Зчитайте вміст файлу
            byte[] fileBytes = File.ReadAllBytes(filePath);

            byte[] encryptedBytes = null;
            byte[] decryptedBytes = null;

            while (true) // Безкінечний цикл для повторного виконання меню
            {
               Console.WriteLine("Меню:");
               Console.WriteLine("1. Зашифрувати і зберегти файл");
               Console.WriteLine("2. Розшифрувати і зберегти файл");
                        Console.WriteLine("3. Перевірити цілісність даних після
розшифрування");
```

```
            Console.WriteLine("~. Повернутися до головного меню");
            Console.WriteLine("0. Вийти з програми");
            Console.Write("Введіть номер опції: ");

            string userInput = Console.ReadLine();
Console.WriteLine("_____
_____");
            switch (userInput)
            {
               case "1":
                  // Опція 1: Зашифрувати файл і зберегти
                  Console.WriteLine("Ви вибрали опцію 1.");
                  encryptedBytes = EncryptFile(fileBytes, key, IV);

                  // Створити повний шлях до файлу на робочому столі
                                 string desktopFilePath = Path.Combine(desktopPath,
"coder_document.txt");

                  // Зберегти зашифрований файл на робочому столі
                  File.WriteAllBytes(desktopFilePath, encryptedBytes);

                      Console.WriteLine("Файл успішно зашифровано і збережено на
робочому столі.");
Console.WriteLine("_____
_____");
                  break;

               case "2":
                  // Опція 2: Розшифрувати файл і зберегти
                  Console.WriteLine("Ви вибрали опцію 2.");

                  // Файл, який буде розшифровуватися
                  string filePath2 = Path.Combine(desktopPath, "coder_document.txt");

                  // Розшифрувати файл
                  decryptedBytes = DecryptFile(File.ReadAllBytes(filePath2), key, IV);
```

```
                    // Створити повний шлях до файлу на робочому столі
                                string desktopFilePath2 = Path.Combine(desktopPath,
"decoder_document.txt");

                    // Зберегти розшифрований файл на робочому столі
                    File.WriteAllBytes(desktopFilePath2, decryptedBytes);

                       Console.WriteLine("Файл успішно розшифровано і збережено на
робочому столі.");
Console.WriteLine("_____
_____");
                    break;

                case "3":
                    // Опція 3: Перевірити цілісність розшифрованого тексту
                    Console.WriteLine("Ви вибрали опцію 3.");
                    bool isIntegrityVerified = VerifyIntegrity(decryptedBytes, filePath);

                    if (isIntegrityVerified)
                    {
                        Console.WriteLine("Цілісність документа після розшифрування
підтверджена. Інформація не була змінена.");
                    }
                    else
                    {
                            Console.WriteLine("Увага! Інформація була змінена після
розшифрування.");
                    }
Console.WriteLine("_____
_____");
                    break;

                case "~":
                    // Повертаємося назад до попереднього меню
                    if (menuStack.Count > 0)
```

```
                    {
                        menuStack.Pop(); // Видаляємо останню опцію зі стеку
Console.WriteLine("_____
_____");
                        return;
                    }
                    else
                    {
                        Console.WriteLine("Немає попередніх меню для повернення.");
                    }
                    break;

                case "0":
                    // Опція 0: Вихід з програми
                            Console.WriteLine("Дякую за використання програми. До
побачення!");
                    return;

                default:
                    Console.WriteLine("Невірний вибір. Будь ласка, виберіть дійсний
номер опції.");
                    break;
            }
        }
    }

    // Код RSA
    static void Ka2Menu(Stack<string> menuStack)
    {
Console.WriteLine("_____
_____");
        Console.WriteLine("Ви у розділі - Асиметричне шифрування: RSA");
        try
        {
            // Отримання шляху до робочого столу
```

```
string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);

            // Шляхи до файлів
            string originalFilePath = Path.Combine(desktopPath, "Ka.txt");
            string encryptedFilePath = Path.Combine(desktopPath, "encrypted.txt");
            string decryptedFilePath = Path.Combine(desktopPath, "decrypted.txt");

            RSAParameters publicKey;
            RSAParameters privateKey;
            // Завантаження приватного ключа, якщо він вже існує
            string privateKeyFilePath = Path.Combine(desktopPath, "private_key.txt");
            if (File.Exists(privateKeyFilePath))
            {
               privateKey = LoadKeyFromFile(privateKeyFilePath);
                    publicKey = new RSAParameters(); // Ініціалізуємо пустий об'єкт RSAParameters для уникнення помилок
            }
            else
            {
               // Генерація пари ключів (приватний і публічний ключі RSA)
                                using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider())
               {
                  // Публічний ключ RSA
                  publicKey = rsa.ExportParameters(false);

