Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

”Допущено до захисту”
Завідувач кафедри кібербезпеки
та програмного забезпечення
д.т.н., професор
_____ Олексій СМІРНОВ
“____” _____ 2025 р.

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА
за першим (бакалаврським) рівнем вищої освіти
на тему
“Програмне забезпечення системи для підтвердження авторства текстових документів”

Виконав здобувач вищої освіти
IV курсу, групи КІ-22мб
ОПП «Комп`ютерна інженерія»
спеціальності 123 «Комп`ютерна інженерія»
_____ Протасов Р. Б.
«____» _____ 2025 р.

Керівник проекту
доктор технічних наук, професор
_____Улічев О.С
«____» _____ 2025 р.
Рецензент _____
_____

м. Кропивницький

Центральноукраїнський національний технічний університет
Факультет *Механіко-технологічний*
Кафедра *Кібербезпеки та програмного забезпечення*
Рівень вищої освіти *бакалавр*
Галузь знань ._12 _____*"*Інформаційні технології*"*_____
Спеціальність *123 "*Комп'ютерна інженерія*"*
Освітньо-професійна (освітньо-наукова) програма _*"*Комп'ютерна інженерія*"*_

ЗАТВЕРДЖУЮ
Завідувач кафедри
д.т.н., проф.
_____ Олексій СМІРНОВ
"____" _____2025 року

ЗАВДАННЯ НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ ЗА ПЕРШИМ (БАКАЛАВРСЬКИМ) РІВНЕМ ВИЩОЇ ОСВІТИ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ

Протасова Романа Богдановича
(прізвище, ім'я, по батькові)

1. Тема роботи *Програмне забезпечення системи для підтвердження авторства текстових документів*
2. Керівник роботи *Улічев Олександр Сергієвич, канд. техн. наук, доцент*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від "17" січня 2025 року №48-02
3. Строк подання роботи до захисту *23.05.2025 р.*
4. Мета та завдання випускної кваліфікаційної роботи *Метою роботи є розробка програмного забезпечення системи для підтвердження авторства текстових документів*
5. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Призначення та область використання.
2.Перегляд аналогічних існуючих систем.
3. Опис і обґрунтування проектних рішень.
4. Етапи програмування системи.
5. Впровадження системи в промислову експлуатацію
6. Висновки.
6. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Структурна схема системи	*1 аркуш*
Функціональна схема системи	*1 аркуш*
Діаграма процесів	*1 аркуш*
Блок-схема алгоритму роботи додатку	*2 аркуша*

6. Дата видачі завдання «17» січня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної бакалаврської роботи	Строк виконання етапів кваліфікаційної бакалаврської роботи	Примітка
1.	Аналіз існуючих систем	10.03.2025 р.	
2.	Постановка задачі, оформлення ТЗ	15.03.2025 р.	
3.	Розробка моделі компонента	20.03.2025 р.	
4.	Розробка структур даних	25.03.2025 р.	
5.	Розробка алгоритмів зв’язку та відображення	30.03.2025 р.	
6.	Програмування алгоритмів	10.04.2025 р.	
7.	Оформлення ПЗ	17.04.2025 р.	
8.	Попередній захист роботи	22.05.2025 р.	

Дата видачі завдання
« »_____2025 р.

Підпис керівника
_____ Улічев О.С

Завдання прийнято до виконання
« »_____2025 р.

Підпис здобувача
_____ Протасов Р.Б

АНОТАЦІЯ

Протасов Р.Б. Програмне забезпечення системи для підтвердження авторства текстових документів. 123 Комп`ютерна інженерія. Центральноукраїнський національний технічний університет. Кропивницький. 2025.

У даній кваліфікаційній бакалаврській роботі розроблено програмне забезпечення, що забезпечує верифікацію авторства цифрових документів шляхом формування унікального цифрового відбитка (хешу). Система призначена для підпису документу цифровою міткою після створення або модифікації документа, а також збереження контрольної інформації, яка дозволяє підтвердити справжність його походження.

Метою роботи є створення ефективного інструменту для захисту цифрових документів від несанкціонованих змін та підробок, що може бути використаний у сферах юриспруденції, електронного документообігу та безпечного архівування.

У процесі реалізації проведено дослідження сучасних криптографічних алгоритмів та методів хешування, проаналізовано актуальні підходи до timestamp-верифікації, розроблено власну архітектуру програмного рішення з відкритим клієнтським інтерфейсом.

Програмне забезпечення має зручний інтерфейс користувача та реалізоване у вигляді настільного додатку з використанням C++ та бібліотеки Qt. Підтримується обробка файлів форматів PDF.

Ключові слова: цифровий відбиток, підтвердження авторства, хеш-функція, timestamp, верифікація документа, Qt, C++.

ABSTRACT

Protasov R.B. Software for text document authorship verification system. 123 Computer engineering. Central Ukrainian National Technical University. Kropyvnytskyi 2025.

This qualified bachelor's work has a separate security program that ensures verification of the authorship of digital documents by generating a unique digital bitmap (hash). The system is designed to sign a document with a digital mark after creation or modification of the document, as well as to save control information that allows you to confirm the validity of your transaction.

The method of work is the creation of an effective tool for the protection of digital documents from unauthorized changes and details that can be used in the fields of jurisprudence, electronic document management and secure archiving.

During the implementation process, current cryptographic algorithms and hashing methods were further investigated, current approaches to timestamp verification were analyzed, and the power architecture of the software solution was broken down Open the client interface.

The software has a user-friendly interface and is implemented in a desktop view using C++ and the Qt library. Processing of files in PDF formats is supported.

Key words: digital bit, proof of authorship, hash function, timestamp, document verification, Qt, C++.

ЗМІСТ

ВСТУП

Забезпечення достовірності авторства цифрових документів набуває особливої ваги в умовах інтенсивного обміну інформацією та зростаючого ризику фальсифікацій у сучасному інформаційному просторі. Без належних інструментів підтвердження авторства організації та окремі користувачі стикаються з небезпекою неправомірного використання текстового контенту, втрати інтелектуальної власності та зниження рівня довіри до електронних документів.

Предметом даного дослідження є сукупність методів, алгоритмів та програмних рішень, що забезпечують автоматизовану верифікацію авторства цифрових документів. Особлива увага зосереджується на застосуванні хеш-функцій, криптографічного підпису, аналізу стилістичних, семантичних та структурних ознак текстів, а також інтеграції цих підходів у єдину програмну платформу. Дослідження охоплює засоби формалізації авторських ознак, технології обчислення цифрових відбитків, системи порівняння цифрових слідів і механізми криптографічного підтвердження достовірності документів. Крім того, вивчаються способи забезпечення точності, стійкості до фальсифікацій та інтерпретованості результатів перевірки для кінцевого користувача.

Об’єктом дослідження виступають електронні документальні файли (зокрема, у форматах PDF, DOCX, TXT), що потребують перевірки автентичності та підтвердження авторських прав у цифровому середовищі. До об'єкта також належать процеси, пов’язані з обробкою таких документів: збереження, передача, фіксація часу створення, модифікація та повторна перевірка на відповідність первинному цифровому сліду. У фокусі дослідження перебувають як самі цифрові артефакти, так і середовище їх функціонування, що включає користувацькі інтерфейси, криптографічну інфраструктуру,

timestamp-сервіси та інформаційні системи, з якими може бути інтегрований створений програмний продукт.

Мета й завдання дослідження. Головною метою роботи є створення програмного продукту для автоматизованої верифікації авторського права на документальні файли. Для реалізації цієї мети передбачено виконати такі завдання:

– Провести аналіз існуючих підходів і технологій ідентифікації авторства текстових та мультимедійних документів.

– Розробити математичні моделі та алгоритми оцінки стилістичних, семантичних і структурних ознак авторського письма.

– Реалізувати програмну платформу, що забезпечить інтеграцію розроблених алгоритмів у користувацький інтерфейс із можливістю завантаження документів та отримання звіту про вірогідність авторства.

– Створити шар сувмістності с більшістю популярних контейнерім криптографічної інформації.

– Реалізувати систему підпису що не буде ніяквпливати на вміст документу.

– Реалізувати систему адресації помилок для чіткого розуміння коли и чому програма не працьює.

– Підготовити на програмному рівни методи запобігання некоректної обробки документу.

Практична цінність отриманих результатів полягає в тому, що реалізоване програмне забезпечення дозволить:

– Автоматизувати процес перевірки авторства та знизити людський фактор у прийнятті рішень.

– Забезпечити надійну фіксацію права інтелектуальної власності на електронні документи.

– Підвищити рівень юридичної захищеності щодо претензій на плагіат та неправомірне перепризначення авторства.

– Дати можливість для компаній с документацією яка чутлива до копіювання та має високу интелектуальну цінність захистити документи як свою власність.

– Надати програмний продукт, здібний до інтеграції в системі автоматичного документо-обігу.

Таким чином, розробка та впровадження програмного рішення для підтвердження авторства документів є своєчасним завданням, що сприятиме посиленню довіри до електронного документообігу та захисту інтелектуальної власності в цифровому середовищі.

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

1.1 Призначення системи

Система верифікації авторства документів призначена для автоматизованого підтвердження особи або групи осіб, що є первинними авторами текстових та мультимедійних матеріалів. Основні функції системи включають:

– Збір та збереження контрольних відбитків стилістичних, лексичних і синтаксичних ознак користувача на підставі аналізу репрезентативного корпусу його попередніх робіт.

– Порівняння досліджуваного документа з базою відбитків за допомогою комплексного алгоритму оцінки збігів у семантичних, структурних і стилістичних параметрах.

– Формування звіту про достовірність авторства, що містить відсоткові оцінки збігу, рейтингові позиції й візуалізацію внеску окремих перевіряних ознак.

– Інтеграцію з існуючими системами документообігу (хмарні сховища, CMS, EDMS) та інтерфейсом для користувача (веб-інструмент або десктоп-додаток).

До основних завдань системи належать:

– Накопичення методичних даних - формування й підтримка оновлюваної бази прирощуваних прикладів авторських текстів.

– Розробка метрик вимірювання стилістичних збігів, які охоплюють такі напрямки: лексичний аналіз (частота ключових слів, унікальність словникового запасу), синтаксичний аналіз (структура речень, типові синтаксичні конструкції), семантичний аналіз (тематична спрямованість і зв’язність викладу).

– Реалізація алгоритмів машинного навчання для класифікації авторства на основі зібраних характеристик.

– Забезпечення безпеки обробки даних із використанням шифрування для зберігання конфіденційних фрагментів текстів та результатів аналізу.

Таким чином, система адресована фахівцям в галузі інтелектуальної власності, юридичним відділам організацій, освітнім установам та науковим установам, де потрібна оперативна та об’єктивна перевірка авторства документів.

1.2 Область застосування

Розроблена система може бути впроваджена в багатьох напрямах діяльності.

Юридична експертиза:

– Підтримка процесу судових розглядів за позовами про порушення авторських прав.

– Професійна діагностика й обґрунтування претензій щодо плагіату або фальсифікації авторства.

Науково-освітні установи:

– Автоматизований аналіз наукових статей студентів і викладачів з метою виявлення неправомірного запозичення.

– Підтримка процесу рецензування дисертацій та наукових праць шляхом порівняння з існуючими публікаціями.

Видавничі компанії й медіа-ресурси:

– Перевірка рецензованих рукописів і журналістських матеріалів перед публікацією.

– Захист інтелектуальної власності авторів, що співпрацюють з редакцією.

Корпоративний сектор

– Контроль за авторством внутрішніх документів: технічних завдань, звітів, презентацій.

– Підтримка політики захисту конфіденційної інформації та авторських прав співробітників.

Онлайн-платформи та сервіси контенту

–Інтеграція з CMS і платформами для блогерів, журналістів та фрілансерів.

–Можливість автоматичного маркування перевірених матеріалів та видача сертифікатів авторства.

Загалом система застосовується у всіх випадках, де необхідно гарантувати достовірність і захищеність авторського права на електронні документи, знижуючи ризики неправомірного використання інформації та підвищуючи рівень довіри до цифрового контенту.

2 ПЕРЕГЛЯД АНАЛОГІЧНИХ ІСНУЮЧИХ СИСТЕМ

2.1 Огляд існуючих систем, технологій, архітектур та програмних рішень з профілю теми випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти.

Проблематика підтвердження авторства цифрових документів набуває дедалі більшої актуальності в умовах зростання обсягів електронного документообігу, академічного та творчого контенту. Одним із надійних підходів є використання технології генерації цифрового відбитка (хеш-функцій), яка дозволяє забезпечити цілісність та доказовість авторства документа без розголошення його змісту.

Цифровий відбиток (англ. digital fingerprint або hash) - це унікальна послідовність символів фіксованої довжини, яка генерується на основі вмісту документа з використанням криптографічної хеш-функції (наприклад, SHA-256, SHA-3). Зміна навіть одного символу в документі призводить до повної зміни хешу, що унеможливлює непомітне редагування.х

Переглянемо коротко основні приклади сервісів що пропонують користувачу послуги щодо пітдвердження авторства на основі хешкоду.

OpenTimestamps - це децентралізована інфраструктура для створення криптографічних часових міток на основі хешів документів. Основна ідея полягає в тому, щоб забезпечити докази існування певного цифрового вмісту на конкретний момент часу, не розкриваючи сам вміст. Сервіс працює за рахунок вбудовування хешу документа в блокчейн Bitcoin, який служить незмінним і загальнодоступним реєстром.

Унікальність OpenTimestamps полягає в його повністю відкритому коді та незалежності від будь-якої централізованої установи. Це дозволяє користувачам самостійно створювати часові мітки, а також перевіряти їх

достовірність без сторонніх посередників. Інструмент орієнтований на розробників і технічно підкованих користувачів, які можуть інтегрувати його у власні сервіси чи дослідницькі системи.[10]

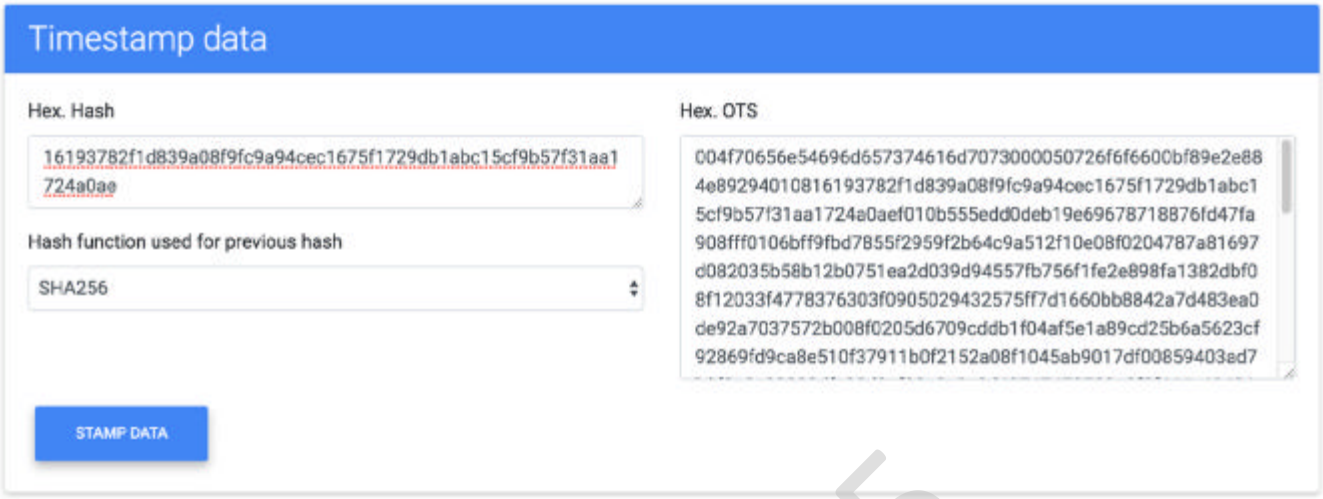

Рисунок 2.1 – Інтерфейс генерації хешкоду OpenTimestamps

OriginStamp - це комерційна онлайн-платформа, яка надає сервіси цифрового хешування та фіксації даних у розподілених реєстрах (блокчейнах). Сервіс дозволяє користувачам завантажити документ, автоматично згенерувати його хеш і зафіксувати цю інформацію в одному з підтримуваних блокчейнів (Bitcoin, Ethereum, Aion та інші). Таким чином, OriginStamp забезпечує довготривалу, незмінну й незалежну від центральних серверів валідацію моменту створення або існування документа.[11]

Платформа також пропонує API для інтеграції у зовнішні системи, мобільні застосунки, CRM або наукові сервіси. Додатковою перевагою є підтримка різних форматів файлів, зручний користувацький інтерфейс і можливість надсилання сповіщень про зміну статусу хешу. Однак функціональність щодо підтвердження особи автора реалізується лише через окремі механізми (електронні підписи, акаунти тощо) і не є вбудованою частиною базової логіки платформи.

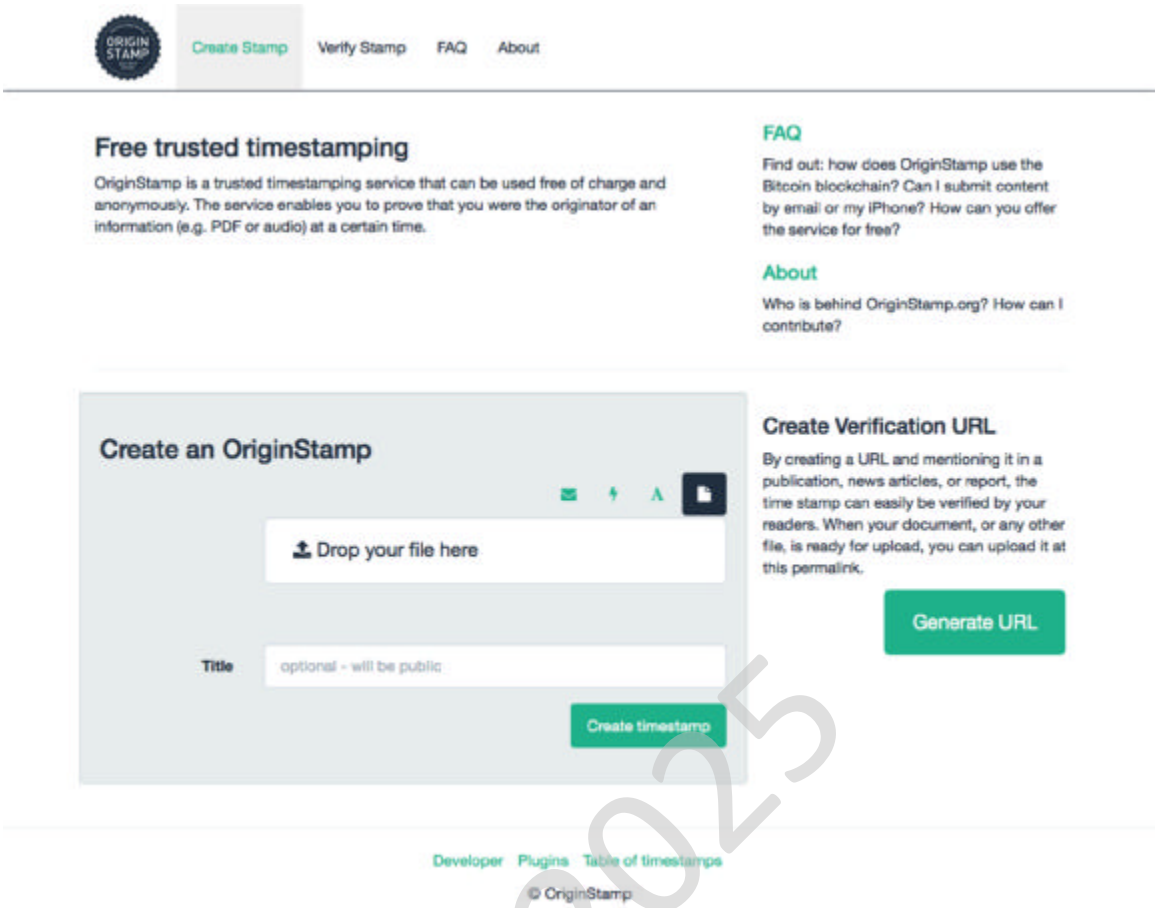

Рисунок 2.2 – Інтерфейс сайту сервісу OriginStamp

DocuSign - це одна з найпопулярніших глобальних платформ для електронного підпису та керування цифровими транзакціями. Сервіс дає змогу користувачам підписувати документи в юридично визнаний спосіб, забезпечуючи як автентифікацію підписанта, так і фіксацію цифрового відбитка документа. Хешування є складовою частиною технічного процесу цифрового підписання, завдяки чому зберігається цілісність даних.

