Хмельницький національний університет
Факультет інформаційних технологій
Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

Медякова Євгена Олександровича

на здобуття ступеня вищої освіти Бакалавра

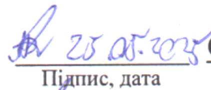
Кросплатформенна система безпечного зберігання інформації з обмеженим доступом

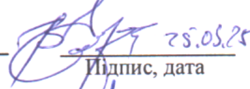
Галузь знань 12 – Інформаційні технології

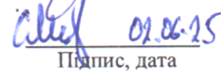
Спеціальність 125 – Кібербезпека

Освітня програма Кібербезпека

КРБКБ.220163.22.01.06 ПЗ

Виконав студент 3 курсу, група КБс-22-1  Євген МЕДЯКОВ
Підпис, дата Ініціали, прізвище

Керівник канд. тех. наук, доцент  Віктор ЧЕШУН
Науковий ступінь, вчене звання Підпис, дата Ініціали, прізвище

Нормоконтролер старший викладач  Сергій МОСТОВИЙ
Науковий ступінь, вчене звання Підпис, дата Ініціали, прізвище

До захисту допускаю:

Зав. кафедри кібербезпеки Юрій КЛЬОЦ
Підпис, дата Ініціали, прізвище

2.06 ____________ 2025р.

Хмельницький, 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Кібербезпеки

Рівень вищої освіти Бакалавр

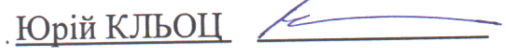
Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека

Освітня програма Кібербезпека

ЗАТВЕРДЖУЮ

Завідувач кафедри кібербезпеки

Юрій КЛЬОЦ 

15 02 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
Медякова Євгена Олександровича

1 Тема роботи Кросплатформенна система безпечного зберігання інформації з обмеженим доступом

Керівник роботи к.т.н, доц. кафедри кібербезпеки Чешун Віктор Миколайович

Затверджено наказом ректора університету від 15 лютого 2025 № ____

2 Строк подання студентом кваліфікаційної роботи на кафедру 1.06.2025

3 Вихідні дані до роботи Кросплатформенна система безпечного зберігання інформації з обмеженим доступом розробляється як програмне рішення з використанням фреймворку Flutter

4 Зміст пояснювальної записки (перелік питань, які потрібно розробити) Аналіз загроз конфіденційній інформації, аналіз актуальності тематики, огляд технологій зберігання інформації, огляд цільових платформ, визначення вимог для розробки системи, побудова архітектури системи та реалізація програмного рішення системи безпечного зберігання інформації

5 Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Діаграма класів

Діаграма активностей

Діаграма послідовностей

6 Консультанти розділів кваліфікаційної роботи

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

7 Дата видачі завдання 16 лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
Вибір і затвердження теми кваліфікаційної роботи	Січень-Лютий	
Ознайомлення з предметною областю	Лютий	
Дослідження існуючих рішень	Лютий	
Постановка задачі	Березень	
Визначення загальних принципів рішення задачі	Березень	
Деталізація принципів рішення задачі	Квітень	
Розробка проєктних рішень	Квітень	
Апробація проєктних рішень	Травень	
Оформлення пояснювальної записки згідно вимог	Травень	
Оформлення графічної частини	Травень	
Захист КР	Червень	

Студент Євген МЕДЯКОВ

Керівник кваліфікаційної роботи Віктор ЧЕШУН

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Кросплатформенна система безпечного зберігання інформації з обмеженим доступом».

Автор роботи: Медяков Євген Олекскандрович.

Керівник роботи: канд. техн. наук, доц. Чешун Віктор Миколайович.

Пояснювальна записка: 78 с., 2 додатки, 25 рисунків, 45 джерел.

Ключові слова: Flutter, AES-256, PBKDF2, біометрична автентифікація, SQLite, офлайн-сховище, кросплатформенність.

Кваліфікаційна робота присвячена розробленню та дослідженню крос-платформеної автономної системи безпечного зберігання конфіденційної інформації.

У роботі проаналізовано сучасні загрози для багатоплатформених середовищ, обґрунтовано вибір Flutter / Dart як єдиної кодової бази. Дані зберігаються локально у SQLite; перед записом вони шифруються алгоритмом AES-256 із деривацією ключів PBKDF2, а майстер-ключі поміщаються у захищені сховища Android Keystore та iOS Keychain. Для підвищення стійкості реалізовано багаторівневу автентифікацію, що поєднує пароль і біометрію; лімітування спроб входу та очищення ключів із пам'яті мінімізують ризик brute-force. У ході реалізації створено відтворюване середовище зборки під Android, iOS, Windows, macOS і Linux; проведено модульні, інтеграційні та системні тести, які підтвердили коректність криптографічних операцій і відсутність критичних уразливостей.

25.05.2025

ABSTRACT

Theme of the qualification work: " Cross-platform system for secure storage of limited access information".

Author of the work: Mediakov Yevhen Oleksandrovych.

Mentor:Ph.D. Cheshun Viktor Mykolaiovych.

Explanatory note: 78 p., 2 appendices, 25 figures, 45 references.

Keywords: Flutter, AES-256, PBKDF2, biometric authentication, SQLite, offline storage, cross-platform.

The qualification work is devoted to the development and research of a cross-platform autonomous system for secure storage of confidential information.

The work analyzes modern threats to multi-platform environments and justifies the choice of Flutter/Dart as a single code base. The data is stored locally in SQLite; before being written, it is encrypted with the AES-256 algorithm with PBKDF2 key derivation, and master keys are placed in secure Android Keystore and iOS Keychain storages. To increase resilience, multi-level authentication combining password and biometrics is implemented; limiting login attempts and clearing keys from memory minimize the risk of brute force. During the implementation, a reproducible build environment was created for Android, iOS, Windows, macOS, and Linux; module, integration, and system tests were conducted, which confirmed the correctness of cryptographic operations and the absence of critical vulnerabilities.

25.05.2025

ЗМІСТ

Вступ 7
1 Теоретичні осонови технологій розробки кросплатформеного рішення зберігання інформації на пристроях 9
1.1 Загрози конфіденційній інформації в багатоплатформених системах......... 9
1.2 Потреба в автономних рішеннях та їх особливості.................................... 11
1.3 Технології безпечного зберігання інформації 12
1.4 Цільові платформи та середовище розгортання 13
1.5 Огляд існуючих рішень 17
1.6 Постановка задачі 18
2 Проєктування та реалізація кросплатформеної системи безпечного зберігання данних з обмеженим доступом 21
2.1 Аналіз до вимог системи 21
2.2 Підготовка середовища розробки 23
2.3 Архітектура застосунку 25
2.4 Створення інтерфейсу користувача 29
2.5 Реалізація функціоналу системи 36
2.6 Висновки 46
3 Тестування та аналіз захищеної системи 48
3.1 Роль тестування в безпечних програмних системах 48
3.2 Підходи до тестування систем з підвищеними вимогами до безпеки 49
3.3 Аналіз стійкості до типових атак 51
3.4 Висновки за результатами тестування 52
3.5 Аналіз перспектив розвитку системи 55
3.6 Висновки 56
Висновки 58
Перелік джерел посилань 60
Додаток А Копії графічної частини 65
Додаток Б Фрагмент програмного коду 68

Зм.	Арк.	№ докум.	Підпис	Дата	КРБКБ.220163.22.01.06 ПЗ			
Виконала		Медяков Є.О.		25.05	Система захисту передачі інформації між об'єктами критичної інфраструктури Пояснювальна записка	Літера	Аркуш	Аркушів
Перевір.		Чешун В.М.		25.05			6	78
Н.контр.		Мостовий С.В		02.06.25		ХНУ, КБс-22-1		
Затвер.		Кльоц Ю.П		2.06.25				

ВСТУП

В умовах сучасного інформаційного суспільства забезпечення безпеки даних стає однією з найважливіших проблем. Постійне зростання обсягів оброблюваної та зберіганої інформації, а також її висока чутливість вимагають розробки нових підходів до зберігання та захисту даних. Зокрема, розробка ефективних систем для зберігання інформації з обмеженим доступом є важливим завданням у галузі комп’ютерної безпеки. Оскільки дані сьогодні можуть зберігатися на різних платформах і пристроях, необхідність у кросплатформених рішеннях для безпечного зберігання стає надзвичайно актуальною.

Кросплатформенні системи забезпечують можливість доступу до даних з різних операційних систем і пристроїв, що дає користувачам гнучкість та зручність. Проте наявність таких можливостей створює додаткові виклики для забезпечення безпеки, оскільки кожна платформа може мати свої уразливості. Одним із ключових аспектів при розробці таких систем є обмеження доступу до даних, яке дозволяє захищати конфіденційну інформацію від несанкціонованого використання.

Одним з ефективних методів забезпечення безпеки є впровадження двофакторної аутентифікації (2FA), яка значно підвищує рівень захисту, вимагаючи від користувача не лише введення пароля, а й підтвердження своєї особи за допомогою другого фактору. Цей додатковий етап може бути представлений у вигляді одноразового коду, надісланого через SMS або генерованого спеціальним додатком, або використання біометричних даних, таких як відбитки пальців, сканування обличчя чи райдужної оболонки ока. Застосування двофакторної аутентифікації дозволяє значно знизити ризик несанкціонованого доступу навіть у разі компрометації пароля користувача.

Ще одним важливим елементом безпеки є використання біометричних ключів, які дозволяють автентифікацію на основі фізичних характеристик людини. Біометричні методи, зокрема відбитки пальців, сканування обличчя та

голосова ідентифікація, стають усе більш популярними завдяки високому рівню точності та зручності. Вони забезпечують додатковий рівень захисту, оскільки біометричні дані важко підробити або викрасти. Інтеграція біометричних систем у кросплатформенні рішення дозволяє створювати надійні механізми контролю доступу до чутливої інформації.

Метою даної роботи є розробка кросплатформенної системи безпечного зберігання інформації з обмеженим доступом. У рамках дослідження буде розглянуто основні принципи та методи забезпечення безпеки даних, проаналізовано існуючі рішення, а також розроблено архітектуру системи з використанням сучасних технологій для створення надійного механізму захисту інформації. Важливим аспектом роботи є розробка механізмів обмеження доступу на основі ролей користувачів, впровадження двофакторної аутентифікації та біометричних ключів для покращення захисту даних.

Актуальність дослідження обумовлена постійним зростанням потреби в безпечному зберіганні конфіденційної інформації в умовах швидкого розвитку мобільних технологій і хмарних обчислень, а також зростаючими вимогами до захисту даних на різних платформах. Тому запропоноване дослідження може мати важливе значення для створення сучасних безпечних систем зберігання інформації та покращення захисту даних у різних галузях.

1 ТЕОРЕТИЧНІ ОСОНОВИ ТЕХНОЛОГІЙ РОЗРОБКИ КРОСПЛАТФОРМЕНОГО РІШЕННЯ ЗБЕРІГАННЯ ІНФОРМАЦІЇ НА ПРИСТРОЯХ

1.1 Загрози конфіденційній інформації в багатоплатформених системах

У сучасному цифровому середовищі значна частина приватної та професійної інформації зберігається на мобільних пристроях, десктопах та в хмарних сервісах. Поширення багатоплатформених додатків, які мають доступ до одних і тих самих даних на різних пристроях, значно розширило можливості користувача, але водночас створило додаткові вектори для атак (рис 1.1). Захист конфіденційної інформації в таких умовах стає не просто бажаним, а обов’язковим елементом будь-якого відповідального програмного рішення [1].