                  // Приватний ключ RSA
                  privateKey = rsa.ExportParameters(true);
                  // Збереження приватного ключа у файл
                  SaveKeyToFile(privateKeyFilePath, privateKey);
               }
            }

            while (true) // Безкінечний цикл для повторного виклику меню
            {
```

```
                Console.WriteLine("Меню:");
                   Console.WriteLine("1. Зашифрувати файл за допомогою алгоритму RSA.");
                    Console.WriteLine("2. Розшифрувати файл за допомогою приватного ключа.");
                Console.WriteLine("3. Перевірити цілісність файлу.");
                Console.WriteLine("~. Повернутися до головного меню");
                Console.WriteLine("0. Вийти з програми");
                Console.Write("Введіть номер опції: ");

                string userInput = Console.ReadLine();
Console.WriteLine("_____");
                switch (userInput)
                {
                   case "1":
                      // Опція 1: Зашифрувати файл і зберегти
                      Console.WriteLine("Ви вибрали опцію 1.");
                      // Зашифрування файлу
                      EncryptFile(originalFilePath, encryptedFilePath, publicKey);
                        Console.WriteLine("Файл успішно зашифровано і збережено на робочому столі.");
Console.WriteLine("_____");
                      break;

                   case "2":
                      // Опція 2: Розшифрувати файл і зберегти
                      Console.WriteLine("Ви вибрали опцію 2.");
                      // Розшифрування файлу
                      if (DecryptFile(encryptedFilePath, decryptedFilePath, privateKey))
                      {
                         // Розшифрування успішне
                          Console.WriteLine("Файл успішно розшифровано і збережено на робочому столі.");
```

```
Console.WriteLine("_____
_____");
                        }
                        else
                        {
                           // Розшифрування не вдалося (дані були пошкоджені)
                            Console.WriteLine("Файл не вдалося розшифрувати і зберегти
на робочому столі. Можливо, дані пошкоджено!");
Console.WriteLine("_____
_____");
                        }
                        break;

                    case "3":
                        // Опція 3: Перевірити цілісність розшифрованого тексту
                        Console.WriteLine("Ви вибрали опцію 3.");
                        // Перевірка цілісності даних перед розшифруванням
                                bool isIntegrityVerified = VerifyIntegrity(originalFilePath,
decryptedFilePath);
                        if (isIntegrityVerified)
                        {
                            Console.WriteLine("Файл був зашифрований і розшифрований
успішно, і цілісність даних підтверджена.");
                        }
                        else
                        {
                                Console.WriteLine("Перевірка цілісності не пройшла. Дані
були змінені.");
                        }
Console.WriteLine("_____
_____");
                        break;

                    case "~":
                        // Повертаємося назад до попереднього меню
                        if (menuStack.Count > 0)
```

```
                    {
                        menuStack.Pop(); // Видаляємо останню опцію зі стеку
Console.WriteLine("_____
_____");
                        return;
                    }
                    else
                    {
                                Console.WriteLine("Немає попередніх меню для
повернення.");
                    }
                    break;

                case "0":
                    // Опція 0: Вихід з програми
                        Console.WriteLine("Дякую за використання програми. До
побачення!");
                    return;

                default:
                    Console.WriteLine("Невірний вибір. Будь ласка, виберіть дійсний
номер опції.");
                    break;
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Помилка: " + ex.Message);
        }
    }

    // Код ЕЦП
    static void Ka2_1Menu(Stack<string> menuStack)
    {
```

```
Console.WriteLine("_____
_____");
            Console.WriteLine("Ви у розділі - Цифровий підпис (ЕЦА)");
                                                    string desktopPath =
Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
            string fullName = "Kayukov Leonid Leonidovych";
            string date = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
            // Створення об'єкта, що містить дані для підпису
            StringBuilder dataToSignBuilder = new StringBuilder();
            byte[] dataToSign = null;
            byte[] signature = null;
            // Створіть повний шлях до файлу на робочому столі
            string desktopFilePath = null;
            string desktopFilePath2 = Path.Combine(desktopPath, "signaturefile2.txt");
            // Створення ключів
            using (var rsa = new RSACryptoServiceProvider())
            {
               try
               {
                  while (true) // Безкінечний цикл для повторного виконання меню
                  {
                     Console.WriteLine("Меню:");
                           Console.WriteLine("1. Створення об'єкта, що містить дані для
підпису");
                     Console.WriteLine("2. Добавляємо підпис до файлу");
                     Console.WriteLine("3. Перевірка підпису в файлі");
                     Console.WriteLine("4. Видалення підпису з файла");
                     Console.WriteLine("~. Повернутися до головного меню меню");
                     Console.WriteLine("0. Вийти з програми");
                     Console.Write("Введіть номер опції: ");

                     string userInput = Console.ReadLine();
Console.WriteLine("_____
_____");
                     switch (userInput)
                     {
```