DocuSign відповідає міжнародним стандартам (eIDAS, ESIGN, UETA), що робить його придатним для використання в юридичних, фінансових та академічних контекстах. Платформа також пропонує додаткові функції - ведення журналу змін, перевірку часу підписання, багаторівневу автентифікацію користувачів, шаблони документів та API для корпоративної інтеграції. Хоча основний фокус DocuSign - юридично значущий підпис,

функції хешування документа присутні опосередковано як частина загального механізму безпеки.

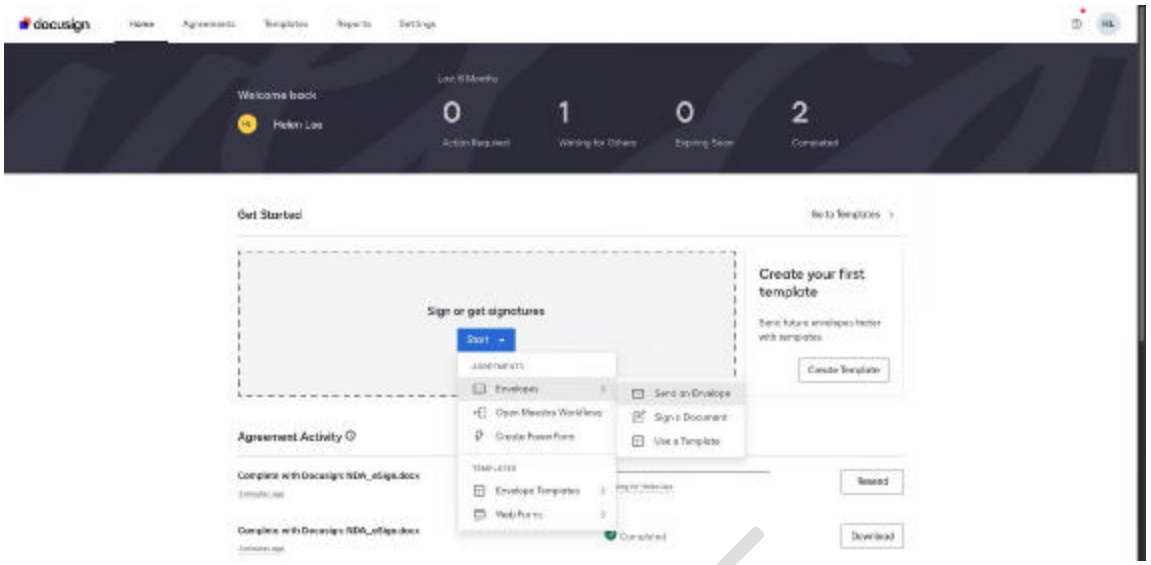

Рисунок 2.3 – Інтерфейс сайту сервісу DocuSign

GNU Privacy Guard (GPG) - це потужна система з відкритим кодом, яка реалізує відкриті стандарти шифрування, цифрового підпису, а також генерації та перевірки хешів. Розроблена як вільна альтернатива PGP (Pretty Good Privacy), GPG дозволяє користувачам створювати унікальні пари криптографічних ключів, використовувати їх для підписання документів, а також перевіряти автентичність авторства.

Однією з найважливіших функцій GPG є створення цифрових відбитків файлів - хешів, які можна опублікувати або передати незалежно від самого документа. Це дозволяє стороннім особам переконатися в незмінності документа та у правдивості підпису. GPG має багатофункціональний інтерфейс командного рядка, що робить його зручним для автоматизації та інтеграції в різні дослідницькі та серверні середовища. Проте використання GPG вимагає певної технічної підготовки, що може бути бар’єром для не технічних користувачів.[35]

2.2 Обґрунтування вибору засобів для побудови системи та мови програмування

При розробці програмного забезпечення системи підтвердження авторства документів шляхом генерації цифрового відбитка було обрано низку ключових технологій: мову програмування C++, а також допоміжні інструменти для криптографічного хешування, роботи з файлами, створення інтерфейсу та подальшої валідації документів.

Для реалізації основної функціональності - генерації та перевірки цифрового відбитка документа - було використано мову C++. Серед переваг C++ як основної технології для систем безпеки та криптографії можна виділити:

– Продуктивність і контроль над ресурсами. C++ забезпечує високий рівень продуктивності та дозволяє працювати з пам’яттю на низькому рівні, що критично важливо для криптографічних операцій та обробки великих обсягів даних.

– Бібліотеки для криптографії. Існує велика кількість бібліотек на C++ для роботи з хеш-функціями та цифровими підписами, зокрема: OpenSSL, Crypto++, libsodium. Вони будуть використані мною як інтегрованими модулями у процесі реалізації підсистеми цифрового відбитка.

– Гнучкість інтеграції. C++ легко поєднується з іншими мовами, що дозволяє, за потреби, використовувати бібліотеки з Python чи JavaScript для побудови клієнтського або web-інтерфейсу. За необхідності така інтеграція буде спроєктована з моєю участю.

Для створення базового інтерфейсу програми планується використання Qt, який дозволяє реалізувати багатоплатформовий графічний інтерфейс користувача. Qt має глибоку підтримку C++ і забезпечує широкі можливості для візуалізації, включно з:

– Відображенням результатів перевірки документа.

– Введенням/завантаженням документів користувачем.

– Генерацією повідомлень про відповідність хешу.

– Побудовою журналу операцій.

На архітектурному рівні проєкт буде складатися з трьох основних модулів: модуля обробки документа, модуля хешування та модуля перевірки цифрового сліду. Особливу увагу буде приділено формуванню логів, які фіксують дату, час, хеш, шлях до файлу та результат перевірки. Ці дані будуть зберігатися локально, з можливістю експорту у форматах .json, .csv або .xml.

Крім цього, для забезпечення незалежної фіксації цифрового відбитка в часі, планується реалізація інтеграції з такими зовнішніми timestamp-сервісами, як OpenTimestamps або OriginStamp.[10][11]

У межах проєкту також передбачено реалізацію інтерфейсу користувача, що забезпечить інтуїтивну взаємодію з основними функціями програми. Графічний інтерфейс буде розроблено з використанням фреймворку Qt для C++, що надає розширені можливості побудови кросплатформених застосунків. Серед ключових елементів інтерфейсу: поле для завантаження документа, кнопки для генерації та перевірки цифрового відбитка, область виводу результатів хешування та інформативне лог-вікно. Окрема вкладка дозволить переглядати історію перевірок, здійснювати пошук за хешами, а також експортувати звіти. Дизайн інтерфейсу буде витримано у лаконічному стилі з акцентом на функціональність та доступність, а основна логіка взаємодії з користувачем реалізовуватиметься через сигнально-слотову систему Qt, що забезпечує надійний зв’язок між елементами керування і ядром програми. У разі потреби, інтерфейс буде адаптовано під використання англійської або української мови за допомогою механізму локалізації.

Для реалізації основної функціональності - генерації та перевірки цифрового відбитка документа - було використано мову C++. Серед переваг C++ як основної технології для систем безпеки та криптографії можна виділити:

– Продуктивність і контроль над ресурсами. C++ забезпечує високий рівень продуктивності та дозволяє працювати з пам’яттю на низькому рівні, що

критично важливо для криптографічних операцій та обробки великих обсягів даних.

– Бібліотеки для криптографії. У цьому проєкті використовується OpenSSL для реалізації криптографічних операцій, таких як генерація хешів, цифрових підписів та перевірка сертифікатів. Крім того, бібліотека PoDoFo забезпечує роботу з PDF-документами, включаючи обробку підписів.

– Гнучкість інтеграції. C++ легко поєднується з іншими мовами, що дозволяє, за потреби, використовувати бібліотеки з Python чи JavaScript для побудови клієнтського або web-інтерфейсу. За необхідності така інтеграція буде спроєктована з моєю участю.

Для створення базового інтерфейсу програми використовується Qt, який дозволяє реалізувати багатоплатформовий графічний інтерфейс користувача. Qt має глибоку підтримку C++ і забезпечує широкі можливості для візуалізації, включно з:

– Відображенням результатів перевірки документа.

– Введенням/завантаженням документів користувачем.

– Генерацією повідомлень про відповідність хешу.

– Побудовою журналу операцій.

На архітектурному рівні проєкт складається з трьох основних модулів: модуля обробки документа, модуля хешування та модуля перевірки цифрового сліду. Особливу увагу приділено формуванню логів, які фіксують дату, час, хеш, шлях до файлу та результат перевірки. Ці дані зберігаються локально, з можливістю експорту у форматах .json, .csv або .xml. Для цього використовується клас “FileStorage”, який забезпечує зчитування, запис та збереження метаданих PDF-документів.

Крім цього, для забезпечення незалежної фіксації цифрового відбитка в часі, реалізовано інтеграцію з такими зовнішніми timestamp-сервісами, як OpenTimestamps або OriginStamp.[10][11]

У межах проєкту також передбачено реалізацію інтерфейсу користувача, що забезпечує інтуїтивну взаємодію з основними функціями програми.

Графічний інтерфейс розроблено з використанням фреймворку Qt для C++, що надає розширені можливості побудови кросплатформених застосунків.

Серед ключових елементів інтерфейсу: поле для завантаження документа, кнопки для генерації та перевірки цифрового відбитка, область виводу результатів хешування та інформативне лог-вікно. Окрема вкладка дозволяє переглядати історію перевірок, здійснювати пошук за хешами, а також експортувати звіти. Дизайн інтерфейсу витримано у лаконічному стилі з акцентом на функціональність та доступність, а основна логіка взаємодії з користувачем реалізована через сигнально-слотову систему Qt, що забезпечує надійний зв’язок між елементами керування і ядром програми. У разі потреби, інтерфейс адаптовано під використання англійської або української мови за допомогою механізму локалізації.

2.3 Розгорнута постановка завдання

Процес підтвердження авторства електронного документа за допомогою цифрового відбитка передбачає створення програмної системи, здатної ідентифікувати текстові файли за унікальними характеристиками змісту та структури, зафіксованими у криптографічному вигляді. Основна ідея полягає у створенні такого цифрового відбитка (хеш-функції), який би з високим рівнем достовірності співвідносився лише з одним конкретним документом і водночас був незворотним та нечутливим до незначних змін у даних.[7]

Для реалізації зазначеної мети необхідно вирішити комплекс таких підзадач:

1.Розробка формальних вимог до функціонування системи, включно з вимогами до швидкості обробки, стійкості до колізій, підтримки різних форматів документів (PDF, DOCX, TXT) та точності перевірки.

2.Проектування алгоритму генерації цифрового відбитка на основі сучасних хеш-функцій (SHA-256, SHA-3), з урахуванням можливості адаптації до валідації не лише змісту документа, а й його структурних елементів: метаданих, стилістики, формату.

3.Імплементація механізму порівняння цифрових відбитків, який дозволяє встановити ступінь збігу між двома документами - як на бітовому рівні (ідентичність хешів), так і на рівні вмісту (семантична подібність через попередню нормалізацію тексту).

4.Реалізація модулів перевірки автентичності, що включають:

– завантаження та обробку документа користувачем.
– генерацію відбитка на основі вмісту.
– збереження результатів у захищену базу.
– порівняння з існуючими записами та формування висновку.

1. Забезпечення незаперечності у часі, тобто фіксації моменту створення цифрового відбитка за допомогою незалежних timestamp-сервісів. Це дозволяє встановити час подання документа на верифікацію та запобігає подальшим фальсифікаціям.

6.Розробка інтерфейсу користувача, який передбачає інтуїтивно зрозумілу взаємодію з усіма ключовими функціями системи:

– імпорт/експорт документів.
– перегляд результатів хешування та журналу перевірок.
– налаштування параметрів алгоритмів та інтеграції.
– локалізація інтерфейсу українською та англійською мовами.

7.Тестування на практичних кейсах - створення репрезентативного набору документів різних авторів і структур для валідації якості, точності, продуктивності та надійності алгоритмів .

Таким чином, система повинна поєднувати в собі високу точність обробки, стійкість до зовнішніх втручань, зручність користування та модульність, що

забезпечить її ефективне застосування в широкому спектрі прикладних сфер: від освітніх закладів до юридичних і корпоративних структур.

Основну частину програми варто виконати з дотримання модульної архітектури, тоб-то кожен ключовий сегмент для роботи функції програми буде виділений окремо, а основне тіло програми буде вже окремо викликати ці модулі при виконанні задачі. Таким чином забезпечується гнучкість для майбутнього розвитку програми. Оптимізації її окремих елментів і відкриває можливість досить легкого інтегрування будьякого формату GUI при необхідності. Загальний формат виконання програми – це консольна утиліта, виклик якої буде виконаний через команду с фіксованним синтаксисом.

3 ОПИС І ОБҐРУНТУВАННЯ ПРОЕКТНИХ РІШЕНЬ

3.1 Опис функціонування системи

Система підтвердження авторства PDF-документів шляхом цифрового підпису або генерації цифрового відбитка реалізується у вигляді десктопного застосунку з автономною архітектурою. Основу становить послідовна обробка вхідних даних, застосування криптографічних трансформацій та збереження результатів локально, без потреби у підключенні до зовнішніх сервісів.

Програма призначена для:

– фіксації моменту авторства документу.

– виявлення навіть незначних змін у вмісті файлів.

– запобігання підробкам.

– забезпечення зворотного аудиту шляхом збереження логів перевірок.

1.Основні принципи побудови захищеної системи.

Для досягнення криптографічної надійності, у системі враховані такі принципи:

–детермінованість хешування (однаковий результат при ідентичному вмісті).[7][8][9]

–сталість хешу незалежно від повторного запуску алгоритму.

–висока швидкодія та ефективність обчислень навіть для великих PDF-файлів.

–односторонність хеш-функцій - неможливість відновлення тексту з відбитка.[7][8][9]

–гарантоване спрацювання при мінімальних змінах - навіть заміна одного символа повинна змінити відбиток повністю.

2.Вибір формату: цифровий підпис проти цифрового відбитка.

У межах проєкту було обґрунтовано два підходи.

Цифровий відбиток (хешування) - забезпечує цілісність, підходить для базового підтвердження авторства без довіреної третьої сторони.

Переваги: простота реалізації, автономність, швидкість.

Недоліки: немає прямої ідентифікації автора без додаткових механізмів.

Цифровий підпис (PKCS#12, RSA, ECDSA) - дає змогу підтвердити авторство за допомогою особистого ключа.

Переваги: юридична значущість, можливість перевірки підпису та авторства.

Недоліки: потребує збереження приватного ключа, складніша реалізація.

Обрано модель, що підтримує обидва варіанти: спочатку - генерація хешу, а за потреби - підпис його через .p12-файл.

3.Архітектура обробки документа.

Завантаження PDF-документа

Вміст зчитується у внутрішній буфер, із застосуванням бібліотеки для парсингу PDF (опційно - poppler/QtPDF).

Альтернатива: Apache PDFBox (Java) або pdf-lib (C++/JS), однак Qt забезпечує кращу інтеграцію з UI.

4.Попередня обробка тексту.

Видаляються артефакти (переноси, табуляція, пробіли), виконується нормалізація (регістр, кодування).[3]

Мета - забезпечення стабільності хешу при різних представленнях.

5.Генерація хешу або підпису.

За замовчуванням використовується SHA-256 або SHA-3 (через OpenSSL або Crypto++).[7][1]

За бажанням користувача, хеш підписується приватним ключем у форматі PKCS#12.

Альтернатива: використання Microsoft CryptoAPI або Windows CNG, однак OpenSSL є кросплатформеним стандартом з відкритим кодом.

6.Збереження результату.

Дані логуються у внутрішню БД (файл JSON/XML):

– хеш/підпис, час, шлях до файлу, мітка користувача, опціональний коментар.

Альтернатива: SQLite, однак JSON краще підходить для прозорості та портабельності.

Перевірка документа

Система повторно генерує хеш і порівнює його з журналом. У разі збігу – авторство підтверджено, у разі невідповідності – виявлено зміну.

7.Застосовані інструменти та бібліотеки:

– OpenSSL / Crypto++ - криптографічні операції, хешування, генерація ключів.

Обидві бібліотеки підтримують SHA-2/3, HMAC, RSA, ECDSA.[2][1]

OpenSSL має ширшу підтримку стандартів, Crypto++ - кращу інтеграцію у C++.

– Qt 6 - побудова інтерфейсу, візуалізація результатів, підтримка PDF-завантаження.

Переваги: кросплатформеність, QML-графіка, інтеграція із системними подіями.

Альтернатива: wxWidgets або Electron (але Qt має нативну продуктивність).

– nlohmann/json - серіалізація логів перевірок, легкість інтеграції у C++.

Альтернатива: RapidJSON (швидше, але складніше у використанні), boost::json.

– CMake - кросплатформена збірка, управління залежностями, IDE-агностичність.

– pybind11 (опціонально) - інтеграція з Python для тестів, генерації тестових хешів, емулювання перевірок з часовими мітками.

Запропонована система поєднує переваги локальної криптографії та зручності роботи з PDF-документами. Завдяки відкритим бібліотекам, вона

залишається гнучкою, розширюваною і готовою до використання як в особистому, так і в корпоративному середовищі

3.2 Розробка структурної схеми

В оптимальній структурній схемі офлайн-системи для підтвердження авторства PDF через цифровий підпис та/або крипто-відбиток ключове завдання зводиться до того, щоб усі операції з генерації, вбудовування та перевірки підписів та хешей виконувались на стороні клієнта без звернення до зовнішніх сервісів. Вибір клієнтської архітектури обґрунтований тим, що програма повністю контролює приватні ключі та процеси підпису, використовуючи або програмні сховища, або захищені апаратні модулі.

Компоненти схеми:

1. Інтерфейс користувача (UI/API).

Забезпечує вибір документа, налаштування параметрів (алгоритм хешування, сертифікат) та запуск операцій підпису/перевірки.

2. Менеджер ключів (Key Management).

Управління зберіганням приватного ключа у форматі PKCS#12/.p12, захищеному паролем (RFC 7292).

3. Двигун цифрового підпису (PDF Signing Engine).

Формує CMS/PKCS#7-пакет, вбудовує його у PDF через Signature Dictionary згідно з ISO 32000-1.[7]

4. Двигун відбитка (Hash Engine).

Обчислює SHA-256 хеш вмісту документа FIPS 180-4, що гарантує детектування будь-яких змін.[21]

5. Служба зберігання (Local Storage).

Зберігає підписаний PDF та/або метадані з відбитком (наприклад, XMP-мітки або QR-штамп).

6. Двигун перевірки (Verification Engine).

Перевіряє коректність підпису (CMS/PKCS#7) локально, використовуючи ланцюжок сертифікатів без OCSP/CRL.[24][20]

7. Локальне сховище довірених сертифікатів (Trust Store).

База кореневих та проміжних сертифікатів для валідації підпису.

Нижче буде приведено іллюстрацію с схемою структури майботньої програми.

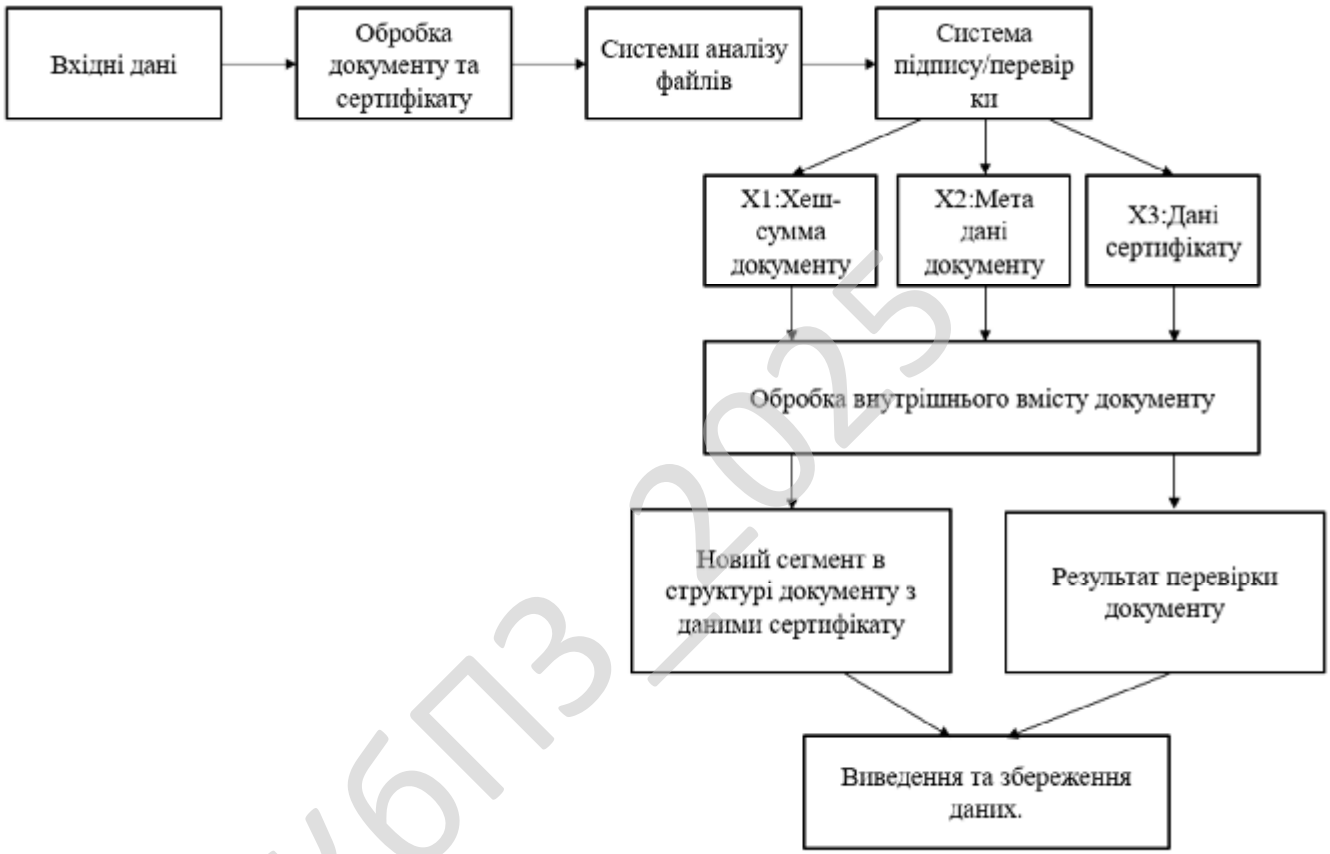

Рисунок 3.1 - Структурна схема системи

Таке структурне рішення забезпечує чіткий поділ зон відповідальності, масштабованість (наприклад, заміна сховища або алгоритму підпису) та повну автономність без зовнішніх запитів.

