МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

БАТЬКІВСЬКА Катерина Володимирівна

Інтеграція Telegram-бота з Midjourney AI для обробки зображень із застосуванням криптографічних алгоритмів / Telegram bot integration with Midjourney AI for image processing using cryptographic algorithms

спеціальність: 125 – Кібербезпека та захист інформації

освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконала студентка групи
КБм -21
К. В. Батьківська

Науковий керівник
д. філософії, С. В. Кулина

Кваліфікаційну роботу
Допущено до захисту:

«____» _____ ___2025 р.

Завідувач кафедри

_____ **В.В.Яцків**

ТЕРНОПІЛЬ – 2025

Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки
Освітній ступінь «магістр»
спеціальність: 125 – Кібербезпека та захист інформації
освітньо-професійна програма – Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ В. В. Яцків
«____» _____ 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
БАТЬКІВСЬКА Катерина Володимирівна
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Інтеграція Telegram-бота з Midjourney AI для обробки зображень із застосуванням криптографічних алгоритмів / Telegram bot integration with Midjourney AI for image processing using cryptographic algorithms
керівник роботи д. філософії С. В. Кулина
затверджені наказом по університету від 20 грудня 2024 року № 938.
2. Строк подання студентом закінченої кваліфікаційної роботи 5 грудня 2025 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
– дослідити методи криптографічного захисту цифрових зображень та їх застосування в Telegram-ботах;
– проаналізувати алгоритми цифрового підпису та визначити їх ефективність для обробки графічних файлів;
– розробити алгоритм вбудованого підпису для захисту медіа-файлів без зміни вмісту зображення;
– інтегрувати Telegram-бот із Midjourney AI та забезпечити безпечний обмін даними через APIFrame;
– створити програмні модулі генерації та перевірки підпису, включаючи обробку SHA-256, payload та прив'язку до chatId і userId;
– реалізувати та протестувати повний цикл підписання й верифікації зображень, включно з завантаженням у Google Drive;

– провести експериментальний аналіз продуктивності криптографічних алгоритмів та визначити оптимальний підхід для практичного застосування.

5. Перелік графічного матеріалу у роботі:

− модуль створення підпису за алгоритмом RSA;
− алгоритм підписання зображення;
− алгоритм перевірки автентичності зображення;
− інтерфейс для роботи з модулями підпису;
− модуль підпису з використанням RSA;
− інтерфейс для роботи з вбудованим підписом RSA-EMBED;
− ініціалізація роботи користувача з Telegram-ботом;

6. Консультанти розділів кваліфікаційної роботи

	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 20 грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз сучасних технологій обробки зображень за допомогою штучного інтелекту	12.2024 р. – 03.2025 р.	
2	Використання криптографічних алгоритмів в обробці зображень	03.2025 р. – 06.2025 р.	
3	Проектування та реалізація інтеграції telegram-бота з Midjourney AI	06.2025 р. – 11.2025 р.	

Студентка _____ Батьківська К. В.
(підпис)

Керівник роботи _____ д. філософії, Кулина С. В.
(підпис)

АНОТАЦІЯ

Батьківська К. Т. Інтеграція Telegram-бота з Midjourney AI для обробки зображень із застосуванням криптографічних алгоритмів. – Рукопис.

Дослідження на здобуття освітнього ступеня «магістр» за спеціальністю 125 «Кібербезпека та захист інформації», освітньо-професійна програма «Кібербезпека». – Західноукраїнський національний університет, Тернопіль, 2025.

У роботі досліджено методи захисту цифрових зображень та розроблено алгоритм вбудованого цифрового підпису, що забезпечує автентичність і цілісність файлів. Запропонований підхід поєднує криптографічні механізми хешування та підпису для підвищення рівня довіри до оброблених зображень.

Ключові слова: ЦИФРОВИЙ ПІДПИС, ХЕШУВАННЯ, MIDJOURNEY, TELEGRAM-БОТ, АВТЕНТИФІКАЦІЯ.

ABSTRACT

Batkivska K. T. Integration of a Telegram Bot with Midjourney AI for Image Processing Using Cryptographic Algorithms. – Manuscript.

Research for obtaining the education level «Master» in specialty 125 «Cybersecurity and Information Protection», educational and professional program «Cybersecurity». – West Ukrainian National University, Ternopil, 2025.

The thesis investigates methods of protecting digital images and develops an embedded digital signature algorithm that ensures the authenticity and integrity of files. The proposed approach combines cryptographic hashing and signature mechanisms to enhance trustworthiness and security of processed images.

Keywords: DIGITAL SIGNATURE, HASHING, MIDJOURNEY, TELEGRAM BOT, AUTHENTICATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

PNG – Portable Network Graphics, формат зображень.

SHA-256 – Secure Hash Algorithm 256, криптографічна хеш-функція.

RSA – Rivest–Shamir–Adleman, алгоритм асиметричного цифрового підпису.

DSA – Digital Signature Algorithm, алгоритм цифрового підпису.

ECDSA – Elliptic Curve Digital Signature Algorithm.

RSA-EMBED – вбудований варіант RSA для підпису PNG-файлів.

HTTP – HyperText Transfer Protocol, протокол передачі даних, що використовується в комп'ютерних мережах.

JSON – JavaScript Object Notation, формат обміну даними.

URL – Uniform Resource Locator, унікальна адреса будь-якого ресурсу в Інтернеті.

TLS – Transport Layer Security, протокол захисту передавання даних.

Web API – Web Application Program Interface, вебінтерфейс для обробки webhook-подій.

ВСТУП

Актуальність роботи. Стрімкий розвиток технологій штучного інтелекту та месенджер-платформ сприяє появі інтегрованих систем, які здатні автоматично генерувати, обробляти та захищати цифровий візуальний контент. Однією з найбільш популярних моделей генерації зображень є Midjourney AI, що забезпечує високоякісну реконструкцію та художню стилізацію зображень на основі текстових промптів. У поєднанні з Telegram-ботами такі системи відкривають нові можливості для автоматизації, креативних сервісів, цифрового маркетингу та безпечного обміну графічними матеріалами.

Однак широке застосування генеративних моделей породжує і суттєві ризики, пов’язані з автентичністю, цілісністю та захистом зображень. Зокрема, збільшується кількість фальсифікацій, deepfake-матеріалів, спотворень метаданих, а також атак перехоплення даних під час передавання. Для забезпечення достовірності графічної інформації потрібні надійні криптографічні механізми, інтегровані безпосередньо в процес генерації та обробки зображень. Саме тому створення системи, яка поєднує генеративні можливості Midjourney з криптографічними алгоритмами підпису та верифікації, є актуальним завданням сучасної інформаційної безпеки.

Мета і завдання дослідження. Метою магістерської роботи є розроблення комплексної системи, яка інтегрує Telegram-бота з Midjourney AI для генерації та обробки зображень з одночасним застосуванням криптографічних методів забезпечення їх автентичності.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати сучасні технології генерації та обробки зображень на основі штучного інтелекту;

– дослідити існуючі підходи до інтеграції Telegram-ботів із системами AI;

– визначити можливості застосування методів хешування, електронного підпису та криптографічного вбудовування даних;

– розробити алгоритм цифрового підпису та вибрати оптимальні криптографічні механізми для забезпечення автентичності зображень;

– створити інтегровану архітектуру Telegram-бота з підтримкою генерації зображень через Midjourney;

– реалізувати програмні модулі підпису, верифікації й вбудовування криптографічних даних у файли зображень;

– провести експериментальне тестування роботи системи у середовищі, наближеному до реальних умов експлуатації.

Об’єкт дослідження – процеси забезпечення автентичності та цілісності цифрового контенту, що генерується моделями штучного інтелекту.

Предмет дослідження – методи та алгоритми генерації, обробки й криптографічного захисту зображень згенерованих AI-системами.

Методи дослідження. У роботі використано методи теоретичного аналізу, порівняльного аналізу криптографічних алгоритмів, алгоритмічного синтезу, експериментального моделювання, статистичного аналізу, а також методи побудови архітектур програмних систем.

Наукова новизна одержаних результатів. Наукова новизна полягає у створенні та дослідженні алгоритму RSA-EMBED – механізму цифрового підпису зображень з вбудовуванням криптографічного підпису безпосередньо у файл. На відміну від традиційних підходів, алгоритм забезпечує збереження підпису разом із зображенням, підвищуючи стійкість даних до модифікації та спрощуючи процедуру верифікації.

Практичне значення роботи. Розроблена інтегрована система Telegram-бота дозволяє автоматично генерувати зображення за допомогою Midjourney, підписувати їх криптографічним ключем, перевіряти автентичність, а також

зберігати результати у хмарному середовищі. Запропонований алгоритм RSA-EMBED забезпечує захист від підміни даних, атак перехоплення, модифікацій файлу та сприяє підвищенню достовірності цифрового контенту.

Результати дослідження. У роботі розроблено архітектуру інтегрованої системи, реалізовано Telegram-бота для взаємодії з Midjourney, створено криптографічний модуль підпису та верифікації зображень, впроваджено схему вбудовування цифрового підпису та виконано експериментальну перевірку працездатності алгоритму.

Публікації та апробація КР.

Батьківська, К., Дзівак, О., Кулина С. Порівняльний аналіз криптографічних методів захисту зображень у сервісах генерації контенту. Інноваційні підходи до розвитку технологій та економіки (IADTE 2025). – Свалява: ЗУНУ, 2025. – С. 209–211.

Батьківська, К., Кулина С. Методи виявлення підроблених або змінених зображень із застосуванням криптографічних хеш-функцій. Кібербезпека та комп’ютерно-інтегровані технології (КБКІТ-2025): Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів. – Тернопіль, 2025. – С. 118–120.

1. АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ОБРОБКИ ЗОБРАЖЕНЬ ЗА ДОПОМОГОЮ ШТУЧНОГО ІНТЕЛЕКТУ

У сучасну епоху цифровізації та масового поширення мультимедійних даних технології обробки зображень за допомогою штучного інтелекту (ШІ) стали ключовим інструментом у багатьох галузях - від мистецтва й дизайну до кібербезпеки та судової експертизи. Зростання популярності генеративних моделей, таких як Midjourney, DALL-E, Stable Diffusion, суттєво змінило підходи до створення, редагування й аналізу візуального контенту.

Водночас активне впровадження таких технологій породжує нові виклики у сфері безпеки, автентифікації даних і захисту від фальсифікацій. Саме тому для спеціалістів з кібербезпеки важливо розуміти принципи роботи сучасних AI-систем обробки зображень, можливості їх інтеграції з іншими платформами (наприклад, Telegram-ботами) та роль криптографічних механізмів у забезпеченні достовірності даних.

1.1. Аналіз існуючих рішень для інтеграції Telegram-ботів із системами AI

Для реалізації AI-функціональності в Telegram-ботах широко використовують .NET-платформу разом із офіційними бібліотеками для Bot API. Зокрема, бібліотека Telegram.Bot є найпопулярнішою .NET-клієнтською реалізацією Telegram Bot API. Наприклад, навчальні матеріали описують створення .NET Telegram-ботів, які звертаються до сервісів штучного інтелекту (наприклад, ChatGPT) через HTTP-запити (HyperText Transfer Protocol). Архітектура таких рішень зазвичай така- бот на .NET отримує повідомлення від користувача через Telegram Bot API (єдина точка взаємодії - HTTPS ендпоїнт *api.telegram.org/bot<token>/METHOD*), формує текстовий “промпт” та робить

запит до AI-сервісу. Запити до зовнішніх API (OpenAI, Midjourney тощо) виконуються через стандартні клієнти HTTP (наприклад, HttpClient або RestClient [6]), де в заголовках передається ключ авторизації.

Основна перевага .NET – зручність використання готових NuGet-пакетів та ASP.NET Core як сервера. Наприклад, Chris Hammond демонструє створення Telegram-бота на .NET 6, який підключається до Telegram API через Telegram.Bot і до ChatGPT через HttpClient. Аналогічно, інші розробники використовують Microsoft Bot Framework з ASP.NET Core для обробки вхідних повідомлень і виклику AI через API зовнішніх сервісів.

Для спілкування з Midjourney ми використаємо APIFRAME. APIFRAME - це проксі-сервіс, що надає REST-інтерфейси для популярних моделей генерації зображень, зокрема Midjourney. Розробник може зареєструватися в APIFRAME, отримати API-ключ і викликати готові ендпоїнти. У таблиці 1.1 представлено порівняння варіантів інтеграції з Midjourney.

Таблиця 1.1

Порівняння варіантів інтеграції з Midjourney

Критерій	APIFRAME (REST API)	Прямий доступ через Discord-бота	Інші сторонні API-шлюзи
Складність інтеграції	Низька - готові REST-ендпоїнти (/imagine)	Висока - потрібен власний Discord-бот, обробка WebSocket-подій	Середня - залежить від сервісу
Необхідність власної інфраструктури	Не потрібна	Потрібен Discord-бот та проксі-сервер	Частково потрібна
Доступ до Midjourney	Через стандартизований REST-інтерфейс	Через Discord API та зміни політик	Залежить від реалізації сервісу

Прод. табл. 1.1

Автентифікація	API-ключ (Bearer Token)	Discord Bot Token + авторизація через Discord	API-ключ або OAuth2
Швидкість отримання результатів	Висока	Залежить від Discord	Від середньої до високої
Надійність роботи	Висока	Середня - Discord змінює правила	Залежить від API
Гнучкість параметрів промптів	Висока	Висока, але складно реалізувати	Середня або висока
Безпека передачі даних	HTTPS	Стандартна безпека Discord API	HTTPS
Масштабованість під навантаження	Висока	Обмежена - rate limits	Середня
Простота інтеграції з .NET	Максимальна - JSON + HttpClient	Низька - WebSocket та Discord events	Середня

Наприклад, ендпоїнт /imagine приймає текстовий запит (prompt) і повертає згенероване зображення або посилання на нього. APIFRAME описує, що «звичайні сценарії використання включають генерацію зображень з текстових промптів» і надає деталізовані приклади API-викликів. Таким чином, патерн інтеграції полягає в наступному- Telegram-бот передає текст від користувача своєму серверу (.NET), сервер формує запит до APIFRAME (заголовки «Content-Type- application/json» і «Authorization- Bearer <API-ключ>»), отримує URL (Uniform Resource Locator) або файл зображення від Midjourney і надсилає результат назад у чат ботом. Усе передавання даних здійснюється через захищений HTTPS-канал.

З архітектурної точки зору, розділення логіки працює так:

– Telegram-бот (.NET) приймає повідомлення через Webhook або Long Polling, перевіряє користувацький запит, формує промпт.

– APIFRAME - посередник- отримує HTTP-запит з промптом, виконує запит до Midjourney (наприклад через Discord), обробляє та повертає готове зображення або ідентифікатор зображення.

– Midjourney генерує зображення за вказаним промптом.

– Сервер бота отримує результат і пересилає його користувачеві в Telegram-чат.

На практиці існують готові рішення та бібліотеки для подібних інтеграцій. Наприклад, вихідний код на GitHub демонструє Telegram-бот з підтримкою Midjourney (через Selenium та Discord) і ChatGPT. Інші розробники використовують власні реалізації на Python чи C#. Для .NET наявні офіційні SDK- OpenAI видає OpenAI .NET Client (за специфікацією OpenAPI) для зручності виклику своїх моделей. Бібліотека Telegram.Bot у комбінації з такими SDK дає повний стек- Telegram → .NET → AI API.

Особливості інтеграції з Midjourney через APIFRAME такі:

– Побудова промптів - згідно з документацією Midjourney, ефективні промпти – короткі та точні фрази з ясними описами. Наприклад, замість простого «кіт» можна сказати «фентезійний кіт-воїн у середньовічних обладунках». Варто уникати довгих списків речей у одному запиті – краще виокремлювати ключові параметри (предмет, стиль, освітлення, колір тощо).

– Обробка запитів - бот отримує повідомлення, визначає, чи це команда на створення зображення, екстрагує параметри (наприклад, "/imagine космос, зоряна ніч, деталі 8K"), потім формує JSON-запит (JavaScript Object Notation) і відправляє його на endpoint APIFRAME. Після отримання відповіді бот надсилає користувачу згенероване зображення.

– Зберігання зображень - збереження контенту може відбуватись локально (файлова система чи БД) або в хмарі (наприклад, Azure Blob, Amazon S3). Часто боти зберігають тільки URL зображення і деякі метадані (ID запиту,

час створення). Для Telegram може використовуватися метод sendPhoto з Bot API, який приймає file_id чи зовнішній URL. При цьому важливо зберігати й метадані для лімітів- Telegram накладає обмеження на розмір файлу та швидкість надсилання.

При інтеграції слід дотримуватись стандартних практик безпеки:

– HTTPS та сертифікати- всі запити до Telegram Bot API та до AI-сервісів здійснюються через HTTPS (Telegram Bot API вимагає HTTPS-адресу для Webhook). TLS (Transport Layer Security) забезпечує конфіденційність даних при передачі (налаштування HTTPS з перевіркою сертифіката на стороні бота).

– Уникати викладати ключі (bot token, API-ключі) у відкритому коді. Використовуйте захищені середовища виконання (.NET Secret Manager, змінні середовища) і HTTPS-з'єднання, щоб токени були в зашифрованому каналі.

– Обмежувати доступ тим, хто може користуватися ботом, перевіряючи user_id або роль користувача в чаті. Для публічних ботів слід дотримуватися загальних правил Telegram – наприклад, додавати в чорні списки, обмежувати кількість викликів команд.

– Рекомендовано надсилати не більше одного повідомлення в секунду у той самий чат, та не більше ~30 повідомлень/секунду у цілому [1] (агреґований ліміт). Перевищення викликає помилки 429 Too Many Requests. Зовнішні AI-сервіси теж можуть мати власні обмеження (наприклад, OpenAI лімітує кількість запитів). Реалізація повинна обробляти відповіді з кодом 429 та чекати перед повторною спробою.

– Необхідно логувати всі запити і відповіді як у бота, так і у сервісів AI. Це допомагає відстежувати зловживання, налагоджувати помилки і аналізувати сесії користувачів. Логи повинні бути захищені і доступні тільки адміністратору.

– Можна використовувати багатофакторну аутентифікацію для адміністративних команд бота, а також перевіряти вміст промптів на заборонені теми.

1.2 Основи роботи Midjourney AI

Midjourney - це потужна генеративна система на основі штучного інтелекту, призначена для створення зображень за текстовими описами (text-to-image) [4, 17]. Вона використовує архітектуру глибоких нейронних мереж, подібну до diffusion models, які поступово «очищають» випадковий шум, формуючи з нього осмислене зображення відповідно до заданого текстового промпту [4].

Принцип роботи Midjourney складається з кількох ключових етапів:

1. Обробка тексту - введений користувачем запит (prompt) перетворюється в числове представлення за допомогою моделей природної мови (Language Models).

2. Дифузійний процес - система генерує зображення шляхом ітераційного відновлення деталей із випадкового шуму, використовуючи векторні ознаки тексту.

3. Оптимізація та рендеринг - застосовуються додаткові фільтри, кольорокорекція й композиційні покращення для досягнення візуальної узгодженості.