Рисунок 1.1 – Розповсюджені загрози в користувацьких системах

Основними загрозами є:

– несанкціонований доступ до пристрою або файлів користувача – особливо актуально для мобільних пристроїв які можуть бути втрачені або викрадені зловмисниками;

– атаки на рівні операційної системи – у тому числі root/jailbreak [2], що дозволяє обійти стандартні механізми безпеки;

– витік даних через мережу – особливо при використанні ненадійних точок доступу Wi-Fi або некоректній реалізації серверної взаємодії;

– компрометація хмарних сховищ – більшість популярних сервісів зберігає ключі шифрування на боці провайдера [3];

– фішинг і підбір паролів – користувачі часто обирають слабкі або повторно використані паролі, що спрощує компрометацію [4].

У багатоплатформенних системах загроза посилюється через фрагментацію архітектури. Наприклад, Android має відкриту екосистему та варіативність виробників, що створює труднощі з оновленням безпеки. iOS – навпаки, закрита платформа, але й тут можливі атаки через вразливості нульового дня. На десктопах, особливо на Windows, поширені зловмисні програми, які використовують соціальну інженерію чи вбудовуються у легітимні процеси.

Крім того, користувачі часто нехтують рекомендаціями з безпеки: не оновлюють систему, не використовують біометричний захист, не встановлюють ПЗ з офіційних джерел, використовують слабкі паролі та ігнорують можливість підключення другого фактору авторизації. Це все робить систему вразливою навіть у разі наявності в ній добре реалізованих технічних засобів захисту.

З огляду на це, побудова системи безпечного зберігання даних має передбачати комплексний підхід: включати надійне шифрування, багаторівневу автентифікацію, захищене локальне сховище та мінімізацію мережевих взаємодій [5].

Особливої уваги потребує сценарій повної автономності системи – тобто здатність працювати без постійного з’єднання з Інтернетом, без серверної синхронізації або зовнішньої залежності або в умовах повністю відсутного зв’язку з причин порушення обладнання або відсутності електропостачання

протягом довго часу. Саме на реалізацію такого типу рішення й орієнтовано дану роботу.

1.2 Потреба в автономних рішеннях та їх особливості

З огляду на вказані вище загрози, дедалі актуальнішим стає запит на рішення, які не залежать від зовнішніх серверів, хмарних сховищ чи постійного мережевого підключення. Такі рішення називають автономними (offline-first або standalone) [6]. Вони не тільки знижують вразливість до мережевих атак, але й підвищують контроль користувача над власними даними.

Автономні системи зберігання даних особливо важливі в таких умовах:

– коли користувачі оперують чутливою або приватною інформацією (паролі, медичні записи, особисті нотатки);

– недовіра до хмарних провайдерів або політик збереження даних [7];

– у середовищах з обмеженим доступом до Інтернету (віддалені регіони, польові умови, військові задачі);

– коли критично важливо забезпечити повний контроль над шифрувальними ключами та автентифікацією.

Головна перевага автономного підходу – це повна локальність обробки даних [8]. Користувач може бути впевнений, що вся інформація залишається лише на його пристрої. Шифрування, дешифрування, генерація ключів та автентифікація здійснюються локально, а будь-який зовнішній доступ фізично неможливий без компрометації пристрою. Проте автономні рішення мають і свої виклики, такі як:

– складніше реалізувати резервне копіювання без мережі;

– втрата доступу до пристрою без пароля або біометрії означає повну втрату даних [9];

– необхідно враховувати відмінності в механізмах зберігання між платформами.

Незважаючи на це, автономні системи зберігають свою привабливість, оскільки пропонують максимальну приватність і контроль [10]. Саме такі вимоги стали основою для розробки даної системи, яка працює без синхронізації, серверів чи сторонніх API, покладаючись виключно на ресурси пристрою користувача.

1.3 Технології безпечного зберігання інформації

У сфері кібербезпеки захист інформації в програмних продуктах досягається шляхом комплексного застосування криптографічних методів, багаторівневої автентифікації та апаратних засобів безпеки. З огляду на стрімкий розвиток мобільних і кросплатформенних технологій, особливої уваги набувають локальні механізми безпечного зберігання даних та захисту доступу до них на стороні клієнта.

Одним із ключових компонентів захисту є локальне або віддалене шифрування даних. У мобільних додатках часто використовуються вбудовані засоби операційних систем:

Android Keystore [11] та iOS Keychain [12] дозволяють безпечно зберігати ключі, токени та іншу чутливу інформацію, використовуючи апаратне шифрування та ізольоване середовище виконання (TEE або Secure Enclave).

Для зберігання структурованих даних широко застосовуються локальні СУБД на кшталт SQLite [13], що можуть бути розширені програмним шифруванням за допомогою алгоритмів, таких як AES, ChaCha20 тощо.

У сучасній практиці безпечне зберігання реалізується шляхом поєднання шифрування даних перед збереженням у базу та захищеного зберігання самих ключів.

Надійний контроль доступу до конфіденційної інформації є невіддільною частиною будь-якої захищеної системи. Одним з ефективних підходів є багаторівнева автентифікація (multi-factor authentication, MFA) [14], яка поєднує принаймні два незалежні фактори:

– щось, що знає користувач – пароль, PIN-код;

– щось, що має користувач – мобільний пристрій або токен;

– щось, чим є користувач – біометричні характеристики (відбиток пальця, розпізнавання обличчя тощо).

У мобільних застосунках широкого поширення набув комбінований підхід, коли вхід до програми захищено паролем, а критичні дії додатково підтверджуються біометрично [15].

Біометричні технології набули значного поширення як зручний та ефективний метод ідентифікації користувачів. У мобільних системах вони реалізовані на рівні апаратного забезпечення, що гарантує високий рівень довіри. Найпоширеніші біометричні модальності на сучасних системах є відбитки пальців (Fingerprint), розпізнавання обличчя (Face ID / Face Unlock) та розпізнавання голосу або райдужної оболонки ока [16]. Використання біометрії дозволяє зменшити ризики, пов’язані з викраденням або вгадуванням паролів, та слугує ефективним бар’єром при спробі несанкціонованої зміни або видалення критичної інформації.

1.4 Цільові платформи та середовище розгортання

Під час розробки кросплатформених програмних рішень для зберігання конфіденційної інформації одним із ключових чинників є вибір операційних систем, які охоплюють найбільшу частку сучасного користувацького середовища (рисунок 1.2) [17]**.** Це дозволяє забезпечити доступ до

функціональності додатку з різних типів пристроїв, а також адаптувати засоби захисту до особливостей відповідної платформи.

Рисунок 1.2 – Графік популярності операційних систем

У цьому контексті система має бути орієнтована на найбільш поширені операційні системи, серед яких: Android, iOS, macOS, Windows, Linux. Кожна з них має власні архітектурні особливості, засоби захисту та політики безпеки, які необхідно враховувати при проектуванні та реалізації програмного забезпечення.

Android – мобільна операційна система, яка була офіційно представлена у 2008 році компанією Google. Вона базується на ядрі Linux, але її архітектура суттєво відрізняється від традиційних Linux-дистрибутивів через наявність власного середовища виконання – Android Runtime (ART), що замінила попередню Dalvik VM, яка забезпечувала інтерпретацію Java-коду. З погляду безпеки, Android підтримує механізми ізоляції процесів, контроль дозволів (permissions), а також Keystore API, що дозволяє зберігати криптографічні ключі в захищеному середовищі [18].

Однією з характерних рис Android є відкритість і варіативність: виробники пристроїв можуть змінювати як зовнішній вигляд системи, так і її внутрішню реалізацію. Це створює як додаткові можливості для інтеграції з апаратною частиною, так і потенційні проблеми сумісності чи безпеки, пов’язані з фрагментацією ОС.

iOS – операційна система для мобільних пристроїв від компанії Apple Inc., представлена у 2007 році. Вона базується на Unix-подібному ядрі Darwin, має закритий вихідний код і призначена виключно для фірмових пристроїв Apple (iPhone, iPad). На відміну від Android, доступ до системних ресурсів у iOS суворо регламентований, а встановлення сторонніх застосунків можливе лише через офіційний магазин App Store.

Серед засобів безпеки, які реалізовано в iOS [19], слід відзначити:

– Keychain Services – API для зберігання облікових записів і ключів;

– Secure Enclave – окремий процесорний модуль для ізольованого зберігання біометричних даних і виконання криптографічних операцій;

– підтримку автентифікації через Touch ID або Face ID.

Таке поєднання дозволяє реалізувати високий рівень захисту без зниження зручності для користувача, що є особливо важливим у контексті роботи з чутливою інформацією.

macOS – настільна операційна система від Apple, яка є логічним продовженням iOS у десктопному середовищі. Вона також базується на ядрі Darwin, поєднуючи в собі стабільність UNIX-систем і графічні інновації Apple. Вперше macOS була представлена у 2001 році як спадкоємець Mac OS Classic.

Особливістю macOS є можливість інтеграції з іншими продуктами Apple, а також підтримка таких засобів захисту [20]:

– FileVault 2 – повне шифрування диску за допомогою алгоритму AES;

– System Integrity Protection (SIP) – механізм, що блокує модифікацію системних файлів навіть для адміністратора;

– Keychain – вбудоване сховище паролів і ключів.

Крім того, перехід Apple на власні процесори архітектури ARM (Apple Silicon) додатково зміцнив апаратну безпеку в нових моделях пристроїв.

Windows – найбільш розповсюджена операційна система для персональних комп’ютерів, яка має широке застосування в корпоративному та домашньому середовищі. Сучасні версії мають розвинуту систему безпеки [21], яка включає:

– BitLocker – засіб шифрування накопичувачів;

– Data Protection API (DPAPI) – механізм, що дозволяє зберігати ключі й паролі у захищеному вигляді;

– Windows Hello – система автентифікації, що підтримує біометричні дані.

Windows також підтримує UWP (Universal Windows Platform), що дозволяє створювати додатки з єдиним API для всіх пристроїв Microsoft – ПК, планшетів, Xbox. Хоча це не є мобільною ОС, її присутність у кросплатформенних рішеннях є необхідною для охоплення широкої аудиторії користувачів.

Linux – відкрите сімейство операційних систем, що базується на ядрі Linux і має широкий спектр дистрибутивів (Ubuntu, Debian, Fedora тощо). Завдяки відкритому коду, Linux активно використовується у наукових, технічних і серверних середовищах. З огляду на безпеку, Linux дозволяє реалізувати [22]:

– шифрування диску за допомогою LUKS (Linux Unified Key Setup);
захист доступу через AppArmor або SELinux;

– ручне керування ключами та сертифікатами з використанням GnuPG.

Linux також популярний у вбудованих системах та IoT, що робить його доцільним у контексті кросплатформенних мобільних або гібридних рішень.

Отже усі розглянуті операційні системи мають власні механізми безпеки, які потрібно враховувати під час розробки. Розгортання кросплатформеного застосунку в умовах різних ОС вимагає адаптації криптографічних і автентифікаційних процесів до кожної платформи, водночас зберігаючи єдину логіку роботи системи.Саме тому одним із найоптимальніших рішень є використання фреймворка Flutter, який дозволяє забезпечити спільну кодову

базу для всіх зазначених платформ [23], із можливістю доступу до нативних API безпеки, реалізованих на кожній ОС.

1.5 Огляд існуючих рішень

Забезпечення безпечного зберігання даних є одним із фундаментальних завдань у галузі кібербезпеки. У сучасних системах застосовується низка методів і технологій, спрямованих на забезпечення конфіденційності, цілісності та доступності інформації. Залежно від архітектури рішення, способу зберігання, використання мережевих технологій та рівня загроз можна виокремити кілька основних підходів.

Хмарні сервіси зі шифруванням на стороні сервера Більшість сучасних платформ такі як Google Drive, Dropbox, OneDrive тощо, реалізують зберігання даних з автоматичним шифруванням на стороні сервера. Такі сервіси забезпечують доступність даних із різних пристроїв, резервне копіювання та масштабованість.