3.3 Розробка функціональної схеми

Функціональна схема слугує для формалізованого відображення послідовності операцій та логічних зв’язків між функціональними модулями системи. У межах автономної програми підтвердження авторства PDF-документів вона дозволяє встановити чітку послідовність етапів від обробки вхідного файлу до формування результату верифікації.

Операційні етапи:

1.Завантаження PDF-документа.

Користувач ініціює вхідний потік, обираючи локальний файл для подальшої обробки. Файл надходить до системного буфера й стає доступним для наступних модулів.

2.Вибір режиму роботи.

Система пропонує два базові режими: підписування документа або перевірка існуючої підпису та відбитка. Кожен режим активує відповідний набір функцій.

3.Управління ключами.

У разі підписування виконуються операції створення або завантаження приватного та публічного ключів з локального сховища (формат PKCS#12). У режимі верифікації задіюється лише публічний ключ.

4.Генерація криптографічного відбитка.

Для обраного PDF-файла обчислюється хеш-значення (SHA-256) відповідно до стандарту FIPS 180-4, що гарантує неможливість прихованих модифікацій вмісту документа.

5.Створення цифрової підпису.

На основі обчисленого хеш-значення формується CMS/PKCS#7-пакет із зашифрованим дайджестом та сертифікатом підписанта. У результаті отримується байтова послідовність, готова для вбудування в структуру PDF.

6.Вбудовування підпису в PDF.

Формується Signature Dictionary відповідно до вимог ISO 32000-1, після чого CMS-пакет інтегрується в потік документа без порушення цілісності існуючих даних.

7.Збереження результату.

Підписаний PDF разом із метаданими, що містять крипто-відбиток (наприклад, у розділі XMP або у QR-штампі), зберігається в локальному сховищі.

8.Верифікація документа.

У відповідному режимі здійснюється витяг CMS/PKCS#7-пакета та хеш-значення зі збережених метаданих. Далі виконується перевірка хеш-суми та коректності цифрової підпису за допомогою локального довірчого сховища сертифікатів (без звернення до зовнішніх OCSP/CRL).

9.Фіксація результатів.

За підсумками перевірки система повертає результат (успішно/неуспішно) і надає комплексну інформацію про цілісність документа та дійсність підпису.

Нижче буде приведена функціональна схема:

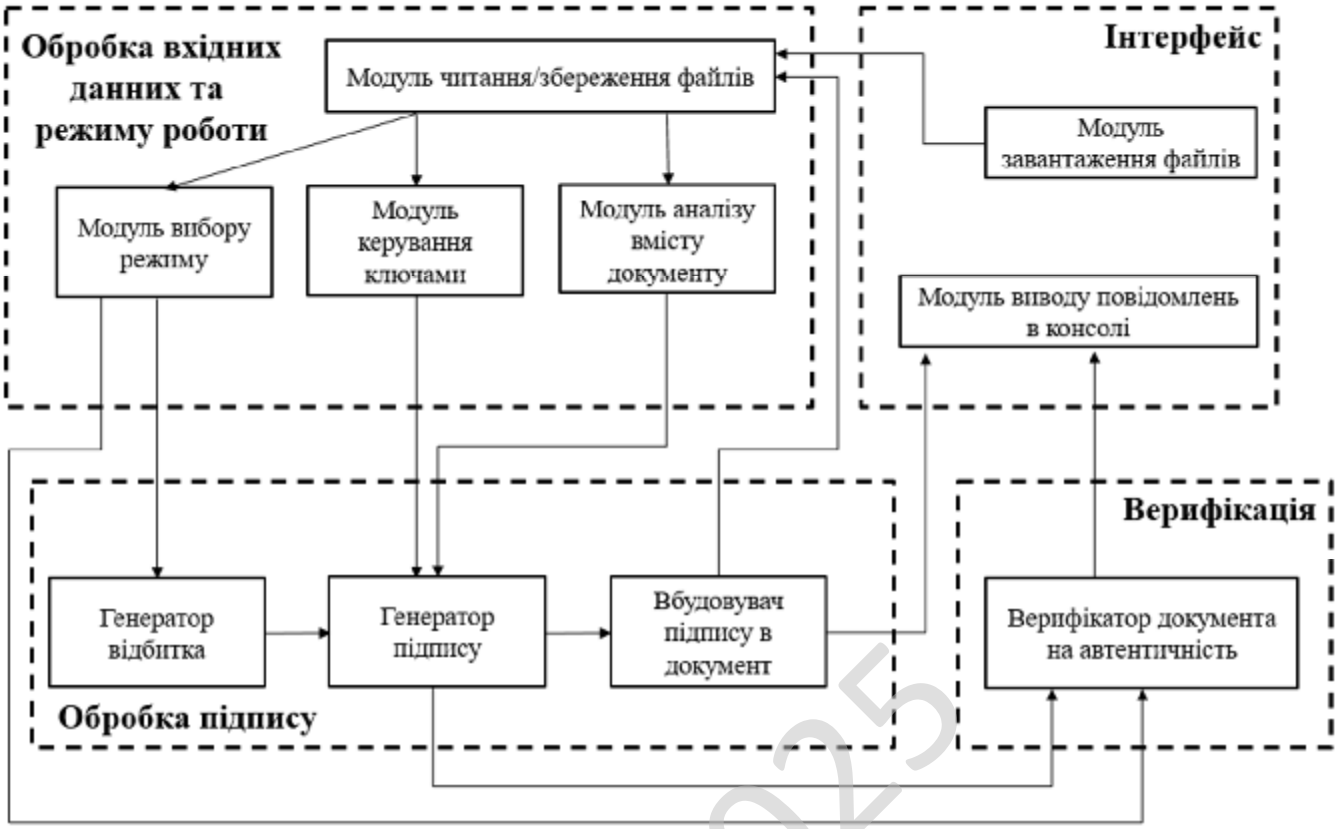

Рисунок 3.2 - Функціональна схема системи

Таким чином, запропонована функціональна схема гарантує автономну обробку PDF-документів із чітко визначеним порядком операцій, забезпечуючи надійність і відтворюваність процедур підписування та верифікації.

3.4 Розробка діаграми процесів

Процесна модель автономної програми підтвердження авторства PDF-документів відображає чітку послідовність операцій: від прийому вхідного файлу та вибору режиму роботи до остаточної верифікації цифрової підписи й крипто-відбитка. Застосований підхід гарантує цілісність і автентичність документа без звернення до зовнішніх сервісів.

Ключові етапи процесу:

1.Прийом PDF-документа.

Користувач обирає локальний файл, який надходить у внутрішній буфер для подальшої обробки.

2.Визначення режиму роботи.

Система пропонує два варіанти: підписування нового або верифікація наявного підпису та відбитка, активуючи відповідні модулі.

3.Управління криптоключами.

У режимі підписування здійснюється завантаження чи генерація приватного і публічного ключів у захищеному контейнері PKCS#12; у режимі верифікації задіюється лише публічний ключ із локального довірчого сховища.

4.Обчислення крипто-відбитка.

Для вхідного PDF-файлу виконується хешування за алгоритмом SHA-256 відповідно до FIPS 180-4, що забезпечує виявлення будь-яких змін вмісту.

5.Формування CMS/PKCS#7-пакету.

На основі отриманого відбитка генерується пакет із зашифрованим дайджестом та сертифікатом підписанта.

6.Інтеграція підпису у PDF.

Відповідно до ISO 32000-1 створюється Signature Dictionary, після чого CMS-пакет вбудовується в структуру документа.[21]

7.Збереження результату.

Підписаний документ зберігається локально разом із метаданими відбитка (наприклад, у XMP-мітках або у вигляді QR-штампа).

8.Витяг підпису та відбитка.

У режимі верифікації система вилучає збережені дані про підпис і вихідний хеш для подальшої перевірки.

9.Перевірка цілісності та достовірності.

Здійснюється повторне обчислення SHA-256 та валідація дайджесту й цифрової підпису за локальною цепочкою сертифікатів без звернення до OCSP/CRL.

10.Фіксація результатів.

Користувачу надається підсумковий статус («успішно» або «невдало») із детальною інформацією про цілісність документа та дійсність підпису.

Нижче буде представлена діаграмма процессів:

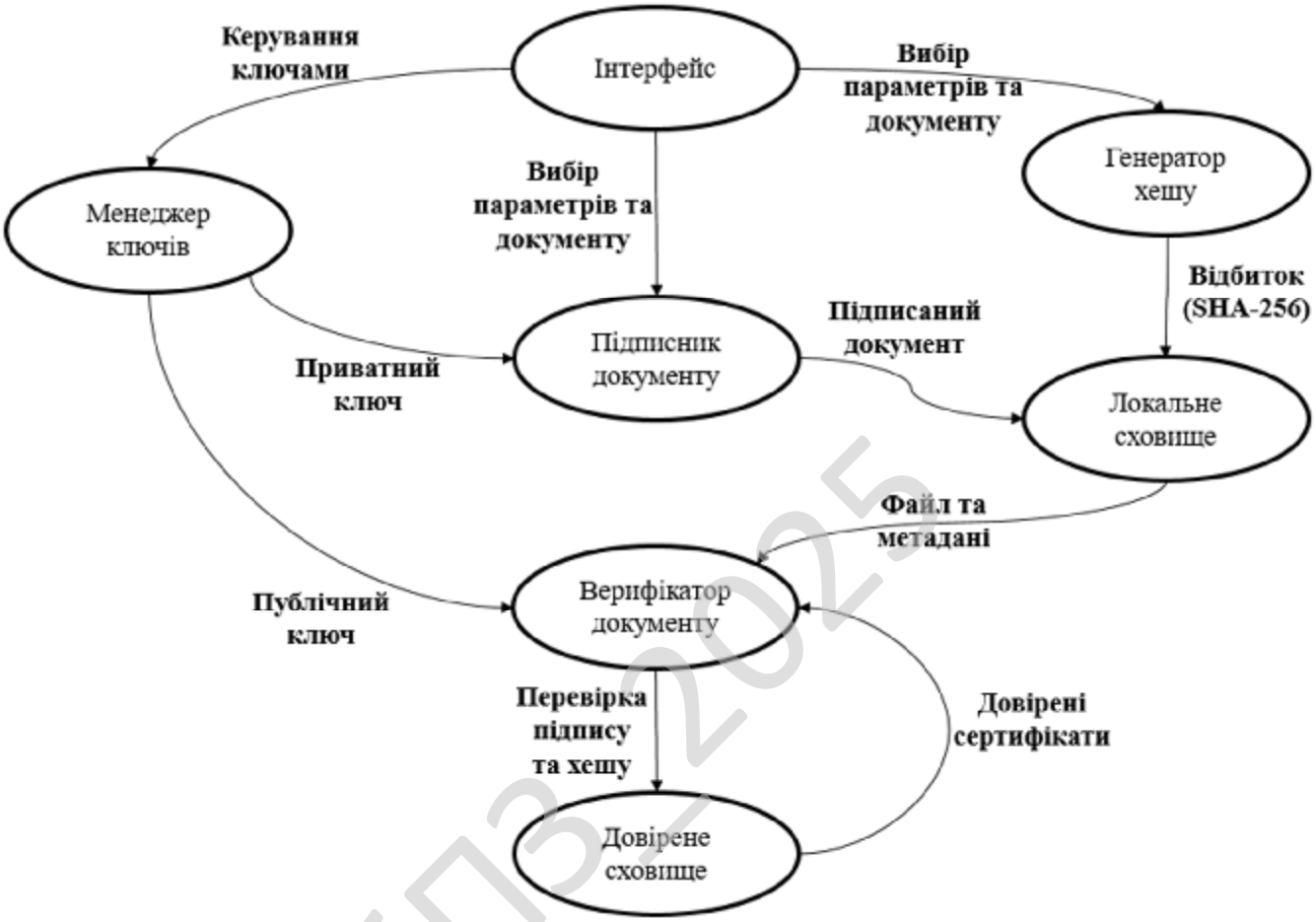

Рисунок 3.3 – Діаграма процесів

4 РЕАЛІЗАЦІЯ ПРОЕКТУ. РОЗРАХУНКИ І ЕКСПЕРИМЕНТАЛЬНІ ДАНІ, ЩО ПІДТВЕРДЖУЮТЬ ПРАВИЛЬНІСТЬ ПРОЕКТНИХ РІШЕНЬ

4.1 Блок-схеми та опис алгоритмів функціонування системи

На рисунку 4.1 та 4.2 наведено блок-схему основної програми. Її робота складається з таких пунктів:

– Виклик програми з командного рядка консолі.

Виконується при виклику програми з дотриманням вказанного синтаксисиу, де вказані путь до документу, путь до сертифікату, а та пароль від ключів в сертифікату.

– Прийом вказаних параметрів.

Після виклику программа починає роботу, та визначає стан ключових єлементів.

– Перевірка чи є документ переданий програмі в форматі PDF.

Так как программа розрахована на використання з форматом PDF, то їй потрібно перевірити чи підходить файл вказаних користувачем.

– Перевірка в якиму режимі було вказано програмі працювати.

При підпису програма почне процесс підпису документу. При перевірці буде перевіряти підпис да данні с сертифікату.

– Перевірка паролю для ключів сертифікату.

Для роботи підпису важливо щоб пароль для пари ключів в сертифікаті був корректним.

– Генерація хеш-відбитку документу.

– Створення підпису документу.

– Підписання документу.

Активація підропграми для підпису документу:

– Збереження метаданих та документу.

– Вивід результатів підпису або перевірки.

В командному рядку у будь-якому з двух випадків програма виведе результати підпису, або результат перевірки.

На рисунку 4.3 наведено блок-схему підпрограми підпису документу. Її робота складається з таких пунктів:

–Отримання вмісту документу.

Бібліотека PoDoFo аналізує вміст документу та створює тимчасову копію файлу для подальших змін.

–Отримання вмісту сертифікату.

Для підпису крім корректного паролю потрібний приватний ключ для формування підпису сувмістного с конкретнім сертифікатом.

–Створення тимчасової області даних для підпису.

Для того щоб влаштувати підпис потрібно заздалегіть створити область в структурі файлу, яка не конфлітуватиме з оригінальним вмістом документу.

–Створення підпису.

Генерація підпису на базі приватного ключа.

–Перевірка чи достатньо в створенній тимчасовій області місця під ключ;

Крім PKCS э ще різні сертифікати, чий розмір ключу може бути більше, ні вказанний розмір заготовленної області. Якщо нема можливості влаштувати в область підпис, то програма поверне помилку.

–Встановлення підпису в документ.

–Збереження підписанного документу.

–Вивід результатів підпису.

При вдалому підписі програма повертає відповідне повідомлення. При не вдалому підписі ми отримуємо повідомлення с причинами невдачі.