```
                    case "1":
                        // Виконати дії для опції 1
                        Console.WriteLine("Ви вибрали опцію 1.");
                        // Створення об'єкта, що містить дані для підпису
                        dataToSignBuilder.AppendLine("Full Name: " + fullName);
                        dataToSignBuilder.AppendLine("Date: " + date);
                        Console.WriteLine(dataToSignBuilder);
Console.WriteLine("_____
_____");
                        break;

                    case "2":
                        // Виконати дії для опції 2
                        Console.WriteLine("Ви вибрали опцію 2.");
                        // Зчитування файлу
                        Console.Write("Введіть назву файла з робочого столу: ");
                        string faleName1 = Console.ReadLine();
                        string filePath = Path.Combine(desktopPath, faleName1);
                        if (!File.Exists(filePath))
                        {
                            Console.WriteLine("Файл не існує.");
                            continue; // або інша логіка, яка вам потрібна
                        }
                        byte[] fileData = File.ReadAllBytes(filePath);

                        // Робимо копію файла на робочому столі
                             Console.Write("Введіть нову назву для файла в якому буде
збережений підпис: ");
                        string faleName2 = Console.ReadLine();
                        desktopFilePath = Path.Combine(desktopPath, faleName2);
                        File.WriteAllBytes(desktopFilePath, fileData);
                        // Робимо копію файла на робочому столі
                        //File.WriteAllBytes(desktopFilePath2, fileData);

                        // Зчитування файлу
                        byte[] fileCopyData = File.ReadAllBytes(desktopFilePath);
```

```
                    // Додаємо дані з файлу до об'єкта для підпису
dataToSignBuilder.Append(Encoding.UTF8.GetString(fileCopyData));
                                                                    dataToSign =
Encoding.UTF8.GetBytes(dataToSignBuilder.ToString());

                    // Зберігаємо підпис у файлі на робочому столі
                    //File.WriteAllText(desktopFilePath, signatureHex);
                        //Console.WriteLine("Копію файлу з підписом створено на
робочому столі.");

                    // Підписуємо дані
                    signature = SignData(dataToSign, rsa);

                            // Зберігаємо підпис у файлі на робочому столі, не
перезаписуючи його
                    AppendSignatureToFile(desktopFilePath, signature);
                        Console.WriteLine("Підпис збережено в файлі на робочому
столі.");
Console.WriteLine("_____
_____");
                    break;

                case "3":
                    // Виконати дії для опції 3
                    Console.WriteLine("Ви вибрали опцію 3.");

                    // Перевіряємо наявність підпису в файлі
                            bool hasSignature = HasSignatureInFile(desktopFilePath,
signature);

                    if (hasSignature)
                    {
                       Console.WriteLine("Підпис знайдено в файлі.");
                    }
                    else
```

```
                {
                    Console.WriteLine("Підпис не знайдено в файлі.");
                }
Console.WriteLine("_____
______");
                break;

            case "4":
                // Виконати дії для опції 4
                Console.WriteLine("Ви вибрали опцію 4.");

                // Видаляємо підпис з файлу
                RemoveSignatureFromFile(desktopFilePath, signature);
Console.WriteLine("_____
______");
                break;

            case "~":
                // Повертаємося назад до попереднього меню
                if (menuStack.Count > 0)
                {
                    menuStack.Pop(); // Видаляємо останню опцію зі стеку
Console.WriteLine("_____
______");
                    return;
                }
                else
                {
                                Console.WriteLine("Немає попередніх меню для
повернення.");
                }
                break;

            case "0":
                // Вихід з програми
```

```
                    Console.WriteLine("Дякую за використання програми. До побачення!");
                    return;

                default:
                    Console.WriteLine("Невірний вибір. Будь ласка, виберіть дійсний номер опції.");
                    break;
                }
            }

        }
        finally
        {
            // Звільняємо ресурси ключів
            rsa.PersistKeyInCsp = false;
        }
    }
}

// Код Хеш-функції: SHA-256, SHA-384, та SHA-512
static void Ka2_2Menu(Stack<string> menuStack)
{
Console.WriteLine("_____
_____");
        Console.WriteLine("Ви у розділі - Хеш-функції: SHA-256, SHA-384, та SHA-512");
                                                    string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        string inputString = "Hello, World!";
        // Обчислити SHA-256, SHA-384 і SHA-512 хеші
        string sha256Hash = CalculateSHA256Hash(inputString);
        string sha384Hash = CalculateSHA384Hash(inputString);
        string sha512Hash = CalculateSHA512Hash(inputString);
        try
        {
```