Midjourney працює переважно через інтеграцію з платформою Discord, що дозволяє користувачам взаємодіяти із системою через команди типу /imagine [4]. Основною перевагою є висока художня якість зображень, різноманітність стилів і гнучкість у налаштуванні параметрів (розмір, співвідношення сторін, деталізація, стилізація тощо).

З технічної точки зору, Midjourney базується на дифузійних моделях, близьких за принципом до Stable Diffusion - відкритої системи, що поєднує латентний простір (latent space) із глибоким навчанням. Такий підхід дозволяє генерувати складні структури та текстури, зберігаючи при цьому ефективність і контроль якості.

Переважна більшість сучасних систем генерації зображень, включно з Midjourney (хоча точна його архітектура є комерційною таємницею), базується на концепції Дифузійних моделей (Diffusion Models - DMs) (рис. 1.1).

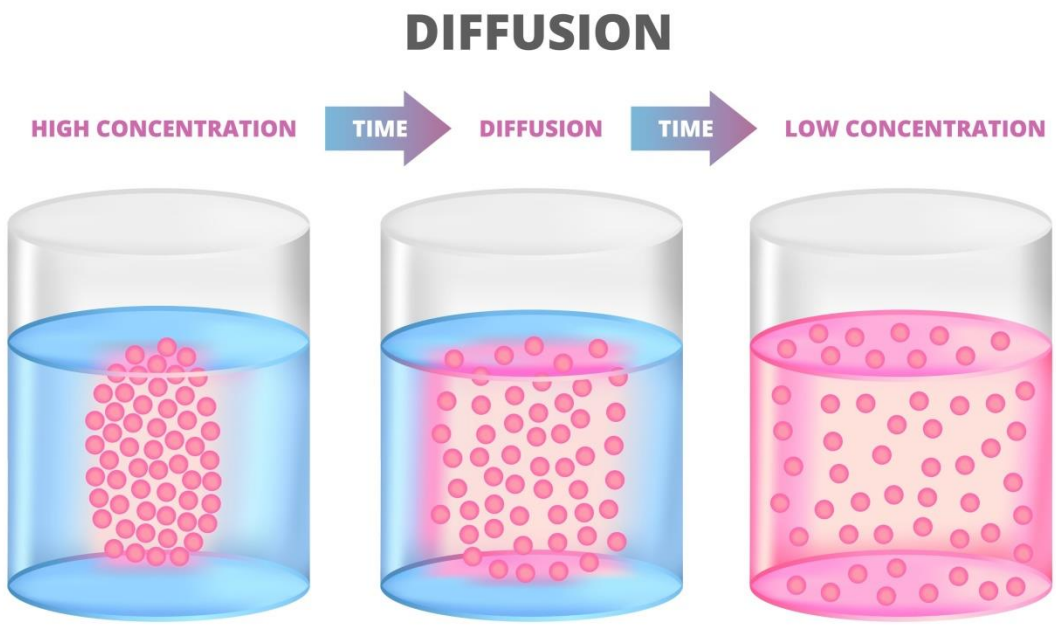


Рисунок 1.1 – Абстрактне графічне представлення поняття Diffusion

DMs являють собою ймовірнісні генеративні моделі, які працюють у два етапи:

1. Прямий процес - поступове додавання гауссового шуму до оригінального зображення, що призводить до його повного руйнування та перетворення на випадковий шум.

2. Зворотний процес - модель навчається відкатувати цей процес, тобто послідовно передбачати та видаляти шум, починаючи з чистого шуму, аж до повного відновлення чіткого зображення, що і є процесом генерації.

Генерація зображень на основі тексту досягається через механізм умовлення. Текстовий запит (prompt) обробляється за допомогою мовної моделі (наприклад, CLIP або T5), яка перетворює його на векторне представлення (ембединг). Цей ембединг інтегрується на різних етапах зворотного процесу дифузії, керуючи процесом видалення шуму таким чином, щоб результат відповідав семантиці вхідного тексту.

Midjourney демонструє високу чутливість до якості та структури вхідного тексту – явище, відоме як Prompt Engineering [4]. Ефективний prompt повинен не лише містити об'єкти, а й вказувати художній стиль, освітлення, композицію та технічні характеристики (наприклад, співвідношення сторін). З точки зору кібербезпеки, це створює як ризики, так і можливості:

– Ризики - використання prompt для генерації зображень, що порушують законодавство, авторські права, або створюють deepfakes.

– Можливості - моделі можуть бути навчені розпізнавати "токсичні" або заборонені запити, забезпечуючи внутрішній механізм фільтрації контенту на рівні вхідних даних.

Головним викликом, породженим розвитком таких систем, є проблема аутентичності цифрового контенту [18]. Надзвичайна якість генерації ускладнює візуальну відмінність між справжніми та синтетичними зображеннями:

– Deepfakes - створення переконливих, але фальшивих візуальних доказів, що є загрозою для репутації та правових процесів.

– Захист авторських прав - необхідність розробки методів для ідентифікації походження згенерованого зображення (чи воно створене AI, і

якщо так, то якою системою), що безпосередньо призводить до необхідності вбудовування цифрових водяних знаків та криптографічних міток.

1.3. Застосування криптографії в обробці зображень

Для забезпечення цілісності та автентичності цифрових зображень застосовують методи хешування та цифрового підпису. У середовищі .NET зазвичай використовують простір імен System.Security.Cryptography (SHA-256 - Secure Hash Algorithm 256, RSA – Rivest–Shamir–Adleman, ECDSA – Elliptic Curve Digital Signature Algorithm, тощо). Також популярна бібліотека BouncyCastle, що містить реалізації широкого спектру алгоритмів (SHA, RSA, ECDSA) для C# [9]. Хеш-функція (наприклад, SHA-256) створює унікальну «відбиток» (дайджест) файлу зображення - при будь-якій зміні даних хеш зміниться. Цифровий підпис використовує приватний ключ автора - спочатку формується хеш даних, потім він шифрується цим ключем. Отримана підпис перевіряється відкритим ключем - так гарантовано встановлюється, що зображення саме від цього автора і що його вміст не змінювався.

Хешування застосовується для перевірки цілісності файлу. Наприклад, у .NET можна прочитати байти файлу та порахувати SHA-256-хеш. Використовуючи клас SHA256 з System.Security.Cryptography, обчислюємо хеш зображення, як наведено на рисунку 1.2.

```
using (SHA256 sha = SHA256.Create())
{
    byte[] data = File.ReadAllBytes("image.jpg");
    byte[] hash = sha.ComputeHash(data);
    string hashString = BitConverter.ToString(hash).Replace("-", "");
    Console.WriteLine(hashString);
}
```

Рисунок 1.2 – Обчислення хешу зображення

Подібний підхід показано у прикладі Microsoft - mySHA256.ComputeHash(fileStream) для файлу [2]. Отриманий SHA-256-хеш (довжиною 32 байти) може зберігатися разом із зображенням для подальшої перевірки.

Хеш у безпековому контексті служить «відбитком» вмісту зображення. Порівняння збереженого хешу з щойно обчисленим дозволяє виявити будь-яку зміну даних [2]. Таким чином, якщо хеш зображення відрізняється від очікуваного – це свідчить про спотворення чи підміну файлу.

Цифровий підпис прив'язує зображення до конкретного автора. В .NET можна використовувати RSA або ECDSA для підпису [7]. Послідовність така - спочатку обчислюємо хеш зображення, потім підписуємо цей хеш приватним ключем автора. Наприклад, з RSA, створюємо ключі та підписуємо SHA-256-хеш (рис. 1.3).

```
using (RSA rsa = RSA.Create(2048))
{
    byte[] data = File.ReadAllBytes("image.jpg");
    byte[] signature = rsa.SignData(data, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    // signature містить цифровий підпис
}
```

Рисунок 1.3 – Створення підпису за допомогою алгоритму RSA

У цьому коді *rsa.SignData* сам порахує SHA-256-хеш і зашифрує його приватним ключем RSA. Аналогічно можна використати *RSACryptoServiceProvider.SignData(...)* з переданим об'єктом SHA256 (як у документації). Для ECDSA замість RSA використовують *ECDsa.Create()* і метод SignData (*HashAlgorithmName.SHA256*) - підпис формується за допомогою еліптичної кривої.

Підпис гарантує, що лише власник приватного ключа міг його згенерувати. При перевірці ми розшифровуємо підпис відкритим ключем і

порівнюємо отриманий хеш зі свіжим хешем зображення. Якщо вони співпадають, зображення не було змінено і дійсно походить від того, хто підписав. Таким чином цифровий підпис забезпечує і цілісність, і автентичність даних. Перевірка здійснюється відкритим ключем. Наприклад, після отримання зображення і підпису використовуємо VerifyData з RSA для перевірки, як показано на рисунку 1.4.

```
using (RSA rsa = RSA.Create())
{
    rsa.ImportParameters(publicKeyParams); // імпортуємо RSA-параметри (модуль та експоненту)
    byte[] data = File.ReadAllBytes("image.jpg");
    bool isValid = rsa.VerifyData(data, signature, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}
```

Рисунок 1.4 – Перевірка валідності підпису

Тут метод VerifyData на основі того ж SHA-256 і відкритого ключа перевіряє підпис. Для ECDSA використовується аналогічний метод *ecdsa.VerifyData(data, signature, HashAlgorithmName.SHA256)*. Якщо перевірка проходить успішно, підтверджується справжність зображення та авторство.

Криптографічні дані часто зберігають у метаданих зображень (EXIF, XMP, IPTC). Наприклад, у поле UserComment чи інші власні теги можна записати хеш або підпис зображення. У .NET це робиться через Image.PropertyItems, щоб записати рядок у метадані (рис. 1.5).

```
var img = Image.FromFile("image.jpg");
var item = img.PropertyItems[0];        // беремо шаблон PropertyItem
item.Id = 0x9286;                       // 0x9286 – тег UserComment
item.Type = 2;                          // тип ASCII
item.Value = Encoding.UTF8.GetBytes("Автор: Example");
item.Len = item.Value.Length;
img.SetPropertyItem(item);
img.Save("image_with_meta.jpg");
```

Рисунок 1.5 – Вбудування користувацької інформації в метадані

Фрагмент коду на рисунку 1.5 додає в зображення EXIF-тег із закоментованим текстом, що демонструє загальний підхід. Аналогічно можна зберегти в метадані хеш чи цифровий підпис у вигляді тексту, щоб перевірити ці дані, програмно читають *PropertyItem* (рис. 1.6).

```
var img = Image.FromFile("image_with_meta.jpg");
var commentItem = img.GetPropertyItem(0x9286);
string comment = Encoding.UTF8.GetString(commentItem.Value);
```

Рисунок 1.6 – Перевірка метаданих зображення

Після цього можна порівнювати збережені метадані з обчисленими значеннями (наприклад, хешем) - так забезпечується додаткова перевірка цілісності та нерозривного зв'язку даних з автором. Зовнішні бібліотеки (наприклад, MetadataExtractor чи GroupDocs.METADATA for .NET) також спрощують читання/запис метаданих зображень [10].

Для приховання інформації в зображеннях застосовують стеганографію та водяні знаки. На відміну від підпису, стеганографія приховує сам факт передачі повідомлення. Наприклад, OpenStego - відкрите Java-застосування для вбудовування секретних даних або водяних знаків у зображення [11]. У .NET існують бібліотеки, що реалізують LSB-методи. Наприклад, бібліотека Stegodon дозволяє легко зашифрувати рядок у пікселях.

```
string imgPath = "cover.png";
string secret = "my hidden message";
Bitmap steg = Stegodon.Stegodon.Encrypt(new Bitmap(imgPath), secret);
steg.Save("stego_image.png");
```

Рисунок 1.7 – Вбудування бінарних даних у зображення

Як показано на рисунку 1.7, код вбудовує бінарні дані рядка у найменш значущі біти пікселів зображення. Припустимо, у тексті "my hidden message" закладено пароль чи хеш; після передавання отримувач може викликати Stegodon.Decrypt щоб витягти дані. Таким чином такий спосіб схованого обміну може служити додатковим каналом збереження секретної інформації чи маркером авторства.

При передачі зображень по мережі використовується шифрування транспортного рівня. У стандартному веб-HTTPS-з'єднанні застосовується протокол TLS [14], який встановлює конфіденційний канал між клієнтом і сервером. TLS використовує обмін ключами, симетричне шифрування даних і контроль цілісності (MAC). Наприклад, HTTPS забезпечує, що дані зображення передаються зашифрованими, і ніхто не може їх підмінити чи прочитати "на льоту".

У месенджері Telegram для захисту передачі застосовується протокол MTProto. У звичайних ("cloud") чатах зображення шифруються симетричним алгоритмом AES-256 з ключами, похідними від SHA-256-гешу повідомлення. Крім того, MTProto підтримує PFS (Perfect Forward Secrecy) - це означає, що ключі оновлюються для кожного повідомлення, що захищає історію листування навіть при компрометації ключа [13, 19]. Для секретних чатів Telegram взагалі використовується повне e2e-шифрування (ключі є тільки в учасників). Таким чином при пересиланні зображень через HTTPS чи MTProto їх вміст зашифрований на транспортному рівні і недоступний стороннім.

2. ВИКОРИСТАННЯ КРИПТОГРАФІЧНИХ АЛГОРИТМІВ В ОБРОБЦІ ЗОБРАЖЕНЬ

2.1. Огляд криптографічних алгоритмів для захисту даних

Криптографічний захист даних забезпечує конфіденційність, цілісність та доступність інформації (так звана тріада ЦРУ). Конфіденційність означає, що дані доступні лише авторизованим користувачам, цілісність означає, що вони не були змінені без відома автора, а автентичність джерела (автентифікація) гарантується електронними підписами. Публічне відображення конфіденційності та цілісності досягається за допомогою шифрування та хешування- симетричні та асиметричні алгоритми шифрування захищають інформацію від несанкціонованого доступу, а криптохеші захищають інформацію від змін. Наприклад, під час шифрування дані перетворюються на зашифрований текст, щоб лише власник секретного ключа міг відновити оригінальне повідомлення. Функції криптохешування (MD5, SHA-1/2/3) створюють фіксовані дайджести, які змінюються навіть при незначних змінах у вхідному файлі. Загалом, сучасні системи безпеки поєднують кілька алгоритмів та схем (шифрування даних, обмін ключами, цифрові підписи), щоб гарантувати цілісність, автентифікацію та конфіденційність обміну.

Симетричне шифрування використовує один і той самий секретний ключ як для шифрування, так і для дешифрування даних (рис. 2.1). Ці алгоритми (наприклад, DES, AES, Blowfish) здебільшого дуже швидкі та ефективні при обробці великих обсягів інформації. Зокрема, симетричне шифрування характеризується високою продуктивністю (передача сотень мегабайт за секунду на сучасному обладнанні).

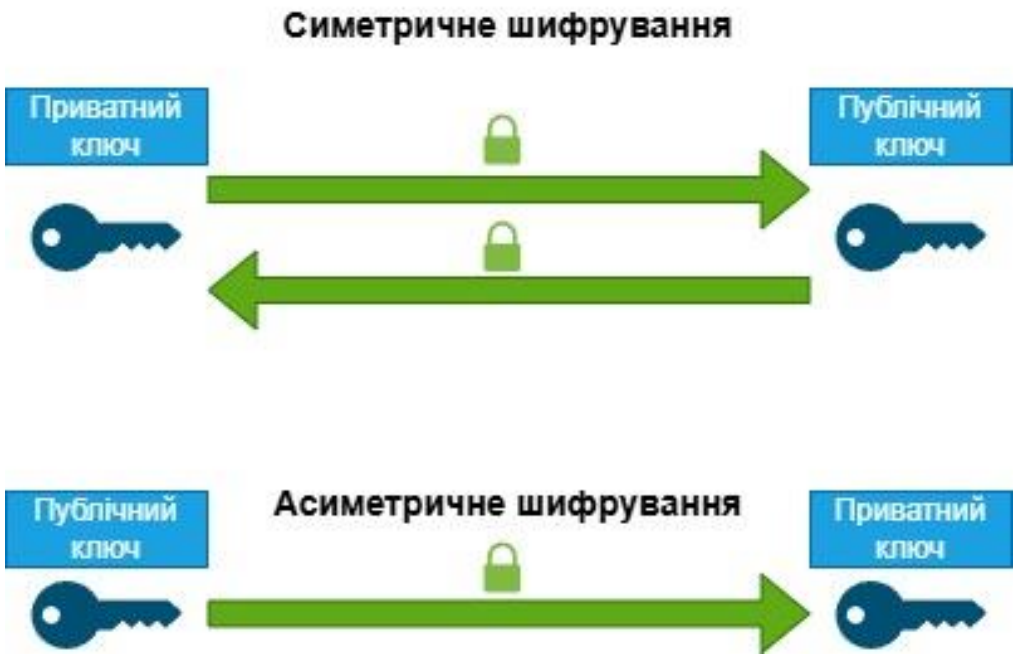


Рисунок 2.1 – Схема симетричного і асиметричного шифрування

Однак серед недоліків є необхідність безпечного обміну секретним ключем між учасниками комунікації - якщо ключ перехоплюється, комунікація стає вразливою.

Наприклад, стандарт DES (Data Encryption Standard) вперше був затверджений для захисту телекомунікацій у 1970-х роках. Він використовує 56-бітний ключ та блокову структуру. Однак через короткий ключ DES був «зламаний» методом грубої сили (у 1999 році група дослідників зламала зашифроване повідомлення приблизно за 22 години). Сьогодні DES вважається небезпечним і практично не використовується.

Натомість, AES (Advanced Encryption Standard) став «золотим стандартом» для блокових шифрів, схвалений NIST у 2001 році, він підтримує ключі довжиною 128, 192 або 256 бітів. AES розроблений як криптографічно стабільна схема зі змінною довжиною ключа. На практиці AES вважається дуже безпечним та ефективним, тому його широко використовують для шифрування даних під час зберігання та передачі інформації. Таблиця продуктивності

показує, що AES-128 у режимі CBC може досягати кількох сотень мегабайт за секунду (наприклад, ~656 000 КБ/с при шифруванні 16-байтових блоків), тоді як AES-256 дещо повільніший (~249 000 КБ/с) через більший ключ. Зі збільшенням розміру блоку або даних продуктивність зростає для всіх варіантів ключів.

Асиметричні (публічні) алгоритми використовують пару ключів-відкритий ключ (для шифрування або перевірки підпису) та закритий ключ (для дешифрування або створення підпису) (рис. 2.1). Найвідомішим асиметричним алгоритмом є RSA (Rivest-Shamir-Adelman). У RSA закритий ключ складається з двох великих простих чисел, а відкритий ключ є їх добутком. Безпека RSA базується на складності розкладання великого числа на множники; при достатньо довгому ключі (2048+ бітів) розкладання є невиправдано дорогим.

Ще одним популярним асиметричним алгоритмом є ElGamal (1985). ElGamal базується на протоколі Діффі-Хеллмана - його безпека пов'язана з дискретним логарифмом у групі [20]. Система ElGamal використовується в PGP/GPG та подібних криптосистемах для шифрування даних та цифрових підписів. На відміну від RSA, ElGamal трохи повільніша через додаткове піднесення до степеня та використовується рідше, але добре підходить для схем на основі Діффі-Хеллмана.