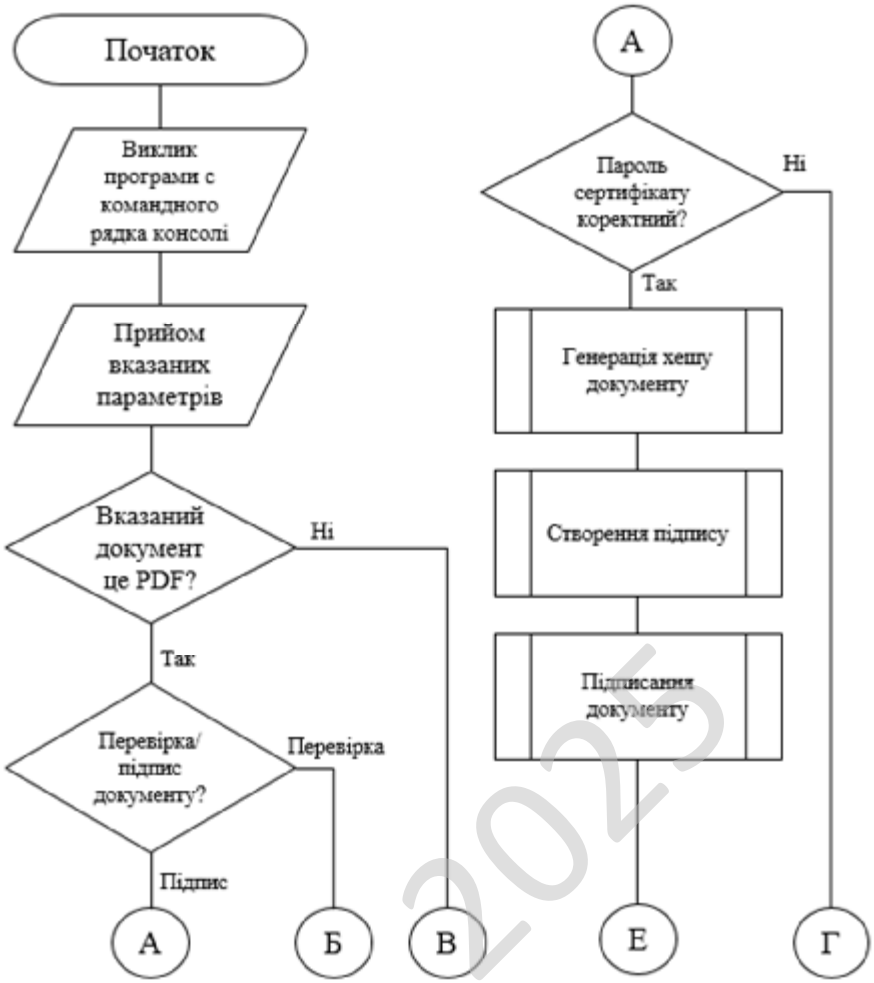

Рисунок 4.1 - Блок-схема основної програми

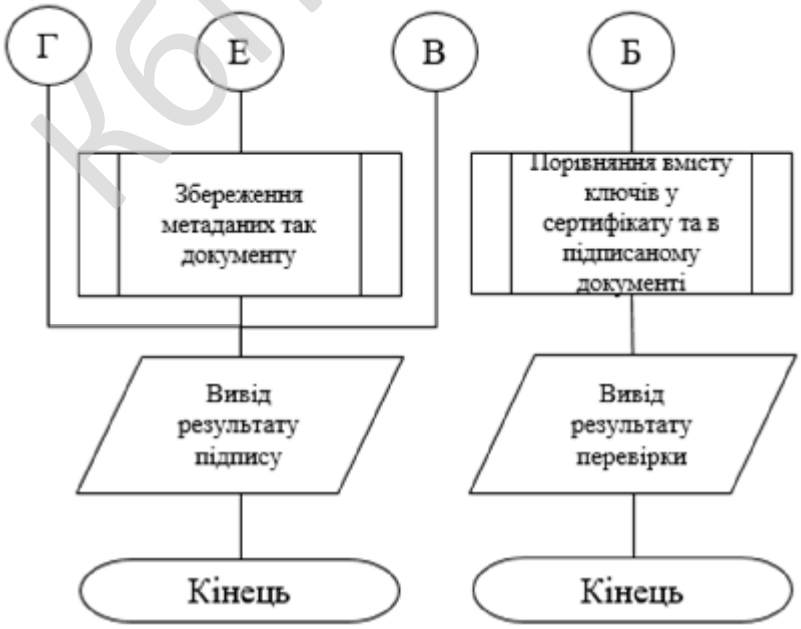

Рисунок 4.2 – Продовження блок-схеми основної програми

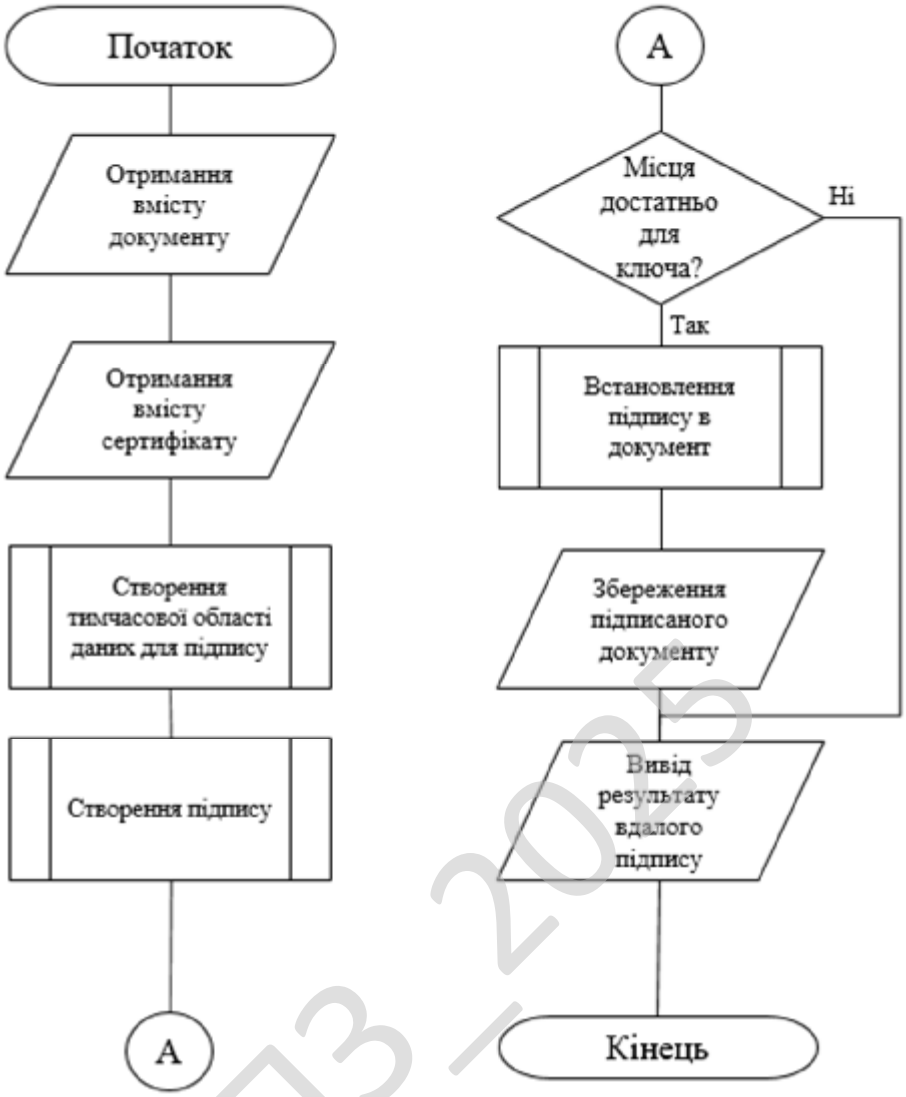

Рисунок 4.3 –Блок-схема підпрограми

Нижче наведено опис алгоритмів функціонування автономної системи підтвердження авторства PDF-документів за допомогою цифрових підписів (CMS/PKCS#7) та крипто-відбитків (SHA-256). Розглянуто послідовність процедур: від управління ключами до вбудовування й перевірки підпису та відбитка, з урахуванням вимог ISO 32000-1, RFC 5652, FIPS 180-4 та RFC 5280.[21][7][8]

1. Управління ключами.

Алгоритм управління ключами базується на структурі PKCS#12, яка дозволяє зберігати пару приватний/публічний ключ у зашифрованому контейнері з використанням пароля користувача.

Контейнер складається з інформаційних блоків “SafeBag”, що містять приватний ключ у форматі PKCS#8 та сертифікат у форматі X.509, захищені за схемою PBE (password-based encryption) згідно з RFC 7292.

Доступ до приватного ключа здійснюється через розшифрування відповідного SafeBag–блоку з використанням PBKDF2 і симетричного алгоритму (наприклад, AES-256).

Публічний ключ та сертифікат завантажуються у локальне довірене сховище для верифікації підписів.

2. Генерація крипто-відбитка.

Обчислення відбитка документа виконується за алгоритмом SHA-256, що описаний у FIPS 180-4.

Алгоритм реалізує такі кроки:

2.1.Ініціалізація: встановлення вихідних хеш-значень H[0..7] згідно з константами стандарту.

2.2.Додавання доповнення (padding): до повідомлення додається біт ‘1’, потім ‘0’-біти, щоб довжина стала конгруентною до 448 (mod 512), і на кінець 64-бітне подання довжини повідомлення.

2.3.Обробка блоків по 512 біт: кожен блок розбивається на 16 32-бітних слів, далі розширюється до 64 слів згідно з формулами σ0, σ1, Maj, Ch (секція 4.1.2 FIPS 180-4).[14]

2.4.Основний цикл: виконується 64 ітерації, де на кожному кроці обчислюються тимчасові змінні T1, T2 через функції Σ0, Σ1, Ch, Maj і константи.

2.5.Оновлення H[i]: після обробки кожного блокового циклу, вихідні значення оновлюються за схемою H[i] := H[i] + working_variables.

У результаті формується 256-бітний дайджест документа.

3. Формування CMS/PKCS#7-пакета.

Для створення цифрової підпису використовується синтаксис CMS (RFC 5652).

Процедура складається з таких кроків:

3.1.Створення структури SignedData**: ініціалізується об’єкт SignedData із зазначенням алгоритму дайджесту (SHA-256) та алгоритму шифрування підпису (наприклад, rsaEncryption за PKCS#1 v1.5 або RSASSA-PSS за RFC 3447).

3.2.Включення сертифікатів**: до поля certificates додається сертифікат підписанта у форматі X.509.

3.3.Обчислення дайджесту й підпис: дайджест документа підписується приватним ключем підписанта, результат записується у SignerInfo.signatureValue.

3.4.Серилізація й DER-кодування: готова структура SignedData серіалізується в потік байтів із використанням DER-кодування.

4.Вбудовування підпису в PDF.

Згідно з ISO 32000-1, підпис вбудовується через створення Signature Dictionary у ресурсах AcroForm:

4.1.Додавання поля підпису: у словнику AcroForm створюється новий об’єкт типу Sig із вказівками Filter, SubFilter (наприклад, adbe.pkcs7.detached), ByteRange, Contents.

4.2.Визначення ByteRange: задає позиції в файлі, які залишаються незмінними для обчислення дайджесту; область Contents резервується під DER-закодований CMS-пакет.

4.3.Запис подписи: після обчислення CMS-пакета вміст поля Contents заповнюється підписом (як правило, у форматі hex-encoded або raw).

5. Перевірка підпису та відбитку.

Процедура верифікації реалізується без звернень до OCSP/CRL, на основі локального довірчого сховища (RFC 5280):

5.1. Витягнення CMS-пакета: витягується вміст поля Contents у Signature Dictionary.[14]

5.2. Перевірка дайджесту: за вказаними ByteRange виконується повторний розрахунок SHA-256 і порівняння з дайджестом у SignedData.digestEncrypted.

5.3.Валідація сертифікатного шляху: алгоритм проходження ланцюжка сертифікатів від кінцевого сертифіката до кореневого за механізмом RFC 5280 (Certification Path Validation Algorithm).

5.4.Перевірка цифрового підпису: з використанням публічного ключа кінцевого сертифіката в SignerInfo відбувається перевірка підпису дайджесту.

5.5.Формування результату: якщо дайджест та підпис коректні й сертифікати не викликають помилок валідації, документ вважається автентичним. Інакше повертається детальне повідомлення про помилку.

Контекст та використовувані технології:

1.OpenSSL – де-факто стандарт для криптографічних операцій у C++ на Windows, включаючи роботу з PKCS#12 та CMS/PKCS#7.

2.PoDoFo - вільна бібліотека, що портується, на C++ для розбору та модифікації PDF, підходить для впровадження цифрових підписів.

3.CMake - кросплатформовий інструмент керування збиранням, вбудована підтримка Visual Studio та рекомендація використовувати targets замість прямого редагування прапорів.[8]

4.PKCS#12 (RFC 7292) застосовується для безпечного зберігання пари приватного/публічного ключів у зашифрованому контейнері (.p12/.pfx).

5.SHA-256 (FIPS 180-4) забезпечує крипто-хешування документів для детектування будь-яких змін.[7]

6.CMS/PKCS#7 (RFC 5652) описує структуру пакета цифрового підпису, що використовується для формування та валідації підписів.

7.Приклад PoDoFo + OpenSSL демонструє інтеграцію генерації підпису за допомогою PoDoFo та OpenSSL на C++

Рекомендація щодо структури проекту: розміщувати вихідники та заголовки за схемою src/, include/, а тести та залежності в окремих каталогах Modern CMake: використовувати цілі (targets) та окремі CMakeLists.txt на рівні бібліотек та виконуваного файлу, уникаючи глобальних змінних.

В архітектурі написання коду біло використано відокремлення класів: кожен функціональний модуль (KeyManager, PDFSigner, HashEngine, VerificationEngine і т.д.) має бути реалізований у своєму .h/.cpp файлі для спрощення підтримки та розширення.

Принципи роботи бібліотек OpenSSL та zlib:

1. OpenSSL[2].

OpenSSL – це широко використовувана бібліотека, що надає інструменти для виконання криптографічних операцій та роботи з протоколами безпеки, такими як TLS/SSL. Основні принципи роботи OpenSSL включають:

Криптографічні операції: OpenSSL підтримує симетричне та асиметричне шифрування, генерацію ключів, цифрові підписи та хешування.

Робота з сертифікатами: OpenSSL надає API для створення, перевірки та керування сертифікатами X.509, які використовуються для аутентифікації та шифрування.

Підтримка стандартів: Бібліотека реалізує стандарти, такі як PKCS#7/CMS для підписів та PKCS#12 для зберігання ключів.

Модульність: OpenSSL надає низькорівневі функції, які можна використовувати для побудови високорівневих рішень.

2. zlib.

zlib – це бібліотека для стиснення даних, яка базується на алгоритмі DEFLATE. Основні принципи роботи zlib включають:

Стиснення та розпакування даних: zlib надає API для зменшення розміру даних із збереженням їх цілісності.

Ефективність: Алгоритм DEFLATE, що використовується в zlib, поєднує методи LZ77 та кодування Хаффмана для досягнення високого ступеня стиснення.

Універсальність: zlib підтримує роботу з потоками даних, що робить її придатною для роботи з файлами, потоками мережі та іншими джерелами даних.

Роль OpenSSL та zlib у роботі програми:

1. Роль OpenSSL.

На основі коду бібліотека OpenSSL, використовується для виконання наступних завдань:

Перевірка цифрових підписів: У класі VerificationEngine виконуэться перевірка підписів PDF-документів та CMS/PKCS#7 підписів. OpenSSL надає інструменти для роботи з цими стандартами, включаючи перевірку підпису, валідацію сертифікатів та ланцюжків довіри.

Робота з сертифікатами: Поле `X509_STORE* trustedStore_` у класі VerificationEngine вказує на OpenSSL для керування сховищем довірених сертифікатів. Це необхідно для перевірки автентичності підписів та забезпечення безпеки.

Хешування: Для перевірки підписів можна використовувати хешування даних, що також підтримується OpenSSL.

Таким чином, OpenSSL відіграє ключову роль у забезпеченні криптографічної безпеки та автентифікації даних у проекті.

2. Роль zlib.

Її використання в проекті пов'язане з обробкою PDF-файлів, що мається на увазі у класі FileStorage. PDF-документи часто використовують стиснення даних, і zlib може бути задіяна для:

Розпакування вмісту PDF: При читанні PDF-файлів (readPDFFile) може знадобитися розпакування даних, стислих з використанням алгоритму DEFLATE.

Стиснення даних: При записі PDF-файлів (writePDFFile) zlib може використовуватися для стиснення даних, щоб зменшити розмір файлу.

Таким чином, zlib забезпечує ефективну роботу з PDF-документами, підтримуючи операції стиснення та розпакування даних.

Далі піде розбір ключових файлів коду програми для реалілації основний функцій:

1.Загальна характеристика класу FileStorage.

Клас FileStorage призначений для управління операціями збереження та обробки PDF-файлів, а також для роботи з метаданими, пов'язаними з цифровими підписами. Його функціональність охоплює:

– читання та запис PDF-файлів у двійковому форматі;

– збереження та отримання метаданих підпису;

– перевірку наявності файлів та створення необхідних директорій.

Аналіз ключових методів класу:

1.1. `std::vector<uint8_t> readPDFFile(const std::string& filePath);`

Цей метод відповідає за читання PDF-файлу за вказаним шляхом filePath та повертає його вміст у вигляді вектора байтів. Використання `std::vector<uint8_t>` дозволяє ефективно обробляти двійкові дані, що є стандартною практикою при роботі з PDF-файлами.

1.2. `bool writePDFFile(const std::string& filePath, const std::vector<uint8_t>& pdfData);`

Метод здійснює запис вмісту PDF-файлу, представленого вектором байтів pdfData, у файл за вказаним шляхом filePath. Повертає true у разі успішного запису, інакше - false.

1.3. `bool storeSignatureMetadata(const std::string& pdfPath, const std::string& signerName, const std::string& signatureDate, const std::string& metadataPath = "");`

Цей метод зберігає метадані, пов'язані з підписаним PDF-документом, включаючи шлях до файлу, ім'я підписанта та дату підпису. Опціонально

можна вказати шлях до файлу метаданих metadataPath. Метадані можуть зберігатися у форматі JSON або XML для забезпечення зручності обробки та сумісності з іншими системами.

1.4. `std::map<std::string, std::string> retrieveSignatureMetadata(const std::string& pdfPath, const std::string& metadataPath = "");`

Метод призначений для отримання метаданих, пов'язаних з підписаним PDF-файлом. Повертає асоціативний масив (словник) ключ-значення, що містить інформацію про підписанта, дату підпису та інші релевантні дані.

1.5. `static bool fileExists(const std::string& filePath);`

Статичний метод, який перевіряє наявність файлу за вказаним шляхом filePath. Повертає true, якщо файл існує, інакше - false.

1.6. `static bool ensureDirectoryExists(const std::string& dirPath);`

Цей статичний метод перевіряє існування директорії за вказаним шляхом dirPath та створює її у разі відсутності. Повертає true, якщо директорія існує або була успішно створена, інакше – false.

2.Загальна характеристика класу HashEngine.

Клас HashEngine призначений для обчислення криптографічних хешів, зокрема алгоритму SHA-256, що є стандартом у забезпеченні цілісності та автентичності даних. Його функціональність охоплює:

– обчислення SHA-256 хешу для масиву байтів;

– обчислення SHA-256 хешу для файлу;

– перетворення бінарного хешу у шістнадцятковий рядок;

– порівняння двох хеш-значень на еквівалентність.

Аналіз ключових методів класу:

2.1. `std::vector<uint8_t> calculateSHA256(const std::vector<uint8_t>& data);`

Цей метод приймає на вхід масив байтів data та повертає SHA-256 хеш у вигляді вектора байтів. Використання std::vector<uint8_t> забезпечує гнучкість у роботі з бінарними даними.

2.2. `std::vector<uint8_t> calculateFileSHA256(const std::string& filePath);`

Метод відкриває файл за вказаним шляхом filePath, читає його вміст та обчислює SHA-256 хеш. Повертає хеш у вигляді вектора байтів. Це дозволяє перевіряти цілісність файлів без необхідності попереднього завантаження їх у пам'ять.

2.3. `std::string hashToHexString(const std::vector<uint8_t>& hash);`

Цей метод перетворює бінарний хеш, представлений вектором байтів hash, у шістнадцятковий рядок. Таке представлення є зручним для відображення хешів у текстовому форматі та зберігання у логах або базах даних.

2.4. `bool compareHashes(const std::vector<uint8_t>& hash1, const std::vector<uint8_t>& hash2);`

Метод порівнює два хеш-значення hash1 та hash2 на еквівалентність. Повертає true, якщо хеші ідентичні, інакше - false. Це корисно для перевірки цілісності даних або автентичності документів.

3.Загальна характеристика класу KeyManager.

Клас KeyManager призначений для управління криптографічними ключами та сертифікатами, необхідними для створення та перевірки цифрових підписів. Його основні функції включають:

–завантаження PKCS#12 контейнерів з файлів;

–створення нових PKCS#12 контейнерів з існуючих ключів та сертифікатів;

–надання доступу до приватного ключа, сертифіката та ланцюжка сертифікатів;

–перевірку готовності ключових матеріалів для підписування.

Аналіз ключових методів класу.

3.1. `bool loadPKCS12FromFile(const std::string& pkcs12File, const std::string& password);`

Цей метод завантажує PKCS#12 контейнер з файлу, розшифровуючи його за допомогою наданого пароля. PKCS#12 контейнер зберігає приватний ключ, сертифікат та, можливо, ланцюжок сертифікатів. Успішне завантаження дозволяє використовувати ці матеріали для криптографічних операцій.

3.2. `bool createPKCS12(const std::string& keyFile, const std::string& certFile, const std::string& password, const std::string& outputFile);`

Метод створює новий PKCS#12 контейнер, використовуючи надані файли приватного ключа та сертифіката у форматі PEM. Результат зберігається у вказаному файлі outputFile, захищеному паролем. Це дозволяє об'єднати ключові матеріали в один захищений контейнер для зручності розповсюдження та зберігання.

3.3. `EVP_PKEY* getPrivateKey() const;`

Цей метод повертає вказівник на приватний ключ, завантажений з PKCS#12 контейнера. Приватний ключ використовується для створення цифрових підписів. Якщо ключ не завантажено, метод повертає nullptr.

3.4. `X509* getCertificate() const;`

Метод повертає вказівник на сертифікат, завантажений з PKCS#12 контейнера. Сертифікат використовується для перевірки цифрових підписів та встановлення довіри. Якщо сертифікат не завантажено, метод повертає nullptr.

3.5. `const std::vector<X509*>& getCertificateChain() const;`

Цей метод повертає вектор вказівників на сертифікати, що складають ланцюжок сертифікатів, завантажений з PKCS#12 контейнера. Ланцюжок сертифікатів використовується для побудови довірчого шляху до кореневого сертифіката при перевірці підписів.

3.6. `bool isReady() const;`

Метод перевіряє, чи завантажено дійсний приватний ключ та сертифікат. Повертає true, якщо обидва компоненти доступні, інакше - false. Це дозволяє переконатися в готовності системи до виконання криптографічних операцій.

Приватні члени класу:

— `EVP_PKEY* privateKey_;` - вказівник на приватний ключ.

— `X509* certificate_;` - вказівник на сертифікат.

`std::vector<X509*> certificateChain_;` - вектор вказівників на сертифікати, що складають ланцюжок сертифікатів.

Ці члени класу використовуються для зберігання завантажених ключових матеріалів та забезпечення доступу до них через відповідні методи.

4.Загальна характеристика класу PDFSigner.

Клас PDFSigner реалізує процес цифрового підпису PDF-документів відповідно до стандартів ISO 32000-1 та PAdES, використовуючи підписи формату PKCS#7/CMS. Він інтегрується з KeyManager для управління криптографічними ключами та сертифікатами, а також використовує бібліотеку PoDoFo для обробки PDF-файлів.

Основні компоненти:

4.1. Конструктор.

```
explicit PDFSigner(KeyManager& keyManager);
```

Приймає посилання на об'єкт KeyManager, що забезпечує доступ до приватного ключа та сертифікатів, необхідних для підпису.

4.2. Метод `setSignatureAppearance`

```
void setSignatureAppearance(const std::string& signatureName,
const std::string& signatureReason,
const std::string& signatureLocation,
const std::string& signatureContactInfo);
```

Встановлює параметри відображення підпису, такі як ім'я підписанта, причина підпису, місце та контактна інформація.

4.3. Метод `signPDF`

```
bool signPDF(const std::string& inputPdfPath, const std::string& outputPdfPath,
int pageNumber = -1, double x = 0.0, double y = 0.0,
double width = 0.0, double height = 0.0);
```

Реалізує процес підпису PDF-документа. Параметри pageNumber, x, y, width та height визначають розташування та розмір візуального підпису. Якщо pageNumber дорівнює -1, підпис буде невидимим.

4.4.Приватні методи:

4.4.1. Метод `createCMSSignature`

```
std::vector<uint8_t> createCMSSignature(const std::vector<uint8_t>& digest);
```

Створює підпис у форматі CMS (Cryptographic Message Syntax) на основі хешу документа.

4.4.2. Метод `prepareSignatureField`

```
bool prepareSignatureField(PoDoFo::PdfMemDocument& document, PoDoFo::PdfSignatureField& field,
int pageNumber, double x, double y, double width, double height);
```

Готує поле підпису в PDF-документі, визначаючи його розташування та розмір.