```
            while (true) // Безкінечний цикл для повторного виконання меню
            {
                Console.WriteLine("Меню:");
                Console.WriteLine("1. Хеш-функції: SHA-256");
                Console.WriteLine("2. Хеш-функції: SHA-384");
                Console.WriteLine("3. Хеш-функції: SHA-512");
                Console.WriteLine("~. Повернутися до головного меню");
                Console.WriteLine("0. Вийти з програми");
                Console.Write("Введіть номер опції: ");

                string userInput = Console.ReadLine();
Console.WriteLine("_____
_____");
                // Обробка помилок вводу
                //if (!int.TryParse(userInput, out int option) || (option < 0 || option > 3))
                //{
                     //Console.WriteLine("Невірний ввід. Будь ласка, введіть дійсний
номер опції.");
                  // continue;
                //}

                switch (userInput)
                {
                   case "1":
                      // Виконати дії для опції 1
                      Console.WriteLine("Ви вибрали опцію 1.");
                      string faleName1 = "SHA256Hash.txt";
                      string filePath1 = Path.Combine(desktopPath, faleName1);
                      //Перевірка наявності файлу перед збереженням
                      if (File.Exists(filePath1))
                      {
                          Console.WriteLine("Файл вже існує. Ви бажаєте перезаписати
його? (Y/N)");
                         string overwriteChoice = Console.ReadLine();
                         if (overwriteChoice.ToLowerInvariant() != "y")
                         {
```

```
                    Console.WriteLine("Операція скасована.");
                    continue;
                }
            }
            // Зберегти SHA-256 хеш у файл
            SaveHashToFile(filePath1, sha256Hash);
            Console.WriteLine("Hashes 256 saved to files.");
            // Перевірка цілісності даних при читанні
            bool isIntegrityVerified1 = VerifyDataIntegrity(filePath1);

            if (isIntegrityVerified1)
            {
                Console.WriteLine("Цілісність даних підтверджена.");
            }
            else
            {
                Console.WriteLine("Цілісність даних порушена.");
            }
Console.WriteLine("_____
_____");
            break;

        case "2":
            // Виконати дії для опції 2
            Console.WriteLine("Ви вибрали опцію 2.");
            string faleName2 = "SHA384Hash.txt";
            string filePath2 = Path.Combine(desktopPath, faleName2);
            //Перевірка наявності файлу перед збереженням
            if (File.Exists(filePath2))
            {
                Console.WriteLine("Файл вже існує. Ви бажаєте перезаписати
його? (Y/N)");
                string overwriteChoice = Console.ReadLine();
                if (overwriteChoice.ToLowerInvariant() != "y")
                {
                    Console.WriteLine("Операція скасована.");
```

```
                    continue;
                }
            }
            // Зберегти SHA-384 хеш у файл
            SaveHashToFile(filePath2, sha384Hash);
            Console.WriteLine("Hashes 384 saved to files.");
            // Перевірка цілісності даних при читанні
            bool isIntegrityVerified2 = VerifyDataIntegrity(filePath2);

            if (isIntegrityVerified2)
            {
                Console.WriteLine("Цілісність даних підтверджена.");
            }
            else
            {
                Console.WriteLine("Цілісність даних порушена.");
            }
Console.WriteLine("_____
_____");
            break;

        case "3":
            // Виконати дії для опції 3
            Console.WriteLine("Ви вибрали опцію 3.");
            string faleName3 = "SHA512Hash.txt";
            string filePath3 = Path.Combine(desktopPath, faleName3);
            //Перевірка наявності файлу перед збереженням
            if (File.Exists(filePath3))
            {
                Console.WriteLine("Файл вже існує. Ви бажаєте перезаписати
його? (Y/N)");
                string overwriteChoice = Console.ReadLine();
                if (overwriteChoice.ToLowerInvariant() != "y")
                {
                    Console.WriteLine("Операція скасована.");
                    continue;
```

```
                    }
                }
                // Зберегти SHA-512 хеш у файл
                SaveHashToFile(filePath3, sha512Hash);
                Console.WriteLine("Hashes 512 saved to files.");
                // Перевірка цілісності даних при читанні
                bool isIntegrityVerified3 = VerifyDataIntegrity(filePath3);

                if (isIntegrityVerified3)
                {
                    Console.WriteLine("Цілісність даних підтверджена.");
                }
                else
                {
                    Console.WriteLine("Цілісність даних порушена.");
                }
Console.WriteLine("_____
_____");
                break;

            case "~":
                // Повертаємося назад до попереднього меню
                if (menuStack.Count > 0)
                {
                    menuStack.Pop(); // Видаляємо останню опцію зі стеку
Console.WriteLine("_____
_____");
                    return;
                }
                else
                {
                                Console.WriteLine("Немає попередніх меню для
повернення.");
                }
                break;
```

```
                case "0":
                    // Вихід з програми
                        Console.WriteLine("Дякую за використання програми. До побачення!");
                    return;

                default:
                    Console.WriteLine("Невірний вибір. Будь ласка, виберіть дійсний номер опції.");
                    break;
                }
            }
        }
        // Обробка винятків
        catch (Exception ex)
        {
            Console.WriteLine($"Помилка: {ex.Message}");
        }
    }