Однак у такому підході ключі шифрування зазвичай зберігаються у провайдера, що створює потенційні ризики витоку або несанкціонованого доступу у випадку компрометації серверної інфраструктури. Деякі сервіси як MEGA або Tresorit використовують end-to-end шифрування, де дані шифруються ще на пристрої користувача, а в хмару передаються лише в зашифрованому вигляді. У цьому випадку тільки користувач володіє ключами доступу до своїх файлів, що значно підвищує безпеку.

Проте навіть у таких рішеннях присутній ризик компрометації, якщо зловмисник отримає доступ до пристрою або ключів автентифікації. Існують рішення, які повністю зберігають та обробляють дані локально, без передачі до зовнішніх серверів. Такі засоби використовуються, зокрема, у додатках для паролів Bitwarden, 1Password у локальному режимі, фінансового обліку або

зберігання нотаток. У цьому підході шифрування виконується локальним механізмом, а ключі можуть зберігатися як у пам’яті пристрою, так і в спеціалізованих сховищах Keychain, Keystore тощо.

Головною перевагою є ізоляція даних від мережі, а отже – зменшення ризиків віддалених атак. Недоліком є складність забезпечення резервного копіювання та синхронізації між пристроями.

На сучасних пристроях підтримується використання апаратних механізмів захисту, таких як Trusted Execution Environment (TEE) або Secure Enclave [24], що дозволяють обробляти криптографічні операції в ізольованому середовищі. Це знижує ризик крадіжки ключів навіть при наявності зловмисного ПЗ в основній операційній системі.

У контексті мобільного застосування все частіше використовуються кросплатформенні фреймворки, які дозволяють створювати універсальні додатки з єдиною логікою зберігання й шифрування даних. Безпека таких рішень залежить від правильного використання платформних API для безпечного зберігання та від якості реалізації криптографічних алгоритмів на клієнтській стороні.

Отже, хоча сучасні підходи до зберігання даних пропонують різноманітні засоби захисту, більшість із них передбачає залежність від зовнішніх сервісів або мережевої інфраструктури. У той же час, підхід до повністю автономного локального зберігання з клієнтським шифруванням і багаторівневою автентифікацією залишається актуальним у контексті підвищених вимог до приватності.

1.6 Постановка задачі

У сучасних умовах стрімкого розвитку інформаційних технологій та зростання обсягів конфіденційної інформації, яку користувачі зберігають на

своїх пристроях, особливої актуальності набуває проблема захищеного локального зберігання даних. Зважаючи на часті випадки компрометації персональної інформації, що зберігається у хмарних сховищах або передається мережею, зростає попит на автономні системи безпеки, які не залежать від зовнішньої інфраструктури. Саме на ці виклики й має відповідати розроблена система – кросплатформенне рішення для безпечного зберігання інформації з обмеженим доступом.

Метою даного дослідження є розробка програмного засобу, що забезпечує захищене локальне зберігання даних на мобільних пристроях з операційними системами Android та iOS, а також на десктопних системах Windows, macOS і Linux. Реалізація має бути кросплатформенною, побудованою з урахуванням принципів безпеки за замовчуванням, мінімізації поверхні атаки та врахування особливостей кожної з цільових операційних систем.

У рамках досягнення поставленої мети необхідно вирішити наступні науково-технічні задачі.

На етапі проєктування обираються інструменти, які дозволяють створити ефективне, безпечне та портоване рішення. Як основна платформа використовується фреймворк Flutter із мовою програмування Dart, що забезпечує підтримку як мобільних, так і десктопних операційних систем. Для реалізації локального зберігання структурованих даних використовується база даних SQLite. У мобільному середовищі застосовується бібліотека sqflite, а для десктопних систем використовується sqflite_ffi – розширення, що реалізує взаємодію з SQLite через Foreign Function Interface, забезпечуючи сумісність із macOS, Windows та Linux.

Для обробки чутливої інформації та безпечного зберігання криптографічних ключів використовується flutter_secure_storage, яка забезпечує інтеграцію з Android Keystore та iOS Keychain. На десктопних платформах, де відсутні вбудовані механізми захищеного сховища, ключі зберігаються у

зашифрованому вигляді з використанням симетричного шифрування на основі пароля, введеного користувачем.

Конфіденційність даних забезпечується шляхом застосування симетричного алгоритму шифрування AES (Advanced Encryption Standard). Шифрування здійснюється безпосередньо перед збереженням у базу даних, а розшифрування – у момент доступу до даних. Це виключає можливість зберігання незашифрованої інформації в системі.

Автентифікація користувача реалізована через парольний захист, що використовується для генерації основного ключа. У випадку, якщо пристрій підтримує біометричну автентифікацію (відбитки пальців, розпізнавання обличчя), вона може використовуватись для підтвердження критичних операцій, таких як видалення або редагування інформації.

Ключовою особливістю системи є її повна автономність: обробка та зберігання інформації здійснюється локально без взаємодії з мережевими службами або сторонніми серверами. Це мінімізує ризики, пов’язані з витоком або перехопленням даних через Інтернет.

Фінальним етапом є тестування розробленого рішення, яке проводиться на реальних пристроях і в середовищах емуляції для Android, iOS, Windows, macOS та Linux. Перевіряється сумісність, стабільність, стійкість до атак, відповідність функціональним вимогам та зручність користування.

Таким чином, у результаті виконання вищенаведених задач очікується створення повноцінного кросплатформенного засобу локального зберігання даних з підвищеним рівнем безпеки, що не залежить від сторонніх інфраструктурних компонентів і здатен забезпечити конфіденційність інформації на широкому спектрі пристроїв.

2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ КРОСПЛАТФОРМЕНОЇ СИСТЕМИ БЕЗПЕЧНОГО ЗБЕРІГАННЯ ДАННИХ З ОБМЕЖЕНИМ ДОСТУПОМ

2.1 Аналіз до вимог системи

Розробка кросплатформенної системи безпечного зберігання інформації потребує всебічного аналізу вимог, які визначають її функціональні можливості, технічні характеристики та критерії ефективності й безпеки.

У межах аналізу вимог до системи було виокремлено кілька ключових напрямків, що охоплюють питання безпеки, доступності, продуктивності та розширюваності.

Функціональні вимоги включають в себе важливі аспекти роботи системи з даними та взаємодії з користувачем. Зокрема, система повинна забезпечувати надійний механізм автентифікації, який передбачає створення користувацького пароля під час першого запуску додатку та регулярну автентифікацію користувача в подальших сеансах роботи.

Пароль користувача повинен відповідати високим вимогам безпеки та складності паролю, для посилення захисту важливих операцій (видалення, редагування даних) передбачено додаткову можливість біометричної перевірки (відбитки пальців або розпізнавання обличчя), що значно підвищує рівень безпеки користувацьких даних. Система повинна реалізовувати повний цикл роботи з інформацією: створення, перегляд, редагування і видалення записів (рисунок 2.1).

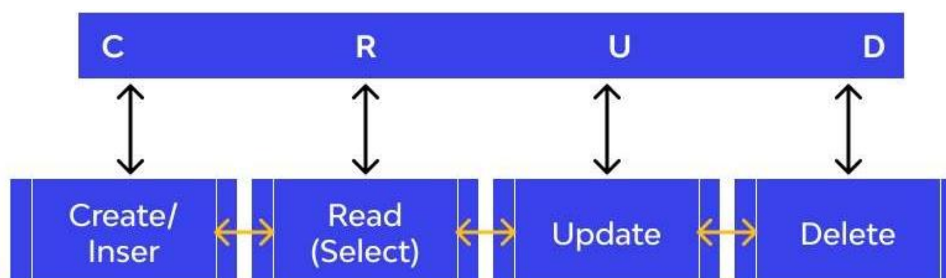


Рисунок 2.1 – Схема CRUD системи

Зберігання даних організовується виключно локально, без використання зовнішніх серверів чи хмарних сховищ, що є важливим для гарантування конфіденційності.

Обов’язковим елементом захисту даних є використання алгоритму симетричного шифрування AES з довжиною ключа 256 біт [25], що дозволяє досягти високого рівня криптографічного захисту (рисунок 2.2)[26].

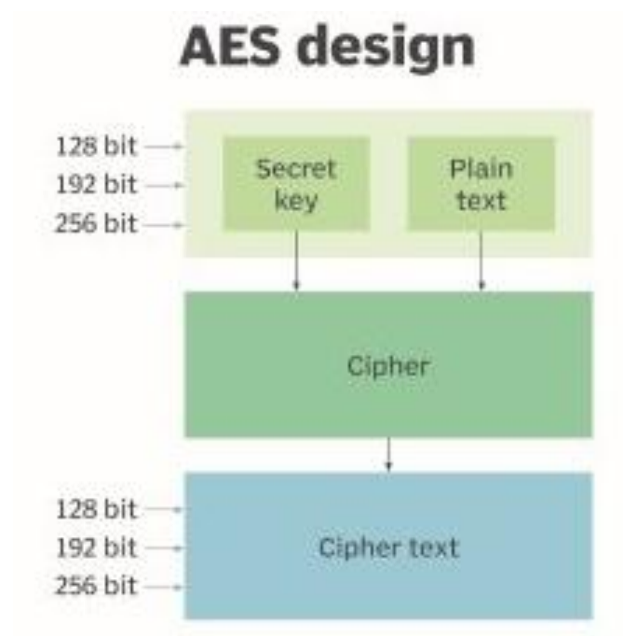


Рисунок 2.2 – Схема AES шифрування

Серед нефункціональних вимог особливу увагу приділено питанням безпеки, продуктивності та кросплатформенності. Безпека забезпечується шляхом запобігання brute-force атакам, обмежуючи кількість спроб входу [27] та гарантуючи автоматичне очищення криптографічних ключів з оперативної пам’яті після закінчення їх використання. Система має бути стійкою до різноманітних загроз, у тому числі й при фізичному доступі до пристрою.

Кросплатформенність забезпечується за рахунок використання фреймворка Flutter [28], що дозволяє створити єдину кодову базу для Android, iOS, Windows, macOS і Linux, таким чином спрощуючи процес розробки та підтримки системи. При цьому інтерфейс користувача повинен бути

уніфікованим і зрозумілим для всіх платформ, із врахуванням сучасних стандартів доступності.

Щодо продуктивності, то вимоги передбачають високу швидкість роботи системи, зокрема швидке шифрування та дешифрування.

Автономність є ключовим критерієм, що передбачає повну незалежність системи від будь-яких зовнішніх ресурсів або мережевих сервісів. Це рішення максимально знижує ризики витоку інформації і забезпечує стабільність роботи незалежно від зовнішніх факторів.

Важливим критерієм є також масштабованість і гнучкість архітектури, яка дозволяє додавати нові функції у майбутньому, наприклад, експорт даних або інтеграцію з додатковими системами автентифікації. Система повинна бути готовою до адаптації без значних змін її внутрішньої архітектури.

Таким чином, детально проаналізовані вимоги дозволяють сформувати чітке бачення системи, яка відповідає сучасним стандартам безпеки, продуктивності та зручності використання на різних платформах.

2.2 Підготовка середовища розробки

Для успішної реалізації кросплатформенної системи безпечного зберігання інформації важливим етапом є правильна організація та підготовка середовища розробки. Вибір відповідного програмного забезпечення, інструментів та бібліотек безпосередньо впливає на ефективність розробки, її якість та зручність підтримки системи в подальшому.