Криптографічні хеш-функції перетворюють довільні вхідні дані на дайджест фіксованого розміру [21]. Наприклад, MD5 видає 128-бітний код, SHA-1 видає 160-бітний код, SHA-256 видає 256-бітний код тощо. У будь-якому випадку, результат має фіксовану довжину незалежно від довжини вхідних даних. Наведений нижче малюнок ілюструє цю властивість - навіть незначна зміна вхідного повідомлення видає принципово інший хеш. Таке «кластеризація вхідних даних у фіксований дайджест» робить хеш-функції незворотними та чутливими до будь-якої модифікації даних [22].

Серйозною слабкістю MD5 та SHA-1 є можливість побудови колізій. Наприклад, MD5 давно визнано криптографічно зламаним [32] - існують практичні атаки, які знаходять колізії за незначний проміжок часу, тому MD5 більше не рекомендується для криптографічних цілей [23]. Аналогічно, SHA-1 фактично був зламаний (перша публічна колізія була продемонстрована у 2017 році [24]). Багато організацій, включаючи NIST, рекомендують перейти на SHA-2 або SHA-3 з 2011 року [25]. У свою чергу, SHA-2 (SHA-256, SHA-384, SHA-512 та їх варіанти) наразі вважаються безпечними - у повній версії SHA-2 за весь час практичного використання не було виявлено жодної катастрофічної вразливості [26]. У 2015 році було схвалено SHA-3 (Keccak) - схему іншого дизайну [27], розроблену як резервний варіант та для розширення криптографічної товщини. Важливо, що SHA-3 не замінює SHA-2 (оскільки радикальних атак на SHA-2 не виявлено [33]), вона може бути використана за необхідності та підвищує загальну надійність набору алгоритмів [34].

Серед симетричних алгоритмів з великими ключами AES вважається високобезпечним – наразі для його повного зламу залишаються лише атаки методом грубої сили [28]. Blowfish також стабільний, але 64-бітний розмір блоку обмежує його масштабованість (є атаки на дуже великі обсяги). DES та 3DES більше не доступні- 3DES, хоча й використовує подвійне (або потрійне) шифрування, характеризується низькою ефективністю та поступово замінюється новими алгоритмами (як показано в літературі, 3DES є "застарілим"). У таблиці 2.1 представлено порівняння симетричних алгоритмів шифрування.

Серед асиметричних схем RSA є безпечною за умови, що ключ достатньо великий (поточний мінімум ~2048 бітів). Її вразливість полягає в розв'язності задачі факторизації великого числа, яка спирається на експоненціальне зростання складності. Ель-Гамаль має подібний рівень стабільності, заснований на дискретному логарифмі.

Таблиця 2.1

Порівняння симетричних алгоритмів шифрування

Алгоритм	Рівень безпеки	Швидкодія	Стійкість до атак	Особливості / недоліки
AES	Дуже високий	Дуже висока	Немає практичних атак	Оптимальний для великих даних
Blowfish	Високий	Висока	Складні атаки на великих об'ємах	64-бітний блок обмежує застосування
DES	Низький	Середня	Повністю зламаний	Застарілий
3DES	Середній	Дуже повільний	Meet-in-the-middle атаки	Застарілий, замінений AES

Криптографія на основі еліптичних кривих (ECC) забезпечує високий рівень безпеки з меншими ключами- наприклад, 256-бітний ключ забезпечує безпеку, еквівалентну 3072-бітному RSA. Водночас, криптографічні алгоритми не враховують зловмисників з квантовим комп'ютером- алгоритми, засновані на дискретному логарифмі, теоретично можуть бути зламані алгоритмом Шора. У таблиці 2.2 представлено порівняння симетричних алгоритмів шифрування.

Таблиця 2.2

Порівняння асиметричних алгоритмів шифрування

Алгоритм	Рівень безпеки	Довжина ключа	Стійкість до атак	Недоліки/ особливості	Швидкість
RSA	Високий	2048-4096 біт	Факторизація великих чисел	Дуже повільний для великих даних	Низька
ElGamal	Високий	2048+ біт	Дискретний логарифм	Великі шифротексти	Низька
ECC	Дуже високий	224-521 біт	Еліптичний дискретний логарифм	Можливі ризики в постквантовій криптографії	Вища, ніж RSA

На практиці цілісність зображення перевіряється шляхом порівняння хеш-значень. Після створення або отримання зображення воно хешується один раз, а отримане значення зберігається (зазвичай у надійному сховищі або цифровому підписі). Пізніше, під час перевірки, обчислюється хеш поточного файлу та порівнюється з оригіналом. Якщо зображення не було змінено, обидва хеші будуть точно збігатися; будь-яке зовнішнє втручання, навіть на рівні одного пікселя, призведе до різниці між хешами. Такий підхід використовується, зокрема, в цифровій криміналістиці та системах зберігання контенту. Документовано, що експерти з цифрових доказів присвоюють кожному файлу унікальний «відбиток пальця» (хеш) і можуть виявити найменші зміни в будь-який час, порівнюючи оригінальний та поточний хеші. Це забезпечує гарантовану перевірку цілісності візуального контенту під час передачі даних або архівування.

MD5 (Message Digest 5) - один із найстаріших алгоритмів хешування. Він обробляє довільні вхідні дані та створює 128-бітне (32-символьне шістнадцяткове) хеш-значення. Перевагою MD5 є дуже висока обчислювальна швидкість, що робить його зручним для швидкої перевірки великих файлів. У 1990-х роках MD5 широко використовувався для перевірки цілісності даних (включаючи зображення). Його головна перевага - швидкість і простота - стала недостатньою на тлі відсутності криптографічної стійкості [35].

SHA-1 (Secure Hash Algorithm 1) формує 160-бітний хеш. На момент розробки він забезпечував більшу стійкість до колізій, ніж MD5, і довгий час використовувався в багатьох протоколах захисту даних. SHA-1 зараз вважається застарілим для цілей цілісності, і рекомендуються новіші алгоритми.

Сімейство SHA-2 було представлено NSA та опубліковано як стандарт у 2001 році. Воно включає низку функцій з різною довжиною вихідного хешу (SHA-224, SHA-256, SHA-384, SHA-512 тощо). Найпоширенішими є SHA-256 (256 бітів) та SHA-512 (512 бітів). Велика довжина хешу забезпечує високу

криптографічну стійкість- обчислювальна складність атаки типу "день народження" робить практично неможливим пошук двох різних зображень з однаковим хешем (SHA-256 вимагає ~2^128 операцій.

SHA-3 (також відомий як Keccak) - це найновіше сімейство стандартів (схвалене NIST у 2015 році). На відміну від SHA-2, SHA-3 використовує губчасту конструкцію та має іншу внутрішню структуру. Він підтримує ті ж розміри хешів (224, 256, 384, 512 бітів) і навіть гнучкий режим випадкового виводу (SHAKE), але вважається більш стійким до потенційних атак на конструкцію SHA-2. SHA-3 зазвичай працює повільніше, ніж SHA-2 на традиційних процесорах, але його архітектура добре підходить для апаратних реалізацій.

Далі розглянемо принцип створення цифрового підпису файлу та концепції захисту зображення за допомогою цифрового підпису. Наведена нижче схема показує стандартну процедуру створення та перевірки цифрового підпису (рис. 2.2).

Рисунок 2.2 – Схема формування і перевірки цифрового підпису

Спочатку обчислюється дайджест за допомогою криптографічної хеш-функції (наприклад, SHA-256) на основі вхідного файлу (наприклад, зображення). Потім цей дайджест шифрується закритим ключем автора (операція «Підпис»), в результаті чого створюється цифровий підпис, який передається разом із файлом або зберігається поруч із ним. Після перевірки одержувач розшифровує підпис за допомогою відкритого ключа автора та порівнює отриманий хеш із хешем, розрахованим самим автором на тому ж файлі; збіг підтверджує, що зображення не було змінено, а автентичність автора встановлена. Такий підхід (підписання дайджесту замість всього файлу) значно пришвидшує операції та зменшує обчислювальне навантаження під час роботи з великими графічними даними.

Шифрування зображень захищає контент від перехоплення або підробки. Цей підхід особливо корисний, коли зображення передаються мережею або зберігаються в хмарі. У цьому розділі обговорюються технічні принципи шифрування зображень, аналізується вплив форматів зображень на складність шифрування та окреслюються ключові переваги та обмеження криптографічних методів у цьому контексті.

Головною перевагою шифрування зображень є забезпечення конфіденційності під час передачі або зберігання. Воно гарантує, що навіть у разі порушення безпеки каналу або сховища контент залишається недоступним. Воно також забезпечує сумісність з принципами автентичності та цілісності у поєднанні з цифровими підписами або хеш-функціями. До недоліків належить висока обчислювальна складність для великих зображень, особливо при використанні бітового або форматно-зберігального шифрування. Крім того, деякі методи можуть впливати на розмір файлу, ускладнюючи його транспортування або зберігання в обмеженому середовищі.

У наступному прикладі (рис. 2.3) демонструється шифрування зображення PNG (Portable Network Graphics) за допомогою AES у режимі CBC.

```
byte[] imageData = File.ReadAllBytes("input.png");
using (Aes aes = Aes.Create())
{
    aes.Key = Convert.FromBase64String("AAECAwQFBgcICQoLDA0ODxAREhMUFRYXGBkaGxwdHh8=");
    aes.IV = Convert.FromBase64String("EBESExQVFhcYGRobHB0eHw==");
    aes.Mode = CipherMode.CBC;
    aes.Padding = PaddingMode.PKCS7;
    using (ICryptoTransform encryptor = aes.CreateEncryptor())
    {
        byte[] cipherData = encryptor.TransformFinalBlock(imageData, 0, imageData.Length);
        File.WriteAllBytes("output.enc", cipherData);
    }
}
```

Рисунок 2.3 – Шифрування зображення PNG за допомогою AES у режимі CBC

Після зчитування байтів файлу застосовується симетричне шифрування з фіксованим ключем та вектором ініціалізації. Це дозволяє повністю приховати вміст, навіть якщо файл зберігається у відкритому доступі.

Процес дешифрування відбувається у зворотному порядку (рис. 2.4)- той самий ключ та вектор застосовуються до зашифрованого потоку байтів, і результатом є повністю відновлений файл зображення.

```
byte[] cipherData = File.ReadAllBytes("output.enc");
using (Aes aes = Aes.Create())
{
    aes.Key = Convert.FromBase64String("AAECAwQFBgcICQoLDA0ODxAREhMUFRYXGBkaGxwdHh8=");
    aes.IV = Convert.FromBase64String("EBESExQVFhcYGRobHB0eHw==");
    aes.Mode = CipherMode.CBC;
    aes.Padding = PaddingMode.PKCS7;
    using (ICryptoTransform decryptor = aes.CreateDecryptor())
    {
        byte[] decryptedData = decryptor.TransformFinalBlock(cipherData, 0, cipherData.Length);
        File.WriteAllBytes("recovered.png", decryptedData);
    }
}
```

Рисунок 2.4 – Дешифрування зображення PNG за допомогою AES у режимі CBC

Отже, шифрування є невід'ємною частиною захисту зображень у цифровому середовищі. Його ефективність залежить не лише від обраного алгоритму, але й від характеристик графічного формату, режиму роботи та правильної реалізації. Приклади на платформі .NET демонструють, як шифрування може бути інтегровано в сучасні прикладні рішення, забезпечуючи конфіденційність візуального контенту в реальних системах.

2.2. Розробка алгоритму цифрового підпису забезпечення автентичності зображень

Для забезпечення автентичності зображень запропоновано власний алгоритм «RSA-EMBED», що ґрунтується на цифровому підписі RSA з вбудовуванням підпису безпосередньо у файл зображення.

RSA-EMBED поєднує добре відомі переваги RSA-підпису (сила перевірки, широка сумісність) із можливістю зберегти підпис разом з даними. На відміну від ECDSA/DSA, де реалізації можуть бути менш стандартизовані або вимагати більшої кваліфікації, RSA-EMBED забезпечує простий і надійний механізм в одній програмній реалізації. Крім того, численні виміри показують, що для режиму верифікації RSA зазвичай є найшвидшим серед популярних алгоритмів підпису

Процес RSA-EMBED розбито на дві основні фази- генерація підпису та перевірка підпису. Обидві фази включають взаємопов’язані логічні блоки: генерацію підпису та його перевірку.

1. **Алгоритм генерація підпису** представлено на рисунку 2.5.

Рисунок 2.5 – Алгоритм підписання зображення

Запропонований алгоритм підписання зображення полягає у тестуванні автентичності перед відправкою та містить наступні блоки:

- Читання зображення - зчитується файл зображення у вигляді байтового масиву (наприклад, PNG, JPEG).
- Хешування - до байтового вмісту застосовується криптографічна хеш-функція SHA-256. SHA-256 формує 256-бітний дайджест від даних.
- Створення RSA-підпису - обчислений хеш зашифровується приватним RSA-ключем, утворюючи підпис. У .NET це виконується, наприклад, методом

rsa.SignData(data, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1), який сам обчислює SHA-256 від data і підписує його.

- Вбудовування підпису - отримані байти підпису додаються до файлу зображення. Технічно це може бути додавання наприкінці файла або у спеціальному службі (метаданих) формату. У нашому алгоритмі резервується фіксована область в кінці файлу (наприклад, останні 276 байт для RSA-2048) під службові маркери та підпис.
- Збереження файлу - після вбудовування підпису сформований файл зберігається. Тепер один файл містить і само зображення, і цифровий підпис.

2. На рисунку 2.6 представлено алгоритм перевірки підпису, що полягає у автентифікації зображення.

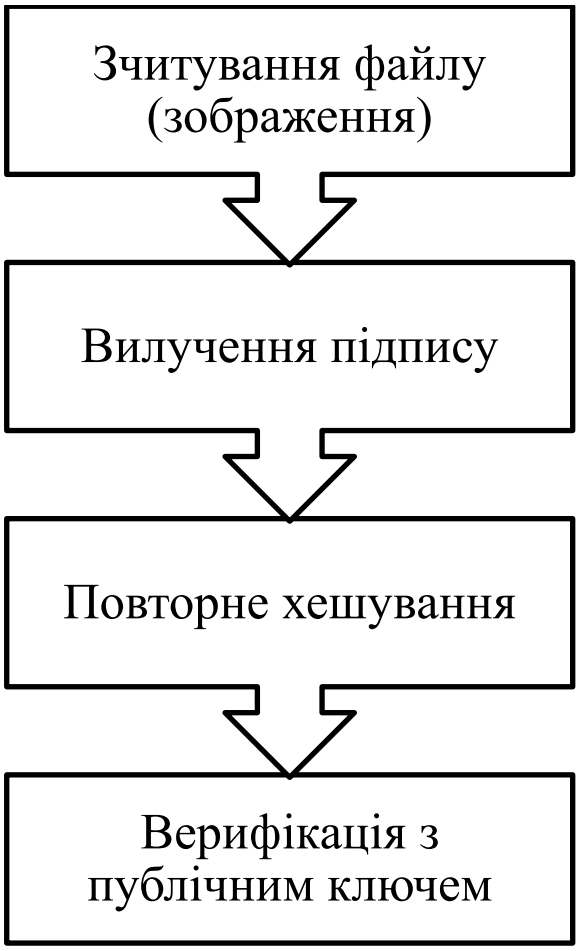


Рисунок 2.6 – Алгоритм перевірки підпису зображення

Запропонований алгоритм перевірки підпису зображення містить наступні блоки:

- Зчитування файлу - відкривається файл із вбудованим підписом.
- Вилучення підпису - з файлу виділяються початкові байти зображення та окремо – вбудований блок підпису (за відомою довжиною або маркерами).
- Повторне хешування - до витягнених байтів зображення застосовується та сама хеш-функція SHA-256 і отримується новий дайджест.
- Верифікація з публічним ключем - публічним RSA-ключем розшифровують вбудований підпис, порівнюючи одержаний хеш з обчисленим. Якщо вони збігаються, підпис вважається дійсним, і зображення не було змінено. Якщо ні, то виявляється спотворення даних або невірний ключ.

Суть процедури полягає в тому, що підпис створюється на геш-функції від даних, а не від файлу напряму. Це звичайна схема електронного підпису, коли дані хешуються і підписуються закритим ключем, а при перевірці відкритим ключем дешифрують підпис, і порівнюють з обчисленим хешем. Якщо будь-яка частина даних змінилася, результати не співпадуть і перевірка не пройде. Такий підхід гарантує цілісність, оскільки навіть бітова зміна у файлі призведе до помилки верифікації.

У реалізації RSA-EMBED на C# використовуються стандартні класи з System.Security.Cryptography.

Для зберігання ключів відповідно до конкретного користувача ми використовуємо реалізацію IUserKeyStore, що містить колекцію приватних ключів для кожного юзера. За допомогою методу RSA.Create() ми створюємо об’єкт RSA, з якого ми можемо експортувати приватний ключ з допомогою методу ExportRSAPrivateKey та додати його в наше сховище. Якшо ж ми вже створювали приватний ключ відповідно до даного користувача і алгоритму, то ми просто дістаємо приватний ключ зі сховища і імпортуємо його для подальшого використання (рис. 2.7).

```
5 references
public RSA GetOrCreateRsa(long chatId, long userId)
{
    var key = Key("RSA", chatId, userId);

    if (!_keys.TryGetValue(key, out var privateKeyBytes))
    {
        using var tmp = RSA.Create(2048);
        privateKeyBytes = tmp.ExportRSAPrivateKey();
        _keys[key] = privateKeyBytes;
    }

    var rsa = RSA.Create();
    rsa.ImportRSAPrivateKey(privateKeyBytes, out _);
    return rsa;
}
```

Рисунок 2.7 – Генерація ключів RSA

Для підпису використовують SignData. Цей метод обчислює хеш від imageBytes алгоритмом SHA-256 і підписує результат приватним ключем RSA. Його параметри- масив даних, назва хеш-алгоритму, схема доповнення. Результат – масив байтів з підписом. При помилці хешування чи недійсному ключі викидається CryptographicException або ArgumentException (рис. 2.8).

```
byte[] payload;
using (var sha = SHA256.Create())
{
    payload = sha.ComputeHash(Combine(
        pngBytes,
        BitConverter.GetBytes(chatId),
        BitConverter.GetBytes(userId)
    ));
}

var signature = rsa.SignData(payload, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
```

Рисунок 2.8 – Виконання підпису

Після отримання підпису необхідно додати його до файлу. Є кілька підходів залежно від формату зображення. Наприклад, для JPEG-файлу можна відкрити потік запису в кінець, але краще використовувати спеціальні сегменти. Часто в JPEG використовується *APPn*-сегмент (application-specific), куди можна поміщати додаткові дані.