    // Код Аутентифікація та обмін ключами: Diffie-Hellman Key Exchange
    static void Ka2_3Menu(Stack<string> menuStack)
    {
Console.WriteLine("_____
_____");
            Console.WriteLine("Ви у розділі - Аутентифікація та обмін ключами: Diffie-Hellman Key Exchange");
                                                    string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        try
        {
            while (true) // Безкінечний цикл для повторного виконання меню
            {
                Console.WriteLine("Меню:");
                    Console.WriteLine("1. Аутентифікація та обмін ключами: Diffie-Hellman Key Exchange");
```

```
                Console.WriteLine("~. Повернутися до головного меню");
                Console.WriteLine("0. Вийти з програми");
                Console.Write("Введіть номер опції: ");

                string userInput = Console.ReadLine();
Console.WriteLine("_____
_____");

                switch (userInput)
                {
                   case "1":
                      // Генерація великих простих чисел та генератора
                      using (var rsa = new RSACryptoServiceProvider(2048))
                      {
                         var parameters = rsa.ExportParameters(true);

                         // Перевірка довжини модуля RSA
                         if (parameters.Modulus.Length < 2048 / 8)
                         {
                            Console.WriteLine("Недостатня довжина модуля RSA.");
                            return;
                         }

                         BigInteger prime = new BigInteger(parameters.Modulus);
                               BigInteger generator = new BigInteger(new byte[] { 2 }); //
Загальноприйнятий генератор для Діффі-Геллмана

                         // Обрані Особою1 та Особою2 приватні ключі
                                        using (SecureString alicePrivateKeySecure =
GeneratePrivateKey(prime))
                                         using (SecureString bobPrivateKeySecure =
GeneratePrivateKey(prime))
                         {
                                                        BigInteger alicePrivateKey =
ConvertToBigInteger(alicePrivateKeySecure);
```

```
                                                    BigInteger bobPrivateKey =
ConvertToBigInteger(bobPrivateKeySecure);

                        // Обчислення відкритих ключів
                            BigInteger alicePublicKey = CalculatePublicKey(generator,
prime, alicePrivateKey);
                             BigInteger bobPublicKey = CalculatePublicKey(generator,
prime, bobPrivateKey);

                        // Обмін публічними ключами
                          BigInteger sharedKeyA = CalculateSharedKey(bobPublicKey,
prime, alicePrivateKey);
                         BigInteger sharedKeyB = CalculateSharedKey(alicePublicKey,
prime, bobPrivateKey);

                        // Перевірка, що обидва спільних ключі рівні
                        Console.WriteLine("Спільний ключ A: " + sharedKeyA);
                        Console.WriteLine("Спільний ключ B: " + sharedKeyB);

                        if (sharedKeyA == sharedKeyB)
                        {
                              Console.WriteLine("Спільні ключі рівні. Збереження у
файл...");

                           // Захист виводу шляху файлу
                           string fileName = "shared_key.txt";
                           string filePath = Path.Combine(desktopPath, fileName);
                           SaveKeyToFile(filePath, sharedKeyA);
                        }
                        else
                        {
                           Console.WriteLine("Спільні ключі не рівні.");
                        }
                    }
                }
```

```
Console.WriteLine("_____
_____");
                    break;

                case "~":
                    // Повертаємося назад до попереднього меню
                    if (menuStack.Count > 0)
                    {
                        menuStack.Pop(); // Видаляємо останню опцію зі стеку
Console.WriteLine("_____
_____");
                        return;
                    }
                    else
                    {
                                Console.WriteLine("Немає попередніх меню для
повернення.");
                    }
                    break;

                case "0":
                    // Вихід з програми
                        Console.WriteLine("Дякую за використання програми. До
побачення!");
                    return;

                default:
                    Console.WriteLine("Невірний вибір. Будь ласка, виберіть дійсний
номер опції.");
                    break;
                }
            }
        }
        // Обробка винятків
        catch (Exception ex)
        {
```

```
            Console.WriteLine($"Помилка: {ex.Message}");
        }
    }

    // AES
    // Метод для шифрування файлу
    static byte[] EncryptFile(byte[] plainBytes, byte[] key, byte[] IV)
    {
        using (Aes aesAlg = Aes.Create())
        {
            aesAlg.Key = key;
            aesAlg.IV = IV;

            // Створення об'єкта, який виконує процес шифрування
                ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
aesAlg.IV);

            using (MemoryStream msEncrypt = new MemoryStream())
            {
                    using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
encryptor, CryptoStreamMode.Write))
                {
                    // Запис зашифрованих даних в потік
                    csEncrypt.Write(plainBytes, 0, plainBytes.Length);
                }

                // Повернення зашифрованих даних у вигляді масиву байтів
                return msEncrypt.ToArray();
            }
        }
    }

    // Метод для розшифрування файлу
    static byte[] DecryptFile(byte[] encryptedBytes, byte[] key, byte[] IV)
    {
        using (Aes aesAlg = Aes.Create())
```

```
            {
                aesAlg.Key = key;
                aesAlg.IV = IV;