Основним інструментом для створення програмного забезпечення було обрано фреймворк Flutter, що базується на мові програмування Dart. Використання Flutter забезпечує високу продуктивність розробки завдяки єдиній кодовій базі, яка компілюється у нативні застосунки для мобільних Android, iOS

та десктопних платформ Windows, macOS, Linux (рисунок 2.3). Для початку роботи з Flutter необхідно встановити Flutter SDK та Dart SDK [29].

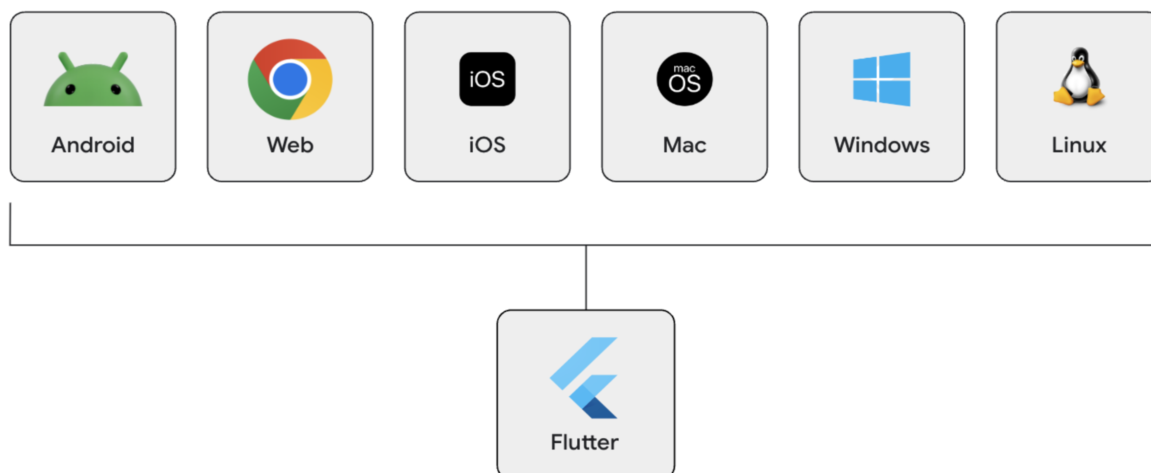


Рисунок 2.3 – Цільові платформи фреймворку Flutter

Для мобільних платформ, таких як Android та iOS, необхідна установка інтегрованих середовищ розробки: Android Studio з набором Android SDK та Xcode для розробки на iOS. Android Studio дозволяє використовувати емулятори Android [30], проводити тестування та налагодження застосунків. Xcode, у свою чергу, забезпечує аналогічні функції для пристроїв Apple [31].

Для десктопних платформ розробка потребує додаткових інструментів. Для Windows використовується Visual Studio з інстальованими компонентами для розробки застосунків на базі C++ та підтримкою Flutter [32]. Для macOS та Linux важливим є наявність відповідних компіляторів, що підтримуються Flutter, а також базових засобів командного рядка та графічних бібліотек (наприклад, GTK на Linux та Cocoa на macOS).

Окрім середовища розробки, важливою складовою є вибір необхідних бібліотек, розробники Flutter створили власне середовище пошуку різноманітних бібліотек pub.dev [33].

Використовуючи ключові слова сформовано список бібліотек які забезпечують функціональність системи.

Серед ключових бібліотек виділяються:

– flutter_secure_storage – для безпечного зберігання криптографічних ключів у захищених сховищах (Android Keystore, iOS Keychain);

– sqflite – бібліотека для взаємодії з базою даних SQLite на мобільних платформах;

– sqflite_ffi – розширення SQLite для десктопних платформ, що забезпечує однакову логіку зберігання даних на всіх підтримуваних ОС;

– encrypt – для реалізації алгоритму шифрування AES із довжиною ключа 256 біт;

– local_auth – бібліотека, що дозволяє інтегрувати біометричну автентифікацію (відбиток пальця, Face ID);

– shared_preferences – для зберігання локальних налаштувань і конфігурацій, які не потребують високого рівня безпеки.

Для забезпечення зручності та прозорості процесу розробки важливо налаштувати систему контролю версій, наприклад Git, що дозволяє ефективно керувати змінами у кодовій базі та полегшує командну взаємодію. Всі перераховані компоненти утворюють комплексне середовище, необхідне для створення, тестування та подальшого розвитку системи безпечного зберігання інформації.

Таким чином, ретельно підібране та налаштоване середовище розробки значно спрощує процес створення програмного рішення, дозволяючи зосередитися на розробці функціоналу, а не на вирішенні технічних проблем із сумісністю та інтеграцією.

2.3 Архітектура застосунку

Архітектура розробленого кросплатформенного застосунку була спроєктована з урахуванням принципів простоти та зрозумілості, відповідно до

концепції KISS (Keep It Simple, Stupid) [34]. Цей підхід передбачає мінімізацію складності коду, що значно спрощує підтримку, налагодження та подальший розвиток додатку. Враховуючи цільову спрямованість проєкту на високу безпеку та автономність, було вирішено уникнути використання додаткових складних архітектурних шаблонів на кшталт BLoC чи Redux, натомість обравши простішу, зрозумілішу та ефективнішу структуру.

Весь код додатку чітко розподілений на три основні шари, що забезпечують ясне розділення логіки та зручність у роботі:

Перший шар – Data Layer (шар даних), який відповідає за прямий доступ до бази даних SQLite через бібліотеки sqflite та sqflite_ffi, а також за взаємодію з безпечним сховищем ключів flutter_secure_storage. Цей шар містить логіку роботи з локальними даними, що забезпечує надійне зберігання та швидкий доступ до інформації.

Другий шар – Domain Layer (доменний шар), включає в себе основну бізнес-логіку додатку, таку як шифрування та дешифрування інформації, генерація та управління криптографічними ключами, автентифікація користувача. Використання чітко визначених інтерфейсів забезпечує простоту та прозорість цього шару.

Третій шар – Presentation Layer (шар представлення), відповідає за реалізацію інтерфейсу користувача на базі Flutter-віджетів. Цей шар розроблено з мінімальними залежностями від інших частин системи, що робить його максимально гнучким і зручним для адаптації до різних платформ.

Використання принципу KISS у цьому проєкті дозволить досягти високої продуктивності розробки, легкості в розумінні логіки роботи системи, зручності у подальшій підтримці та розширенні функціоналу без шкоди для безпеки та ефективності рішення [35].

Для створення проекту використовуючи середовище розробки Flutter необхідно встановити розширення для AndroidStudio, які дозіволяють використовувати мову програмування Dart, для цього в налаштування Android

Studio перейдемо на вкладку розширення та завантажимо необхідні розширення зображені на рисуноку 2.4. Перелік розширень можна модифікувати за потреби, кожному розробнику під себе, тому крім критично необхідних Flutter та Dart додаємо рішення генераторів коду, які за ключовими словами вказані користувачем автоматично створить програмний код, що покращує працездатність розробника, та полегшує завдання які потребують однакових дій.

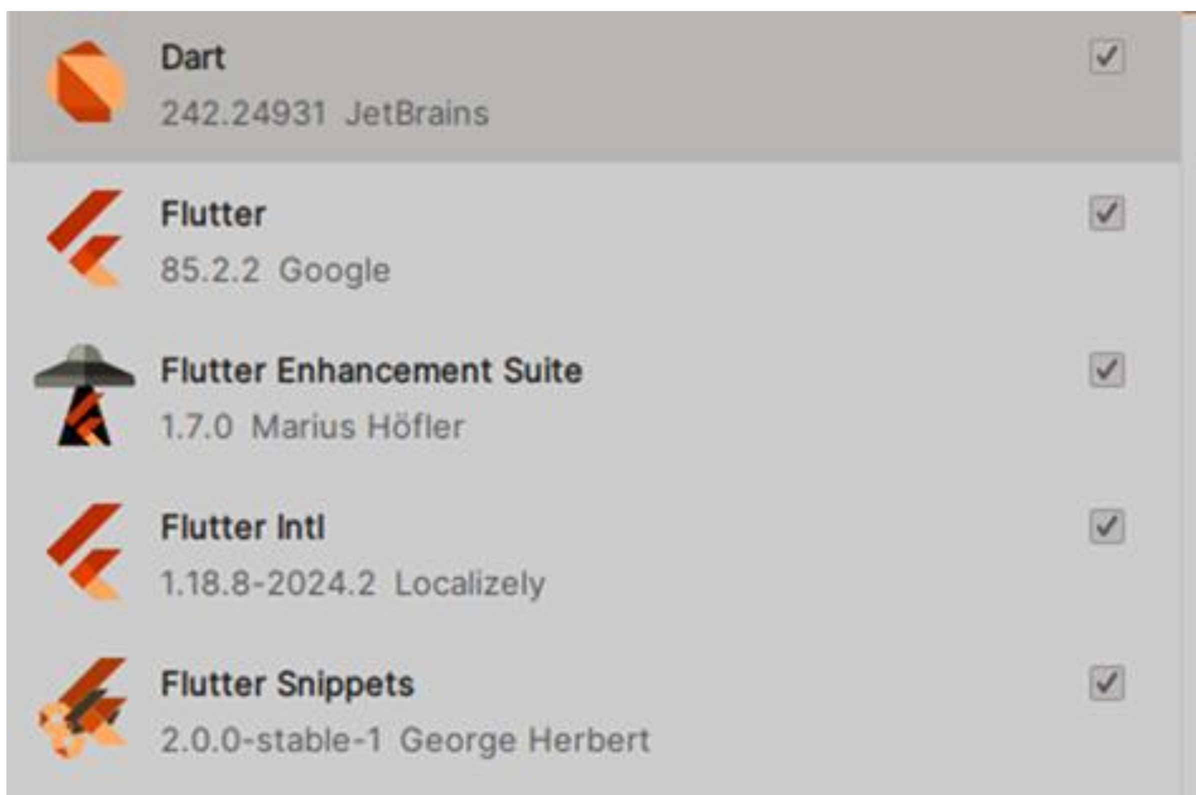


Рисунок 2.4 – Необхідні розширення для Flutter

Після встановлення необхідних розширень та перезапуску середовища розробки використовуючи вбудований термінал необхідно перевірити працездатність фреймворку Flutter за допомогою консольної команди flutter doctor -v перевіряємо чи встановленні усі необхідні елементи для розробки програмного забезпечення на платформи визначенні в постановці завдання, для цього необхідно погодитись з ліцензійними умовами розробки для Android використовуючи термінал системи.

Створюємо новий проект використовуючи середовище програмування Android Studio як зображено на рисунку 2.5. Важливо відмітити, що при створені проекту потрібно перевірити чи правильно вказано шлях до Flutter та вказати усі необхідні для проекту платформи.

Створивши новий проект Flutter автоматично створює тестовий проект представлений простим обрахунком натискань на іконку.

Рисунок 2.5 – Діалогове вікно створення нового проекту

Щоб перевірити чи правильно все встановлено виконаємо запуск цієї програми на доступних платформ та перевіримо чи не виникає помилок при запуску компіляції коду. Створимо файлову систему програми відповідно до запланованої архітектури (рисунок 2.6).