Створюємо новий сегмент APP15 зі своїми даними (зазвичай починаємо з унікального ідентифікатора і зберігаємо там бінарний підпис). Стандартні парсери JPEG ігноруватимуть невідомі APPn-сегменти. Альтернативно, в найпростішому випадку можна просто дописати байти підпису в кінець файлу після маркера EOI (FF D9) – більшість декодерів JPEG проігнорують додаткові дані. У будь-якому разі, ми зберігаємо у файл- [оригінальні байти зображення] + [маркер/заголовок служби] + [байти RSA-підпису] (рис. 2.9).

```
var lengthBytes = BitConverter.GetBytes(signature.Length);
var result = new byte[pngBytes.Length + Magic.Length + lengthBytes.Length + signature.Length];

Buffer.BlockCopy(pngBytes, 0, result, 0, pngBytes.Length);
Buffer.BlockCopy(Magic, 0, result, pngBytes.Length, Magic.Length);
Buffer.BlockCopy(lengthBytes, 0, result, pngBytes.Length + Magic.Length, lengthBytes.Length);
Buffer.BlockCopy(signature, 0, result, pngBytes.Length + Magic.Length + lengthBytes.Length, signature.Length);
```

Рисунок 2.9 – Вбудування підпису

При відкритті файлу з підписом потрібно відокремити початкові дані зображення від вбудованого блоку підпису. Залежно від структури ми або знаємо точну довжину підпису (256 байт) і додаткові заголовки, або читаємо, наприклад, усі байти після певного маркера. Потім виконуємо перевірку, як показано на рисунку 2.10.

```
var originalPng = new byte[magicIndex];
Buffer.BlockCopy(signedPngBytes, 0, originalPng, 0, magicIndex);

using var rsa = _keyStore.GetOrCreateRsa(chatId, userId);

byte[] payload;
using (var sha = SHA256.Create())
{
    payload = sha.ComputeHash(Combine(
        originalPng,
        BitConverter.GetBytes(chatId),
        BitConverter.GetBytes(userId)
    ));
}

return rsa.VerifyData(payload, signature, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
```

Рисунок 2.10 – Верифікація підпису

Метод VerifyData поверне true, якщо підпис відповідає хешу. При неправильному ключі або зламаних байтах воно поверне false. Код програмної реалізації цього модуля представлено в додатку А

2.3. Порівняльний аналіз середовищ розробки модулів шифрування даних

Різні середовища пропонують широкі можливості для криптографії, часто через вбудовані або сторонні бібліотеки. .NET (C#) має вбудований простір імен System.Security.Cryptography, який делегує операції ОС. Для додаткових алгоритмів і функцій існують сторонні пакети. Python не містить у стандартній бібліотеці всієї криптофункціональності, але має популярний пакет PyCA Cryptography, який підтримує симетричне й асиметричне шифрування, підписи тощо. Також доступні PyCryptodome, M2Crypto та інші. Java пропонує Java Cryptography Architecture (JCA/JCE) – масштабовану систему провайдерів з API для підписів, гешів, шифрів, генерації ключів тощо. За замовчуванням включені провайдери (SunJCE), є альтернативи як BouncyCastle. C++ не має стандартної бібліотеки для криптографії, тому використовує сторонні – найвідоміші приклади – OpenSSL, Crypto++, libsodium, Botan тощо. Ці бібліотеки написані на C/C++ і забезпечують майже всі алгоритми, але вимагають підключення і налаштування. JavaScript (Node.js) має вбудований модуль crypto, який є обгорткою над OpenSSL, що забезпечує гешування, HMAC, симетричне шифрування і функції підпису/перевірки. Також є окремі бібліотеки наприклад, jsrsasign, libsodium wrappers.

За продуктивністю лідирують компільовані мови з низькорівневим доступом до алгоритмів. C++ зазвичай найшвидший, оскільки код виконується напряму і може використовувати апаратне прискорення (AVX, AES-NI). .NET (C#) та Java мають JIT-компіляцію, тому при тривалому виконанні можуть

досягати високої швидкості, близької до C/C++ (залежить від оптимізації). Операції з великими числами (RSA, ECC) в .NET делегуються швидким нативним провайдерам (CNG/OpenSSL), тому також ефективні. Python інтерпретований мову з меншим швидкодією, тому криптоопераці у «чистому» Python відчутно повільніші, хоча оптимізовані C-бібліотеки, як PyCryptodome, підвищують продуктивність. Дослідження показують, що реалізація AES на Go значно швидша, ніж на Python, а RSA навпаки може бути повільнішим. Node.js (JavaScript) – за рахунок того, що crypto використовує OpenSSL, може бути досить швидким у базових операціях, але має однопоточну модель виконання, що враховується при масштабуванні. Взагалі- найшвидші – C++ та Java/.NET, середні – Node.js, найповільніші – Python.

Усі розглянуті мови підтримують роботу з API і сервісами. Python та JavaScript відомі простотою HTTP-запитів і мають сотні готових клієнтів для API (включаючи Google Drive, Telegram). .NET (C#) теж має офіційні клієнтські бібліотеки (наприклад, Google APIs .NET Client Library для Drive) і популярний пакет Telegram.Bot. Java має свій Google API Client та бібліотеки для Telegram, хоча їх екосистема менша за Python/JS. C++ в плані інтеграції поступається- доводиться використовувати бібліотеки на кшталт libcurl і специфічні обгортки (наприклад, *tgbot-cpp*), що ускладнює розробку [29]. З таблиці офіційних прикладів видно- для .NET існує готова Telegram.Bot API-бібліотека, для Node.js – популярні Telegraf та node-telegram-bot-api, для Python - python-telegram-bot/Telethon [30]. У загальному інтеграційний потенціал найбільший у Python/Node (велика кількість бібліотек), гарний у .NET/Java (офіційні SDK), найскладніший – у C++ (невелика кількість готових бібліотек) [31].

.NET (C#) у версіях .NET Core/.NET 5+ є крос-платформовим – працює на Windows, Linux, macOS (за умови наявності потрібних ОС-бібліотек для крипто). Документація Microsoft підтверджує, що .NET використовує на різних ОС відповідні криптопровайдери, тож код C# може запускатися скрізь. Python

працює практично на всіх системах (Windows, Linux, MacOS, навіть ARM) – досить встановити інтерпретатор. Java традиційно «пишеш один раз – запускаєш скрізь» (JVM присутня на основних ОС). Node.js (JavaScript) теж легко переноситься між ОС. C++ теоретично крос-платформовий, але потребує перекомпіляції під кожну ЦА (Windows/Linux/macOS), а також налаштування залежностей (бібліотеки). У цілому- Python, Java, Node.js, .NET Core мають високу кросплатформеність, C++ – дещо меншу через необхідність перекомпіляції.

Сучасні середовища пропонують різні механізми захисту секретів. .NET має вбудовані засоби (Data Protection API, DPAPI) для шифрування даних, зберігання сертифікатів в сховищі Windows, а також легку інтеграцію з Azure Key Vault. Java використовує KeyStore/JKS для зберігання ключів із захистом паролем, також легко інтегрується з HSM чи Vault. Python і Node.js таких «з коробки» не мають- зазвичай доводиться зберігати ключі у файлах або змінних середовища і використовувати сторонні модулі (наприклад, python-keyring, node-keytar) для доступу до системного сховища або хмарних секрет-менеджерів. C++ не має стандартних механізмів – можна використовувати системні API (Windows CryptoAPI, macOS Keychain) або сторонні рішення. Отже, найбільш зручне (вбудоване) безпечне зберігання ключів є у .NET і Java, а в інших мовах потрібні додаткові зусилля.

Python та JavaScript мають одну з найбільших спільнот розробників, обсяг документації та прикладів – гігантський. Java також дуже поширена в корпоративному середовищі, тому великі ресурси і форуми є. .NET (C#) підтримується Microsoft і має хорошу офіційну документацію і активну спільноту, хоча трохи менш розгалужену, ніж у Python/JS. C++ – поширена мова, але для криптографії часто залучаються спеціалізовані бібліотеки, в яких документація може бути складнішою. У цілому- для пошуку рішень та прикладів найзручніші Python/JS/Java з їх великими спільнотами; .NET також

має достатньо ресурсів (добре документовані класи та NuGet-пакети); C++ доведеться розбиратися через офіційну документацію бібліотек та рішення на форумах. Порівняння технологій в контексті розробки модулів шифрування даних представлено в таблиці 2.3.

Таблиця 2.3

Порівняння технологій в контексті розробки модулів шифрування даних

Критерій	.NET (C#)	Python	Java	C++	JavaScript (Node.js)
Крипто-бібліотеки	Вбудовано, стабільно	Широкі, OpenSSL-залежні	Повні, Bouncy Castle	Потужні, низькорівневі	Базові, не всі алгоритми
Продуктив-ність	Висока (JIT)	Низька	Висока	Дуже висока	Середня (однопотокова)
Інтеграція API	SDK, OAuth, Telegram	Бібліотеки, проста інтеграція	Хороша підтримка API	Обмежена, вручну	Відмінна для веб API
Кросплатформеність	.NET Core, повна	Повна	JVM-базована	Повна, але складна збірка	Повна (V8)
Збереження ключів	DPAPI, KeyVault	Ручне або нестандартне	KeyStore, HSM	Ручне, OpenSSL	Ручне, без стандарту
Спільнота/Документація	Активна, MSDN	Дуже активна	Велика, корпоративна	Складна, експертна	Масова, багато прикладів

Як бачимо з порівняння представленого в таблиці 2.3, технології мають чіткі критерії своєї оцінки, а саме:

– Криптобібліотеки. Усі мови мають потужну підтримку (зазвичай через сторонні пакети). .NET та Java пропонують вбудовані засоби (CNG/Лінії, JCA), Python і Node.js покладаються на популярні бібліотеки з відкритим кодом (Cryptography, Crypto), C++ – на OpenSSL/Crypto++.

– Продуктивність найкраща у C++ (5), Java і .NET помірна (4), Python найнижча (2). Node.js, завдяки OpenSSL, має середню швидкість (3). Це важливо при генерації підписів/ключів у реальному часі.

– Інтеграція. Python і Node.js («скриптові» мови) мають найпотужніші клієнтські SDK для Telegram, Google Drive тощо, C# і Java – також добре, але C++ – лише базові бібліотеки (libcurl), що ускладнює розробку.

– Кросплатформеність. Усі, крім C++, дуже зручні, .NET Core, Python, Java, Node.js легко запускаються на різних ОС; для C++ потрібна перекомпіляція.

– Зберігання ключів найзахищеніше в .NET (DPAPI, Key Vault) та Java (KeyStore), тоді як Python/JS/C++ потребують додаткових рішень.

– Спільнота/документація. Python, JS, Java мають найбільші спільноти (рейтинг 5). .NET також широко поширений і добре документований, але трохи поступається в обсязі. C++/спеціалізовані бібліотеки мають менше «стандартного» матеріалу.

Обрано .NET (C#) через збалансованість усіх критеріїв.

По-перше, .NET має вбудовані криптографічні API, що використовують надійні механізми ОС (CNG/Windows CryptoAPI чи OpenSSL на Linux), тож розробник отримує готові реалізації RSA/ECDSA/AES тощо з мінімальними зусиллями.

По-друге, продуктивність C# достатня для підписування зображень у режимі бота (більша, ніж у Python), особливо з урахуванням JIT-оптимізацій.

По-третє, інтеграція з Telegram API у C# дуже зручна- існує популярна бібліотека Telegram.Bot, яка дозволяє швидко налаштувати прийом та відправку повідомлень з фотографіями. Також у C# доступні SDK для Google Drive та інших сервісів, якщо знадобляться.

На додачу, .NET Core забезпечує кросплатформеність – бот можна запускати на сервері Linux чи Windows без значних змін коду. Безпека ключів у .NET теж на високому рівні- можна використовувати DataProtection API або Azure Key Vault для захисту приватних ключів.

Нарешті, велика спільнота .NET/C# і гарна документація гарантують підтримку та приклади реалізації. Таким чином, .NET (C#) поєднує у собі розвинуті крипто-бібліотеки, прийнятну швидкодію, зручну інтеграцію з Telegram та іншими сервісами, а також безпечні інструменти для обробки ключів – що робить цю платформу оптимальною для роботи з Telegram-ботом, який займається електронним підписом зображень.

3. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ІНТЕГРАЦІЇ TELEGRAM-БОТА З MIDJOURNEY AI

Розробка інтегрованої системи генерації та обробки зображень базувалася на архітектурі, яка поєднує бота Telegram як клієнтський інтерфейс, проміжний API для маршрутизації запитів та генеративну модель Midjourney як ядро обчислювального модуля. Telegram було обрано як комунікаційну платформу завдяки своїй відкритій інфраструктурі Bot API, підтримці передачі зображень та метаданих, асинхронній обробці запитів та можливості інтеграції криптографічних функцій без потреби у сторонньому клієнтському програмному забезпеченні. В результаті бот Telegram виступає універсальною точкою доступу до сервісів штучного інтелекту, з можливістю авторизації, генерації, підпису та доставки зображень.

3.1. Розробка архітектури Telegram-бота

Розроблений Telegram-бот є комплексною системою (рис. 3.1), яка поєднує можливості генерації зображень за допомогою Midjourney, криптографічної обробки та інтеграції з Google Drive для зберігання результатів. Архітектура побудована на основі .NET Web API (Web Application Program Interface) та використовує подійну й асинхронну модель роботи.

Система складається з таких ключових компонентів:

– Web API, що приймає webhook-запити від зовнішнього сервісу APIFrame (інтерфейс до Midjourney).

– Telegram Bot Client, який обробляє повідомлення користувачів.

– Фоновий сервіс (IHostedService), що запускає Telegram-бота та координує роботу внутрішніх модулів.

– Сервіс виклику Midjourney - MidjourneyInvoker.
– Сервіс зберігання та обробки запитів - ImageProcessingService.
– Сервіс Google Drive - GoogleService.
– Агрегатор подій - EventAggregator.
– Слухач webhook-подій - HookListener.
– Основний бізнес-сервіс для Telegram - TelegramService.
– CryptographyService – сервіс для підпису і перевірки зображень.

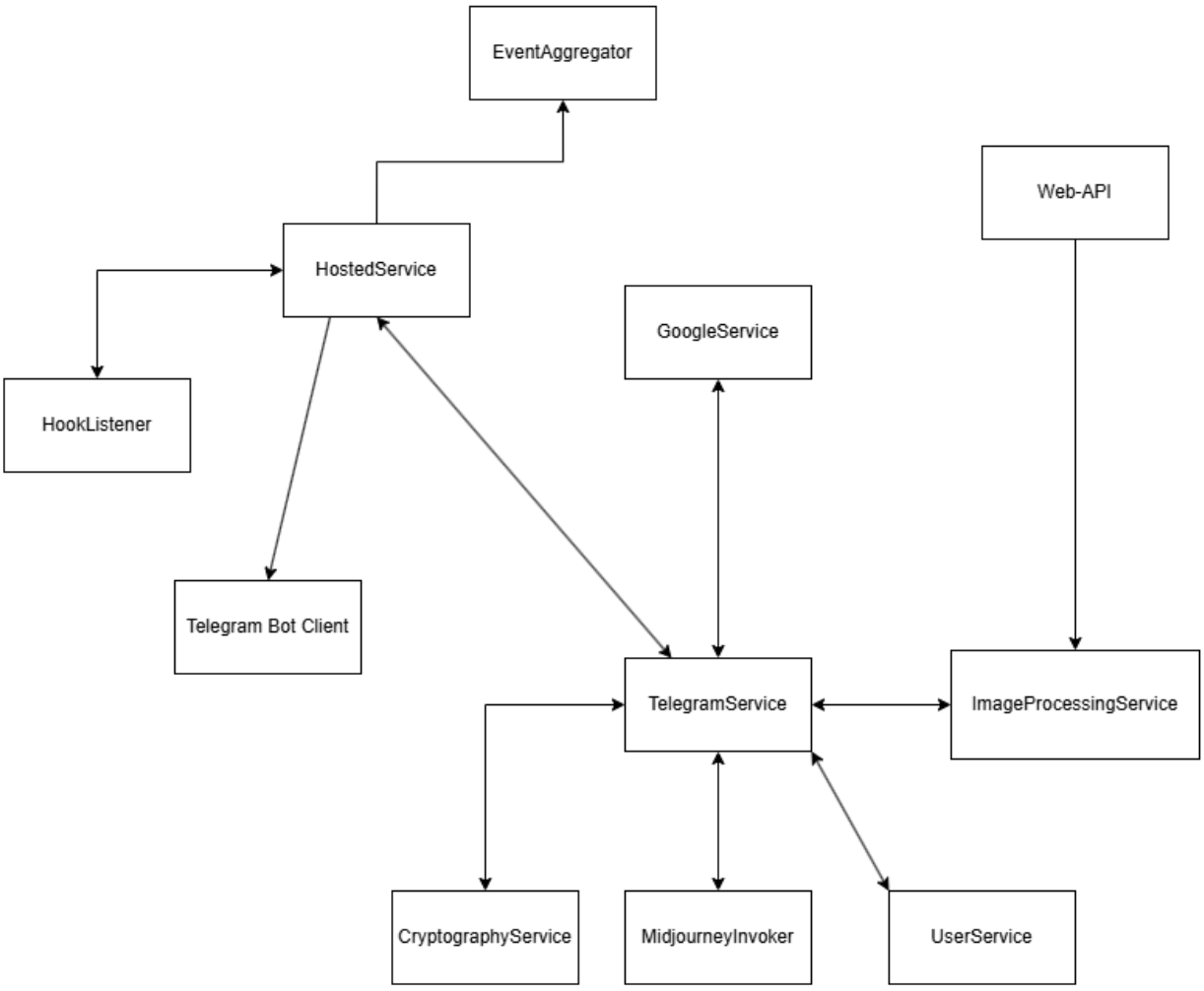


Рисунок 3.1 – Схема компонентів системи

Усі компоненти, що представлені на рисунку 3.1 взаємодіють між собою за принципами SOLID, інверсії залежностей та чіткої сегментації відповідальностей, що забезпечує масштабованість, модульність та простоту подальшого розширення.

Архітектура створеного Telegram-бота являє собою комплексне технічне рішення, у якому поєднані механізми взаємодії користувача з Telegram Bot API, алгоритми генерації зображень через Midjourney, логіка обробки проміжних статусів та можливість збереження результатів у хмарному середовищі Google Drive. Система створена на основі технології .NET Web API та орієнтована на асинхронну обробку подій, масштабованість та незалежність окремих підсистем одна від одної.

У межах системи реалізована інтеграція декількох ключових функціональних підсистем, кожна з яких відповідає за конкретний етап життєвого циклу користувацького запиту. Центральною частиною виступає Web API, який приймає webhook-запити від зовнішнього сервісу APIFrame, що слугує інтерфейсом до Midjourney. Telegram Bot Client реалізує механізм взаємодії з користувачами через Telegram і відповідає за отримання команд, повідомлень, callback-подій та відправлення результатів. Асинхронний фоновий сервіс на базі IHostedService забезпечує запуск Telegram-бота після старту веб-додатка, а також координує процеси, пов’язані з обробкою оновлень, ініціалізацією сервісів та синхронізацією роботи різних компонентів системи.

Функціональність генерації зображень реалізована через спеціалізований сервіс MidjourneyInvoker, який інкапсулює логіку взаємодії з APIFrame. Усі запити на генерацію prompt’ів створюються, відправляються та відстежуються саме через цей компонент. Інформація про поточний стан кожного запиту зберігається та оновлюється у сервісі обробки запитів IImageProcessingService, який відповідає за створення, пошук, модифікацію та систематизацію даних про завдання генерації.