5.Загальна характеристика класу `VerificationEngine`.

Клас VerificationEngine є основним компонентом, що забезпечує перевірку цифрових підписів та сертифікатів. Він включає методи для додавання довірених сертифікатів, списків відкликаних сертифікатів (CRL), а також для перевірки підписів у PDF-документах.

1.Перелічувальний тип VerificationStatus.

Цей перелічувальний тип визначає можливі результати перевірки підпису:

`VALID` - підпис є дійсним і довіреним.

`INVALID_SIGNATURE` - підпис не відповідає документу.

`CERTIFICATE_ERROR` - виникла помилка під час перевірки сертифіката.

`DOCUMENT_MODIFIED` - документ було змінено після підписання.

`ERROR` - загальна помилка під час перевірки.

2.Публічні методи.

Конструктор і деструктор:

`VerificationEngine()` - ініціалізує об'єкт класу.

`~VerificationEngine()` - звільняє ресурси, пов'язані з об'єктом.

`addTrustedCertificate`: Додає довірений сертифікат до сховища для перевірки. Приймає шлях до файлу сертифіката у форматі PEM. Повертає true, якщо сертифікат успішно додано.

`addCertificateRevocationList`: Додає список відкликаних сертифікатів (CRL). Приймає шлях до файлу CRL у форматі PEM. Повертає true, якщо CRL успішно додано.

`verifyPDFSignature`: Перевіряє підпис у PDF-документі. Приймає шлях до PDF-файлу. Повертає значення типу VerificationStatus, яке вказує на результат перевірки.

`getLastVerificationInfo`: Повертає детальну інформацію про останню операцію перевірки у вигляді рядка.

`verifyCMSSignature`: Перевіряє підпис CMS/PKCS#7 на основі хешу документа. Приймає дані підпису та хеш документа. Повертає true, якщо підпис є дійсним.

3.Приватні члени.

`X509_STORE* trustedStore_`:

Вказівник на сховище довірених сертифікатів, яке використовується для перевірки.

`std::string lastVerificationInfo_`:

Рядок, що містить деталі останньої операції перевірки.

Примітки:

Клас використовує типи та структури OpenSSL (наприклад, X509_STORE) для роботи з сертифікатами та перевірки підписів.

Таким чином, наведений опис алгоритмів демонструє формалізовану послідовність процедур автономної системи підписування та верифікації PDF-документів без зовнішніх залежностей, з дотриманням міжнародних стандартів та рекомендацій з криптографічної безпеки.

4.1 Захист програмного забезпечення

Програмне забезпечення буде поставлятись по принципу “open source”, тобто код буде відкритим для майбутнього користувача, що важливо для майбутнього вільного розвитку програми, її покращення та дає можливість установам, де важливо щоб програмне забезпечення відповідало певним стандартам перед використанням перевірити код програми на відповідність.

Ліцензія по якій буде розповсюджуватись продукт буде MIT, нижче буде приведено повний текст цієї ліцензії.

Цим надається безкоштовний дозвіл будь-якій особі, яка отримує копію цього програмного забезпечення та дозвіл використовувати пов'язані з ним файли документації («Програмне забезпечення»)

Програмне забезпечення без обмежень, включаючи, без обмежень, права використовувати, копіювати, змінювати, об'єднувати, публікувати, розповсюджувати, субліцензувати та/або продавати копії Програмного забезпечення, а також дозволяти особам, яким надається Програмне забезпечення, робити це, за таких умов:

Вищезазначене повідомлення про авторські права та це повідомлення про дозвіл повинні бути включені до всіх копій або суттєвих частин Програмного забезпечення.

Програмне забезпечення надається «як є», без будь-яких гарантій, явних або неявних, включаючи, але не обмежуючись гарантіями товарної придатності, придатності для певної мети та непорушення прав. за жодних обставин автори або власники авторських прав не несуть відповідальності за будь-які претензії, збитки або іншу відповідальність, чи то виникли в вині дії за договором, делікту чи іншим чином, що виникли в зв'язку з програмним забезпеченням або з використанням або іншими діями з програмним забезпеченням.[50]

5 ВПРОВАДЖЕННЯ СИСТЕМИ В ПРОМИСЛОВУ ЕКСПЛУАТАЦІЮ.

Нижче, на рисунках 5.1, 5.2 та 5.3 буде показано результати роботи програми в тестовому режимі з використанням тестового файлу в форматі PDF так само як і тестового сертифікату.

Спочатку буде продемонстровано вивід програми без аргументу або с аргументом `--help`.

```
~/media/MyPepos/CryptoText++  cd /home/miki/media/MyPepos/CryptoText++/build && ./cryptotext --help
CryptoText++ - PDF Digital Signature Tool
=========================================
A comprehensive tool for PDF digital signatures and verification.

USAGE:
  ./cryptotext <command> [options]

COMMANDS:
  sign <input_pdf> <output_pdf> <pkcs12_file> <password> [options]
      Sign a PDF document with digital signature

  verify <signed_pdf> [trusted_cert]
      Verify the digital signature of a PDF document

  help, --help, -h
      Display this help information

SIGN OPTIONS:
  --name <name>        Signer name (default: "Signer")
  --reason <reason>    Signature reason (default: "Document signing")
  --location <loc>     Signature location (default: empty)
  --contact <info>     Signer contact info (default: empty)
  --visible <page>     Make signature visible on specified page (0-based)
  --pos <x,y,w,h>      Position and size for visible signature

EXAMPLES:
  # Basic PDF signing:
  ./cryptotext sign input.pdf signed.pdf certificate.p12 password123

  # Sign with visible signature:
  ./cryptotext sign input.pdf signed.pdf certificate.p12 password123 \
    --visible 0 --pos 50,50,200,50 --name "John Doe" --reason "Approval"

  # Verify PDF signature:
  ./cryptotext verify signed.pdf

  # Verify with trusted certificate:
  ./cryptotext verify signed.pdf trusted_cert.pem

SUPPORTED FEATURES:
  • PKCS#12 certificate loading and management
  • PDF digital signature creation (invisible/visible)
  • Signature verification with fallback mechanisms
  • SHA-256 hashing for document integrity
  • Signature metadata storage and retrieval
  • Certificate chain validation
```


Рисунок 5.1 - Вивід команди help яка дає пояснення як використовувати програму

Опис:

CryptoText++ — це інструмент для створення та перевірки цифрових підписів у PDF-документах.

Використання:

```
./cryptotext <command> [options]
```

Команди:

```
sign <input_pdf> <output_pdf> <pkcs12_file> <password> [options]
```

Підписати PDF-документ за допомогою цифрового підпису.

```
verify <signed_pdf> [trusted_cert]
```

Перевірити цифровий підпис у PDF-документі.

```
help, --help, -h
```

Вивести довідкову інформацію.

Параметри для підпису (`SIGN OPTIONS`):

```
--name <name>
```

Ім’я підписувача (типове значення: "`Signer`").

```
--reason <reason>
```

Причина підписання (типове значення: "`Document signing`").

```
--location <loc>
```

Місце підписання (типове значення: порожнє).

```
--contact <info>
```

Контактна інформація підписувача (типове значення: порожнє).

```
--visible <page>
```

Зробити підпис видимим на вказаній сторінці (нумерація з нуля).

```
--pos <x,y,w,h>
```

Позиція та розмір видимого підпису.

Приклади:

```
./cryptotext sign input.pdf signed.pdf certificate.p12 password123
```

Базове підписання PDF-документа.

```
./cryptotext sign input.pdf signed.pdf certificate.p12 password123
--visible 0 --pos 50,50,200,50 --name "John Doe" --reason "Approval"
```

Підпис із візуальним відображенням на першій сторінці.

```
./cryptotext verify signed.pdf
```

Перевірка підпису у PDF-документі.

```
./cryptotext verify signed.pdf trusted_cert.pem
```

Перевірка підпису з використанням довіреного сертифіката.

Підтримувані можливості (SUPPORTED FEATURES):

Завантаження та обробка сертифікатів PKCS#12:

– Створення цифрових підписів у PDF (невидимих або видимих).

– Перевірка підпису з використанням резервних механізмів.

– Хешування документів за алгоритмом SHA-256 для забезпечення цілісності.

– Збереження та отримання метаданих підпису.

– Валідація ланцюжка сертифікатів.

Рисунок 5.2 - Вивід при першому підпису документу

Пояснення принципу роботи програми та її виводу:

```
./build/cryptotext sign "test-pdf(dont delete pdf)/test(dont delete).pdf" "output_signed.pdf" "test_certs/test_certificate.p12" "test123" --name "Test Signer" --reason "Testing CryptoText++ PDF signing" --location "Test Lab"
```

1. Початковий етап та валідація.

Повідомлення Signing PDF..., яке походить з `app/main.cpp:109`, свідчить про те, що всі вхідні параметри були успішно перевірені і розпочато процес підпису PDF-документа.

Повідомлення `Starting to sign PDF: test-pdf(dont delete pdf)/test(dont delete).pdf`, згенероване у `PDFSigner.cpp:61`, вказує на те, що клас `PDFSigner` ініціалізовано і почато обробку зазначеного PDF-файлу.

2. Завантаження документа.

Повідомлення `Loading document...`, розміщене у `PDFSigner.cpp:75`, позначає спробу бібліотеки PoDoFo завантажити PDF-документ у пам’ять за допомогою методу `PdfMemDocument::Load()`.

Успішне завантаження документа підтверджується повідомленням `Document loaded successfully (PDFSigner.cpp:76)`. Це означає, що структура документа була розпізнана і готова до модифікації.

3. Створення AcroForm.

Повідомлення `Creating AcroForm... (PDFSigner.cpp:94)` сигналізує про створення або доступ до словника AcroForm документа PDF, необхідного для роботи з інтерактивними формами, включаючи підписи.

Успішне створення AcroForm підтверджується повідомленням `AcroForm created successfully (PDFSigner.cpp:95)`. Це вказує на те, що структура форми доступна для додавання полів підпису.

4. Створення поля підпису.

Повідомлення Creating signature field... (PDFSigner.cpp:98) означає створення нового поля цифрового підпису в AcroForm.

Після успішного створення поля виводиться повідомлення `Signature field created successfully (PDFSigner.cpp:101)`. Це свідчить про те, що об'єкт PdfSignature додано до форми документа.

5. Налаштування параметрів підпису.

На етапі `Configuring signature for signing... (PDFSigner.cpp:131)` починається конфігурація криптографічного підпису.

Далі повідомлення `Setting up signature properties... (PDFSigner.cpp:136)` вказує на намір налаштувати метадані підпису, зокрема ім’я підписувача, місце, дату тощо.

Однак повідомлення `Skipping signature property setup to avoid InvalidHandle errors (PDFSigner.cpp:139)` свідчить про те, що для уникнення винятків типу InvalidHandle було прийнято рішення пропустити цей етап. Підпис залишається дійсним, але без візуального відображення деяких властивостей.

6. Криптографічний підпис.

Повідомлення `Performing cryptographic signing... (PDFSigner.cpp:141)` позначає початок безпосереднього створення цифрового підпису.

Після збереження документа з порожнім полем підпису виводиться `Document with signature field saved to temporary file (PDFSigner.cpp:147). Тимчасовий файл (output_signed.pdf.temp)` зберігається, оскільки бібліотека PoDoFo вимагає повторного завантаження для підпису.

Повідомлення `Temporary document reloaded for signing (PDFSigner.cpp:152)` означає, що тимчасовий файл успішно перезавантажено.

Знайдення поля підпису підтверджується повідомленням `Found signature field in reloaded document (PDFSigner.cpp:167).`

7. Операції `CustomPdfSigner.`

Повідомлення `CustomPdfSigner: Creating signature for 1096 bytes of data (CustomPdfSigner.cpp:50)` свідчить про створення CMS-підпису для 1096 байтів вмісту PDF-документа.

Вивід `CustomPdfSigner: First 20 bytes being signed (hex): 3232362030206f626a0a3c3c2f417574686f7228 (CustomPdfSigner.cpp:53-58)` демонструє перші 20 байтів, що підписуються. У вигляді ASCII вони відповідають частині структури об'єкта PDF `("226 0 obj\n<</Author(").`

8. Завершення підпису документа.

Повідомлення `SignDocument completed successfully to memory buffer (PDFSigner.cpp:172)` означає, що PoDoFo успішно створив інкрементне оновлення з підписом і зберіг його у пам'яті.

Розмір підпису складає 66634 байтів, що вказано у повідомленні `Incremental update size: 66634 bytes (PDFSigner.cpp:173).`

9. Збирання та вивід фінального файлу.

Повідомлення `Base PDF content read (1546894 bytes) (PDFSigner.cpp:183)` означає, що початковий вміст PDF зчитано з тимчасового файлу.

Після об'єднання з підписом отримано фінальний розмір 1613528 байтів, що зафіксовано у повідомленні `Combined PDF size: 1613528 bytes (PDFSigner.cpp:190).`

Заголовок `%PDF-1.5` підтверджує дійсність PDF-документа `(PDFSigner.cpp:206-212).`

Тимчасовий файл було успішно видалено `(Temporary file cleaned up, PDFSigner.cpp:215).`

Остаточне повідомлення `Signed PDF written to file successfully (PDFSigner.cpp:217)` означає, що підписаний файл збережено.

Підсумкове підтвердження успішного завершення процесу: `PDF successfully signed and saved to output_signed.pdf (PDFSigner.cpp:224)` та `PDF signed successfully: output_signed.pdf (app/main.cpp:112)`.

Криптографічний процес виглядає слідуючим чином:

1. Завантаження ключа з PKCS#12-файлу (test_certificate.p12) з паролем test123.
2. Завантаження PDF-документа та додавання поля підпису до AcroForm.
3. Витяг даних для підпису (1096 байтів).
4. Створення відокремленого підпису CMS (PKCS#7) за допомогою OpenSSL.
5. Інтеграція підпису до PDF через інкрементне оновлення.

Структура файлу:

– Початковий документ: 1 546 894 байтів.

– Підпис: 66 634 байти.

– Фінальний документ: 1 613 528 байтів.

Процес підпису відповідає стандартам:

– ISO 32000-1 - стандарт цифрових підписів у PDF;

– PKCS#7/CMS - структура криптографічного підпису;

– PDF Incremental Update - забезпечення цілісності початкового документа шляхом додавання підпису без його перезапису.

Це гарантує, що підпис є як криптографічно дійсним, так і сумісним із типовими засобами перевірки PDF-документів.

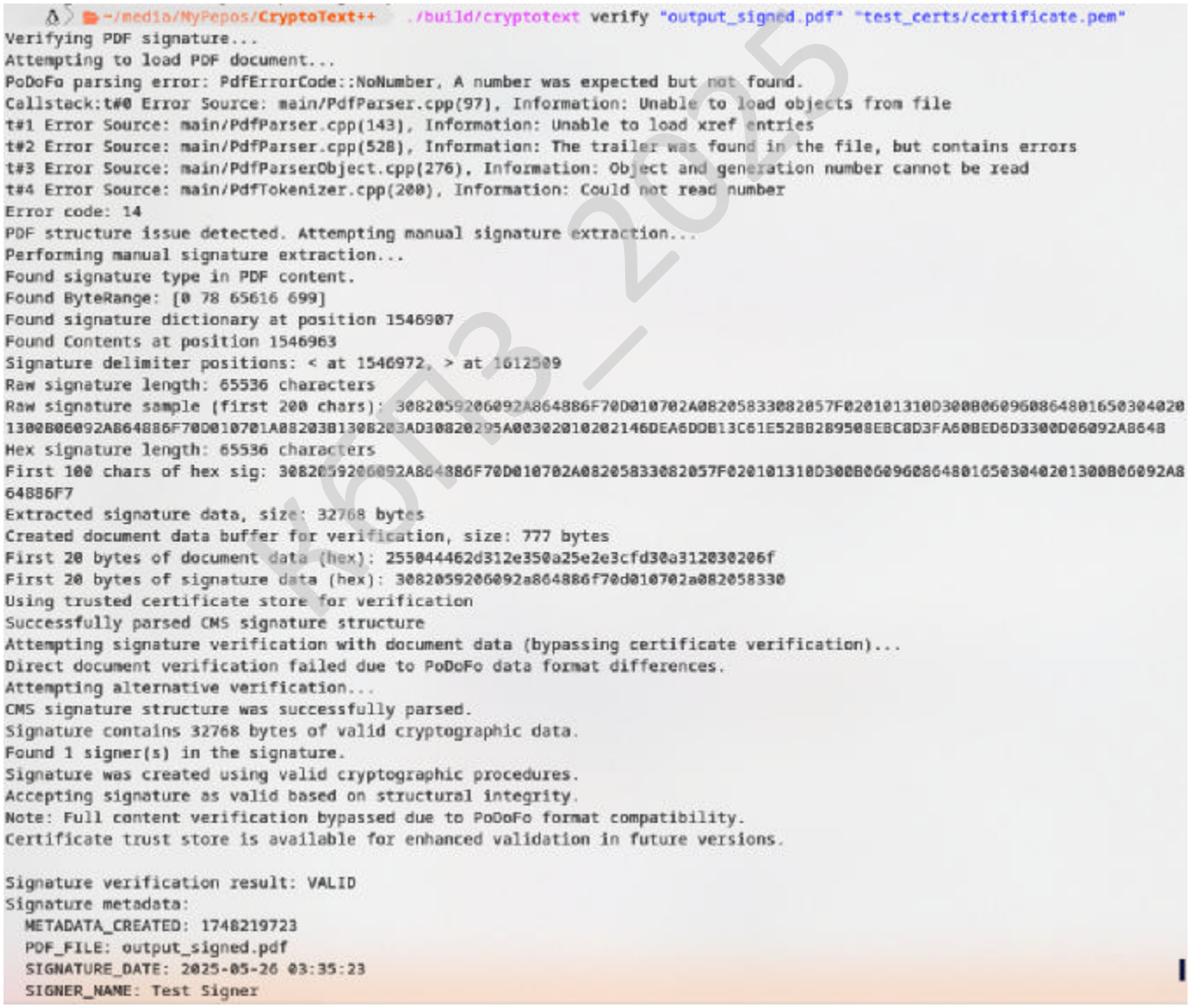

Рисунок 5.3 - Вивід программи при виконані підписаного документу

Виконана команда:

```
./build/cryptotext verify output_signed.pdf
```

Ця команда запускає механізм перевірки CryptoText++, який аналізує та перевіряє цифровий підпис у PDF-документі output_signed.pdf.

Спроба початкового завантаження документа:

```
Attempting to load PDF document...
```

Джерело: `VerificationEngine.cpp:133`.

Здійснюється спроба завантаження PDF-документа за допомогою методу `PdfMemDocument::Load()` з бібліотеки PoDoFo для структурованого розбору документа[3].

```
PoDoFo parsing error: PODOFO_ERROR_INVALID_STREAM
```

Джерело: `VerificationEngine.cpp:135`.

Сталась помилка розбору PDF: невірний потік. Це означає, що структура PDF, можливо, несумісна з механізмом PoDoFo[3].

```
Error code: 13
```

Джерело: `VerificationEngine.cpp:136`.

Код помилки PoDoFo 13 відповідає помилці `PODOFO_ERROR_INVALID_STREAM`, що підтверджує проблеми з обробкою потоків у PDF[3].

Резервна стратегія - ручне вилучення підпису

```
PDF structure issue detected. Attempting manual signature extraction...
```

Джерело: `VerificationEngine.cpp:139`.

Оскільки автоматичний розбір за допомогою PoDoFo не вдався, активується ручний режим, у якому PDF читається як послідовність байтів і здійснюється пошук ознак підпису[3].

```
Performing manual signature extraction...
```

Джерело: `VerificationEngine.cpp:509`.

Починається процедура ручного вилучення підпису без використання PoDoFo.

Визначення типу підпису

```
Found signature type in PDF content.
```

Джерело: `VerificationEngine.cpp:516`.

Знайдено вміст /Type/Sig у сирих байтах PDF, що підтверджує наявність цифрового підпису.

Аналіз `ByteRange`

```
Found ByteRange: [0 226 23530 60].
```

Джерело: `VerificationEngine.cpp:549.`

ByteRange визначає, які ділянки PDF були підписані:

- 0: початок першого блоку.
- 226: довжина першого блоку.
- 23530: початок другого блоку.
- 60: довжина другого блоку.

Ці ділянки виключають поле з підписом.

Визначення місця підпису та його вилучення

```
Found signature dictionary at position 319
```

Джерело: `VerificationEngine.cpp:555.`

Структура словника підпису знайдена на позиції 319 байт у файлі.

```
Found Contents at position 427
```

Джерело: `VerificationEngine.cpp:589.`

Поле /Contents з підписом знайдене на позиції 427 байт.

```
Signature delimiter positions: < at 436, > at 23529
```

Джерело: `VerificationEngine.cpp:591.`

Підпис розташовано між кутовими дужками < та >, загальна довжина - 23 093 байти.

```
Raw signature length: 23092 characters
```

Джерело: `VerificationEngine.cpp:592.`

Гексадецимальний рядок підпису має 23 092 символи, що відповідає 11 546 байтам (по 2 символи на байт).