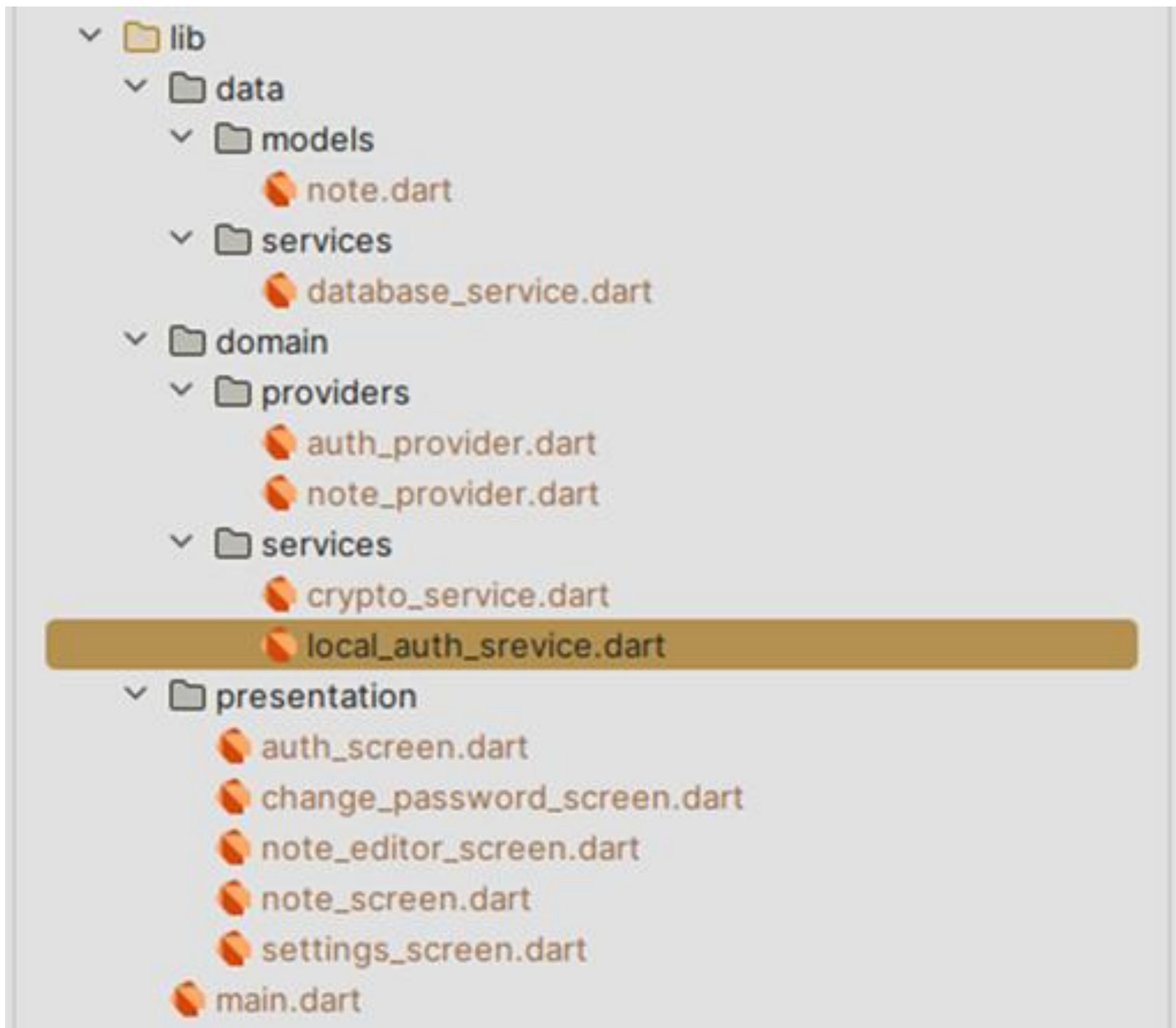


Рисунок 2.6 – Файлова структура додатку

Побудувавши файлову структуру додатку, переходимо до наповнення інтерфейсу користувача.

2.4 Створення інтерфейсу користувача

Платформа Flutter забезпечує високий рівень гнучкості у створенні інтерфейсів завдяки унікальному підходу до побудови UI, заснованому на концепції "все є віджетом". У Flutter будь-який візуальний елемент – від простого

текстового поля до складного макету екрана – реалізується як окремий віджет [36], який можна легко комбінувати, вкладати або модифікувати. Це надає розробнику повний контроль над виглядом і поведінкою інтерфейсу, дозволяючи досягти високої адаптивності та стилістичної узгодженості.

Інтерфейси у Flutter створюються декларативно. Це означає, що розмітка будується не покроково, а описує стан інтерфейсу в кожен момент часу. Коли стан змінюється – змінюється і відповідний набір віджетів. Такий підхід дозволяє уникати багатьох типових помилок, що виникають у імперативних фреймворках, і суттєво спрощує логіку підтримки та розширення коду в майбутньому.

У межах цього проєкту інтерфейс побудований з урахуванням принципів узгодженості між різними екранами. Застосовуються спільні компоненти та шаблони, зокрема: віджети TextFormField з вбудованою валідацією, діалогові вікна підтвердження через AlertDialog, структура Scaffold з AppBar для створення каркасів екранів. На більшості сторінок реалізовані інтуїтивно зрозумілі кнопки дій, збереження або повернення, із відповідною стилізацією згідно стандартів Material Design (рисунок 2.7) [37].

Рисунок 2.7 – Елементи дизайну програми

Крім того, кожен екран дотримується єдиного візуального стилю: шрифти, кольорова гама, відступи та структура контенту підібрані таким чином, щоб забезпечити комфортну навігацію для користувача незалежно від його досвіду чи обраної платформи. Всі елементи добре масштабуються і адаптуються до розмірів екрану, що дозволяє використовувати додаток як на мобільних, так і на десктопних пристроях без втрати зручності.

Окрему увагу приділено способам навігації між екранами. Вони реалізовані за допомогою MaterialPageRoute, що забезпечує анімацію переходів у стилі платформи та зручність у передаванні параметрів між сторінками. У результаті вдалося досягти консистентного досвіду користування, де всі елементи працюють очікувано й послідовно незалежно від контексту використання.

Інтерфейс користувача складається з низки функціональних екранів, кожен із яких має своє призначення та структуру, але водночас дотримується загальної логіки і стилістики всього додатку.

Екран автентифікації є першим кроком взаємодії користувача з додатком. Він дозволяє створити або ввести пароль для доступу до системи. Особливістю цього екрану є інтерактивні елементи, такі як перемикання видимості пароля, автоматична установка фокусу на поле введення, що дозволяє використовувати додаток навіть особам з обмеженими можливостями та для користувачів які використовують додаткове обладнання для введення та виведення інформації до додатку а також адаптивне відображення інструкцій залежно від того, чи це перший запуск програми.

Користувачу надається зворотний зв'язок у випадку помилок або некоректного вводу за допомогою багатофункціонального Scaffold який широко використовується на усіх актуальних платформах, а також візуальні індикатори процесу перевірки пароля. Це створює враження плавності й безпечності роботи ще з перших секунд використання (рисунок 2.8).

```
Widget build(BuildContext context) {
        padding: const EdgeInsets.all( value: 24.0),
        child: Form(
          key: _formKey,
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              const Icon( icon: Icons.lock_outline, size: 80, color: Colors.teal),
              const SizedBox(height: 32),
              Text(
                 data: widget.isFirstTime
                    ? 'Create a Secure Password'
                    : 'Enter Your Password',
                style: Theme.of(context).textTheme.headlineSmall,
                textAlign: TextAlign.center,
              ), // Text
              const SizedBox(height: 8),
              Text(
                 data: widget.isFirstTime
                    ? 'This password will be used to encrypt your notes'
                    : 'Unlock your Local Keep',
                textAlign: TextAlign.center,
              ), // Text
              const SizedBox(height: 32),
              TextFormField(...), // TextFormField
              if (widget.isFirstTime) ...[
                const SizedBox(height: 16),
                TextFormField(...), // TextFormField
              ],
              const SizedBox(height: 24),
              if (_errorMessage != null) ...[
                Text(
                   data: _errorMessage!,
                  style: const TextStyle(color: Colors.red),
                  textAlign: TextAlign.center,
                ), // Text
                const SizedBox(height: 16),
              ],
              SizedBox(...), // SizedBox
            ]
```

Рисунок 2.8 – Фрагмент коду екрану аутентифікації

Екран зміни пароля забезпечує користувачу можливість оновити облікові дані. Він складається з трьох полів для вводу: поточного пароля, нового пароля та підтвердження нового пароля. Кожне з них має окрему функцію керування видимістю тексту, що підвищує зручність і приватність введення. Валідація на цьому екрані враховує відповідність нового пароля до критеріїв безпеки, а також його відмінність від поточного. У разі виникнення помилок виводиться текстове пояснення, яке дозволяє користувачу швидко виправити ситуацію (рисунок 2.9).

```
class _ChangePasswordScreenState extends State<ChangePasswordScreen> {
  Widget build(BuildContext context) {
        backgroundColor: Theme.of(context).colorScheme.primaryContainer,
      ), // AppBar
      body: Padding(
        padding: const EdgeInsets.all( value: 16.0),
        child: Form(
          key: _formKey,
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              if (_errorMessage != null)
                Padding(
                  padding: const EdgeInsets.only(bottom: 10.0),
                  child: Text(
                     data: _errorMessage!,
                    style: TextStyle(
                      color: Theme.of(context).colorScheme.error,
                    ), // TextStyle
                    textAlign: TextAlign.center,
                  ), // Text
                ), // Padding
              TextFormField(...), // TextFormField
              const SizedBox(height: 12),
              TextFormField(...), // TextFormField
              const SizedBox(height: 12),
              TextFormField(...), // TextFormField
              const SizedBox(height: 20),
              _isLoading
                  ? const CircularProgressIndicator()
                  : ElevatedButton(
                    onPressed: _submitChangePassword,
                    child: const Text( data: 'Change Password'),
                  ), // ElevatedButton
            ],
          ), // Column
        ), // Form
      ), // Padding
    ); // Scaffold
```

Рисунок 2.9 – Код екрану зміни пароля

Редактор нотаток – це гнучкий інтерфейс для створення та редагування інформації. Він містить поле заголовка та поле змісту, кожне з яких розміщене в межах прокручуваної структури з підтримкою вільного введення тексту. При редагуванні вже існуючої нотатки вгорі автоматично виводиться дата останньої зміни. Якщо користувач намагається залишити екран зі зміненим вмістом, який не був збережений, система автоматично пропонує підтвердити вихід через

діалогове вікно. Це дозволяє уникнути втрати важливих даних через неуважність (рисунок 2.10).

```
60  @override
61  Widget build(BuildContext context) {
62    return PopScope(
63      canPop: !_isEdited,
64      onPopInvokedWithResult: ( bool didPop,  Object? result) async {...},
88      child: Scaffold(
89        appBar: AppBar(
90          backgroundColor: Theme.of(context).colorScheme.primaryContainer,
91          actions: [
92            IconButton(
93              icon: const Icon( icon: Icons.save),
94              onPressed: _saveNote,
95              tooltip: 'Save',
96            ), // IconButton
97          ],
98        ), // AppBar
99        body: Padding(
100         padding: const EdgeInsets.all( value: 16.0),
101         child: Column(
102           crossAxisAlignment: CrossAxisAlignment.start, // Align date to the start
103           mainAxisAlignment: MainAxisAlignment.start,
104           children: [
105             if (widget.note != null) // Only show date for existing notes
106               Padding(...), // Padding
116             Flexible(
117               flex: 1,
118               child: TextField(...), // TextField
132             ), // Flexible
133             Divider(),
134             Expanded(
135               flex: 4,
136               child: TextField(...), // TextField
150             ), // Expanded
151           ],
152         ), // Column
153       ), // Padding
154     ), // Scaffold
155   ); // PopScope
156 }
```

Рисунок 2.10 – Код екрану редактора нотатків

Головний екран додатку відображає список усіх створених користувачем нотаток. Візуалізація реалізована через адаптивну сітку з картками, які динамічно змінюють свій розмір залежно від обсягу вмісту. Кожна нотатка містить текст і дату, а також кнопку для швидкого видалення. При натисканні на нотатку відкривається її редагування, а перед виконанням критичних дій (наприклад, видалення або редагування) здійснюється перевірка біометрії, що