                // Створення об'єкта, який виконує процес розшифрування
                        ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key,
aesAlg.IV);

                using (MemoryStream msDecrypt = new MemoryStream(encryptedBytes))
                {
                            using (CryptoStream csDecrypt = new CryptoStream(msDecrypt,
decryptor, CryptoStreamMode.Read))
                    {
                        using (MemoryStream msPlainText = new MemoryStream())
                        {
                            csDecrypt.CopyTo(msPlainText);

                            // Повернути розшифрований текст як масив байтів
                            return msPlainText.ToArray();
                        }
                    }
                }
            }
        }

        // Метод для перевірки цілісності розшифрованого тексту
        static bool VerifyIntegrity(byte[] decryptedBytes, string filePath)
        {
            byte[] originalFileBytes = File.ReadAllBytes(filePath);

                // Порівняти розшифрований текст і оригінальний файл за допомогою
SequenceEqual
                                                                    return
StructuralComparisons.StructuralEqualityComparer.Equals(decryptedBytes,
originalFileBytes);
        }
```

```
// RSA
// Збереження ключа у файл
static void SaveKeyToFile(string filePath, RSAParameters key)
{
   using (StreamWriter sw = new StreamWriter(filePath))
   {
      sw.WriteLine(Convert.ToBase64String(key.Modulus));
      sw.WriteLine(Convert.ToBase64String(key.Exponent));
      sw.WriteLine(Convert.ToBase64String(key.D));
      sw.WriteLine(Convert.ToBase64String(key.P));
      sw.WriteLine(Convert.ToBase64String(key.Q));
      sw.WriteLine(Convert.ToBase64String(key.DP));
      sw.WriteLine(Convert.ToBase64String(key.DQ));
      sw.WriteLine(Convert.ToBase64String(key.InverseQ));
   }
}

// Завантаження ключа із файлу
static RSAParameters LoadKeyFromFile(string filePath)
{
   RSAParameters key = new RSAParameters();
   using (StreamReader sr = new StreamReader(filePath))
   {
      key.Modulus = Convert.FromBase64String(sr.ReadLine());
      key.Exponent = Convert.FromBase64String(sr.ReadLine());
      key.D = Convert.FromBase64String(sr.ReadLine());
      key.P = Convert.FromBase64String(sr.ReadLine());
      key.Q = Convert.FromBase64String(sr.ReadLine());
      key.DP = Convert.FromBase64String(sr.ReadLine());
      key.DQ = Convert.FromBase64String(sr.ReadLine());
      key.InverseQ = Convert.FromBase64String(sr.ReadLine());
   }
   return key;
}
```

```
        // Зашифрування файлу за допомогою алгоритму RSA
            static void EncryptFile(string inputFile, string outputFile, RSAParameters publicKey)
        {
                                using (RSACryptoServiceProvider rsaEncrypt = new RSACryptoServiceProvider())
            {
              rsaEncrypt.ImportParameters(publicKey);

              using (FileStream inputStream = File.OpenRead(inputFile))
              using (FileStream outputStream = File.Create(outputFile))
              {
                int keySize = rsaEncrypt.KeySize / 8;
                int blockSize = keySize - 11; // Розмір блока для RSA PKCS#1 v1.5
                int bufferSize = blockSize;

                byte[] buffer = new byte[bufferSize];
                int bytesRead;

                while ((bytesRead = inputStream.Read(buffer, 0, bufferSize)) > 0)
                {
                                                                byte[] encryptedBytes = rsaEncrypt.Encrypt(buffer.Take(bytesRead).ToArray(), false);
                    outputStream.Write(encryptedBytes, 0, encryptedBytes.Length);
                }
              }
            }
        }

        // Розшифрування файлу за допомогою приватного ключа
            static bool DecryptFile(string inputFile, string outputFile, RSAParameters privateKey)
        {
            try
            {
```

```
                using (RSACryptoServiceProvider rsaDecrypt = new RSACryptoServiceProvider())
                {
                    rsaDecrypt.ImportParameters(privateKey);

                    using (FileStream inputStream = File.OpenRead(inputFile))
                    using (FileStream outputStream = File.Create(outputFile))
                    {
                        int keySize = rsaDecrypt.KeySize / 8;
                        int blockSize = keySize; // Розмір блока для RSA PKCS#1 v1.5
                        int bufferSize = blockSize;

                        byte[] buffer = new byte[bufferSize];
                        int bytesRead;

                        while ((bytesRead = inputStream.Read(buffer, 0, bufferSize)) > 0)
                        {
                            byte[] decryptedBytes = rsaDecrypt.Decrypt(buffer.Take(bytesRead).ToArray(), false);
                            outputStream.Write(decryptedBytes, 0, decryptedBytes.Length);
                        }
                    }
                }

                return true; // Повертаємо true, якщо розшифрування пройшло успішно
            }
            catch (CryptographicException ex)
            {
                Console.WriteLine("Помилка розшифрування: " + ex.Message);
                return false; // Повертаємо false у випадку помилки розшифрування
            }
        }