Завантаження результатів у Google Drive забезпечує окремий модуль GoogleService, що реалізує авторизацію через OAuth 2.0 та обробляє операції завантаження файлів. Архітектура також включає подійний агрегатор EventAggregator, який виконує роль внутрішнього механізму розсилання подій між компонентами, зберігаючи при цьому низький рівень їхньої зв'язності. Слухач webhook-подій HookListener реагує на оновлення статусу задачі та відповідає за інформування користувача в Telegram про прогрес або завершення генерації.

Подальша робота з вибраними варіаціями, авторизацією та збереженням зображень реалізована у TelegramService – основному сервісі обробки команд і callback-запитів. Саме в ньому відбувається інтерпретація команд /start і /imagine, обробка надісланих користувачем prompt'ів, виклик MidjourneyInvoker, збереження даних про запит, передача інформації у EventAggregator та підтримка інтерактивних сценаріїв через callback-кнопки. TelegramService реалізує повний життєвий цикл взаємодії користувача з ботом-від моменту надсилання команди до завантаження зображення у Google Drive.

Окремим модулем архітектури виступає GoogleService, що забезпечує можливість зберігання зображень на Google Drive. Авторизація реалізована через OAuth 2.0 із використанням офіційної бібліотеки Google APIs. Сервіс створює HTTP-клієнт Google Drive, формує метадані файлу, виконує завантаження та повертає інформацію про результат.

Таким чином, архітектура Telegram-бота сформована як подійно-орієнтована система з асинхронною обробкою запитів, де кожен компонент виконує чітко визначену функцію, а взаємодія між ними здійснюється через механізми інверсії залежностей та внутрішню подійну модель.

3.2. Програмна реалізація архітектури Telegram-бота

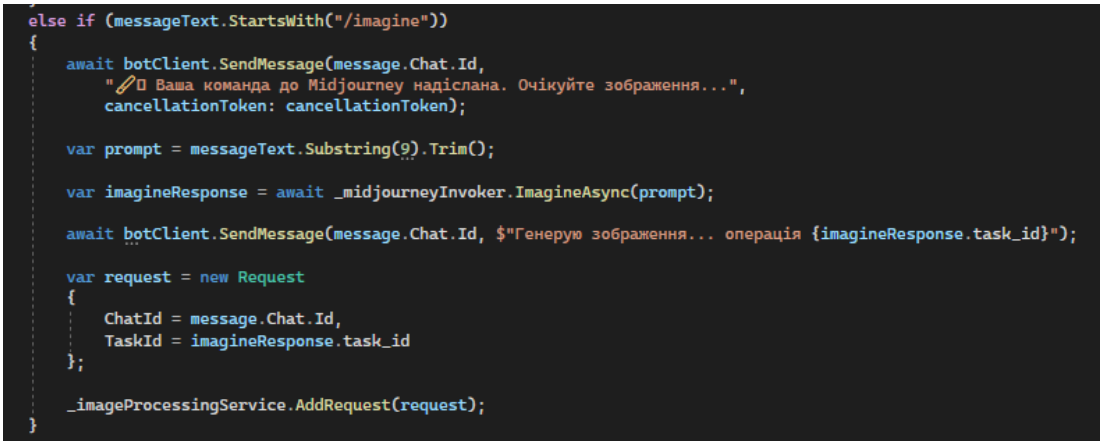
3.2.1. Програмний код реалізації архітектури

Інтеграція Telegram-бота з Midjourney AI реалізована через проміжний HTTP-сервіс APIFrame, який надає REST-інтерфейс для надсилання запитів на запити та отримання інформації про стан завдань генерації зображень. Алгоритм інтеграції побудовано як послідовність асинхронних викликів, що охоплюють етапи формування запиту, його надсилання до APIFrame, збереження ідентифікатора завдання, обробки webhook-повідомлень та подальшої взаємодії з користувачем у Telegram. Кожен логічний крок з відповідними фрагментами коду детально розглянуто нижче.

Початковою точкою алгоритму є команда користувача */imagine*, яка надсилається в чаті з ботом. Ця команда обробляється в методі HandleUpdateAsync класу TelegramService. Сервіс відокремлює текст запиту, викликає метод інтеграції з Midjourney та реєструє нове завдання у внутрішньому сервісі обробки запитів-

На цьому етапі алгоритм виконує кілька важливих дій- інформує користувача про запуск генерації, витягує запит з тексту повідомлення, викликає сервіс інтеграції MidjourneyInvoker та зберігає зв'язок між ChatId користувача та task_id, повернутим APIFrame (рис. 3.2). Сам task_id є ключем, який дозволяє додатково зіставляти відповіді вебхуків з певним користувачем та його запитом.

Алгоритмічно цей етап можна описати так- формується POST-запит на кінцеву точку /imagine, додається токен авторизації для доступу до APIFrame, створюється об’єкт ImagineRequest, який містить prompt користувача, співвідношення сторін зображення та URL webhook-ендпоінта, куди мають надходити повідомлення про стан задачі. Цей об’єкт серіалізується у JSON і передається як тіло HTTP-запиту.



```
else if (messageText.StartsWith("/imagine"))
{
    await botClient.SendMessage(message.Chat.Id,
        "🖌️ Ваша команда до Midjourney надіслана. Очікуйте зображення...",
        cancellationToken: cancellationToken);

    var prompt = messageText.Substring(9).Trim();

    var imagineResponse = await _midjourneyInvoker.ImagineAsync(prompt);

    await botClient.SendMessage(message.Chat.Id, $"Генерую зображення... операція {imagineResponse.task_id}");

    var request = new Request
    {
        ChatId = message.Chat.Id,
        TaskId = imagineResponse.task_id
    };

    _imageProcessingService.AddRequest(request);
}
```

Рисунок 3.2 – Компонент для запуску роботи бота з користувачем

Після отримання відповіді дані десеріалізуються у модель ImagineResponse, яка містить унікальний ідентифікатор задачі task_id, як показано на рисунку 3.3, саме це значення далі зберігається у сервісі ImageProcessingService.

```
1 reference
public async Task<ImagineResponse> ImagineAsync(string prompt, string ratio)
{
    var request = new HttpRequestMessage(HttpMethod.Post, "https://api.apiframe.pro/imagine");
    request.Headers.Add("Authorization", "<API_KEY>");

    var imagineRequest = new ImagineRequest(prompt, ratio, "https://webhook.site/bfc1fc89-2666-4b1c-8bd7-62a955dfbf91", "");

    var stringRequest = JsonConvert.SerializeObject(imagineRequest);

    var content = new StringContent(stringRequest, null, "application/json");
    request.Content = content;
    var response = await _httpClient.SendAsync(request);
    response.EnsureSuccessStatusCode();

    var stringResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine(stringResponse);

    var imagineResponse = JsonConvert.DeserializeObject<ImagineResponse>(stringResponse);

    return imagineResponse;
}
```

Рисунок 3.3 – Компонент для відправки запиту у Midjourney AI

Важливим моментом є використання методу EnsureSuccessStatusCode(), що генерує виняток у разі неуспішного HTTP-коду відповіді. Це дозволяє централізовано обробляти помилки інтеграції та, за потреби, додати механізм повторних спроб або логування. Після того як APIFrame прийняв запит до

Midjourney, подальша комунікація зі сторони цього сервісу відбувається через webhook-виклики на адресу, вказану у полі webhook_url. На стороні .NET Web API ця взаємодія реалізована в контролері ImageProcessingController (рис. 3.4).

```
[ApiController]
[Route("[controller]")]
0 references
public class ImageProcessingController : ControllerBase
{
    [HttpPost("imagine-hook")]
    0 references
    public async Task<IActionResult> ImagineHookAsync([FromServices] IImageProcessingService imageProcessingService)
    {
        using var reader = new StreamReader(Request.Body);
        var body = await reader.ReadToEndAsync();

        var contentType = Request.ContentType;

        Console.WriteLine($"Content-Type: {contentType}");
        Console.WriteLine($"Body: {body}");

        var imagineResponse = JsonConvert.DeserializeObject<ImagineHookResponse>(body);

        imageProcessingService.UpdateRequest(imagineResponse);

        return Ok();

    }
```

Рисунок 3.4 – Отримання повідомлення про стан генерації зображення

На цьому етапі алгоритм інтеграції отримує нові дані про стан задачі генерації. Тіло HTTP-запиту читається як сирий текст, логування Content-Type та вмісту тіла дозволяє діагностувати можливі проблеми формату. Далі JSON-рядок десеріалізується в об’єкт ImagineHookResponse, який містить інформацію про статус (processing, finished тощо), URL-адреси згенерованих зображень та відповідний task_id. Оновлення стану відбувається через метод UpdateRequest сервісу IImageProcessingService, де за task_id знаходиться відповідна заявка користувача і доповнюється новими даними.

Таким чином, Web API-контролер у цьому алгоритмі виконує роль "точки входу" для зовнішніх повідомлень про прогрес генерації та є зв’язувальною ланкою між APIFrame і внутрішньою подійною моделлю застосунку. Сервіс IImageProcessingService відповідає за зберігання стану всіх активних запитів. Після виклику UpdateRequest(imagineResponse) він оновлює інформацію про відповідний запит, зберігає надані APIFrame статус та посилання на

зображення, а потім генерує подію оновлення через EventAggregator. Це дозволяє відокремити логіку оновлення даних від логіки реагування на ці зміни.

Сервіс даних не працює безпосередньо з Telegram і не надсилає жодних повідомлень користувачу. Натомість він повідомляє решту системи, що для певного task_id з'явилася нова інформація, і передає цей факт у вигляді події. Далі обробкою займається окремий компонент. Обробником події оновлення запиту є клас HookListener, який підписується на події RequestUpdate через EventAggregator. Саме в ньому реалізовано алгоритм взаємодії з Telegram-користувачем на етапах "генерується" та "завершено" (рис. 3.5).

```
1 reference
public async void RequestUpdated(object? sender, string taskId)
{
    var request = _imageProcessingService.GetByTaskId(taskId);

    switch (request.ImagineHookResponse.status)
    {
        case "processing":
            {
                await _telegramBotClient.SendMessage(request.ChatId, $"Генерую зображення... операція {taskId}");

            }
            break;

        case "finished":
            {
                var keyboard = new InlineKeyboardMarkup(new[]
                {
                    new[]
                    {
                        InlineKeyboardButton.WithCallbackData("u 1", $"choose:0:{taskId}"),
                        InlineKeyboardButton.WithCallbackData("u 2", $"choose:1:{taskId}"),
                        InlineKeyboardButton.WithCallbackData("u 3", $"choose:2:{taskId}"),
                        InlineKeyboardButton.WithCallbackData("u 4", $"choose:3:{taskId}"),
                        InlineKeyboardButton.WithCallbackData("redo", "redo"),
                        InlineKeyboardButton.WithUrl("Full", @$"{request.ImagineHookResponse.original_image_url}")
                    }
                });

                await _telegramBotClient.SendMessage(request.ChatId, $"Зображення згенеровано... операція {taskId}");

                using var httpClient = new HttpClient();

                var imageBytes = await httpClient.GetByteArrayAsync(request.ImagineHookResponse.original_image_url);

                using var stream = new MemoryStream(imageBytes);

                await _telegramBotClient.SendPhoto(request.ChatId, InputFile.FromStream(stream), replyMarkup: keyboard);
            }
            break;
    }
```

Рисунок 3.5 – Представлення згенерованих зображень користувачу

Алгоритм на цьому кроці працює так- за ідентифікатором задачі отримується повний об'єкт запиту, далі аналізується поточний статус. Якщо статус "processing", користувач одержує інформаційне повідомлення про те, що

генерація триває. Якщо статус "finished", система формує inline-клавіатуру з варіантами u1–u4, кнопкою повторної генерації та посиланням на повнорозмірне зображення, завантажує згенерований файл за посиланням original_image_url і надсилає його користувачу разом із клавіатурою. Таким чином, завершальний етап інтеграції з Midjourney не лише доставляє результат, але й дає можливість продовжити роботу з ним у Telegram (зокрема вибрати варіацію або зберегти зображення у Google Drive).

У підсумку алгоритм інтеграції з Midjourney AI реалізовано як наскрізний асинхронний процес, що починається з команди /imagine у Telegram, переходить до форимування HTTP-запиту на APIFrame, зберігає task_id і пов’язує його з користувачем, продовжується обробкою webhook-повідомлень у Web API, оновленням стану задачі у IImageProcessingService та генерацією події, а завершується реакцією HookListener, який інформує користувача про результати і надсилає йому згенероване зображення. Завдяки чіткому розподілу відповідальностей та інкапсуляції інтеграційної логіки у спеціальних сервісах, система залишається розширюваною, тестованою та стійкою до змін зовнішнього API.

У межах дослідження та реалізації бота використовується чотири криптографічні алгоритми, кожен із яких виконує окрему роль. RSA, DSA та ECDSA застосовуються переважно для порівняння, оцінки їхньої продуктивності та формування метрик, необхідних для аналітичної частини магістерської роботи. Основним же алгоритмом, який використовується у практичному сценарії, є модифікована версія RSA з вбудованим підписом у структуру PNG-файлу (RSA-EMBED). Саме цей метод дозволяє підписувати зображення без зміни їхнього вмісту та одночасно вбудовувати криптографічні дані в сам файл, що робить процес прозорим для користувача та сумісним із сервісами зберігання на кшталт Google Drive.

Для алгоритмів, що повертають зовнішній підпис (RSA, DSA, ECDSA), використовується спільний інтерфейс, який показано на рисунку 3.6.

```
5 references
public interface IImageSigner
{
    8 references
    string Algorithm { get; }
    4 references
    byte[] Sign(byte[] imageBytes, long chatId, long userId);
    5 references
    bool Verify(byte[] imageBytes, long chatId, long userId, byte[] signature);
}
```

Рисунок 3.6 – Інтерфейс для роботи з модулями підпису

Це дозволяє централізовано керувати різними реалізаціями та порівнювати їхню продуктивність. RSA-варіант підписує SHA-256 хеш від байтового представлення PNG-файлу (рис. 3.7)-

```
    2 references
    public byte[] Sign(byte[] imageBytes, long chatId, long userId)
    {
        using var rsa = _keyStore.GetOrCreateRsa(chatId, userId);
        var payload = ImagePayloadBuilder.BuildPayloadRsa(imageBytes);

        return rsa.SignData(payload, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    }

    3 references
    public bool Verify(byte[] imageBytes, long chatId, long userId, byte[] signature)
    {
        using var rsa = _keyStore.GetOrCreateRsa(chatId, userId);
        var payload = ImagePayloadBuilder.BuildPayloadRsa(imageBytes);

        return rsa.VerifyData(payload, signature, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    }
}
```

Рисунок 3.7 – Модуль підпису з використанням алгоритму RSA

Алгоритми DSA та ECDSA реалізуються аналогічно із використанням відповідних класів DSA та ECDsa та включають в payload додаткові поля (chatId, userId) для жорсткої прив’язки підпису до конкретного користувача.

Основним алгоритмом, що застосовується в роботі бота, є варіант RSA-EMBED, який вбудовує підпис безпосередньо в PNG-файл.

Саме цей варіант використовується при підписі зображень, що потім завантажуються на Google Drive, його ідея полягає в наступному:

– оригінальний PNG-файл не модифікується на рівні пікселів;

– підпис додається у вигляді службового “хвоста” в кінці файлу;

– структура хвоста- MAGIC-рядок для ідентифікації SIGPNGMIDJOUNEY + довжина підпису (int32 в Little Endian) + байти підпису.

Більшість PNG-декодерів ігнорують додаткові байти після кінця файлу, тому зображення коректно відображається у звичайних переглядачах, але при цьому містить у собі криптографічний підпис, а для вбудованого підпису у системі використовується окремий інтерфейс (рис. 3.8).

```
3 references
public interface IEmbeddedImageSigner
{
    4 references
    string Algorithm { get; }

    2 references
    byte[] SignAndEmbed(byte[] pngBytes, long chatId, long userId);

    2 references
    bool VerifyEmbedded(byte[] signedPngBytes, long chatId, long userId);
}
```

Рисунок 3.8 – Інтерфейс для роботи з модулями підпису

У рамках запропонованого алгоритму перевірка автентичності включає в сете наступне:

– виділення оригінального PNG із підписаного файлу;

– повторне обчислення SHA-256 від originalPng + chatId + userId;

– перевірку RSA-підпису отриманого payload.

Для зручної роботи з кількома алгоритмами реалізовано сервіс CryptographyService, який інкапсулює логіку виклику конкретних реалізацій та збирання метрик.

3.2.2. Принципи роботи з телеграм-ботом обробки зображень

Робота з ботом розпочинається з моменту його додавання до переліку контактів у Telegram. Користувач знаходить бота за його унікальним ім’ям або посиланням, після чого натискає кнопку “Start” або надсилає команду /start. Ця команда ініціює встановлення первинного сеансу взаємодії та запускає базову логіку сервісу.

Команда */start* також слугує точкою входу у процес роботи з ботом-користувач отримує інструкції щодо подальших кроків (рис. 3.9), зокрема про те, як згенерувати зображення через Midjourney, як ініціювати підпис та як завантажити результат у Google Drive.

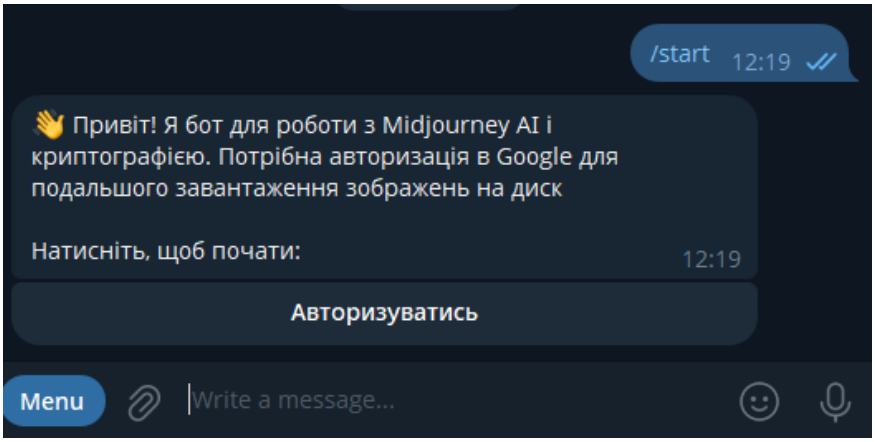


Рисунок 3.9 – Ініціалізація роботи користувача з ботом

Після первинного запуску (рис. 3.10) користувачеві необхідно завершити процес авторизації, аби бот міг виконувати операції зі збереження підписаних зображень у його Google Drive. Для цього бот надсилає спеціальне повідомлення з кнопкою «Авторизуватись». Натиснувши її, користувач автоматично переходить у браузер, де відкривається сторінка OAuth-автентифікації Google.