додає додатковий рівень захисту. На цьому ж екрані розміщено плаваючу кнопку для створення нової нотатки, яка завжди знаходиться під рукою користувача (рисунок 2.11).

```
19   class _NotesScreenState extends State<NotesScreen> {
140    Widget build(BuildContext context) {
141      final notes = Provider.of<NoteProvider>(context).notes;
142
143      return Scaffold(
144        appBar: AppBar(
145          title: const Text('Local Keep'),
146          backgroundColor: Theme.of(context).colorScheme.primaryContainer,
147          actions: [
148            IconButton(...),  // IconButton
153            IconButton(...),  // IconButton
158          ],
159        ),  // AppBar
160        body: _isLoading
161            ? const Center(child: CircularProgressIndicator())
162            : notes.isEmpty
163                ? const Center(
164                    child: Text('No notes yet. Tap + to create one.'),
165                  )  // Center
166                : Padding(
167                    padding: const EdgeInsets.all(8.0),
168                    child: MasonryGridView.count(
169                      crossAxisCount: 2,
170                      mainAxisSpacing: 8,
171                      crossAxisSpacing: 8,
172                      itemCount: notes.length,
173                      itemBuilder: (context, index) {
174                        final note = notes[index];
175                        return _buildNoteCard(note);
176                      },
177                    ),  // MasonryGridView.count
178                  ),  // Padding
179        floatingActionButton: FloatingActionButton(
180          onPressed: _createNote,
181          backgroundColor: Colors.teal,
182          foregroundColor: Colors.white,
183          tooltip: 'Add Note',
184          child: const Icon(Icons.add),
185        ),  // FloatingActionButton
186      );  // Scaffold
187    }
```

Рисунок 2.11 – Код головного екрану

Екран налаштувань представлений у вигляді списку параметрів з використанням стандартних елементів ListTile. Він дозволяє змінити пароль або

повністю скинути всі дані, включаючи пароль і створені нотатки. Остання дія вимагає підтвердження через діалогове вікно, що запобігає випадковому видаленню важливої інформації. Таким чином, користувач має контроль над своїми даними, не виходячи за межі програми (рисунок 2.12).

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text( data: 'Settings'),
      backgroundColor: Theme.of(context).colorScheme.primaryContainer,
    ), // AppBar
    body: ListView(
      children: [
        ListTile(
          leading: const Icon( icon: Icons.password),
          title: const Text( data: 'Change Password'),
          onTap: () {
            Navigator.of(context).push(
              route: MaterialPageRoute(builder: ( BuildContext _) => const ChangePasswordScreen()),
            );
          },
        ), // ListTile
        ListTile(
          leading: Icon( icon: Icons.delete_forever, color: Colors.red[700]),
          title: Text( data: 'Reset Password & Data', style: TextStyle(color: Colors.red[700])),
          onTap: () => _showResetConfirmationDialog(context),
        ), // ListTile
        const Divider(),
      ],
    ), // ListView
  ); // Scaffold
}
```

Рисунок 2.12 – Фрагмент коду екрану налаштувань

Усі екрани побудовані з урахуванням єдиної логіки навігації, збереження інтерфейсної цілісності та забезпечення позитивного користувацького досвіду відповідно до постановки завдання.

2.5 Реалізація функціоналу системи

Усі операції зберігання нотаток реалізовано через сервіс DatabaseService, що оперує SQLite-базою даних. Для Android та iOS використовується бібліотека

sqflite, а для десктопних систем – sqflite_ffi, що забезпечує однакову логіку доступу до даних на всіх підтримуваних платформах.

Функція insertNote приймає об’єкт нотатки, попередньо шифрує її вміст, після чого формує мапу для збереження у SQLite. Метод повертає ідентифікатор вставленого запису (рисунок 2.13).

```
static const String databaseText = '''
        CREATE TABLE notes(
          id INTEGER PRIMARY KEY AUTOINCREMENT,
          title TEXT,
          content TEXT,
          created_at TEXT,
          updated_at TEXT
        )
      ''';
1 usage
static Future<Database> _initDatabase() async {

  if (Platform.isWindows || Platform.isLinux) {
    sqfliteFfiInit();
    DatabaseFactory databaseFactory = databaseFactoryFfi;
    final String dbPath = await databaseFactoryFfi.getDatabasesPath();
    final String path = join( part: dbPath, 'local_keep.db');
    Database db = await databaseFactory.openDatabase(
      path,
      options: OpenDatabaseOptions(
        version: 1,
        onCreate: ( Database db, int version) async {
          await db.execute( sql: databaseText);
        },
      ),
    );
    ;
    return db;
  } else {
    final String dbPath = await getDatabasesPath();
    final String path = join( part: dbPath, 'local_keep.db');
    return await openDatabase(
      path,
      version: 1,
      onCreate: ( Database db, int version) async {
        await db.execute( sql: databaseText);
      },
    );
```

Рисунок 2.13 – Код ініціалізації бази данних

Функція getNotes зчитує всі записи з таблиці notes, сортує їх за датою редагування, розшифровує вміст кожного запису та повертає список об’єктів Note. Якщо ключ шифрування відсутній або пошкоджений, генерація винятку попереджає про помилку в логіці шифрування.

Функція updateNote приймає оновлену нотатку, проводить повторне шифрування її вмісту та оновлює відповідний запис у базі даних за ідентифікатором. Це дозволяє підтримувати цілісність зашифрованих даних.

Функція deleteNote приймає ідентифікатор нотатки й видаляє відповідний запис із таблиці. У випадку потреби очищення всієї бази застосовується функція clearNotes, яка повністю очищає таблицю notes (рисунок 2.14).

```
static Future<int> insertNote(Note note) async {...}
// Get all notes
1 usage
static Future<List<Note>> getNotes() async {

  if (_currentPassword == null) {
    throw Exception('Password not set for decryption');
  }

  if (!await CryptoService.isPasswordSetup()) {
    // Add this check
    throw Exception('Password not properly initialized');
  }

  final Database db = await database;
  final List<Map<String, Object?>> maps = await db.query( table: 'notes', orderBy: 'updated_at DESC');
  final List<Note> notes = <Note>[];
  for (var Map<String, Object?> map in maps) {
    // Decrypt content
    final String encryptedContent = map['content'] as String;
    final String decryptedContent = await CryptoService.decrypt(
      encryptedData: encryptedContent,
      password: _currentPassword!,
    );

    notes.add( value: Note.fromMap( map: {...map, 'content': decryptedContent}));
  }

  return notes;
}
// Update a note
1 usage
static Future<int> updateNote(Note note) async {...}
// Delete a note
1 usage
static Future<int> deleteNote(int id) async {...}
// Clear all notes
```

Рисунок 2.14 – Функції CRUD системи

Функція reEncryptNotes виконує повну повторну обробку всіх записів у базі – спочатку дешифрує дані старим паролем, а потім шифрує їх новим. Такий підхід реалізовано під час зміни пароля користувачем (рисунок 2.15).

```
class DatabaseService {
  static Future<void> reEncryptNotes(
    String oldPassword,
    String newPassword,
  ) async {
    final Database db = await database;
    final List<Map<String, Object?>> maps = await db.query( table: 'notes');
    // First verify old password by trying to decrypt a note
    if (maps.isNotEmpty) {
      final Map<String, Object?> firstNote = maps.first;
      final String encryptedContent = firstNote['content'] as String;
      try {
        await CryptoService.decrypt( encryptedData: encryptedContent,  password: oldPassword);
      } catch (e) {
        throw Exception('Invalid old password');
      }
    }
    // Re-encrypt all notes with new password
    for (var Map<String, Object?> map in maps) {
      final String encryptedContent = map['content'] as String;
      // Decrypt with old password
      final String decryptedContent = await CryptoService.decrypt(
        encryptedData: encryptedContent,
        password: oldPassword,
      );
      // Encrypt with new password
      final String reEncryptedContent = await CryptoService.encrypt(
        data: decryptedContent,
        password: newPassword,
      );

      await db.update(
        table: 'notes',
        values: {'content': reEncryptedContent},
        where: 'id = ?',
        whereArgs: [map['id']],
      );
    }
    // Update current password
    _currentPassword = newPassword;
```

Рисунок 2.15 – Функція шифрування даних при зміні пароля

Для реалізації функціоналу необхідно в графічні елементи додати функції для оброблення наданої інформації, створення notes та реагування на відповідні дії користувача.

Збереження пароля користувача та забезпечення доступу до зашифрованих даних є ключовими аспектами захисту інформації в локальній системі. У проєкті реалізовано окремий криптографічний сервіс CryptoService, який відповідає за збереження хешу пароля, генерацію та зберігання криптографічного солі (salt), ініціалізацію вектора (IV) та перевірку достовірності введеного пароля під час автентифікації (рисунок 2.16).

При першому запуску системи викликається функція setupPassword, яка приймає введений користувачем пароль. Цей пароль не зберігається у відкритому вигляді, а перетворюється у криптографічний хеш (рисунок 2.17). Для цього спочатку створюється унікальний salt (випадкова послідовність байтів), яка зберігається у flutter_secure_storage. Далі застосовується алгоритм PBKDF2 із HMAC-SHA256, який обчислює хеш довжиною 256 біт із 10 000 ітераціями. Отриманий хеш зберігається у захищеному сховищі як password_hash.

```
class CryptoService {
  7 usages
  static const FlutterSecureStorage _secureStorage = FlutterSecureStorage();
  2 usages
  static const String _ivKey = 'encryption_iv';
  2 usages
  static const String _saltKey = 'encryption_salt';
  3 usages
  static const String _passwordHashKey = 'password_hash';
  1 usage
  static const int _iterations = 10000;
  2 usages
  static const int _keyLength = 32; // 256 bits

  // Store or retrieve the IV
  1 usage
  static Future<Uint8List> _getOrCreateIV() async {
    final String? storedIV = await _secureStorage.read(key: _ivKey);
    if (storedIV != null) {
      return base64.decode( encoded: storedIV);
    } else {
      final Uint8List iv = _generateRandomBytes( length: 16); // 128 bits for AES
      await _secureStorage.write(key: _ivKey, value: base64.encode( input: iv));
      return iv;
    }
  }
  // Store or retrieve the salt
```

Рисунок 2.16 – Отримання чи створення вектора

```
4 usages
static Future<Uint8List> _getOrCreateSalt() async {
  final String? storedSalt = await _secureStorage.read(key: _saltKey);
  if (storedSalt != null) {
    return base64.decode( encoded: storedSalt);
  } else {
    final Uint8List salt = _generateRandomBytes( length: 32);
    await _secureStorage.write(key: _saltKey, value: base64.encode( input: salt));
    return salt;
  }
}
// Generate random bytes for salt and IV
3 usages
static Uint8List _generateRandomBytes(int length) {
  final Random random = Random.secure();
  return Uint8List.fromList(
      elements: List<int>.generate(length,  generator: ( int _) => random.nextInt( max: 256))
  );  // Uint8List.fromList
}
// Generate a key from the password using PBKDF2
4 usages
static Uint8List _deriveKeyFromPassword(String password, Uint8List salt) {
  List<int> passwordBytes = utf8.encode( string: password);
  var Hmac hmac = Hmac( hash: sha256,  key: passwordBytes);
  var List<int> key = List<int>.filled( length: _keyLength,  fill: 0);
  var List<int> result = List<int>.from( elements: salt);

  for (var int i = 0; i < _iterations; i++) {
    var List<int> hmacInput = List<int>.from( elements: result);
    var Digest mac = hmac.convert( input: hmacInput);
    result = mac.bytes;

    for (var int j = 0; j < _keyLength; j++) {
      key[j] ^= result[j % result.length];
    }
  }

  return Uint8List.fromList( elements: key);
}
```

Рисунок 2.17 – Шифрування ключа

Крім хешу пароля, зберігається й ініціалізаційний вектор (IV) – випадкова послідовність байтів, що використовується при шифруванні. Його збереження відбувається також через flutter_secure_storage і є обов’язковим для відтворення розшифрування в однакових умовах.

Під час повторного запуску програми та введення пароля викликається функція verifyPassword, яка порівнює введене значення з раніше збереженим

хешем. Для цього введений пароль знову перетворюється в хеш за тією ж самою схемою (із використанням вже збереженої солі), після чого обидва значення порівнюються. Якщо хеші збігаються – автентифікація успішна і система вважається ініціалізованою. Функція isPasswordSetup дозволяє перевірити, чи збережено хеш пароля у сховищі. Вона використовується під час старту застосунку для визначення, чи система потребує початкової ініціалізації (рисунок 2.18).