        // Перевірка цілісності файлу
        static bool VerifyIntegrity(string originalFilePath, string decryptedFilePath)
        {
```

```
    // Зчитуємо байти обох файлів
    byte[] originalFileBytes = File.ReadAllBytes(originalFilePath);
    byte[] decryptedFileBytes = File.ReadAllBytes(decryptedFilePath);

    // Перевіряємо, чи файли мають однаковий розмір
    if (originalFileBytes.Length != decryptedFileBytes.Length)
    {
        return false;
    }

    // Порівнюємо файли байт-по-байту
    for (int i = 0; i < originalFileBytes.Length; i++)
    {
        if (originalFileBytes[i] != decryptedFileBytes[i])
        {
            return false;
        }
    }
    return true;
}

// Електронний підпис
// Метод для створення ЕЦП для даних
public static byte[] SignData(byte[] data, RSACryptoServiceProvider rsa)
{
    // Створення підпису
    return rsa.SignData(data, CryptoConfig.MapNameToOID("SHA256"));
}

// Метод для перевірки наявності підпису в файлі
public static bool HasSignatureInFile(string desktopFilePath, byte[] signature)
{
    // Зчитуємо вміст файлу у вигляді байтів
    byte[] fileData = File.ReadAllBytes(desktopFilePath);

    // Перевірка наявності підпису в байтовому масиві
```

```
        return FindSignatureIndex(fileData, signature) >= 0;
    }

    // Метод для видалення підпису з файлу
        public static void RemoveSignatureFromFile(string desktopFilePath, byte[] signature)
    {
        // Зчитуємо вміст файлу у вигляді байтів
        byte[] fileData = File.ReadAllBytes(desktopFilePath);

        // Пошук позиції підпису у файлі
        int signatureIndex = FindSignatureIndex(fileData, signature);

        if (signatureIndex >= 0)
        {
            // Видаляємо підпис, розширюючи або обрізаючи байтовий масив за необхідністю
            byte[] newData = new byte[fileData.Length - signature.Length];
            Array.Copy(fileData, 0, newData, 0, signatureIndex);
            Array.Copy(fileData, signatureIndex + signature.Length, newData, signatureIndex, fileData.Length - (signatureIndex + signature.Length));

            // Зберігаємо оновлений вміст файлу
            File.WriteAllBytes(desktopFilePath, newData);
            Console.WriteLine("Підпис видалено з файлу.");
        }
        else
        {
            Console.WriteLine("Підпис не знайдено в файлі.");
        }
    }

    // Метод для знаходження позиції підпису у байтовому масиві
    public static int FindSignatureIndex(byte[] data, byte[] signature)
    {
        for (int i = 0; i <= data.Length - signature.Length; i++)
```

```
            {
                bool found = true;
                for (int j = 0; j < signature.Length; j++)
                {
                    if (data[i + j] != signature[j])
                    {
                        found = false;
                        break;
                    }
                }
                if (found)
                {
                    return i;
                }
            }
            return -1; // Підпис не знайдено
        }

        // Метод для додавання підпису до файлу
            public static void AppendSignatureToFile(string desktopFilePath, byte[] signature)
        {
            if (desktopFilePath is null)
            {
                throw new ArgumentNullException(nameof(desktopFilePath));
            }

                using (FileStream fs = new FileStream(desktopFilePath, FileMode.Append, FileAccess.Write))
            {
                fs.Write(signature, 0, signature.Length);
            }
        }

        //SHA-256, SHA-384 і SHA-512
        //Зберегти в файлі
```

```
        static void SaveHashToFile(string filePath, string data)
        {
            using (StreamWriter writer = new StreamWriter(filePath))
            {
                // Збереження даних
                writer.WriteLine(data);

                // Обчислення та збереження хеш-значення
                string hash = CalculateSHA256Hash(data);
                writer.WriteLine($"Hash: {hash}");
            }
        }

        //Перевірка цілісності даних
        static bool VerifyDataIntegrity(string filePath)
        {
            try
            {
                using (StreamReader reader = new StreamReader(filePath))
                {
                    // Зчитування даних
                    string data = reader.ReadLine();

                    // Зчитування збереженого хеш-значення
                    string savedHashLine = reader.ReadLine();
                        string savedHash = savedHashLine.Split(' ')[1]; // Отримання хеш-значення з рядка

                    // Обчислення хеш-значення для перевірки
                    string calculatedHash = CalculateSHA256Hash(data);

                    // Порівняння хеш-значень
                                        return string.Equals(calculatedHash, savedHash, StringComparison.OrdinalIgnoreCase);
                }
            }
```

```
    catch (Exception ex)
    {
        Console.WriteLine($"Помилка: {ex.Message}");
        return false;
    }
}