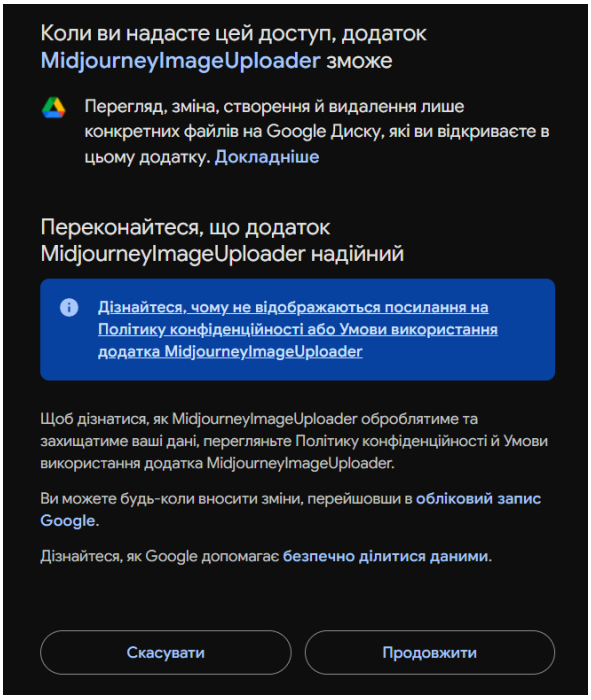


Рисунок 3.10 – Авторизація користувача на Google Drive

Після успішної авторизації в callback буде повідомлення (рис. 3.11)-

Рисунок 3.11 – Повідомлення про успішну авторизацію

Далі ми можемо використати команду */imagine* для того, щоб згенерувати зображення (рис. 3.12). Після того, як операція виконається успішно, сервер відправить нам 4 варіанти зображення з можливістю вибору.

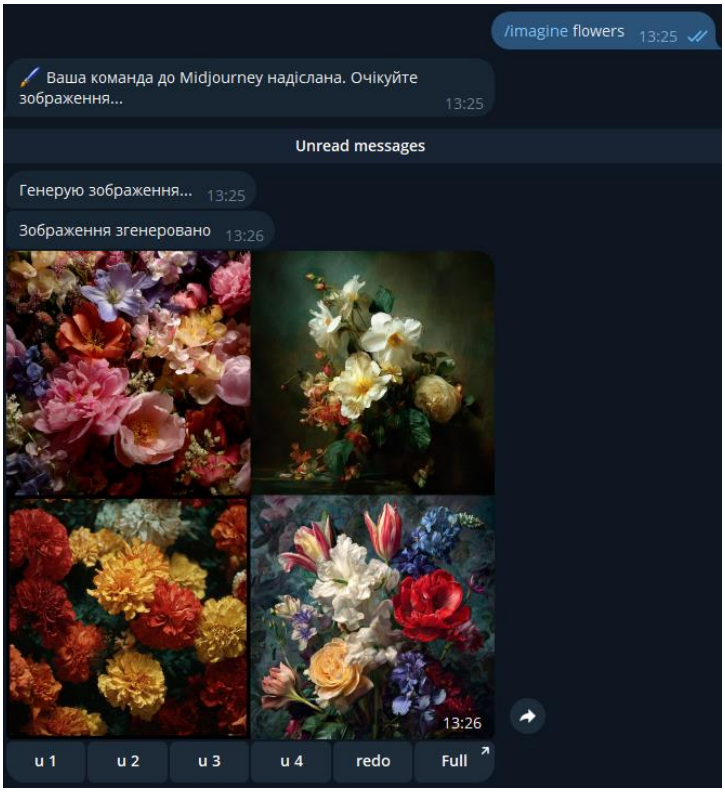


Рисунок 3.12 – Результат команди /imagine

Далі, виберемо перший варіант з допомогою вбудованих кнопок вибору (рис. 3.13).

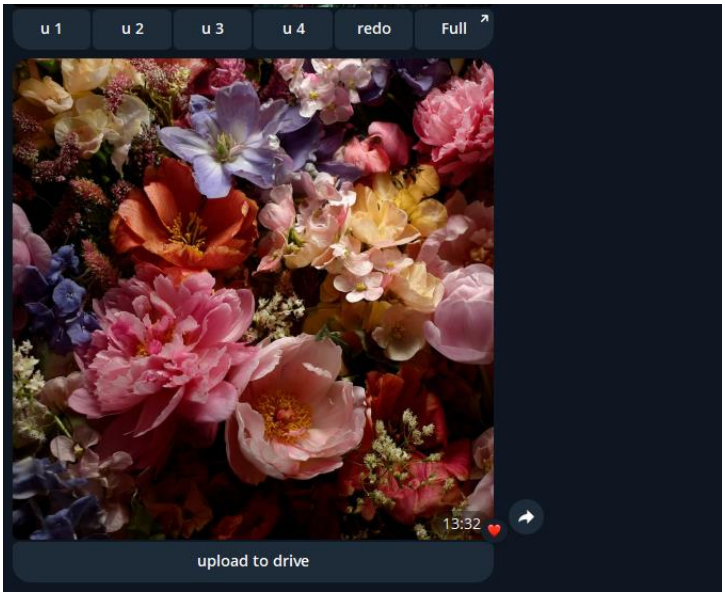


Рисунок 3.13 – Вибір зображення з допомогою вбудованого інтерфейсу

Для того, щоб підписати і завантажити зображення на Google Drive, можемо використати вбудовану кнопку Upload to drive. На стороні сервера, ми виконуємо підписання файлу та завантажуємо його у сховище.

Після успішного підписання і завантаження ми отримаємо повідомлення, як показано на рисунку 3.14-

Рисунок 3.14 – Повідомлення про успішне підписання і завантаження

Перевіримо, чи дійсно зображення завантажено на Google Drive. Так, дійсно, зображення ми завантажили (рис. 3.15).

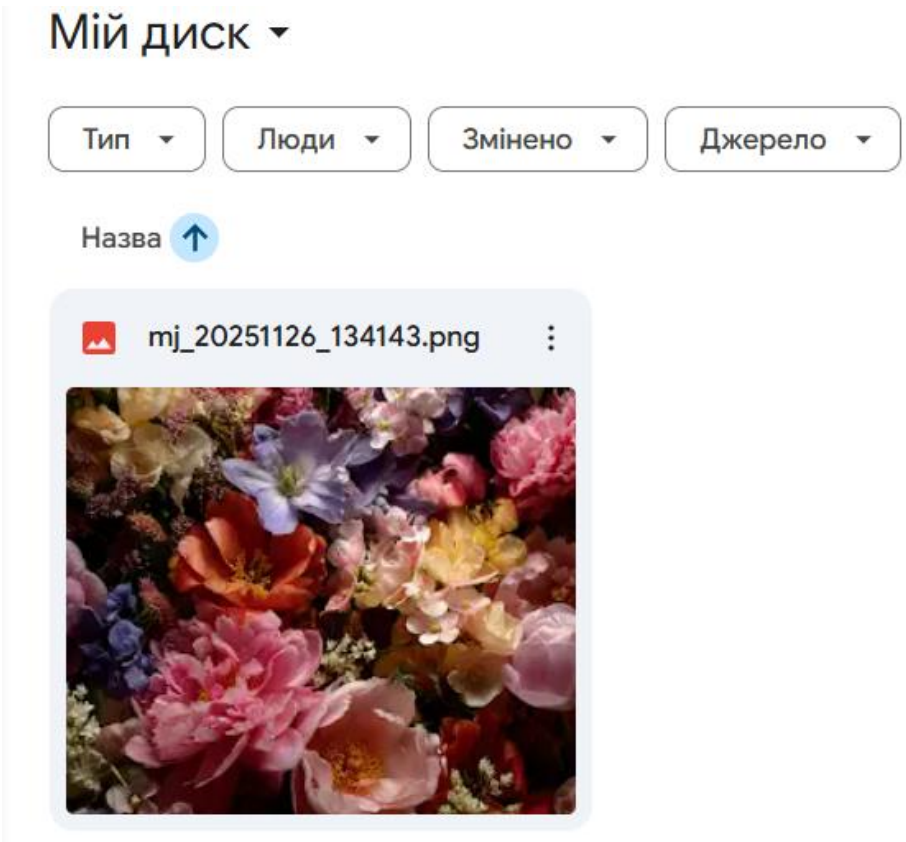


Рисунок 3.15 – Завантажене зображення на Google Drive

Тепер виконаємо перевірку зображення на автентичність, тобто чи дійсно зображення підписане. Для цього використаємо команду */verify* (рис. 3.16).

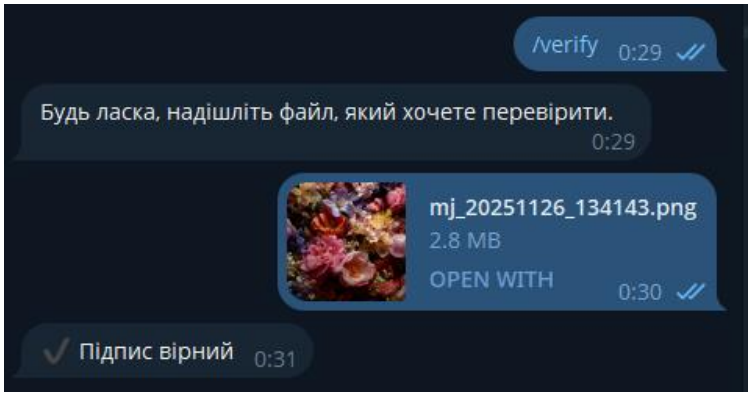


Рисунок 3.16 – Успішна верифікація підпису зображення

Якщо завантажити зображення без вбудованого підпису, то сервіс перевірить його і відправить відповідне повідомлення (рис. 3.17).

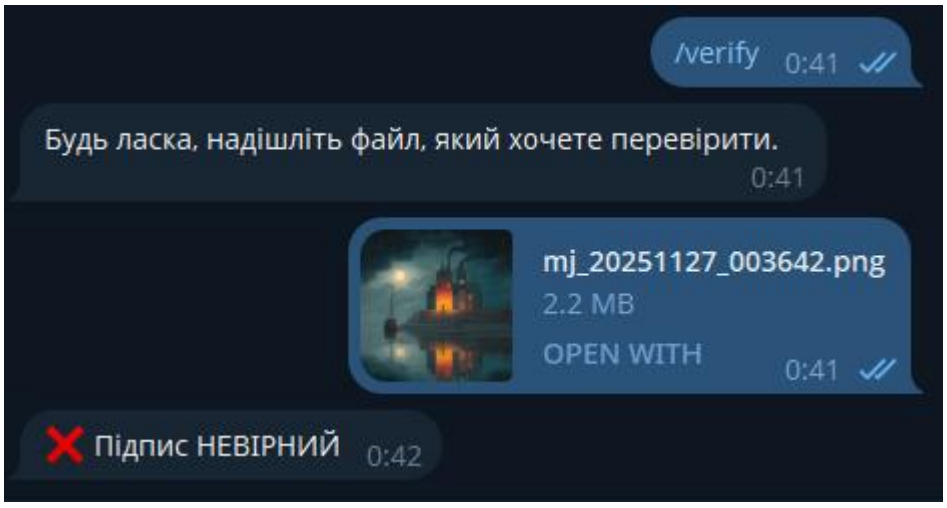


Рисунок 3.17 – Приклад перевірки зображення з помилковим підписом

3.3. Дослідження ефективності інтеграції та безпеки даних розробленої архітектури

3.3.1. Аналіз продуктивності криптографічних методів обробки зображень

Для оцінки швидкодії цифрового підпису було проведено вимірювання часу генерації та перевірки підписів зображень різного розміру (близько 0.9, 2.9, 5.6 та 8.0 МБ). Досліджувалися алгоритми RSA, DSA, ECDSA, а також метод *RSA-EMBED*, де підпис RSA вбудовується безпосередньо в файл зображення.

Результати вимірювань часу наведено у таблицях 3.1 та 3.2.

Таблиця 3.1

Час генерації цифрового підпису для зображень різного розміру

Алгоритм	0.9 МБ, мс	2.9 МБ, мс	5.6 МБ, мс	8.0 МБ, мс
RSA	~11	~25	~40	~60
DSA	~9	~23	~38	~59
ECDSA	~10	~22	~38	~59
RSA-EMBED	~18	~45	~80	~120

Таблиця 3.1 містить середній час генерації підпису для кожного алгоритму залежно від розміру зображення, а таблиця 3.2 – час перевірки підпису.

Видно, що зі збільшенням розміру файлу час обробки зростає майже лінійно – це очікувано, адже основні витрати часу припадають на обчислення хешу зображення (операція з лінійною складністю від обсягу даних).

Таблиця 3.2

Час перевірки цифрового підпису для зображень різного розміру

Алгоритм	0.9 МБ, мс	2.9 МБ, мс	5.6 МБ, мс	8.0 МБ, мс
RSA	~9	~21	~38	~58
DSA	~10	~24	~39	~59
ECDSA	~10	~23	~39	~60
RSA-EMBED	~19	~46	~81	~118

Згідно з отриманими результатами, всі алгоритми демонструють схожу масштабованість- при збільшенні розміру файлу в ~9 разів (від 0.9 МБ до 8.0 МБ) час підписання та перевірки зростає приблизно в 8–10 разів. Це вказує на домінування операцій хешування та вводу/виводу над власне криптографічними обчисленнями при роботі з великими зображеннями. Втім, для менших файлів різниця між алгоритмами помітніша.

При генерації підпису алгоритм ECDSA показав найвищу швидкодію. Це пояснюється використанням коротших ключів та ефективніших обчислень на еліптичних кривих, що суттєво зменшує обчислювальні витрати та підвищує продуктивність. DSA також продемонстрував дещо кращу швидкість підписання, ніж RSA, будучи менш вимогливим до ресурсів порівняно з RSA. Алгоритм RSA виявився найповільнішим при підписанні (через дорогі операції з великими простими числами), хоча забезпечує співставний рівень безпеки. Натомість при перевірці підпису RSA випереджає інші алгоритми- завдяки відносно малому публічному експоненту перевірка RSA-підпису виконується дуже швидко. DSA потребує двох модульних експоненцій при перевірці, тому його швидкодія середня. ECDSA при перевірці виконує кілька операцій з точками кривої, що робить його дещо повільнішим за RSA, але різниця несуттєва (при великих розмірах файлів вона нівелюється затратами на хешування).

Метод RSA-EMBED є функціонально еквівалентним звичайному RSA у плані криптографії, але додає накладні витрати на вбудовування підпису в файл. Як видно з таблиць, це призводить до збільшення загального часу- під час накладення підпису бот мусить додатково записати підписане зображення на диск (додавши службові дані), а при перевірці – зчитати файл та вилучити підпис. Для невеликих файлів цей оверхед незначний, але на найбільшому зображенні (8 МБ) метод RSA-EMBED майже подвоює час обробки в порівнянні зі стандартним RSA. Таким чином, використання вбудованого підпису зручне з точки зору збереження підпису разом із зображенням, проте за це сплачується додатковим часом обробки і ускладненням логіки роботи з файлами.

Продуктивність Telegram-бота при накладенні та перевірці цифрових підписів зображень залежить головно від розміру файлів та вибраного алгоритму. Найкращу швидкодію демонструє ECDSA, в той час як RSA має найвищі затримки при генерації підпису. Вбудовування підпису в зображення сповільнює обробку через додаткові операції вводу/виводу. Попри відмінності, всі розглянуті алгоритми справляються із завданням за прийнятний час (десятки мілісекунд для файлів розміром кілька МБ), тому вибір конкретного алгоритму може визначатися іншими чинниками – рівнем безпеки, сумісністю або зручністю реалізації.

3.3.2. Оцінка безпеки даних

Безпека даних є критично важливою при розробці бота, що обробляє користувацькі зображення та взаємодіє з зовнішніми сервісами. Розглянемо ключові аспекти захисту інформації в даному рішенні та проаналізуємо потенційні загрози.

Цифровий підпис для цілісності та автентичності**.** Застосування електронного цифрового підпису гарантує, що зображення не було змінене або

підроблене під час передачі чи зберігання. Бот, підписуючи зображення закритим ключем, забезпечує криптографічний зв’язок між зображенням і його джерелом. Тільки володіючи відповідним відкритим ключем, можна перевірити підпис і тим самим підтвердити автентичність зображення та особу (або сервіс), що його підписала. Це захищає від непомітного спотворення даних- будь-яка модифікація файлу після підписання призведе до невідповідності хешів і недійсності підпису. Таким чином, користувач може впевнитись, що отримане зображення саме те, яке було створене ботом (або надане для перевірки), і не було змінене третім боком. Цей механізм підвищує довіру до дій бота, особливо якщо зображення передаються через незахищені канали чи зберігаються у зовнішніх сховищах.

Telegram-бот у процесі роботи може використовувати Google Drive API для зберігання або отримання зображень, а також API Midjourney для генерації зображень за текстовим запитом. Для інтеграції з Google Drive API доцільно застосувати протокол OAuth 2.0- бот діє як довірений клієнт, отримуючи від користувача або розробника маркер доступу (access token) з обмеженими правами. Кожен API-запит до Google Drive має включати дійсний токен у заголовку авторизації, який перевіряється серверами Google. Це гарантує, що бот може оперувати лише тими файлами, доступ до яких був наданий.

Для Midjourney API або схожого сервісу генерації зображень, доступ зазвичай захищається унікальним ключем або токеном сесії. Бот повинен зберігати цей ключ конфіденційно і відправляти його тільки через захищені канали (наприклад, HTTPS) при виклику API. Важливо також контролювати, що запити до Midjourney надходять тільки від бота- якщо API Midjourney підтримує валідацію підпису або секрету в запиті, слід скористатися цим механізмом. Таким чином, доступ до зовнішніх сервісів із боку бота ґрунтується на принципі мінімально необхідних привілеїв- бот автентифікується перед API лише

валідними обліковими даними, а обсяг його прав суворо обмежений налаштуваннями OAuth/API ключа.

Усі облікові дані, що використовуються ботом – токен доступу до Telegram Bot API, секретні ключі для цифрового підпису, OAuth-токени Google, API-ключі Midjourney – зберігаються в захищений спосіб. Натомість застосовується механізм змінних середовища або захищених конфігурацій. Доступ до цих змінних є лише у процесу бота, а права доступу на сервері налаштовуються так, щоб сторонні користувачі не могли їх читати.

Передача даних між користувачем і ботом здійснюється через інфраструктуру Telegram, яка шифрує трафік між клієнтом і сервером (протокол TLS) і використовує власний протокол з шифруванням між сервером Telegram і ботом. Це означає, що переглянути або підробити повідомлення на етапі передачі сторонній зловмисник не зможе (за умови використання офіційного API Telegram). Однак слід врахувати, що звичайні чати з ботом не є наскрізно зашифрованими для кінцевих користувачів, тож Telegram як платформа теоретично має доступ до переданого контенту. Це потрібно мати на увазі при обробці конфіденційних зображень – можливо, користувачу варто рекомендувати не надсилати ботам матеріали, які він не довірив би самому Telegram. З боку сервера бота, слід використовувати Webhook з HTTPS або хмарне середовище, що гарантує шифрування каналу Bot API. У налаштуваннях вебхука можна додатково задати секретний токен в URL, щоб підтверджувати достовірність запитів від Telegram.

ВИСНОВКИ

У магістерській роботі вирішено актуальне завдання створення інтегрованої системи генерації, обробки та криптографічного захисту зображень, що базується на поєднанні Telegram-бота з генеративною моделлю Midjourney AI.