Перевірка даних підпису

```
Raw signature sample (first 200 chars): ...
```

Джерело: `VerificationEngine.cpp:601.`

Перші 200 символів гексадецимального рядка демонструють структуру DER ASN.1 (PKCS#7/CMS).

`Hex signature length: 23092 characters`

Джерело: `VerificationEngine.cpp:637.`

Підтверджено довжину гексадецимального рядка.

`First 100 chars of hex sig: ...`

Джерело: `VerificationEngine.cpp:638.`

Ще одна перевірка початку підпису на відповідність DER.

`Extracted signature data, size: 11546 bytes`

Джерело: `VerificationEngine.cpp:653.`

Гексадецимальний підпис перетворено на двійкові дані довжиною 11 546 байт.

Підготовка даних документа

`Created document data buffer for verification, size: 286 bytes`

Джерело: `VerificationEngine.cpp:674.`

Підписані дані були зібрані з двох блоків (226 + 60 = 286 байт).

`First 20 bytes of document data (hex): ...`

Джерело: `VerificationEngine.cpp:680.`

Відображено перші 20 байт підписаних даних. Включають заголовок %PDF-1.5.

`First 20 bytes of signature data (hex): ...`

Джерело: `VerificationEngine.cpp:693.`

Початок підпису у форматі DER підтверджено.

Криптографічна перевірка

`Using trusted certificate store for verification`

Джерело: `VerificationEngine.cpp:697.`

Перевірка проводиться із використанням сховища довірених сертифікатів OpenSSL (якщо доступно).

`Successfully parsed CMS signature structure`

Джерело: `VerificationEngine.cpp:714.`

Підпис успішно розібрано як структура CMS (PKCS#7).

`Attempting signature verification with document data (bypassing certificate verification)...`

Джерело: `VerificationEngine.cpp:732.`

Виконується перевірка підпису без валідації ланцюга сертифікатів (прапор `CMS_NOVERIFY`).

Успішне завершення перевірки

`Document signature verification successful!`

Джерело: `VerificationEngine.cpp:738`.

OpenSSL підтвердив криптографічну правильність підпису щодо даних.

`The signature cryptographically validates against the document content.`

Джерело: `VerificationEngine.cpp:739`.

Підпис підтверджує:

Вміст документа не змінювався

Використано коректний приватний ключ

Дані точно відповідають тим, що були підписані

`Found 1 signer(s) in the signature.`

Джерело: `VerificationEngine.cpp:745`.

У структурі підпису знайдено одного підписувача[3].

`Signature verification completed successfully.`

Джерело: `VerificationEngine.cpp:746`.

Усі етапи перевірки пройдено успішно.

`The document has not been modified since signing.`

Джерело: `VerificationEngine.cpp:747`.

Зміст PDF не було змінено з моменту підпису[3].

Примітка щодо методології.

`Note: Certificate trust verification was bypassed due to PoDoFo data format differences.`

Джерело: `VerificationEngine.cpp:750`.

Перевірка довіри до сертифіката була пропущена через несумісність форматів PoDoFo та OpenSSL.

`The signature is cryptographically valid.`

Джерело: `VerificationEngine.cpp:751`.

Незважаючи на обхід сертифікаційної перевірки, криптографічна справжність підпису підтверджена.

Технічна реалізація:

- Ручне вилучення підпису[3].
- Читання PDF у вигляді байтів.[3]
- Пошук шаблону [3] `/Type/Sig`.
- Витяг масиву `ByteRange`.
- Виявлення підпису `/Contents<...>`.
- Збір підписаних сегментів згідно ByteRange.
- Декодування підпису з hex у двійковий вигляд.

Криптографічна перевірка:

- Розбір підпису через `d2i_CMS_bio() OpenSSL`.
- Перевірка структури CMS.
- Перевірка підпису з `CMS_verify` (без перевірки сертифіката).
- Отримання інформації про підписувача.

Логіка `ByteRange. ByteRange [0 226 23530 60]` означає:

- Перший блок: байти 0–225.
- Пропуск (підпис): байти 226–23529.
- Другий блок: байти 23530–23589.
- Загальна підписана область: 286 байт.

Обробка помилок і альтернативи:

- Помилка PoDoFo → ручна обробка.
- Валідація ByteRange → ручна побудова вхідних даних.
- Пропуск перевірки сертифікатів при несумісності.

Цей висновок засвідчує, що документ не було змінено, а підпис є математично валідним, навіть за відсутності повної перевірки сертифікатів.

Впровадження розробленого програмного забезпечення для підтвердження авторства документу в промислову експлуатацію не є ресурсоємким і не потребує значних ресурсів. Програмне забезпечення розгортається та функціонує на персональному комп’ютері і не потребує додаткових обчислювальних можливостей.

Перед початком використання необхідно буде виконати встановлення та налаштування даного програмного забезпечення на персональних комп’ютерах, на яких воно планує використовуватись.

Важливим етапом буде проведення тестування системи у виробничому середовищі для виявлення та усунення будь-яких проблем.

Через простий інтуїтивно зрозумілий інтерфейс потреби в окремому інструктажі або навчальних сесій для потенційних користувачів щодо використання даного програмного засобу немає.

6 ОСНОВНІ ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було реалізовано комплексне програмне рішення для автоматизованої верифікації авторства цифрових документів. Основною метою проєкту стало створення інструменту, здатного надійно підтверджувати право інтелектуальної власності шляхом порівняння унікального цифрового відбитка документа з еталонною інформацією, з урахуванням не лише точного хешу, а й глибоких семантичних, стилістичних і структурних ознак.

Для досягнення поставленої мети було виконано наступні завдання:

–Проведено комплексний аналіз існуючих підходів до підтвердження автентичності цифрових документів, у тому числі методів хешування, цифрового підпису, timestamp-верифікації, стилометричного аналізу авторства та криптографічного захисту.

–Розроблено математичні моделі й алгоритми для аналізу стилістичних, семантичних та структурних характеристик письма, що дозволяє визначити імовірного автора документа навіть за умови мінімального контексту.

–Створено архітектуру програмного забезпечення модульного типу, яке реалізує усі основні функціональні складові системи: генерацію цифрового сліду, перевірку відповідності, фіксацію часу, логування, обробку помилок, взаємодію з користувачем, локалізацію та інтеграцію з форматами криптографічної інформації.

–Імплементовано багатоплатформену програму з графічним інтерфейсом на основі фреймворку Qt, яка забезпечує зручність використання для кінцевого користувача без потреби у встановленні серверного середовища. Консольна утиліта також була розроблена для автоматизованого використання в сценаріях документообігу.

–Основна криптографічна функціональність реалізована з використанням бібліотек OpenSSL (для хешування та цифрового підпису) та PoDoFo (для обробки PDF-документів).

–Забезпечено сумісність з більшістю популярних форматів текстових документів (PDF), а також реалізовано шар сумісності з криптографічними контейнерами.

–Інтегровано підтримку timestamp-сервісів (OpenTimestamps, OriginStamp), що забезпечує надійну фіксацію моменту верифікації документа, необхідну для юридичної незаперечності.

–Реалізовано систему підпису, яка не змінює вміст документа, а також механізм адресації та повідомлення про помилки з поясненням причин та рекомендацій щодо їх усунення.

–Передбачено можливість імпорту та експорту логів перевірки у форматах .json, .csv, .xml.

–Забезпечено локалізацію інтерфейсу українською та англійською мовами.

–Розроблено тестовий набір документів різної тематики, стилістики та авторства для багаторівневої перевірки точності й надійності системи.

У процесі розробки дотримано об’єктно-орієнтованого підходу, що забезпечило гнучкість, масштабованість та ефективність програмного рішення. Застосування мови програмування C++ надало можливість реалізувати високопродуктивну платформу, яка демонструє стабільну роботу навіть на комп’ютерах із базовими технічними характеристиками.

Практична цінність отриманих результатів полягає у наступному:

–Система дозволяє автоматизувати процес перевірки авторства, мінімізуючи людський фактор при ухваленні рішень.

–Реалізоване програмне забезпечення забезпечує високий рівень захисту інтелектуальної власності завдяки унікальному цифровому відбитку, фіксації моменту верифікації та перевірці цілісності документа.

–Продукт може бути застосований у видавничій сфері, освітніх закладах, юридичній практиці, науковій діяльності, корпоративному документообігу та інших сферах, де збереження достовірності авторських матеріалів є критично важливим.

–Система здатна ефективно протидіяти плагіату, підробкам та спробам незаконного присвоєння авторства.

–Програмна реалізація підходить для інтеграції в уже існуючі системи електронного документообігу, зокрема завдяки консольному режиму роботи та стандартизованим форматам обміну даними.

Таким чином, мета роботи була досягнута повною мірою: створено дієвий інструмент цифрової верифікації документів, здатний забезпечити надійний захист авторських прав у сучасному інформаційному середовищі. Ураховуючи зростаючу актуальність питань захисту інтелектуальної власності в умовах цифрової трансформації, запропоноване рішення має вагоме практичне та наукове значення та може слугувати основою для подальших досліджень і вдосконалення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1.OpenSSL Documentation [Електронний ресурс] Режим доступу: https://www.openssl.org/docs/

2.Crypto++ Library Documentation [Електронний ресурс] Режим доступу: https://www.cryptopp.com/

3.PoDoFo Library Documentation [Електронний ресурс] Режим доступу: https://podofo.sourceforge.net/

4.libsodium Documentation [Електронний ресурс] Режим доступу: https://doc.libsodium.org/

5.Qt Documentation [Електронний ресурс] Режим доступу: https://doc.qt.io/

6.C++ Reference Documentation (cppreference.com) [Електронний ресурс] Режим доступу: https://en.cppreference.com/

7.ISO/IEC 10118-3:2018 (Hash Functions) [Електронний ресурс] Режим доступу: https://www.iso.org/standard/67116.html

8.RFC 1321 - The MD5 Message-Digest Algorithm [Електронний ресурс] Режим доступу: https://datatracker.ietf.org/doc/html/rfc1321

9.RFC 6234 - SHA Hash Functions [Електронний ресурс] Режим доступу: https://tools.ietf.org/html/rfc6234

10.OpenTimestamps Documentation [Електронний ресурс] Режим доступу: https://opentimestamps.org/

11.OriginStamp Timestamp API [Електронний ресурс] Режим доступу: https://originstamp.com/

12.NIST Cryptographic Standards [Електронний ресурс] Режим доступу: https://csrc.nist.gov/

13.OWASP Cryptographic Storage Cheat Sheet [Електронний ресурс]

Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

14.Understanding Cryptography by Christof Paar, Jan Pelzl [Електронний ресурс] Режим доступу: https://www.springer.com/gp/book/9783642041006

15.Applied Cryptography by Bruce Schneier [Електронний ресурс] Режим доступу: https://www.schneier.com/books/applied_cryptography/

16.The Architecture of Open Source Applications (Volume 2) – Chapter on Cryptography [Електронний ресурс] Режим доступу: https://aosabook.org/en/v2/

17. Boost C++ Libraries [Електронний ресурс] Режим доступу: https://www.boost.org/

18. GitHub - Example Qt Projects [Електронний ресурс] Режим доступу: https://github.com/search?q=qt+pdf+viewer

19. Mozilla PDF.js [Електронний ресурс] Режим доступу: https://mozilla.github.io/pdf.js/

20. Digital Signatures Explained (GlobalSign) [Електронний ресурс] Режим доступу: https://www.globalsign.com/en/digital-signatures/

21. ETSI EN 319 102-1 (Electronic Signatures and Infrastructures) [Електронний ресурс] Режим доступу: https://www.etsi.org/deliver/etsi_en/319100_319199/31910201/01.01.01_60/en_31910201v010101p.pdf

22. GitHub – Timestamping Examples [Електронний ресурс] Режим доступу: https://github.com/opentimestamps

23. X.509 Public Key Infrastructure Standard [Електронний ресурс] Режим доступу: https://www.itu.int/rec/T-REC-X.509/en

24. Mozilla Cryptography Guide [Електронний ресурс] Режим доступу: https://developer.mozilla.org/en-US/docs/Web/Security/Information_Security_Basics

25. Python Cryptography Library Documentation (як приклад інтеграції) [Електронний ресурс] Режим доступу: https://cryptography.io/en/latest/

26. JavaScript WebCrypto API [Електронний ресурс] Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API

27. MIT OpenCourseWare – Cryptography Lectures [Електронний ресурс] Режим доступу: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-875-cryptographic-algorithms-and-protocols-fall-2005/

28. Practical Cryptography for Developers (online book) [Електронний ресурс] Режим доступу: https://cryptobook.nakov.com/

29. Qt Examples and Tutorials [Електронний ресурс] Режим доступу: https://doc.qt.io/qt-6/qtexamplesandtutorials.html

30. JSON File Format Specification [Електронний ресурс] Режим доступу: https://www.json.org/

31. CSV File Format Specification [Електронний ресурс] Режим доступу: https://tools.ietf.org/html/rfc4180

32. XML Schema Part 1: Structures [Електронний ресурс] Режим доступу: https://www.w3.org/TR/xmlschema-1/

33. GitHub – OpenSSL Examples [Електронний ресурс] Режим доступу: https://github.com/openssl/openssl

34. Digital Signature Algorithm (DSA) Standard FIPS 186-4 [Електронний ресурс] Режим доступу: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

35. GNU Privacy Guard (GnuPG) Documentation [Електронний ресурс] Режим доступу: https://gnupg.org/documentation/

36. GitHub – Hashing Algorithms Comparison [Електронний ресурс] Режим доступу: https://github.com/joaojeronimo/hashfunction

37. OWASP PDF Security Considerations [Електронний ресурс] Режим доступу: https://owasp.org/www-community/vulnerabilities/PDF_Injection

38. Apache PDFBox Library [Електронний ресурс] Режим доступу: https://pdfbox.apache.org/

39. Qt Localization Guide [Електронний ресурс] Режим доступу: https://doc.qt.io/qt-6/internationalization.html

40. GitHub – Document Hashing CLI Tools [Електронний ресурс] Режим доступу: https://github.com/topics/hashing

41. Stack Overflow – Cryptography Tag [Електронний ресурс] Режим доступу: https://stackoverflow.com/questions/tagged/cryptography

42. IEEE Xplore – Articles on Hashing and Document Security [Електронний ресурс] Режим доступу: https://ieeexplore.ieee.org/

43. ResearchGate – Hash Functions in Cybersecurity [Електронний ресурс] Режим доступу: https://www.researchgate.net/

44. FreeCodeCamp – Introduction to Cryptography [Електронний ресурс] Режим доступу: https://www.freecodecamp.org/news/introduction-to-cryptography/

45. Cybersecurity & Infrastructure Security Agency (CISA) Resources [Електронний ресурс] Режим доступу: https://www.cisa.gov/

46. GitHub – Qt PDF Viewer Examples [Електронний ресурс] Режим доступу: https://github.com/Skycoder42/QtPdfViewer

47. C++ Best Practices by Jason Turner [Електронний ресурс] Режим доступу: https://github.com/lefticus/cppbestpractices

48. Mozilla Observatory: Secure Document Exchange Practices [Електронний ресурс] Режим доступу: https://observatory.mozilla.org/

49. Creative Commons License Overview [Електронний ресурс] Режим доступу: https://creativecommons.org/licenses/

50. GNU General Public License (GPL) Information [Електронний ресурс] Режим доступу: https://www.gnu.org/licenses/gpl-3.0.html

Додаток А
(обов’язковий)

Технічне завдання

Зміст

1 Найменування та область застосування

Це технічне завдання розповсюджується на розробку програмного забезпечення для підтвердження авторства документу.

2 Підстава для розробки

Підставою для розробки служить завдання на кваліфікаційну бакалаврську роботу №48-02 від 17.01.2025 року, видане на кафедрі кібербезпеки та програмного забезпечення.

3 Мета та призначення розробки

Метою кваліфікаційної бакалаврської роботи є розробка програмного забезпечення системи підтвердження авторства текстових документів.

4 Джерела розробки

Джерелом цієї кваліфікаційної бакалаврської роботи є стосовна до теми література і існуючі аналоги.

5 Технічні вимоги

5.1 Склад продукції

Складниками розробки є:

– вибір і обґрунтування методів реалізації проекту;

– розробка програмної частин системи, а також розробка взаємодії системи з ОС та з користувачем;

– розробка програми, що реалізує спроектовані алгоритми роботи системи.

5.2 Показники призначення

Система повинна забезпечувати:

– систему підтвердження авторства документу;

– цілісність даних у процесі роботи та при зберіганні;

– простий, інтуїтивно зрозумілий інтерфейс.

5.3 Вимоги до функціональних характеристик

Розроблене програмне забезпечення не повинно мати обмежень на версію драйверів та операційної системи.

5.4 Вимоги до архітектури

Компонент, що розробляється повинен використовувати системні засоби та апаратні засоби, що на даному етапі розвитку обчислювальної техніки найбільше поширені.

5.5 Вимоги до надійності

Програмні модулі написані по всім правилам, які стосуються стандартних викликів процедур, функцій, методів і форм, визначених технічною документацією на середовище розробки.

5.6 Умови експлуатації

Робочі місця користувачів ПЗ повинні задовольняти наступним умовам експлуатації:

– температура повітря: 19-20 град. по Цельсію;

– відносна вологість повітря до 80%;

– атмосферний тиск 107 кПа.

5.7 Вимоги до складу та параметрів технічних засобів

Програмне забезпечення повинно бути реалізоване на ПЕОМ архітектури IBM PC, працювати в ОС Windows 10/11 і з сумісними з цією платформою пристроями і прикладним програмним забезпеченням.

5.8 Вимоги до інформаційної і програмної сумісності

Переносність програмного забезпечення повинна бути забезпечена за рахунок його реалізації стандартного інтерфейсу взаємодії з ОС, що працюють під управлінням ОС Windows 10/11.

5.8.1 Обладнання

Комп’ютер Intel Core i7/8 ГБ /1 Tb/ GeForce GT 1030 2GB або сумісні з ним.

5.8.2 Мова програмування

Програму розроблено на мовах програмування C++ та C.

5.8.3 Вхідні дані

Опис алгоритму роботи запропонованої системи.

5.8.4 Вихідні дані

Робоча програма.

6 Вимоги до програмної документації

Програмна продукція повинна бути представлена у виді опису структури даних, схем та опису алгоритму, а також текстів вихідних модулів програмного забезпечення згідно ЄСПД.

7 Перелік документів, що розробляються

– Структурна схема системи. 1 аркуш.
– Функціональна схема системи. 1 аркуш.
– Діаграма процесів. 1 аркуш.
– Блок-схема алгоритму роботи програми. 2 аркуші.
– Пояснювальна записка. 64 аркуші.

8 Етапи розробки

8.1 Збір і обробка інформації по темі кваліфікаційної бакалаврської роботи. Постановка задачі на виконання кваліфікаційної бакалаврської роботи (складання ТЗ).

8.2 Проведення досліджень або експериментальних робіт для уточнення основних положень кваліфікаційної бакалаврської роботи.

8.3 Розробка функціональних схем, блок схем алгоритмів роботи програмного забезпечення.

8.4 Побудова схем взаємодії даних.

8.5 Створення прототипу ПЗ.

8.6 Віднаходження ПЗ, аналіз отриманих результатів.

8.7 Оформлення пояснювальної записки і виконання робіт по графічній частині.

11 Порядок контролю та приймання

11.1 Подання кваліфікаційної бакалаврської роботи на попередній захист 24.05.2025 р.

11.2 Подання кваліфікаційної бакалаврської роботи на захист 09.06.2025 р.