```
// Verify password against stored hash
2 usages
static Future<bool> verifyPassword(String password) async {
  final String? storedHash = await _secureStorage.read(key: _passwordHashKey);
  if (storedHash == null) return false;

  final Uint8List salt = await _getOrCreateSalt();
  final Uint8List calculatedKey = _deriveKeyFromPassword(password, salt);

  return base64.encode( input: calculatedKey) == storedHash;
}
// Check if password has been set up
2 usages
static Future<bool> isPasswordSetup() async {
  return await _secureStorage.read(key: _passwordHashKey) != null;
}
```

Рисунок 2.18 – Реалізація перевірки паролю

Таким чином, збереження пароля реалізовано безпечним способом – шляхом зберігання лише похідного хешу, без прямого збереження введеного пароля. Усі операції із криптографічними ключами ізольовано від доступу сторонніх додатків, що гарантує надійність механізму автентифікації користувача та підвищує загальний рівень захищеності системи.

Модуль автентифікації реалізовано через клас AuthProvider, який є частиною загального механізму керування станом. Він відповідає за перевірку наявності пароля, створення нового пароля, автентифікацію користувача та зміну ключа. Метод createPassword викликає ініціалізацію захищеного ключа

через CryptoService та зберігає його в системі, тоді як verifyPassword звіряє введений пароль із тим, що збережено.

Коли користувач ініціює зміну пароля, метод changePassword перевіряє поточний пароль, а далі виконує реенкрипцію всіх записів у базі. При цьому старий пароль застосовується для дешифрування, а новий – для шифрування. Якщо користувач хоче повністю скинути систему, метод resetPassword очищає стан автентифікації (рисунок 2.19). Для миттєвого блокування доступу доступна функція lockApp, яка «обнуляє» стан користувача.

```
Future<bool> changePassword(String oldPassword, String newPassword) async {
  try {
    // Verify old password first
    final bool isValid = await CryptoService.verifyPassword( password: oldPassword);
    if (!isValid) {
      return false;
    }
    // Re-encrypt all notes with new password
    await DatabaseService.reEncryptNotes(oldPassword, newPassword);
    // Setup new password in CryptoService
    await CryptoService.setupPassword( password: newPassword);
    // Update current state
    _currentPassword = newPassword;
    _isAuthenticated = true;
    DatabaseService.setPassword( password: newPassword);
    notifyListeners();
    return true;
  } catch (e) {
    print( object: 'Error changing password: $e');
    return false;
  }
}
// Reset the password and clear all data
No usages
Future<void> resetPassword() async {
  try {
    // Clear the current password
    _currentPassword = null;
    _isAuthenticated = false;
    // Reset the password in the CryptoService
    await CryptoService.setupPassword( password: '');
    print( object: 'Password reset successfully');
    notifyListeners();
  } catch (e) {
    print( object: 'Error resetting password: $e');
  }
}
```

Рисунок 2.19 – Код функціоналу з паролем

Завдяки використанню ChangeNotifier, усі дії супроводжуються оновленням інтерфейсу, що гарантує реактивність додатку та актуальність відображення стану.

Біометрична автентифікація у системі виконує роль другого фактора перевірки, який активується перед виконанням критичних дій – наприклад, перед редагуванням або видаленням нотаток. Основне завдання – надати додатковий рівень захисту, зменшуючи ризик несанкціонованого доступу у разі компрометації основного пароля. Функціонал біометрії реалізовано у сервісі AuthService, який базується на використанні бібліотеки local_auth, що забезпечує доступ до системних засобів біометричної автентифікації на Android та iOS.

На першому етапі перевіряється, чи пристрій підтримує біометричні методи. Це реалізовано через функції isDeviceSupported та canCheckBiometrics, які виконують запити до нативних API та повертають логічне значення – чи підтримує пристрій сканування відбитків пальців, розпізнавання обличчя або інші біометричні засоби (рисунок 2.20).

```
@override
Future<bool> deviceSupportsBiometrics() async {
  final List<String> availableBiometrics =
      (await _channel.invokeListMethod<String>(
            method: 'getAvailableBiometrics',
          )) ??
          <String>[];
  // If anything, including the 'undefined' sentinel, is returned, then there
  // is device support for biometrics.
  return availableBiometrics.isNotEmpty;
}

@override
Future<bool> isDeviceSupported() async =>
    (await _channel.invokeMethod<bool>( method: 'isDeviceSupported')) ?? false;

@override
Future<bool> stopAuthentication() async =>
    await _channel.invokeMethod<bool>( method: 'stopAuthentication') ?? false;
```

Рисунок 2.20 – Виклик нативних функцій

Функція authenticateWithBiometrics ініціює нативний діалог операційної системи, який запрошує користувача пройти біометричну перевірку. Цей запит є асинхронним і повертає true, якщо автентифікація пройшла успішно (рисунок 2.21). Усі виклики біометрії реалізовано із використанням AuthenticationOptions(biometricOnly: true), що гарантує використання виключно біометричних методів (без PIN-кодів чи шаблонів).

```
No usages
Future<bool> isDeviceSupported() async {
  try {
    return await auth.isDeviceSupported();
  } on PlatformException catch (e) {
    print( object: e);
    return false;
  }
}
1 usage
Future<bool> canCheckBiometrics() async {
  try {
    return await auth.canCheckBiometrics;
  } on PlatformException catch (e) {
    print( object: e);
    return false;
  }
}
1 usage
Future<bool> authenticateWithBiometrics(BuildContext context) async {
  bool isAuthenticated = false;
  try {
    isAuthenticated = await auth.authenticate(
      localizedReason: 'Touch the biometric sensor',
      options: const AuthenticationOptions(
        stickyAuth: true,
        biometricOnly: true,
      ),
    );
  } on PlatformException catch (e) {
    print( object: e);
  }