У результаті проведеного дослідження отримано такі основні результати:

1. Проведено аналіз сучасних технологій генерації та обробки зображень за допомогою штучного інтелекту. Визначено, що Midjourney входить до найпотужніших моделей text-to-image і забезпечує високий рівень деталізації та художньої якості зображень.

2. Досліджено підходи до інтеграції Telegram-ботів із системами AI та визначено оптимальну архітектуру для побудови системи генерації та обробки зображень у реальному часі. Обґрунтовано вибір .NET як базової технологічної платформи.

3. Проаналізовано криптографічні механізми, включно з алгоритмами хешування, електронного підпису та стеганографічного вбудовування даних, що можуть бути застосовані для забезпечення автентичності графічної інформації.

4. Розроблено алгоритм RSA-EMBED, який забезпечує створення та вбудовування цифрового підпису безпосередньо у файл зображення. Показано, що цей алгоритм ефективно виявляє модифікації та гарантує автентичність цифрового контенту.

5. Створено Telegram-бота, який виконує генерацію зображень через Midjourney, обробку результатів, криптографічне підписання й перевірку достовірності зображень, а також інтегрується з Google Drive для хмарного зберігання даних.

6. Проведено експериментальне тестування системи, яке підтвердило коректність роботи алгоритмів, стійкість RSA-EMBED до підміни даних та ефективність інтеграційного рішення в умовах реального використання.

Отже, поставлена мета магістерської роботи досягнута повністю, а отримані результати мають практичну цінність та можуть бути використані у системах захисту цифрових матеріалів, бот-платформах, сервісах обробки зображень та інформаційних системах, що потребують криптографічного забезпечення достовірності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Батьківська К., Дзівак О., Кулина С. Порівняльний аналіз криптографічних методів захисту зображень у сервісах генерації контенту // *Інноваційні підходи до розвитку технологій та економіки (IADTE 2025)*. – Свалява: ЗУНУ, 2025. – С. 209–211.
2. Батьківська К., Кулина С. Методи виявлення підроблених або змінених зображень із застосуванням криптографічних хеш-функцій // *Кібербезпека та комп'ютерно-інтегровані технології (КБКІТ–2025)*. – Тернопіль, 2025. – С. 118–120.
3. Telegram Bot API Documentation. Режим доступу: https://core.telegram.org/bots/api
4. Microsoft. System.Security.Cryptography Namespace. Режим доступу: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography
5. APIFRAME Documentation. Midjourney Image Generation API. Режим доступу: https://apiframe.pro/docs/midjourney
6. Midjourney Official Guide. Prompting and Parameters. Режим доступу: https://docs.midjourney.com
7. Hammond C. Building a Telegram Chatbot in .NET 6 with OpenAI API. Режим доступу: https://chrishammond.dev/blog
8. OpenAI API Reference. Режим доступу: https://platform.openai.com/docs/api-reference
9. RSA Digital Signature Implementation in C#. Microsoft Learn. Режим доступу: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.rsa.signdata
10. ECDsa Class Documentation. Microsoft Learn. Режим доступу: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.ecdsa

11. BouncyCastle Cryptography Library for C#. Режим доступу: https://www.bouncycastle.org/csharp

12. MetadataExtractor for .NET. Режим доступу: https://github.com/drewnoakes/metadata-extractor-dotnet

13. OpenStego Project. Режим доступу: https://www.openstego.com

14. Coalition for Content Provenance and Authenticity (C2PA). Режим доступу: https://c2pa.org

15. Telegram MTProto Protocol Overview. Режим доступу: https://core.telegram.org/mtproto

16. OWASP Secure Software Development Lifecycle Guide. Режим доступу: https://owasp.org

17. Stallings W. *Cryptography and Network Security: Principles and Practice.* – Pearson, 2023.

18. Menezes A., van Oorschot P., Vanstone S. *Handbook of Applied Cryptography.* – CRC Press, 2018.

19. Goodfellow I., Bengio Y., Courville A. *Deep Learning.* – MIT Press, 2016.

20. Cloudflare. What Is Public Key Cryptography? Режим доступу: https://www.cloudflare.com/learning/cryptography

21. NIST CSRC. Hash Functions Overview. Режим доступу: https://csrc.nist.gov/projects/hash-functions

22. OWASP Cryptographic Storage Cheat Sheet. Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

23. SSL.com. Introduction to Digital Certificates. Режим доступу: https://www.ssl.com/faqs/what-is-a-digital-certificate

24. SixLabors. ImageSharp Documentation. Режим доступу: https://github.com/SixLabors/ImageSharp

25. Stanford University. *Applied Cryptography Course Notes.* Режим доступу: https://crypto.stanford.edu/notes/

26. Liu S., Sun W. Image Encryption Using Advanced Cryptographic Techniques // *IEEE Access.* – 2022. – Vol. 10. – P. 118450–118462. DOI: https://doi.org/10.1109/ACCESS.2022.3219876

27. Huang Y., Shi Y. Digital Image Integrity Verification Using Hybrid Hash Functions // *ACM Transactions on Multimedia Computing, Communications, and Applications.* – 2020. – Vol. 16, № 3. – P. 1–22. DOI: https://doi.org/10.1145/3391961

28. Roy A., Banerjee S. A Study on Secure Image Transmission Using AES Variants // *IEEE Transactions on Multimedia Systems.* – 2021. – Vol. 23, № 4. – P. 1058–1068. DOI: https://doi.org/10.1109/TMM.2020.3041123

29. Singh A., Sharma M. A Survey on Image Authentication Methods // *Journal of Information Security and Applications.* – Elsevier, 2021. – Vol. 58. – P. 102817. DOI: https://doi.org/10.1016/j.jisa.2021.102817

30. Wang L., Zhao H. Cryptographic Approaches for Image Security in AI-driven Systems // *Proceedings of the ICCT 2023.* – Beijing, 2023. – P. 455–462. DOI: https://doi.org/10.1109/ICCT57878.2023.00082

31. Patel R. Steganographic and Cryptographic Techniques for Image Protection // *Proceedings of the 2020 13th International Conference on Computer Science and Information Technology (ICCSIT).* – 2020. – P. 94–101. Режим доступу: https://dl.acm.org/doi/10.1145/3447682.3447701

32. National Institute of Standards and Technology. *FIPS PUB 180-4: Secure Hash Standard (SHS).* – Gaithersburg, MD: NIST, 2015. Режим доступу: https://csrc.nist.gov/publications/detail/fips/180/4/final

33. National Institute of Standards and Technology. *FIPS PUB 186-5: Digital Signature Standard (DSS).* – NIST, 2023. Режим доступу: https://csrc.nist.gov/publications/detail/fips/186/5/final

34. ISO/IEC 14888-3:2018. *Information technology — Security techniques — Digital signatures with appendix — Part 3: Mechanisms using a hash-function.* Режим доступу: https://www.iso.org/standard/76382.html

35. ISO/IEC 9797-1:2011. *Information technology — Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher.* Режим доступу: https://www.iso.org/standard/50379.html

Додаток А

Програмні коди реалізації алгоритму цифрового підпису забезпечення автентичності зображень

```
using ImageGenerationWebApi.Abstractions;
using System.Security.Cryptography;
using System.Text;

namespace ImageGenerationWebApi.Services
{

public class EmbeddedRsaPngSigner : IEmbeddedImageSigner
   {
      private readonly IUserKeyStore _keyStore;
      private static readonly byte[] Magic = Encoding.ASCII.GetBytes("SIGPNGMIDJOURNEY");

      public string Algorithm => "RSA-EMBED";

      public EmbeddedRsaPngSigner(IUserKeyStore keyStore)
      {
         _keyStore = keyStore;
      }

      public byte[] SignAndEmbed(byte[] pngBytes, long chatId, long userId)
      {
         if (pngBytes == null || pngBytes.Length == 0)
            throw new ArgumentException("PNG bytes are empty", nameof(pngBytes));

         using var rsa = _keyStore.GetOrCreateRsa(chatId, userId);

         byte[] payload;
         using (var sha = SHA256.Create())
         {
            payload = sha.ComputeHash(Combine(
               pngBytes,
               BitConverter.GetBytes(chatId),
               BitConverter.GetBytes(userId)
            ));
         }
         var signature = rsa.SignData(payload, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);

         var lengthBytes = BitConverter.GetBytes(signature.Length);
         var result = new byte[pngBytes.Length + Magic.Length + lengthBytes.Length + signature.Length];

         Buffer.BlockCopy(pngBytes, 0, result, 0, pngBytes.Length);
         Buffer.BlockCopy(Magic, 0, result, pngBytes.Length, Magic.Length);
         Buffer.BlockCopy(lengthBytes, 0, result, pngBytes.Length + Magic.Length, lengthBytes.Length);
         Buffer.BlockCopy(signature, 0, result, pngBytes.Length + Magic.Length + lengthBytes.Length,
signature.Length);
         return result;
      }
      public bool VerifyEmbedded(byte[] signedPngBytes, long chatId, long userId)
      {
         if (signedPngBytes == null || signedPngBytes.Length == 0)
```

```
        return false;
    int magicIndex = LastIndexOf(signedPngBytes, Magic);
    if (magicIndex < 0)
        return false;
    int lengthIndex = magicIndex + Magic.Length;
    if (lengthIndex + sizeof(int) > signedPngBytes.Length)
        return false;
    int sigLength = BitConverter.ToInt32(signedPngBytes, lengthIndex);
    if (sigLength <= 0)
        return false;
    int sigIndex = lengthIndex + sizeof(int);
    if (sigIndex + sigLength > signedPngBytes.Length)
        return false;
    var signature = new byte[sigLength];
    Buffer.BlockCopy(signedPngBytes, sigIndex, signature, 0, sigLength);
    var originalPng = new byte[magicIndex];
    Buffer.BlockCopy(signedPngBytes, 0, originalPng, 0, magicIndex);

    using var rsa = _keyStore.GetOrCreateRsa(chatId, userId);

    byte[] payload;
    using (var sha = SHA256.Create())
    {
        payload = sha.ComputeHash(Combine(
            originalPng,
            BitConverter.GetBytes(chatId),
            BitConverter.GetBytes(userId)
        ));
    }

    return rsa.VerifyData(payload, signature, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}


private static byte[] Combine(params byte[][] arrays)
{
    var length = arrays.Sum(a => a.Length);
    var result = new byte[length];
    int offset = 0;

    foreach (var a in arrays)
    {
        Buffer.BlockCopy(a, 0, result, offset, a.Length);
        offset += a.Length;
    }

    return result;
}

private static int LastIndexOf(byte[] source, byte[] pattern)
{
    if (pattern.Length == 0 || source.Length == 0 || pattern.Length > source.Length)
        return -1;

    for (int i = source.Length - pattern.Length; i >= 0; i--)
    {
        bool match = true;
        for (int j = 0; j < pattern.Length; j++)
```

```
            {
                if (source[i + j] != pattern[j])
                {
                    match = false;
                    break;
                }
            }

            if (match)
                return i;
        }

        return -1;
    }

public class CryptographyService
    {
        private readonly Dictionary<string, IImageSigner> _signers;
        private readonly Dictionary<string, IEmbeddedImageSigner> _embeddedSigners;
        public readonly IMetricSink _metricSink;

        public CryptographyService()
        {
            IUserKeyStore keyStore = new InMemoryUserKeyStore();

            var standardSigners = new IImageSigner[]
            {
                new RsaImageSigner(keyStore),
                new DsaImageSigner(keyStore),
                new EcdsaImageSigner(keyStore)
            };

            var embeddedSigners = new IEmbeddedImageSigner[]
            {
                new EmbeddedRsaPngSigner(keyStore)
            };

            _signers = standardSigners.ToDictionary(s => s.Algorithm, s => s, StringComparer.OrdinalIgnoreCase);
            _embeddedSigners = embeddedSigners.ToDictionary(s => s.Algorithm, s => s,
StringComparer.OrdinalIgnoreCase);
            _metricSink = new InMemoryMetricSink();
        }

        public ImageSignatureDto Sign(byte[] imageBytes, long chatId, long userId, string algorithm)
        {
            if (!_signers.TryGetValue(algorithm, out var signer))
                throw new ArgumentException($"Unknown algorithm: {algorithm}", nameof(algorithm));

            var sw = Stopwatch.StartNew();
            var signature = signer.Sign(imageBytes, chatId, userId);
            sw.Stop();

            _metricSink.Record(new SignatureMetric(
                Algorithm: signer.Algorithm,
                OperationType: SignatureOperationType.Sign,
                ChatId: chatId,
                UserId: userId,
                ImageSizeBytes: imageBytes.Length,
```

```
        SignedImageSizeBytes: null,
        SignatureSizeBytes: signature.Length,
        ElapsedMilliseconds: sw.ElapsedMilliseconds,
        TimestampUtc: DateTime.UtcNow
    ));

    return new ImageSignatureDto(signer.Algorithm, signature);
}

public bool Verify(byte[] imageBytes, long chatId, long userId, ImageSignatureDto dto)
{
    if (!_signers.TryGetValue(dto.Algorithm, out var signer))
        return false;

    var sw = Stopwatch.StartNew();
    bool ok = signer.Verify(imageBytes, chatId, userId, dto.Signature);
    sw.Stop();

    _metricSink.Record(new SignatureMetric(
        Algorithm: signer.Algorithm,
        OperationType: SignatureOperationType.Verify,
        ChatId: chatId,
        UserId: userId,
        ImageSizeBytes: imageBytes.Length,
        SignedImageSizeBytes: null,
        SignatureSizeBytes: dto.Signature.Length,
        ElapsedMilliseconds: sw.ElapsedMilliseconds,
        TimestampUtc: DateTime.UtcNow
    ));

    return ok;
}

public bool VerifyTryAll(byte[] imageBytes, long chatId, long userId, byte[] signature)
{
    foreach (var signer in _signers.Values)
    {
        var sw = Stopwatch.StartNew();
        bool ok = signer.Verify(imageBytes, chatId, userId, signature);
        sw.Stop();

        _metricSink.Record(new SignatureMetric(
            Algorithm: signer.Algorithm,
            OperationType: SignatureOperationType.Verify,
            ChatId: chatId,
            UserId: userId,
            ImageSizeBytes: imageBytes.Length,
            SignedImageSizeBytes: null,
            SignatureSizeBytes: signature.Length,
            ElapsedMilliseconds: sw.ElapsedMilliseconds,
            TimestampUtc: DateTime.UtcNow
        ));

        if (ok)
            return true;
    }

    return false;
```

```
}

public byte[] SignAndEmbed(byte[] pngBytes, long chatId, long userId, string algorithm)
{
   if (!_embeddedSigners.TryGetValue(algorithm, out var signer))
      throw new ArgumentException($"Unknown embedded algorithm: {algorithm}", nameof(algorithm));

   var sw = Stopwatch.StartNew();
   var signedPng = signer.SignAndEmbed(pngBytes, chatId, userId);
   sw.Stop();
   _metricSink.Record(new SignatureMetric(
      Algorithm: signer.Algorithm,
      OperationType: SignatureOperationType.SignEmbedded,
      ChatId: chatId,
      UserId: userId,
      ImageSizeBytes: pngBytes.Length,
      SignedImageSizeBytes: signedPng.Length,
      SignatureSizeBytes: signedPng.Length - pngBytes.Length,
      ElapsedMilliseconds: sw.ElapsedMilliseconds,
      TimestampUtc: DateTime.UtcNow
   ));
   return signedPng;
}
public bool VerifyEmbedded(byte[] signedPngBytes, long chatId, long userId, string algorithm)
{
   if (!_embeddedSigners.TryGetValue(algorithm, out var signer))
      return false;
   var sw = Stopwatch.StartNew();
   bool ok = signer.VerifyEmbedded(signedPngBytes, chatId, userId);
   sw.Stop();
   _metricSink.Record(new SignatureMetric(
      Algorithm: signer.Algorithm,
      OperationType: SignatureOperationType.VerifyEmbedded,
      ChatId: chatId,
      UserId: userId,
      ImageSizeBytes: signedPngBytes.Length,
      SignedImageSizeBytes: signedPngBytes.Length,
      SignatureSizeBytes: null,
      ElapsedMilliseconds: sw.ElapsedMilliseconds,
      TimestampUtc: DateTime.UtcNow
   ));
   return ok;
}
```

Додаток Б

Копії публікацій

ЗБІРНИК ТЕЗ ДОПОВІДЕЙ

ІІ ВСЕУКРАЇНСЬКОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ “ІННОВАЦІЙНІ ПІДХОДИ ДО РОЗВИТКУ ТЕХНОЛОГІЙ ТА ЕКОНОМІКИ”

ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Innovation and science

ЗАКАРПАТСЬКИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЗУНУ

Свалява - 2025

ПОЧЕСНІ ГОЛОВИ КОНФЕРЕНЦІЇ

Микола ДИВАК, д. т. н., професор, проректор з наукової роботи ЗУНУ

СПІВГОЛОВИ КОНФЕРЕНЦІЇ

Едіта ГРАБАР, кандидат педагогічних наук, доцент, директор Закарпатського навчально-наукового центру ЗУНУ
Леся ДУБЧАК, кандидат технічних наук, доцент, завідувач кафедри комп'ютерної інженерії ЗУНУ

ПРОГРАМНИЙ КОМІТЕТ

Мар'ян ТРІПАК, доктор економічних наук, професор кафедри інституту, ректор НРЗВО "Кам'янець-Подільський державний інститут"
Оксана ГОМОТЮК, доктор історичних наук, професор, декан соціально-гуманітарного факультету ЗУНУ
Андрій КОЦУР, к.е.н., доцент, декан факультету економіки та управління
Ігор ЯКИМЕНКО, к.т.н., доцент, декан факультету комп'ютерних інформаційних технологій
Андрій КІЗИМА, к.е.н., доцент, декан факультету фінансів та обліку
Святослав ПИТЕЛЬ, к.е.н., доцент, директор навчально-наукового інституту новітніх освітніх технологій
Василь БРИЧ, д.е.н., професор, директор навчально-наукового інституту інноватики, природокористування та інфраструктури
Віктор РУСІН, к.е.н., доцент, директор Навчально-наукового інституту публічного управління