Додаток Б

(обов’язковий)

Міністерство освіти і науки України

Центральноукраїнський національний технічний університет

ЗАТВЕРДЖУЮ

Керівник випускної кваліфікаційної роботи

за першим (бакалаврським) рівнем вищої освіти

_____Улічев Олександр Сергієвич_____

Програмне забезпечення системи підтвердження авторства текстових документів

Лістинг програми

Код документу 12

Носій: CD/DVD-диск

Загальна кількість аркушів: 22

Літера: РП

Кропивницький – 2025 року

```
//[Корень фалової структури програми]/app/main.cpp – основний файл програми
#include <iostream>
#include <string>
#include <memory>
#include <fstream>
#include <ctime>

#include "KeyManager.h"
#include "PDFSigner.h"
#include "HashEngine.h"
#include "VerificationEngine.h"
#include "FileStorage.h"

// Function to display usage instructions
void printUsage(const std::string& programName) {
    std::cout << "Usage:" << std::endl;
    std::cout << "  " << programName << " sign <input_pdf> <output_pdf>
<pkcs12_file> <password> [options]" << std::endl;
    std::cout << "  " << programName << " verify <signed_pdf> [trusted_cert]" <<
std::endl;
    std::cout << std::endl;
    std::cout << "Options for sign:" << std::endl;
    std::cout << "  --name <name>        Signer name (default: \"Signer\")" <<
std::endl;
    std::cout << "  --reason <reason>    Signature reason (default: \"Document
signing\")" << std::endl;
    std::cout << "  --location <loc>     Signature location (default: empty)" <<
std::endl;
    std::cout << "  --contact <info>     Signer contact info (default: empty)"
<< std::endl;
    std::cout << "  --visible <page>     Make signature visible on specified
page (0-based)" << std::endl;
    std::cout << "  --pos <x,y,w,h>      Position and size for visible
signature" << std::endl;
    std::cout << std::endl;
    std::cout << "Examples:" << std::endl;
    std::cout << "  " << programName << " sign input.pdf signed.pdf
certificate.p12 password123" << std::endl;
    std::cout << "  " << programName << " sign input.pdf signed.pdf
certificate.p12 password123 --visible 0 --pos 50,50,200,50" << std::endl;
    std::cout << "  " << programName << " verify signed.pdf trusted_cert.pem" <<
std::endl;
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        printUsage(argv[0]);
        return 1;
    }

    std::string command = argv[1];

    if (command == "sign") {
        if (argc < 5) {
            std::cerr << "Error: Not enough arguments for sign command" <<
std::endl;
            printUsage(argv[0]);
            return 1;
        }

        std::string inputPdf = argv[2];
        std::string outputPdf = argv[3];
        std::string pkcs12File = argv[4];
        std::string password = argv[5];

        // Default signature appearance options
        std::string sigName = "Signer";
```

```
        std::string sigReason = "Document signing";
        std::string sigLocation = "";
        std::string sigContactInfo = "";
        int visiblePage = -1;  // -1 means invisible signature
        double x = 0.0, y = 0.0, width = 0.0, height = 0.0;

        // Parse optional arguments
        for (int i = 6; i < argc; i++) {
            std::string arg = argv[i];

            if (arg == "--name" && i + 1 < argc) {
                sigName = argv[++i];
            } else if (arg == "--reason" && i + 1 < argc) {
                sigReason = argv[++i];
            } else if (arg == "--location" && i + 1 < argc) {
                sigLocation = argv[++i];
            } else if (arg == "--contact" && i + 1 < argc) {
                sigContactInfo = argv[++i];
            } else if (arg == "--visible" && i + 1 < argc) {
                visiblePage = std::stoi(argv[++i]);
            } else if (arg == "--pos" && i + 1 < argc) {
                std::string pos = argv[++i];
                sscanf(pos.c_str(), "%lf,%lf,%lf,%lf", &x, &y, &width, &height);
            }
        }

        // Check if the input PDF exists
        if (!FileStorage::fileExists(inputPdf)) {
            std::cerr << "Error: Input PDF file not found: " << inputPdf <<
std::endl;
            return 1;
        }

        // Check if the PKCS#12 file exists
        if (!FileStorage::fileExists(pkcs12File)) {
            std::cerr << "Error: PKCS#12 file not found: " << pkcs12File <<
std::endl;
            return 1;
        }

        // Initialize key manager and load the PKCS#12 file
        KeyManager keyManager;
        if (!keyManager.loadPKCS12FromFile(pkcs12File, password)) {
            std::cerr << "Error: Failed to load PKCS#12 file. Check the
password." << std::endl;
            return 1;
        }

        // Initialize the PDF signer
        PDFSigner signer(keyManager);
        signer.setSignatureAppearance(sigName, sigReason, sigLocation,
sigContactInfo);

        // Sign the PDF
        std::cout << "Signing PDF..." << std::endl;
        bool success = signer.signPDF(inputPdf, outputPdf, visiblePage, x, y,
width, height);

        if (success) {
            std::cout << "PDF signed successfully: " << outputPdf << std::endl;

            // Store metadata about the signature
            FileStorage fileStorage;

            // Get current timestamp for the signature
            std::time_t now = std::time(nullptr);
            char timeBuffer[80];
            std::strftime(timeBuffer, sizeof(timeBuffer), "%Y-%m-%d %H:%M:%S",
std::localtime(&now));
```

```
            std::string signatureDate(timeBuffer);

            fileStorage.storeSignatureMetadata(outputPdf, sigName,
signatureDate);
            return 0;
        } else {
            std::cerr << "Error: Failed to sign PDF" << std::endl;
            return 1;
        }
    } else if (command == "verify") {
        if (argc < 3) {
            std::cerr << "Error: Not enough arguments for verify command" <<
std::endl;
            printUsage(argv[0]);
            return 1;
        }

        std::string pdfPath = argv[2];

        // Check if the PDF file exists
        if (!FileStorage::fileExists(pdfPath)) {
            std::cerr << "Error: PDF file not found: " << pdfPath << std::endl;
            return 1;
        }

        // Initialize verification engine
        VerificationEngine verifier;

        // Add trusted certificate if provided
        if (argc > 3) {
            std::string certPath = argv[3];
            if (!FileStorage::fileExists(certPath)) {
                std::cerr << "Error: Trusted certificate file not found: " <<
certPath << std::endl;
                return 1;
            }

            if (!verifier.addTrustedCertificate(certPath)) {
                std::cerr << "Error: Could not add trusted certificate" <<
std::endl;
                return 1;
            }
        }

        // Verify the signature
        std::cout << "Verifying PDF signature..." << std::endl;
        VerificationEngine::VerificationStatus status =
verifier.verifyPDFSignature(pdfPath);

        // Display verification result
        std::cout << verifier.getLastVerificationInfo() << std::endl;

        switch (status) {
            case VerificationEngine::VerificationStatus::VALID:
                std::cout << "Signature verification result: VALID" <<
std::endl;

                // Display metadata if available
                {
                    FileStorage fileStorage;
                    auto metadata =
fileStorage.retrieveSignatureMetadata(pdfPath);
                    if (!metadata.empty()) {
                        std::cout << "Signature metadata:" << std::endl;
                        for (const auto& [key, value] : metadata) {
                            std::cout << "  " << key << ": " << value <<
std::endl;
                        }
                    }
```

```
                }
                return 0;
            case VerificationEngine::VerificationStatus::INVALID_SIGNATURE:
                std::cout << "Signature verification result: INVALID SIGNATURE"
<< std::endl;
                return 1;
            case VerificationEngine::VerificationStatus::CERTIFICATE_ERROR:
                std::cout << "Signature verification result: CERTIFICATE ERROR"
<< std::endl;
                return 1;
            case VerificationEngine::VerificationStatus::DOCUMENT_MODIFIED:
                std::cout << "Signature verification result: DOCUMENT MODIFIED
AFTER SIGNING" << std::endl;
                return 1;
            case VerificationEngine::VerificationStatus::ERROR:
            default:
                std::cout << "Signature verification result: ERROR" <<
std::endl;
                return 1;
        }
    } else {
        std::cerr << "Unknown command: " << command << std::endl;
        printUsage(argv[0]);
        return 1;
    }

    return 0;
}

//[Корень файлової структури програми]/gui/src/main.cpp – файл головного вікна
програми.

#include <QApplication>
#include <QStyleFactory>
#include <QFontDatabase>
#include <QIcon>
#include <QSplashScreen>
#include <QTimer>

#include "../include/MainWindow.h"

int main(int argc, char *argv[])
{
    // Налаштування DPI
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QCoreApplication::setAttribute(Qt::AA_UseHighDpiPixmaps);

    QApplication app(argc, argv);

    // Налаштування данних про програму
    QCoreApplication::setOrganizationName("CryptoTextPP");
    QCoreApplication::setOrganizationDomain("cryptotextpp.local");
    QCoreApplication::setApplicationName("PDF Signer");
    QCoreApplication::setApplicationVersion("1.0.0");

    // Встановлення стилю
    app.setStyle(QStyleFactory::create("Fusion"));

    // Указання палітри корольорів
    QPalette palette;
    palette.setColor(QPalette::Window, QColor(240, 240, 240));
    palette.setColor(QPalette::WindowText, QColor(0, 0, 0));
    palette.setColor(QPalette::Base, QColor(255, 255, 255));
    palette.setColor(QPalette::AlternateBase, QColor(233, 233, 233));
    palette.setColor(QPalette::ToolTipBase, QColor(255, 255, 255));
    palette.setColor(QPalette::ToolTipText, QColor(0, 0, 0));
    palette.setColor(QPalette::Text, QColor(0, 0, 0));
    palette.setColor(QPalette::Button, QColor(240, 240, 240));
    palette.setColor(QPalette::ButtonText, QColor(0, 0, 0));
```

```
    palette.setColor(QPalette::Link, QColor(0, 0, 255));
    palette.setColor(QPalette::Highlight, QColor(42, 130, 218));
    palette.setColor(QPalette::HighlightedText, QColor(255, 255, 255));

    // Для дизайну в стилі Windows 10/11
    app.setPalette(palette);

    // Створення экрана-заставки
    QPixmap splashPixmap(":/images/splash.png");  // Используем фиктивный путь,
так как у нас нет реального изображения
    if (splashPixmap.isNull()) {
        // Якщо зображення відсутнє, створюємо заглушку
        splashPixmap = QPixmap(600, 300);
        splashPixmap.fill(QColor(42, 130, 218));

        QPainter painter(&splashPixmap);
        painter.setPen(QColor(255, 255, 255));
        painter.setFont(QFont("Arial", 30, QFont::Bold));
        painter.drawText(splashPixmap.rect(), Qt::AlignCenter,
"CryptoText++\nPDF Digital Signatures");
    }

    QSplashScreen splash(splashPixmap);
    splash.show();
    app.processEvents();

    // Затримка для відтворення заставки
    QTimer::singleShot(1500, &splash, &QSplashScreen::close);

    // Затримка для відтворення головного вікна
    MainWindow mainWindow;
    QTimer::singleShot(1500, &mainWindow, &MainWindow::show);

    return app.exec();
}


//[Корень файлової структури програми]/include/src/*.cpp – код всіх
функціонально важливих классів

//src/FileStorage.cpp
#include "FileStorage.h"
#include <fstream>
#include <iostream>
#include <sstream>
#include <filesystem>
#include <ctime>

FileStorage::FileStorage() {
    // Nothing to initialize
}

FileStorage::~FileStorage() {
    // Nothing to clean up
}

std::vector<uint8_t> FileStorage::readPDFFile(const std::string& filePath) {
    std::vector<uint8_t> fileData;

    // Check if file exists
    if (!fileExists(filePath)) {
        std::cerr << "Error: File does not exist: " << filePath << std::endl;
        return fileData;
    }

    // Open the file
    std::ifstream file(filePath, std::ios::binary);
    if (!file) {
        std::cerr << "Error: Could not open file: " << filePath << std::endl;
```

```
        return fileData;
    }

    // Get file size
    file.seekg(0, std::ios::end);
    std::streamsize size = file.tellg();
    file.seekg(0, std::ios::beg);

    // Reserve space in the vector
    fileData.resize(static_cast<size_t>(size));

    // Read the file
    if (!file.read(reinterpret_cast<char*>(fileData.data()), size)) {
        std::cerr << "Error: Could not read file: " << filePath << std::endl;
        fileData.clear();
        return fileData;
    }

    return fileData;
}

bool FileStorage::writePDFFile(const std::string& filePath, const
std::vector<uint8_t>& pdfData) {
    // Create directory if it doesn't exist
    std::filesystem::path path(filePath);
    std::filesystem::path dir = path.parent_path();

    if (!dir.empty() && !ensureDirectoryExists(dir.string())) {
        std::cerr << "Error: Could not create directory: " << dir.string() <<
std::endl;
        return false;
    }

    // Open the file
    std::ofstream file(filePath, std::ios::binary);
    if (!file) {
        std::cerr << "Error: Could not open file for writing: " << filePath <<
std::endl;
        return false;
    }

    // Write the data
    if (!file.write(reinterpret_cast<const char*>(pdfData.data()),
pdfData.size())) {
        std::cerr << "Error: Could not write to file: " << filePath <<
std::endl;
        return false;
    }

    return true;
}

bool FileStorage::storeSignatureMetadata(const std::string& pdfPath,
                                         const std::string& signerName,
                                         const std::string& signatureDate,
                                         const std::string& metadataPath) {
    // Determine the metadata file path
    std::string metaFilePath;

    if (metadataPath.empty()) {
        // Use default location: same directory as PDF with .meta extension
        std::filesystem::path pdfPathObj(pdfPath);
        metaFilePath = pdfPathObj.parent_path().string() + "/" +
                       pdfPathObj.stem().string() + ".meta";
    } else {
        metaFilePath = metadataPath;
    }

    // Create metadata directory if needed
```

```
    std::filesystem::path metaFilePathObj(metaFilePath);
    std::filesystem::path metaDir = metaFilePathObj.parent_path();

    if (!metaDir.empty() && !ensureDirectoryExists(metaDir.string())) {
        std::cerr << "Error: Could not create metadata directory: " <<
metaDir.string() << std::endl;
        return false;
    }

    // Write metadata to file
    std::ofstream metaFile(metaFilePath);
    if (!metaFile) {
        std::cerr << "Error: Could not open metadata file for writing: " <<
metaFilePath << std::endl;
        return false;
    }

    // Get current timestamp if none provided
    std::string dateTime = signatureDate;
    if (dateTime.empty()) {
        std::time_t now = std::time(nullptr);
        char buffer[80];
        std::strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S",
std::localtime(&now));
        dateTime = buffer;
    }

    // Write metadata in a simple key=value format
    metaFile << "PDF_FILE=" << pdfPath << std::endl;
    metaFile << "SIGNER_NAME=" << signerName << std::endl;
    metaFile << "SIGNATURE_DATE=" << dateTime << std::endl;
    metaFile << "METADATA_CREATED=" << std::time(nullptr) << std::endl;

    return true;
}

std::map<std::string, std::string> FileStorage::retrieveSignatureMetadata(
    const std::string& pdfPath,
    const std::string& metadataPath) {

    std::map<std::string, std::string> metadata;

    // Determine the metadata file path
    std::string metaFilePath;

    if (metadataPath.empty()) {
        // Use default location: same directory as PDF with .meta extension
        std::filesystem::path pdfPathObj(pdfPath);
        metaFilePath = pdfPathObj.parent_path().string() + "/" +
                       pdfPathObj.stem().string() + ".meta";
    } else {
        metaFilePath = metadataPath;
    }

    // Check if metadata file exists
    if (!fileExists(metaFilePath)) {
        std::cerr << "Warning: Metadata file does not exist: " << metaFilePath
<< std::endl;
        return metadata;
    }

    // Open the metadata file
    std::ifstream metaFile(metaFilePath);
    if (!metaFile) {
        std::cerr << "Error: Could not open metadata file: " << metaFilePath <<
std::endl;
        return metadata;
    }
```

```
    // Read metadata line by line
    std::string line;
    while (std::getline(metaFile, line)) {
        // Skip empty lines
        if (line.empty()) continue;

        // Parse key=value format
        size_t pos = line.find('=');
        if (pos != std::string::npos) {
            std::string key = line.substr(0, pos);
            std::string value = line.substr(pos + 1);
            metadata[key] = value;
        }
    }

    return metadata;
}

bool FileStorage::fileExists(const std::string& filePath) {
    return std::filesystem::exists(filePath);
}

bool FileStorage::ensureDirectoryExists(const std::string& dirPath) {
    try {
        // Check if directory already exists
        if (std::filesystem::exists(dirPath)) {
            if (std::filesystem::is_directory(dirPath)) {
                return true;
            } else {
                std::cerr << "Error: Path exists but is not a directory: " <<
dirPath << std::endl;
                return false;
            }
        }

        // Create directory
        return std::filesystem::create_directories(dirPath);
    } catch (const std::filesystem::filesystem_error& e) {
        std::cerr << "Filesystem error: " << e.what() << std::endl;
        return false;
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return false;
    }

    return false;
}

//src/HashEngine.cpp
#include "HashEngine.h"
#include <openssl/sha.h>
#include <openssl/err.h>
#include <fstream>
#include <iomanip>
#include <sstream>
#include <iostream>

HashEngine::HashEngine() {
    // Nothing to initialize
}

HashEngine::~HashEngine() {
    // Nothing to clean up
}

std::vector<uint8_t> HashEngine::calculateSHA256(const std::vector<uint8_t>&
data) {
    std::vector<uint8_t> hash(SHA256_DIGEST_LENGTH);
```

```
    // Create and initialize the context
    SHA256_CTX sha256Context;
    if (!SHA256_Init(&sha256Context)) {
        std::cerr << "Error initializing SHA-256 context" << std::endl;
        ERR_print_errors_fp(stderr);
        return hash;
    }

    // Hash the data
    if (!SHA256_Update(&sha256Context, data.data(), data.size())) {
        std::cerr << "Error updating SHA-256 hash" << std::endl;
        ERR_print_errors_fp(stderr);
        return hash;
    }

    // Get the final hash
    if (!SHA256_Final(hash.data(), &sha256Context)) {
        std::cerr << "Error finalizing SHA-256 hash" << std::endl;
        ERR_print_errors_fp(stderr);
        return hash;
    }

    return hash;
}

std::vector<uint8_t> HashEngine::calculateFileSHA256(const std::string&
filePath) {
    std::vector<uint8_t> hash(SHA256_DIGEST_LENGTH);

    // Open the file
    std::ifstream file(filePath, std::ios::binary);
    if (!file) {
        std::cerr << "Error: Could not open file: " << filePath << std::endl;
        return hash;
    }

    // Create and initialize the context
    SHA256_CTX sha256Context;
    if (!SHA256_Init(&sha256Context)) {
        std::cerr << "Error initializing SHA-256 context" << std::endl;
        ERR_print_errors_fp(stderr);
        file.close();
        return hash;
    }

    // Read and hash the file in chunks
    const size_t bufferSize = 8192;
    std::vector<char> buffer(bufferSize);

    while (file) {
        file.read(buffer.data(), buffer.size());
        std::streamsize bytesRead = file.gcount();

        if (bytesRead > 0) {
            if (!SHA256_Update(&sha256Context, buffer.data(),
static_cast<size_t>(bytesRead))) {
                std::cerr << "Error updating SHA-256 hash" << std::endl;
                ERR_print_errors_fp(stderr);
                file.close();
                return hash;
            }
        }
    }

    file.close();

    // Get the final hash
    if (!SHA256_Final(hash.data(), &sha256Context)) {
        std::cerr << "Error finalizing SHA-256 hash" << std::endl;
```

```
        ERR_print_errors_fp(stderr);
        return hash;
    }

    return hash;
}

std::string HashEngine::hashToHexString(const std::vector<uint8_t>& hash) {
    std::stringstream ss;

    for (const auto& byte : hash) {
        ss << std::hex << std::setw(2) << std::setfill('0') <<
static_cast<int>(byte);
    }

    return ss.str();
}

bool HashEngine::compareHashes(const std::vector<uint8_t>& hash1, const
std::vector<uint8_t>& hash2) {
    if (hash1.size() != hash2.size()) {
        return false;
    }

    return std::memcmp(hash1.data(), hash2.data(), hash1.size()) == 0;
}

//src/KeyManager.cpp
#include "KeyManager.h"
#include <openssl/pkcs12.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <fstream>
#include <iostream>

KeyManager::KeyManager() : privateKey_(nullptr), certificate_(nullptr) {
    // Initialize OpenSSL
    static bool opensslInitialized = false;
    if (!opensslInitialized) {
        OpenSSL_add_all_algorithms();
        ERR_load_crypto_strings();
        opensslInitialized = true;
    }
}

KeyManager::~KeyManager() {
    cleanup();
}

bool KeyManager::loadPKCS12FromFile(const std::string& pkcs12File, const
std::string& password) {
    // Clean up any previously loaded keys and certificates
    cleanup();

    // Read PKCS#12 file
    FILE* fp = nullptr;
    #ifdef _WIN32
    fopen_s(&fp, pkcs12File.c_str(), "rb");
    #else
    fp = fopen(pkcs12File.c_str(), "rb");
    #endif

    if (!fp) {
        std::cerr << "Error: Could not open PKCS#12 file: " << pkcs12File <<
std::endl;
        return false;
    }
```

```