//Використання SHA-256
static string CalculateSHA256Hash(string input)
{
    using (SHA256 sha256 = SHA256.Create())
    {
        // Конвертує введені дані в байтовий масив
        byte[] inputBytes = Encoding.UTF8.GetBytes(input);

        // Обчислює SHA-256 хеш
        byte[] hashBytes = sha256.ComputeHash(inputBytes);

        // Перетворює байти хешу в рядок в шістнадцятковому форматі (hex)
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < hashBytes.Length; i++)
        {
            sb.Append(hashBytes[i].ToString("x2"));
        }

        // Повертає отриманий SHA-256 хеш у вигляді рядка
        return sb.ToString();
    }
}

//Використання SHA-384
static string CalculateSHA384Hash(string input)
{
    using (SHA384 sha384 = SHA384.Create())
    {
        // Конвертує введені дані в байтовий масив
```

```
        byte[] inputBytes = Encoding.UTF8.GetBytes(input);

        // Обчислює SHA-384 хеш
        byte[] hashBytes = sha384.ComputeHash(inputBytes);

        // Перетворює байти хешу в рядок в шістнадцятковому форматі (hex)
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < hashBytes.Length; i++)
        {
            sb.Append(hashBytes[i].ToString("x2"));
        }

        // Повертає отриманий SHA-384 хеш у вигляді рядка
        return sb.ToString();
    }
}

//Використання SHA-512
static string CalculateSHA512Hash(string input)
{
    using (SHA512 sha512 = SHA512.Create())
    {
        // Конвертує введені дані в байтовий масив
        byte[] inputBytes = Encoding.UTF8.GetBytes(input);

        // Обчислює SHA-512 хеш
        byte[] hashBytes = sha512.ComputeHash(inputBytes);

        // Перетворює байти хешу в рядок в шістнадцятковому форматі (hex)
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < hashBytes.Length; i++)
        {
            sb.Append(hashBytes[i].ToString("x2"));
        }

        // Повертає отриманий SHA-512 хеш у вигляді рядка
```

```
            return sb.ToString();
        }
    }

    //Diffie-Hellman Key Exchange
    // Генерація випадкового приватного ключа
    static SecureString GeneratePrivateKey(BigInteger prime)
    {
        using (RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider())
        {
                byte[] randomBytes = new byte[32]; // змініть розмір відповідно до потреб
            rng.GetBytes(randomBytes);
                // Отримання випадкового приватного ключа та конвертація його в SecureString
                return ConvertToSecureString(BigInteger.Remainder(BigInteger.Abs(new BigInteger(randomBytes)), prime - 2) + 1);
        }
    }

    // Конвертація великого числа в SecureString
    static SecureString ConvertToSecureString(BigInteger number)
    {
        SecureString secureString = new SecureString();
        // Додає кожну цифру з рядка представлення числа в SecureString
        foreach (char digit in number.ToString())
        {
            secureString.AppendChar(digit);
        }
        return secureString;
    }

    static BigInteger ConvertToBigInteger(SecureString secureString)
    {
        IntPtr ptr = IntPtr.Zero;
        try
```

```
        {
            // Конвертація SecureString в Unicode-рядок (string) через BSTR (Basic String)
                                                                    ptr = System.Runtime.InteropServices.Marshal.SecureStringToBSTR(secureString);
            string str = System.Runtime.InteropServices.Marshal.PtrToStringBSTR(ptr);

            // Кодування Unicode-рядка в масив байтів
            byte[] bytes = Encoding.Unicode.GetBytes(str);

            // Створення об'єкта BigInteger з масиву байтів
                // Об'єкт BigInteger, який представляє велике ціле число, отримане з рядка, отриманого із захищеного рядка.
            return new BigInteger(bytes);
        }
        finally
        {
            // Очищення виділеної пам'яті для BSTR
            if (ptr != IntPtr.Zero)
            {
                System.Runtime.InteropServices.Marshal.ZeroFreeBSTR(ptr);
            }
        }
    }

    // Обчислення відкритого ключа
        static BigInteger CalculatePublicKey(BigInteger generator, BigInteger prime, BigInteger privateKey)
    {
        // Відкритий ключ (BigInteger), отриманий шляхом взяття генератора до ступеня приватного ключа по модулю простого числа.
        return BigInteger.ModPow(generator, privateKey, prime);
    }

    // Обчислення спільного ключа
```

```
        static BigInteger CalculateSharedKey(BigInteger publicKey, BigInteger prime, BigInteger privateKey)
        {
            // Спільний ключ (BigInteger), отриманий шляхом взяття публічного ключа до ступеня приватного ключа по модулю простого числа.
            return BigInteger.ModPow(publicKey, privateKey, prime);
        }

        // Збереження ключа у файл
        static void SaveKeyToFile(string filePath, BigInteger key)
        {
            using (StreamWriter writer = new StreamWriter(filePath))
            {
                writer.Write(key.ToString());
            }
        }
    }
```