ОРГАНІЗАЦІЙНИЙ КОМІТЕТ

Олег ПІЦУН - к.т.н., доцент, доцент кафедри комп'ютерної інженерії ЗУНУ
Віктор АНТОНЮК - к.н. з держ.упр., заступник директора з наукової роботи Закарпатського навчально-науково центру
Михайло ШКІЛЬНЯК - д. е. н., професор, завідувач кафедри менеджменту, публічного управління та персоналу
Андрій КРИСОВАТИЙ, доктор економічних наук, професор, завідувач кафедри фінансів ім. С. І. Юрія
Павло ПОПОВИЧ - д.т.н., професор, завідувач кафедри транспорту і логістики
Оксана ТУЛАЙ - д.е.н., професор, завідувач кафедри міжнародних відносин та дипломатії
Оксана МАРУХЛЕНКО - д.н. з держ.упр., завідувач кафедри управління факультету економіки та управління Київського столичного університету імені Бориса Грінченка
Олександр МОЛНАР - к.е.н., доцент, завідувач кафедри економіки, підприємництва та торгівлі економічного факультету ДВНЗ «УжНУ»
Андрій ГІРНЯК - д.пс.н., професор, завідувач кафедри психології та соціальної роботи
Тетяна ПОСПЄЛОВА - д.н. з держ.упр., доцент, професор кафедри управління факультету економіки та управління Київського столичного університету імені Бориса Грінченка
Аліна ПОМАЗА-ПОНОМАРЕНКО - д.н. з держ. управління, завідувач науково-дослідної лабораторії з дослідження проблем управління у сфері цивільного захисту Національного університету цивільного захисту України (м. Харків)
Вікторія ШВЕДУН - д.н. з держ. управління, професор, професор кафедри економіки та публічного управління Національного аерокосмічного університету «Харківський авіаційний інститут» (м. Харків
Михайло ГАЗУДА - д.е.н., професор, кафедра економіки, підприємництва та торгівлі економічного факультету ДВНЗ «УжНУ»
Вікторія ГОТРА - д.е.н., професор, кафедра економіки, підприємництва та торгівлі економічного факультету ДВНЗ «УжНУ»
Андрій ДЕРЛИЦЯ - к.е.н., в.о. проректора з наукової та міжнародної діяльності НРЗВО "Кам'янець-Подільський державний інститут"
Ірина СИДОР - к.е.н., доцент, доцент кафедри фінансів ім.С.І.Юрія
Руслан РОЗУМ - к.т.н., доцент, доцент кафедри транспорту і логістики
Андрій ВІТРОВИЙ - к.т.н., доцент, доцент кафедри економічної експертизи та землевпорядкування
Ольга СТОЛЯРЕНКО - к.пс.н., доцент, завідувач кафедри соціальної роботи, психології та соціокультурної діяльності ім. Т.Сосновської
Фаєрчук Вадим, Галунька Богдан, Лисик Марія, Шпак Марта, Петришин Наталія - студенти ФКІТ ЗУНУ

8. Розсіхін В., Бабічев О., Марукленко О., Кравчук О., Штікун О. Розвиток територіальних комунікацій 'потужний, як factor of socio-ecological development of territories // Cuestiones Políticas. 2023. Т. 41 № 77. С. 205-226. DOI: https://doi.org/10.46398/cuestpol.4177.14

9. Шевченка М., Марукленко О., Трач О., Шведенко П., Дубович О. Використання аналітики даних у публічному управлінні для запобігання корупції в умовах гібридної війни // Pakistan Journal of Criminology. 2024. Т. 16 № 2. С. 943-958. DOI: https://doi.org/10.62271/pjc.16.2.943.958

10. Державна служба статистики України. Регіональні статистичні дані. 2023. URL: http://www.ukrstat.gov.ua

11. Pospelova T., Maruhlenko O., Rudenko V. Компетентнісний підхід у державному регулюванні соціальної сфери: парадигмальні зрушення в контексті соціально-економічної політики // Європейський науковий журнал Економічних та Фінансових інновацій. 2025. № 1 (15). З. 309-318. URL: https://journal.eae.com.ua/index.php/journal/article/view/415

ПОРІВНЯЛЬНИЙ АНАЛІЗ КРИПТОГРАФІЧНИХ МЕТОДІВ ЗАХИСТУ ЗОБРАЖЕНЬ У СЕРВІСАХ ГЕНЕРАЦІЇ КОНТЕНТУ

Кулина Сергій
PhD, доцент кафедри кібербезпеки
sersks@wunu.edu.ua
Батьківська Катерина
здобувач вищої освіти, гр. КБм-11
katerynabatkivska@gmail.com
Західноукраїнський національний університет
Олександр Дзівак
аспірант
ole.dzvk@gmail.com
Західноукраїнський національний університет

У сучасному цифровому світі стрімко зростає попит на сервіси генерації візуального контенту з використанням штучного інтелекту. Такі сервіси, як Midjourney, DALL·E, Stable Diffusion та інші, дозволяють створювати високоякісні зображення на основі текстових запитів користувачів [1]. З одночасним розширенням функціональності та масовим впровадженням таких інструментів зростає також обсяг оброблюваної та переданої інформації, яка часто містить персональні або конфіденційні дані.

Питання безпечної передачі та зберігання зображень, згенерованих штучним інтелектом, стає особливо актуальним, особливо коли йдеться про використання таких сервісів у поєднанні з месенджерами, наприклад, ботами Telegram. Ці боти виступають проміжною ланкою між користувачем та сервісом ШІ, забезпечуючи зручний інтерфейс, але водночас створюючи нові ризики витоку або перехоплення даних.

Саме тому використання методів криптографічного захисту є критично важливим для забезпечення конфіденційності, цілісності та автентичності переданого контенту.

Існує кілька підходів до захисту зображень, найпоширенішими з яких є:

- симетричне шифрування (AES, DES, Blowfish);
- асиметричне шифрування (RSA, ECC);
- гібридні методи (поєднання симетричних та асиметричних алгоритмів);
- стеганографічні методи (приховування інформації в зображеннях);
- хешування та цифрові підписи (SHA-2, SHA-3, HMAC).

Кожен із них має свої переваги та недоліки. Симетричні алгоритми добре підходять для швидкого шифрування великих обсягів даних (наприклад, файлів PNG або JPG), але вимагають надійного обміну ключами. Асиметричні

алгоритми, навпаки, забезпечують високу безпеку під час передачі, але працюють повільніше та менш ефективно для великих обсягів.

Telegram-боти є надзвичайно популярним інтерфейсом для взаємодії зі AI-сервісами, оскільки вони забезпечують зручний доступ до інструментів через мобільні пристрої та настільні комп'ютери [2].

Однак, передача зображень через API Telegram створює певні вразливості:

- зображення зберігаються на серверах Telegram;
- механізми Telegram не гарантують наскрізного шифрування для ботів;
- трафік HTTP/HTTPS може бути перехоплений, якщо проксі-сервер або API неправильно налаштовано.

Відповідно, для мінімізації таких ризиків у ботах Telegram реалізовують наступні механізми [3]:

- локальний захист зображень на стороні бота перед надсиланням користувачеві;
- тимчасове зберігання файлів з обмеженим терміном дії (TTL);
- створення зашифрованих посилань з одноразовим доступом;

У таблиці 1 представлено порівняння поширених криптографічних методів, що інтегруються в телеграм боти.

Таблиця 1

Порівняння поширених криптографічних методів

Метод	Безпека	Швидкодія	Складність реалізації	Підходить для Telegram-бота
AES	Висока	Висока	Низька	Так
RSA	Висока	Низька	Середня	Частково (для передачі ключів)
ECC	Висока	Середня	Висока	Так
SHA-256	Середня	Висока	Низька	Так (для перевірки цілісності)
Стеганографія	Низька	Середня	Висока	Ні (небезпечна при стисненні зображень)

Як бачимо з таблиці 1, найкращим рішенням для Telegram-ботів є використання AES у поєднанні з RSA або ECC для обміну ключами, а також хешування для перевірки цілісності зображення [4].

У випадку використання Telegram-бота, інтегрованого з Midjourney, користувач надсилає текстовий запит боту. Бот, у свою чергу, передає запит до API Midjourney, отримує згенероване зображення та шифрує його за допомогою AES [5]. Ключ AES генерується для кожної сесії та шифрується відкритим ключем користувача (RSA або ECC). Таким чином, зображення може бути розшифровано лише на стороні користувача.

Додатково, бот генерує SHA-256 хеш зображення, який зберігається окремо або надсилається разом з файлом для перевірки цілісності. У роботі проаналізовано сучасні криптографічні методи, які можна використовувати для

ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»
ГАЛИЦЬКИЙ ФАХОВИЙ КОЛЕДЖ ІМ. В'ЯЧЕСЛАВА ЧОРНОВОЛА

КІБЕРБЕЗПЕКА ТА КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ (КБКІТ – 2025)

науково-практична конференція
молодих вчених, аспірантів та студентів

28–29 серпня 2025
Тернопіль

Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів «Кібербезпека та комп'ютерно-інтегровані технології» (КБКІТ - 2025), Тернопіль, 2025. - 154 с.

Редакційна колегія:

Василь ЯЦКІВ – доктор технічних наук, професор, завідувач кафедри кібербезпеки, Західноукраїнський національний університет.

Михайло КАСЯНЧУК – доктор технічних наук, професор, професор кафедри кібербезпеки, Західноукраїнський національний університет.

Ігор ЯКИМЕНКО – кандидат технічних наук, доцент, декан факультету комп'ютерних інформаційних технологій, Західноукраїнський національний університет.

Лідія ТИМОШЕНКО – кандидат економічних наук, доцент, завідувач кафедри кібербезпеки та програмного забезпечення, Національний університет «Одеська політехніка».

Наталя СТЕФУРАК – кандидат фізико-математичних наук, завідувач відділенням комп'ютерних технологій, Галицький фаховий коледж ім. В'ячеслава Чорновола.

Наталія ЯЦКІВ – кандидат технічних наук, доцент, доцент кафедри спеціалізованих комп'ютерних систем, Західноукраїнський національний університет.

Степан ІВАСЬЄВ – кандидат технічних наук, доцент, доцент кафедри кібербезпеки, Західноукраїнський національний університет.

Тарас ЦАВОЛИК – кандидат технічних наук, доцент, доцент кафедри кібербезпеки, Західноукраїнський національний університет.

Людмила БАБАЛА – кандидат економічних наук, доцент, доцент кафедри кібербезпеки, Західноукраїнський національний університет.

Сергій КУЛИНА – PhD, доцент кафедри кібербезпеки, Західноукраїнський національний університет.

Ігор ІГНАТЄВ – викладач кафедри кібербезпеки, Західноукраїнський національний університет.

Аліна ДАВЛЕТОВА – викладач кафедри кібербезпеки, Західноукраїнський національний університет.

Головний редактор: Михайло КАСЯНЧУК
Технічний редактор: Аліна ДАВЛЕТОВА

Адреса редакції:
Західноукраїнський національний університет, кафедра кібербезпеки,
вул. Олени Теліги 8, м. Тернопіль 46003
Контакти:
*e-mail:*conferencekb@gmail.com

КРИПТОГРАФІЧНІ МЕТОДИ ЗАХИСТУ ІНФОРМАЦІЇ

УДК 004

Катерина БАТЬКІВСЬКА, Сергій КУЛИНА

Західноукраїнський національний університет

МЕТОДИ ВИЯВЛЕННЯ ПІДРОБЛЕНИХ АБО ЗМІНЕНИХ ЗОБРАЖЕНЬ ІЗ ЗАСТОСУВАННЯМ КРИПТОГРАФІЧНИХ ХЕШ-ФУНКЦІЙ

Вступ. У цифрову епоху зображення стали одним із ключових носіїв інформації, що використовуються в медіа, освіті, електронній комерції, державних установах та правовій системі. З розвитком технологій редагування та створення візуального контенту (зокрема DeepFake, GAN) значно зросла кількість підроблених зображень, які можуть бути використані для дезінформації, фальсифікації доказів або маніпуляції громадською думкою [1]. Це зумовлює необхідність створення надійних методів перевірки автентичності цифрових файлів. Одним із найефективніших і водночас простих інструментів є криптографічні хеш-функції, які забезпечують контроль цілісності та виявлення будь-яких змін у зображенні [2].

Мета: дослідження методів виявлення підроблених або змінених зображень із застосуванням криптографічних хеш-функцій.

1. Принципи роботи криптографічні хеш-функції

Криптографічна хеш-функція перетворює будь-яку кількість даних на короткий, унікальний ідентифікатор - хеш-код. Якщо змінити навіть один піксель, колір або метадані (EXIF) у файлі, нове хеш-значення буде повністю відрізнятися від оригіналу. Ця властивість, відома як ефект лавини, дозволяє надійно виявляти навіть незначні підробки.

Основні вимоги до безпечної хеш-функції:

– односторонність - неможливо відновити вихідні дані з хешу;

– стійкість до колізій - важко знайти два різні файли з однаковим хешем;

– висока продуктивність - швидке обчислення хешів для великої кількості зображень.

Серед сучасних алгоритмів найпоширенішими є MD5, SHA-1, SHA-256 та SHA-3 [3]. Однак, MD5 та SHA-1 більше не вважаються безпечними через доведені колізії. На противагу їм алгоритми сімейства SHA-2 (зокрема SHA-256) забезпечують високу стійкість до атак, а SHA-3 забезпечує ще вищий рівень безпеки завдяки принципу губчастої конструкції.

2. Практичні аспекти виявлення змінених зображень

Для перевірки автентичності зображення використовується наступний алгоритм дій:

– обчислення хешу оригінального файлу (SHA-256 або SHA-3);

– збереження цього хешу в безпечному сховищі або базі даних;

– під час повторної перевірки, обчислення нового хешу та порівняння його з оригіналом;

– якщо хеші не збігаються, зображення було змінено або пошкоджено.

Графічно алгоритм зображено на рисунку 1.

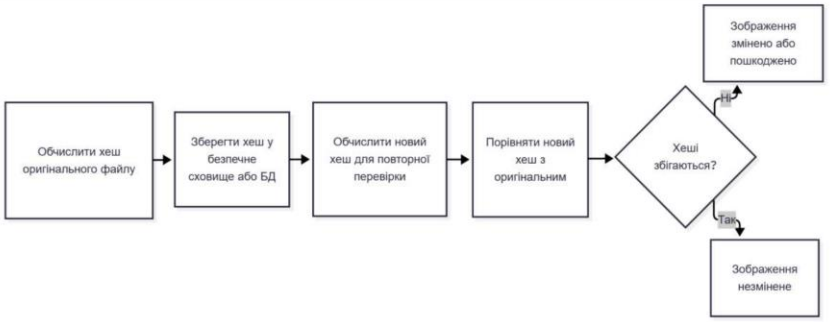


Рисунок 1 - Алгоритм перевірки автентичності зображення

Алгоритм, представлений на рисунку 1, дозволяє легко виявити будь-які спроби несанкціонованого редагування, навіть якщо зміни невидимі для людського ока. Для підвищення рівня безпеки система може бути доповнена цифровими підписами (DSA або RSA), які гарантують автентичність автора, а також використанням стеганографічних водяних знаків, які приховують хеш безпосередньо в піксельних даних зображення [4].

3. Порівняльний аналіз алгоритмів хешування

MD5 та SHA-1 – це ранні криптографічні хеш-функції, але зараз вважаються застарілими через відомі вразливості до колізій. MD5 забезпечує високу швидкість, але низький рівень безпеки, і більше підходить для неформального контролю даних, ніж для захисту. SHA-1 дещо надійніший, але також більше не рекомендується для критично важливих застосувань.

На противагу цьому, SHA-256, який є частиною сімейства SHA-2, забезпечує значно вищу безпеку, зберігаючи при цьому хорошу продуктивність. Він широко використовується для перевірки цілісності файлів, цифрових підписів та в блокчейн-системах. Найновіша – SHA-3, яка базується на принципово іншій конструкції (Keccak). Вона демонструє дуже високу стійкість до атак і використовується в критично важливих інформаційних системах, зокрема для захисту цифрових доказів та в урядових криптографічних стандартах. На відміну від SHA-2, який використовує класичну схему Меркла-Дамгарда, SHA-3 реалізує губкоподібну (sponge) конструкцію, що дозволяє гнучко налаштовувати параметри безпеки та ефективно працювати в умовах обмежених ресурсів. Обидва алгоритми підтримуються сучасними криптографічними протоколами (TLS, PGP, S/MIME) і рекомендовані до використання такими організаціями, як NIST та ISO. В умовах зростаючих загроз цифровій безпеці вони забезпечують надійний фундамент для побудови систем автентифікації, верифікації цифрового контенту та зберігання доказів у юридичній практиці.

Алгоритми SHA-256 та SHA-3 вважаються сучасним стандартом криптографічного хешування завдяки їхній високій стійкості до криптоаналітичних атак, включаючи колізії та атаки на основі прообразів, що робить їх надійним інструментом для забезпечення цілісності файлів, автентифікації цифрових повідомлень, захисту електронних підписів, а також для використання в технологіях блокчейн та цифровій криміналістиці, де цілісність даних є критично важливою.

Таблиця 1 - Порівняльний аналіз алгоритмів хешування

Алгоритм	Довжина хешу (біт)	Рівень безпеки	Стійкість до колізій	Швидкодія	Застосування
MD5	128	Низький	Уразливий	Висока	Лише базова перевірка
SHA-1	160	Середній	Часткова	Висока	Обмежене використання
SHA-256	256	Високий	Висока	Середня	Перевірка цілісності файлів
SHA-3	256/512	Дуже високий	Дуже висока	Середня	Критичні системи, захист доказів

Як видно з таблиці 1, незважаючи на високу продуктивність алгоритм MD5 є повністю скомпрометованим алгоритмом, оскільки для нього вже давно виявлені колізії, що робить його непридатним для використання в системах безпеки. SHA-1, хоча й демонструє кращу стабільність, також не рекомендується для використання в сучасних рішеннях через часткові колізії, виявлені у 2017 році. Алгоритм SHA-256 забезпечує оптимальний баланс між продуктивністю та безпекою: він ефективно працює з великими зображеннями у форматах PNG та JPG, має високу стійкість до колізій та є фактичним стандартом для перевірки цілісності файлів у більшості операційних систем та мережевих протоколів [5].

SHA-3 характеризується підвищеною стійкістю до криптоаналітичних атак завдяки іншій структурі (модель губки). Доцільно використовувати його в системах, де довгострокова надійність є критично важливою, наприклад, у судових реєстрах, блокчейнах або при зберіганні цифрових доказів. Таким чином, SHA-256 можна вважати оптимальним рішенням для практичного контролю цілісності зображень, а SHA-3 – стратегічним вибором для майбутніх систем, орієнтованих на підвищення довіри та юридичної значущості цифрових даних[6].

Висновок. Аналіз підтверджує ефективність криптографічних хеш-функцій як універсального механізму виявлення підроблених або змінених зображень. Алгоритми SHA-256 та SHA-3 забезпечують оптимальне поєднання стійкості до колізій, продуктивності та захисту від сучасних атак. Їх інтеграція з цифровими підписами та технологіями блокчейн створює новий рівень безпеки для підтримки довіри до цифрових матеріалів у медіа, правовій та технічній сферах.

Перелік використаних джерел.

1. Verdoliva L. Media Forensics and DeepFakes: An Overview. IEEE Journal of Selected Topics in Signal Processing, 2020, Vol. 14, No. 5, pp. 910–932. DOI: 10.1109/JSTSP.2020.3002103

2. Столлінгс В. Криптографія та мережева безпека: принципи та практика. – Pearson, 2022.

3. Кучер В.І., Бондаренко І.В. Основи криптографії: підручник. – Київ: НАУ, 2020.

4 Menezes A., Van Oorschot P., Vanstone S. Handbook of Applied Cryptography. – CRC Press, 2021.

5. Daemen J., Rijmen V. The Design of Rijndael: AES - The Advanced Encryption Standard. – Springer, 2020.

6. NIST. Secure Hash Standard (SHS): FIPS PUB 180-4. – National Institute of Standards and Technology, 2015.