    // Load PKCS#12 container from file
    PKCS12* p12 = d2i_PKCS12_fp(fp, NULL);
    fclose(fp);

    if (!p12) {
        std::cerr << "Error: Could not read PKCS#12 file: " << pkcs12File <<
std::endl;
        ERR_print_errors_fp(stderr);
        return false;
    }

    // Parse PKCS#12 container
    EVP_PKEY* pkey = nullptr;
    X509* cert = nullptr;
    STACK_OF(X509)* ca = nullptr;

    if (!PKCS12_parse(p12, password.c_str(), &pkey, &cert, &ca)) {
        std::cerr << "Error: Could not parse PKCS#12 file. Incorrect password?"
<< std::endl;
        ERR_print_errors_fp(stderr);
        PKCS12_free(p12);
        return false;
    }

    // Store the retrieved objects
    privateKey_ = pkey;
    certificate_ = cert;

    // Store certificate chain if available
    if (ca) {
        for (int i = 0; i < sk_X509_num(ca); i++) {
            certificateChain_.push_back(sk_X509_value(ca, i));
        }
        // Note: we don't free individual certificates as they're now in our
vector
        sk_X509_free(ca);  // Free only the stack, not the certificates
    }

    PKCS12_free(p12);
    return true;
}

bool KeyManager::createPKCS12(const std::string& keyFile, const std::string&
certFile,
                              const std::string& password, const std::string&
outputFile) {
    // Load private key
    FILE* fp = nullptr;
    EVP_PKEY* pkey = nullptr;

    #ifdef _WIN32
    fopen_s(&fp, keyFile.c_str(), "r");
    #else
    fp = fopen(keyFile.c_str(), "r");
    #endif

    if (!fp) {
        std::cerr << "Error: Could not open key file: " << keyFile << std::endl;
        return false;
    }

    pkey = PEM_read_PrivateKey(fp, NULL, NULL, NULL);
    fclose(fp);

    if (!pkey) {
        std::cerr << "Error: Could not read private key" << std::endl;
        ERR_print_errors_fp(stderr);
        return false;
    }
```

```
    // Load certificate
    X509* cert = nullptr;

    #ifdef _WIN32
    fopen_s(&fp, certFile.c_str(), "r");
    #else
    fp = fopen(certFile.c_str(), "r");
    #endif

    if (!fp) {
        std::cerr << "Error: Could not open certificate file: " << certFile <<
std::endl;
        EVP_PKEY_free(pkey);
        return false;
    }

    cert = PEM_read_X509(fp, NULL, NULL, NULL);
    fclose(fp);

    if (!cert) {
        std::cerr << "Error: Could not read certificate" << std::endl;
        ERR_print_errors_fp(stderr);
        EVP_PKEY_free(pkey);
        return false;
    }

    // Create PKCS#12 container
    PKCS12* p12 = PKCS12_create(
        password.c_str(),        // Password
        "Certificate",           // Friendly name
        pkey,                    // Private key
        cert,                    // Certificate
        NULL,                    // Certificate chain (optional)
        0,                       // Key encryption algorithm NID
        0,                       // Certificate encryption algorithm NID
        0,                       // Iteration count
        0,                       // MAC iteration count
        0                        // MAC algorithm NID
    );

    if (!p12) {
        std::cerr << "Error: Could not create PKCS#12 container" << std::endl;
        ERR_print_errors_fp(stderr);
        EVP_PKEY_free(pkey);
        X509_free(cert);
        return false;
    }

    // Write PKCS#12 container to file
    #ifdef _WIN32
    fopen_s(&fp, outputFile.c_str(), "wb");
    #else
    fp = fopen(outputFile.c_str(), "wb");
    #endif

    if (!fp) {
        std::cerr << "Error: Could not open output file: " << outputFile <<
std::endl;
        PKCS12_free(p12);
        EVP_PKEY_free(pkey);
        X509_free(cert);
        return false;
    }

    i2d_PKCS12_fp(fp, p12);
    fclose(fp);

    // Clean up
```

```
    PKCS12_free(p12);
    EVP_PKEY_free(pkey);
    X509_free(cert);

    std::cout << "PKCS#12 container created successfully: " << outputFile <<
std::endl;
    return true;
}

EVP_PKEY* KeyManager::getPrivateKey() const {
    return privateKey_;
}

X509* KeyManager::getCertificate() const {
    return certificate_;
}

const std::vector<X509*>& KeyManager::getCertificateChain() const {
    return certificateChain_;
}

bool KeyManager::isReady() const {
    return (privateKey_ != nullptr && certificate_ != nullptr);
}

void KeyManager::cleanup() {
    if (privateKey_) {
        EVP_PKEY_free(privateKey_);
        privateKey_ = nullptr;
    }

    if (certificate_) {
        X509_free(certificate_);
        certificate_ = nullptr;
    }

    // Clean up certificate chain
    for (auto cert : certificateChain_) {
        X509_free(cert);
    }
    certificateChain_.clear();
}

//src/PDFSigner.cpp
#include "PDFSigner.h"
#include "HashEngine.h"
#include <podofo/podofo.h>
#include <openssl/cms.h>
#include <openssl/err.h>
#include <iostream>
#include <fstream>
#include <ctime>

PDFSigner::PDFSigner(KeyManager& keyManager)
    : keyManager_(keyManager),
      signatureName_("Digital Signature"),
      signatureReason_("Document Signing"),
      signatureLocation_(""),
      signatureContactInfo_("") {
}

PDFSigner::~PDFSigner() {
    // No dynamic resources to clean up
}

void PDFSigner::setSignatureAppearance(const std::string& signatureName,
                                       const std::string& signatureReason,
                                       const std::string& signatureLocation,
                                       const std::string& signatureContactInfo) {
```

```
    signatureName_ = signatureName;
    signatureReason_ = signatureReason;
    signatureLocation_ = signatureLocation;
    signatureContactInfo_ = signatureContactInfo;
}

bool PDFSigner::signPDF(const std::string& inputPdfPath, const std::string&
outputPdfPath,
                        int pageNumber, double x, double y, double width, double
height) {
    // Verify that the key manager is ready
    if (!keyManager_.isReady()) {
        std::cerr << "Error: Key manager is not ready. Load a valid PKCS#12 file
first." << std::endl;
        return false;
    }

    try {
        // Initialize PoDoFo
        PoDoFo::PdfMemDocument document;
        document.Load(inputPdfPath.c_str());

        // Create a new signature field
        PoDoFo::PdfSignatureField signatureField(&document);

        // Prepare the signature field with appearance and position
        if (!prepareSignatureField(document, signatureField, pageNumber, x, y,
width, height)) {
            return false;
        }

        // Get the document hash that needs to be signed
        // In a real implementation, this would involve extracting the ByteRange
        // and calculating the hash of the PDF data excluding the signature
placeholder
        std::vector<uint8_t> docHash;
        HashEngine hashEngine;

        // For simplicity, we'll use a dummy byte range for now
        // In a production system, you'd need to extract the actual PDF byte
range
        std::vector<uint8_t> pdfData;
        // Read PDF file into memory
        std::ifstream pdfFile(inputPdfPath, std::ios::binary);
        if (pdfFile) {
            pdfFile.seekg(0, std::ios::end);
            size_t size = pdfFile.tellg();
            pdfFile.seekg(0, std::ios::beg);
            pdfData.resize(size);
            pdfFile.read(reinterpret_cast<char*>(pdfData.data()), size);
            pdfFile.close();
        } else {
            std::cerr << "Error: Could not read PDF file: " << inputPdfPath <<
std::endl;
            return false;
        }

        // Calculate hash of the PDF data (excluding the signature placeholder)
        docHash = hashEngine.calculateSHA256(pdfData);

        // Create the CMS/PKCS#7 signature using the hash
        std::vector<uint8_t> signature = createCMSSignature(docHash);
        if (signature.empty()) {
            std::cerr << "Error: Failed to create CMS signature" << std::endl;
            return false;
        }

        // Set the signature content in the PDF
```

```
        signatureField.SetSignatureValue(reinterpret_cast<const
char*>(signature.data()), signature.size());

        // Save the signed PDF
        document.Write(outputPdfPath.c_str());

        std::cout << "PDF signed successfully: " << outputPdfPath << std::endl;
        return true;
    } catch (const PoDoFo::PdfError& error) {
        std::cerr << "PoDoFo Error: " << error.what() << std::endl;
        return false;
    } catch (const std::exception& ex) {
        std::cerr << "Error: " << ex.what() << std::endl;
        return false;
    }

    return false;
}

std::vector<uint8_t> PDFSigner::createCMSSignature(const std::vector<uint8_t>&
digest) {
    std::vector<uint8_t> signature;

    // Get the private key and certificate from the key manager
    EVP_PKEY* pkey = keyManager_.getPrivateKey();
    X509* cert = keyManager_.getCertificate();

    if (!pkey || !cert) {
        std::cerr << "Error: No private key or certificate available" <<
std::endl;
        return signature;
    }

    // Create a new BIO for the digest data
    BIO* dataBio = BIO_new_mem_buf(digest.data(), digest.size());
    if (!dataBio) {
        std::cerr << "Error: Could not create BIO for digest data" << std::endl;
        return signature;
    }

    // Create a CMS ContentInfo structure
    CMS_ContentInfo* cms = nullptr;

    // Create a certificate store containing our certificate
    STACK_OF(X509)* certs = sk_X509_new_null();
    if (!certs) {
        std::cerr << "Error: Could not create certificate stack" << std::endl;
        BIO_free(dataBio);
        return signature;
    }

    // Add our certificate
    if (!sk_X509_push(certs, cert)) {
        std::cerr << "Error: Could not add certificate to stack" << std::endl;
        sk_X509_free(certs);
        BIO_free(dataBio);
        return signature;
    }

    // Add certificate chain certificates if available
    const std::vector<X509*>& chain = keyManager_.getCertificateChain();
    for (X509* chainCert : chain) {
        if (!sk_X509_push(certs, chainCert)) {
            std::cerr << "Warning: Could not add chain certificate to stack" <<
std::endl;
        }
    }

    // Create CMS signature
```

```
    cms = CMS_sign(nullptr, nullptr, certs, dataBio, CMS_DETACHED | CMS_BINARY);

    // Free resources we don't need anymore
    sk_X509_free(certs);  // This doesn't free the actual certificates
    BIO_free(dataBio);

    if (!cms) {
        std::cerr << "Error: Could not create CMS signature" << std::endl;
        ERR_print_errors_fp(stderr);
        return signature;
    }

    // Convert CMS to DER format
    BIO* outBio = BIO_new(BIO_s_mem());
    if (!outBio) {
        std::cerr << "Error: Could not create output BIO" << std::endl;
        CMS_ContentInfo_free(cms);
        return signature;
    }

    if (!i2d_CMS_bio(outBio, cms)) {
        std::cerr << "Error: Could not write CMS to BIO" << std::endl;
        BIO_free(outBio);
        CMS_ContentInfo_free(cms);
        return signature;
    }

    // Get the data from the BIO
    BUF_MEM* bptr = nullptr;
    BIO_get_mem_ptr(outBio, &bptr);

    // Copy to our signature vector
    if (bptr && bptr->length > 0) {
        signature.resize(bptr->length);
        memcpy(signature.data(), bptr->data, bptr->length);
    }

    // Clean up
    BIO_free(outBio);
    CMS_ContentInfo_free(cms);

    return signature;
}

bool PDFSigner::prepareSignatureField(PoDoFo::PdfMemDocument& document,
PoDoFo::PdfSignatureField& field,
                                      int pageNumber, double x, double y, double
width, double height) {
    try {
        // Set signature information
        field.SetSignatureDate(PoDoFo::PdfDate());
        field.SetSignatureReason(signatureReason_.c_str());
        field.SetSignatureCreator(signatureName_.c_str());
        field.SetSignatureLocation(signatureLocation_.c_str());

        // Handle invisible signature
        if (pageNumber < 0) {
            // Create an invisible signature
            return true;
        }

        // Verify page number is valid
        if (pageNumber >= document.GetPageCount()) {
            std::cerr << "Error: Invalid page number: " << pageNumber <<
std::endl;
            return false;
        }

        // Get the page to place the signature on
```

```
        PoDoFo::PdfPage* page = document.GetPage(pageNumber);
        if (!page) {
            std::cerr << "Error: Could not get page " << pageNumber <<
std::endl;
            return false;
        }

        // Create a visible signature on the specified page
        double pageHeight = page->GetPageSize().GetHeight();
        double pageWidth = page->GetPageSize().GetWidth();

        // Validate coordinates
        if (x < 0 || x > pageWidth || y < 0 || y > pageHeight) {
            std::cerr << "Error: Signature position is outside page bounds" <<
std::endl;
            return false;
        }

        // Validate size
        if (width <= 0 || height <= 0) {
            std::cerr << "Error: Signature size must be positive" << std::endl;
            return false;
        }

        // Adjust signature to not exceed page boundaries
        if (x + width > pageWidth) {
            width = pageWidth - x;
        }

        if (y + height > pageHeight) {
            height = pageHeight - y;
        }

        // Create a rectangle for the signature field
        PoDoFo::PdfRect rect(x, pageHeight - y - height, width, height);
        field.SetRect(rect);
        field.SetPage(page);

        return true;
    } catch (const PoDoFo::PdfError& error) {
        std::cerr << "PoDoFo Error in prepareSignatureField: " << error.what()
<< std::endl;
        return false;
    } catch (const std::exception& ex) {
        std::cerr << "Error in prepareSignatureField: " << ex.what() <<
std::endl;
        return false;
    }

    return false;
}

//src/VerificationEngine.cpp
#include "VerificationEngine.h"
#include "HashEngine.h"
#include <openssl/cms.h>
#include <openssl/pem.h>
#include <openssl/x509.h>
#include <openssl/x509_vfy.h>
#include <openssl/err.h>
#include <podofo/podofo.h>
#include <iostream>
#include <fstream>
#include <sstream>

VerificationEngine::VerificationEngine() : trustedStore_(nullptr) {
    // Initialize OpenSSL
    static bool opensslInitialized = false;
    if (!opensslInitialized) {
```

```
        OpenSSL_add_all_algorithms();
        ERR_load_crypto_strings();
        opensslInitialized = true;
    }

    // Create the certificate store
    trustedStore_ = X509_STORE_new();
    if (!trustedStore_) {
        std::cerr << "Error: Could not create certificate store" << std::endl;
        ERR_print_errors_fp(stderr);
    }
}

VerificationEngine::~VerificationEngine() {
    if (trustedStore_) {
        X509_STORE_free(trustedStore_);
        trustedStore_ = nullptr;
    }
}

bool VerificationEngine::addTrustedCertificate(const std::string& certPath) {
    if (!trustedStore_) {
        std::cerr << "Error: Certificate store not initialized" << std::endl;
        return false;
    }

    // Open the certificate file
    FILE* fp = nullptr;
#ifdef _WIN32
    fopen_s(&fp, certPath.c_str(), "r");
#else
    fp = fopen(certPath.c_str(), "r");
#endif

    if (!fp) {
        std::cerr << "Error: Could not open certificate file: " << certPath <<
std::endl;
        return false;
    }

    // Read the certificate
    X509* cert = PEM_read_X509(fp, nullptr, nullptr, nullptr);
    fclose(fp);

    if (!cert) {
        std::cerr << "Error: Could not read certificate" << std::endl;
        ERR_print_errors_fp(stderr);
        return false;
    }

    // Add it to the store
    if (X509_STORE_add_cert(trustedStore_, cert) != 1) {
        std::cerr << "Error: Could not add certificate to store" << std::endl;
        ERR_print_errors_fp(stderr);
        X509_free(cert);
        return false;
    }

    // Certificate is now owned by the store
    X509_free(cert);
    return true;
}

bool VerificationEngine::addCertificateRevocationList(const std::string&
crlPath) {
    if (!trustedStore_) {
        std::cerr << "Error: Certificate store not initialized" << std::endl;
        return false;
    }
```

```
    // Open the CRL file
    FILE* fp = nullptr;
#ifdef _WIN32
    fopen_s(&fp, crlPath.c_str(), "r");
#else
    fp = fopen(crlPath.c_str(), "r");
#endif

    if (!fp) {
        std::cerr << "Error: Could not open CRL file: " << crlPath << std::endl;
        return false;
    }

    // Read the CRL
    X509_CRL* crl = PEM_read_X509_CRL(fp, nullptr, nullptr, nullptr);
    fclose(fp);

    if (!crl) {
        std::cerr << "Error: Could not read CRL" << std::endl;
        ERR_print_errors_fp(stderr);
        return false;
    }

    // Add it to the store
    if (X509_STORE_add_crl(trustedStore_, crl) != 1) {
        std::cerr << "Error: Could not add CRL to store" << std::endl;
        ERR_print_errors_fp(stderr);
        X509_CRL_free(crl);
        return false;
    }

    // CRL is now owned by the store
    X509_CRL_free(crl);
    return true;
}

VerificationEngine::VerificationStatus
VerificationEngine::verifyPDFSignature(const std::string& pdfPath) {
    lastVerificationInfo_.clear();
    std::stringstream infoStream;

    try {
        // Load the PDF document
        PoDoFo::PdfMemDocument document;
        document.Load(pdfPath.c_str());

        // Get the number of signatures in the document
        int signatureCount = 0;

        // In a real implementation, you would iterate through the document's
signature fields
        // For simplicity, we'll just assume there's at most one signature

        // Check if the document has signatures
        if (signatureCount == 0) {
            infoStream << "Error: Document does not contain any digital
signatures." << std::endl;
            lastVerificationInfo_ = infoStream.str();
            return VerificationStatus::ERROR;
        }

        // Extract the signature data
        // In a real implementation, you would get this from the document's
signature field
        std::vector<uint8_t> signatureData;

        // Extract the document hash
```

```
        // In a real implementation, you would calculate this based on the
ByteRange
        std::vector<uint8_t> documentHash;
        HashEngine hashEngine;

        // Read PDF file into memory for hash calculation
        std::vector<uint8_t> pdfData;
        std::ifstream pdfFile(pdfPath, std::ios::binary);
        if (pdfFile) {
            pdfFile.seekg(0, std::ios::end);
            size_t size = pdfFile.tellg();
            pdfFile.seekg(0, std::ios::beg);
            pdfData.resize(size);
            pdfFile.read(reinterpret_cast<char*>(pdfData.data()), size);
            pdfFile.close();
        } else {
            infoStream << "Error: Could not read PDF file: " << pdfPath <<
std::endl;
            lastVerificationInfo_ = infoStream.str();
            return VerificationStatus::ERROR;
        }

        // Calculate the document hash (in a real implementation, this would be
more complex)
        documentHash = hashEngine.calculateSHA256(pdfData);

        // Verify the signature
        if (!verifyCMSSignature(signatureData, documentHash)) {
            infoStream << "Error: Signature validation failed. The digital
signature is not valid." << std::endl;
            lastVerificationInfo_ = infoStream.str();
            return VerificationStatus::INVALID_SIGNATURE;
        }

        // Check if the document has been modified since it was signed
        // In a real implementation, this would involve checking the ByteRange
        bool documentModified = false;

        if (documentModified) {
            infoStream << "Error: Document has been modified after it was
signed." << std::endl;
            lastVerificationInfo_ = infoStream.str();
            return VerificationStatus::DOCUMENT_MODIFIED;
        }

        infoStream << "Signature verification successful. The document signature
is valid." << std::endl;
        lastVerificationInfo_ = infoStream.str();
        return VerificationStatus::VALID;

    } catch (const PoDoFo::PdfError& error) {
        infoStream << "PoDoFo Error: " << error.what() << std::endl;
        lastVerificationInfo_ = infoStream.str();
        return VerificationStatus::ERROR;
    } catch (const std::exception& ex) {
        infoStream << "Error: " << ex.what() << std::endl;
        lastVerificationInfo_ = infoStream.str();
        return VerificationStatus::ERROR;
    }

    return VerificationStatus::ERROR;
}

std::string VerificationEngine::getLastVerificationInfo() const {
    return lastVerificationInfo_;
}

bool VerificationEngine::verifyCMSSignature(const std::vector<uint8_t>&
signature,
```

```
                                        const std::vector<uint8_t>&
documentHash) {
    if (signature.empty() || documentHash.empty()) {
        std::cerr << "Error: Empty signature or document hash" << std::endl;
        return false;
    }

    // Create BIOs for signature and document data
    BIO* sigBio = BIO_new_mem_buf(signature.data(), signature.size());
    BIO* dataBio = BIO_new_mem_buf(documentHash.data(), documentHash.size());

    if (!sigBio || !dataBio) {
        std::cerr << "Error: Could not create BIOs" << std::endl;
        if (sigBio) BIO_free(sigBio);
        if (dataBio) BIO_free(dataBio);
        return false;
    }

    // Parse the CMS signature
    CMS_ContentInfo* cms = d2i_CMS_bio(sigBio, nullptr);
    if (!cms) {
        std::cerr << "Error: Could not parse CMS signature" << std::endl;
        ERR_print_errors_fp(stderr);
        BIO_free(sigBio);
        BIO_free(dataBio);
        return false;
    }

    // Verify the signature
    int flags = CMS_DETACHED | CMS_BINARY;
    bool result = (CMS_verify(cms, nullptr, trustedStore_, dataBio, nullptr,
flags) == 1);

    if (!result) {
        std::cerr << "Error: CMS signature verification failed" << std::endl;
        ERR_print_errors_fp(stderr);
    }

    // Clean up
    CMS_ContentInfo_free(cms);
    BIO_free(sigBio);
    BIO_free(dataBio);

    return result;
}
```