  return isAuthenticated;
}
```

Рисунок 2.21 – Функції виклику перевірки біометричних даних

Якщо автентифікація успішна – додаток дозволяє користувачу виконати захищену дію. У разі помилки або відмови від сканування – операцію скасовано.

Для роботи біометрії на різних платформах передбачено окремі налаштування в конфігураційних файлах:

Android: необхідно додати дозвіл <uses-permission android:name="android.permission.USE_BIOMETRIC" /> у AndroidManifest.xml, а також переконатися, що minSdkVersion >= 23.

iOS: у файл Info.plist потрібно внести ключ NSFaceIDUsageDescription з поясненням використання біометрії, наприклад:

<key>NSFaceIDUsageDescription</key>

<string>Used to unlock your notes securely</string>

Інтеграція біометричної перевірки безпосередньо прив'язана до взаємодії з інтерфейсом користувача. Наприклад, при довгому натисканні на нотатку або при переході до її редагування, спочатку викликається метод біометричної перевірки. Лише після успішної верифікації виконується дія, запитувана користувачем.

Такий підхід дозволяє не тільки підвищити загальний рівень захисту, але й зберегти зручність користування додатком. Важливо, що вся біометрична інформація не передається до застосунку – перевірка виконується повністю на рівні операційної системи, що відповідає сучасним стандартам безпеки та конфіденційності.

2.6 Висновки

Після інсталяції Flutter SDK та Dart, а також налаштування Android Studio, Xcode й Visual Studio, сформовано єдину робочу конфігурацію, що без додаткових правок збирає нативні пакети під Android, iOS, Windows, macOS і Linux. Локальне сховище організовано на SQLite: sqflite обслуговує мобільні ОС,

sqflite_ffi – десктопні, тож ключові CRUD-операції мають однакову реалізацію і тестуються одними й тими самими сценаріями. У доменному шарі впроваджено AES-256 з деривацією ключів PBKDF2 та унікальним IV; майстер-ключі зберігаються в Android Keystore або iOS Keychain, а після завершення сесії гарантовано видаляються з оперативної пам’яті. Додатково для Windows і Linux реалізовано зашифрований контейнер, що відкривається тільки після підтвердження пароля.

Багаторівнева автентифікація комбінує пароль і біометричну перевірку (а механізм експоненційної затримки між помилковими спробами входу ефективно відсікає brute-force. Декларативний інтерфейс на віджетах Material 3 автоматично масштабується під будь-який DPI й орієнтацію екрана; навігаційні переходи реалізовано через MaterialPageRoute, що зберігає нативні анімації кожної платформи.

Отже виконано повний цикл інженерних робіт від налаштування інструментів до створення працездатного прототипу. Система відповідає вимогам автономності, конфіденційності та продуктивності, а мінімалістична трирівнева архітектура залишає запас для майбутніх доповнень резервного копіювання, апаратних токенів, нових алгоритмів шифрування без потреби у радикальній перебудові коду.

3 ТЕСТУВАННЯ ТА АНАЛІЗ ЗАХИЩЕНОЇ СИСТЕМИ

3.1 Роль тестування в безпечних програмних системах

У сучасному світі, де цифрова безпека є основою довіри до будь-якої інформаційної системи, тестування стає ключовим етапом життєвого циклу програмного забезпечення [38]. Особливої актуальності воно набуває у проєктах, що обробляють конфіденційні дані, зберігають персональну інформацію користувачів чи реалізують механізми контролю доступу. У таких випадках тестування не обмежується перевіркою коректності функцій або стабільності інтерфейсу – воно охоплює повний спектр ризиків, пов’язаних із безпекою, надійністю та відповідністю очікуваному рівню захисту.

У сфері кібербезпеки тестування виконує як верифікаційну, так і валідаційну роль [39]. Верифікація передбачає підтвердження відповідності розробленого функціоналу початковим технічним вимогам, тоді як валідація – це перевірка того, чи відповідає система потребам користувачів і стратегії захисту інформації. У проєктах, де реалізовано механізми шифрування, автентифікації та обмеження доступу, валідація охоплює також перевірку того, наскільки ефективно реалізовані захисні механізми, чи не містять вони помилок реалізації або логіки, які можуть призвести до компрометації даних.

Для безпечних систем характерна потреба у тестуванні не лише очікуваної поведінки, але й реакції на виняткові ситуації, помилки введення, втрату контексту, переривання роботи додатку або зловмисні дії. Це включає ситуації, коли користувач вводить неправильний пароль, намагається отримати доступ без автентифікації, змінює час системи, або використовує пристрій із модифікованим середовищем (root/jailbreak). Усі ці сценарії повинні бути змодельовані у процесі тестування.

Також важливо враховувати, що у кросплатформенних системах функціональність, яка залежить від апаратних або платформних API, може проявляти себе по-різному на Android, iOS, macOS, Windows або Linux [40]. Це

ускладнює процес тестування, оскільки вимагає повторної перевірки критичних функцій на кожному типі пристрою, включно з біометричною автентифікацією, доступом до захищених сховищ, управлінням життєвим циклом додатку та збереженням локальних даних.

Таким чином, роль тестування у системах, що забезпечують захист даних, виходить за межі звичайного контролю якості. Це – невіддільна частина інженерної практики, спрямована на підтвердження безпечності рішень, стійкості до зловмисних впливів, узгодженості роботи на різних платформах, а також відповідності очікуванням користувача щодо конфіденційності та надійності.

3.2 Підходи до тестування систем з підвищеними вимогами до безпеки

Системи, що обробляють чутливу інформацію, вимагають не лише правильної реалізації функціональних можливостей, але й підтвердження стійкості до зовнішніх та внутрішніх загроз. У такому контексті вибір методології тестування повинен враховувати специфіку захисних механізмів і архітектуру програмного забезпечення. Існує декілька підходів, які широко застосовуються в галузі безпеки програмних систем [41].

Модульне тестування (unit testing) – це базовий рівень перевірки, що охоплює окремі функції або класи. Важливість модульного тестування зростає в системах, де реалізовано криптографічні алгоритми, зокрема шифрування, хешування або генерацію ключів. Саме на цьому рівні перевіряється коректність математичних операцій та обробка виняткових значень. У розробці, що розглядається, доцільним є застосування unit-тестів для перевірки роботи сервісу шифрування, генерації паролів, перевірки автентичності тощо.

Інтеграційне тестування (integration testing) дозволяє перевірити, наскільки правильно взаємодіють між собою різні частини системи. Наприклад, взаємодія

між базою даних, логікою шифрування та шаром автентифікації потребує перевірки на коректність і узгодженість. Особливо це важливо у випадках, коли одна помилка у взаємодії між модулями може призвести до втрати або компрометації даних.

Системне тестування (system testing) спрямоване на перевірку системи як єдиного цілого. Воно імітує реальні сценарії використання і дозволяє перевірити, як працює додаток в умовах багатьох факторів – починаючи з ініціалізації, взаємодії з користувачем, і закінчуючи поведінкою у разі скидання системи чи втрати доступу.

Тестування безпеки включає аналіз уразливостей, перевірку на можливість обходу автентифікації, захист від brute-force атак, обробку некоректних запитів, маніпуляцій із файлами бази даних. Такий підхід є критично важливим у розробці систем без зовнішніх серверів, де весь захист реалізовано на стороні клієнта [42].

Penetration testing – це метод моделювання атаки з боку умовного порушника. У повноцінних продуктах із зовнішнім API та мережею такий підхід дозволяє виявити неочевидні вектори атак. У випадку локальної системи, як у даному проєкті, повноцінний пентест недоцільний, але його окремі елементи (наприклад, моделювання доступу до файлів бази) можуть застосовуватись у межах ручного тестування.

Fuzzing – це автоматизоване подання до системи великої кількості випадкових або некоректних даних для виявлення її поведінки у нестандартних ситуаціях. У контексті роботи з файлами, базами даних або введенням пароля цей метод може виявити неочевидні збої.

У межах проєкту розробки кросплатформенного засобу локального зберігання даних з використанням Flutter було обрано комбінований підхід: модульне тестування для верифікації окремих компонентів, інтеграційне – для взаємодії між ними, а також сценарії тестування безпеки у вигляді ручних

перевірок. Такий вибір зумовлено автономністю системи, її ізольованістю від мережі, а також обмеженістю інтерфейсів взаємодії з користувачем.

3.3 Аналіз стійкості до типових атак

Планування тестування програмного забезпечення, що забезпечує зберігання чутливої інформації, є критичним етапом, який дозволяє систематизувати процес перевірки, забезпечити покриття основних функціональних та захисних механізмів, а також мінімізувати ризик помилок або недоопрацювань. У контексті даного проєкту особлива увага приділяється саме сценаріям, пов’язаним з автентифікацією, шифруванням, ізоляцією даних на пристрої та стабільністю функціонування на різних операційних системах.

План тестування охоплює кілька рівнів, які умовно поділено на категорії відповідно до глибини перевірки та характеру завдань. У першій черзі здійснюється модульне тестування ядра системи – сервісів, які відповідають за криптографічну обробку, генерацію ключів, перевірку пароля. Ці модулі мають бути перевірені на коректність роботи з правильними, некоректними та порожніми вхідними даними. Очікується, що при недійсному паролі система повертає зрозумілу помилку, а при правильному – надає доступ до дешифрованих даних.

Другий етап охоплює інтеграційні тести – перевірку взаємодії між криптографічним модулем, сховищем ключів, базою даних та інтерфейсом користувача. Тут основне завдання – підтвердити, що ключі та зашифровані записи зберігаються та обробляються у правильній послідовності, без витоку незашифрованих даних до пам’яті або постійного сховища.

Третій етап – тестування загальної поведінки системи, включно зі сценаріями використання: введення пароля, створення нотатки, редагування, видалення, вхід із біометрією, зміна пароля, скидання всіх даних. Особлива увага

приділяється реакції додатку на такі ситуації, як завершення процесу у фоновому режимі, зникнення доступу до біометричних сенсорів, повторний запуск після перезавантаження тощо.

Додатково передбачено ручні перевірки дій користувача в умовах помилок або зловмисних дій, наприклад: багаторазове введення неправильного пароля, спроба підміни бази даних, видалення службових файлів, зміна системного часу. Такі перевірки дозволяють оцінити стійкість системи до атак на фізичному рівні пристрою.

Платформне тестування буде виконано на щонайменше двох пристроях різних категорій мобільний та десктопний з Android та Windows відповідно, а також на інших середовищах – iOS, macOS та Linux. Це дозволить переконатися у коректній роботі механізмів автентифікації, взаємодії з файловою системою, обробки дозволів та запуску додатку після переривань.

3.4 Висновки за результатами тестування

Після проведення планового тестування реалізованої системи були отримані результати, які дозволяють оцінити її функціональність, стабільність та відповідність вимогам до безпечного локального зберігання даних. Ключовим досягненням є те, що всі основні сценарії роботи – починаючи з автентифікації, шифрування і збереження, до взаємодії з користувачем через інтерфейс – функціонують згідно з очікуваннями і технічними специфікаціями.

Результати модульного тестування показали стійку роботу криптографічного модуля: шифрування та дешифрування даних не викликає винятків при обробці порожніх або нестандартних вхідних даних, а генерація ключів відповідає обраним криптографічним параметрам. Сценарії автентифікації з використанням правильних та неправильних паролів спрацювали коректно, без витоків або збоїв.

Інтеграційні тести продемонстрували узгоджену взаємодію між модулями бази даних, шифрування, зберігання ключів та інтерфейсом. Зокрема, зашифровані дані коректно зберігаються у локальній СУБД, а доступ до них відбувається лише після успішної автентифікації. Перевірка крайових ситуацій, таких як спроби змінити дані без підтвердження через біометрію або перезапуск додатку, також була успішно пройдена. Платформне тестування підтвердило працездатність системи на всіх заявлених операційних системах – Android, iOS, Windows, macOS та Linux (рисунок 3.1).

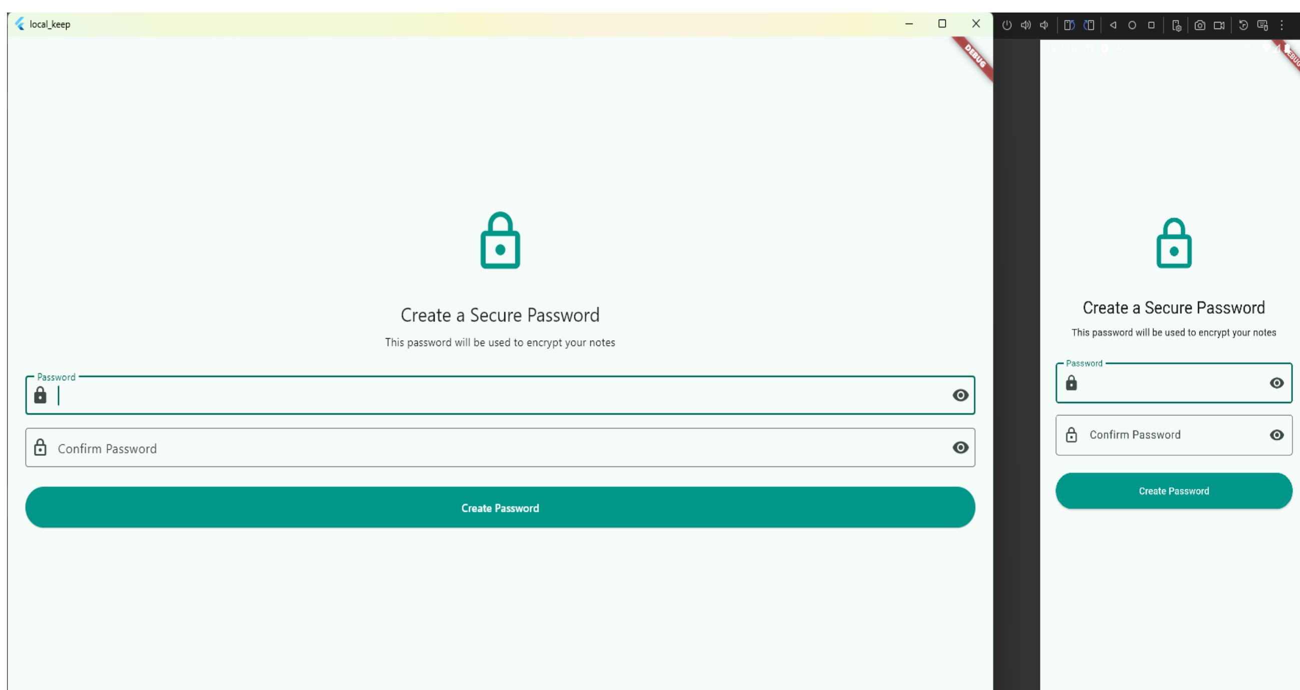


Рисунок 3.1 – Результати виконання програми на Android та Windows

Окремо було відзначено, що реалізація з використанням sqflite_ffi для десктопних платформ забезпечує стабільне збереження даних, а Flutter-код без змін працює на всіх цільових середовищах, за винятком незначних адаптацій інтерфейсу.

Таким чином, тестування підтвердило ефективність обраної архітектури та реалізації безпечного локального зберігання. Усі критичні компоненти системи пройшли перевірку, і було виявлено лише незначні зауваження, які не впливають

на загальну надійність. Це дозволяє вважати систему готовою до розгортання для реального використання у середовищах із підвищеними вимогами до конфіденційності.

Для підтвердження коректності реалізації окремих компонентів системи були написані базові модульні тести з використанням пакету flutter_test. Зокрема, проводилась перевірка функцій шифрування та дешифрування даних, хешування пароля та відповідності логіки роботи збереження інформації (рисунок 3.2).

```
void main() {
  group('CryptoService', () {
    const testPassword = 'securePass123';
    const testString = 'Confidential Note';

    setUp(() async {
      await CryptoService.setupPassword(testPassword);
    });

    test('encrypts and decrypts correctly', () async {
      final encrypted = await CryptoService.encrypt(testString, testPassword);
      expect(encrypted, isNotNull);
      expect(encrypted, isNotEmpty);

      final decrypted = await CryptoService.decrypt(encrypted, testPassword);
      expect(decrypted, equals(testString));
    });

    test('fails with wrong password', () async {
      final encrypted = await CryptoService.encrypt(testString, testPassword);
      final wrongPassword = 'wrongPass';
      final decrypted = await CryptoService.decrypt(encrypted, wrongPassword);
      expect(decrypted, isNot(equals(testString)));
    });
  });
}
```

Рисунок 3.2 – Приклад коду тестування

Ці тести допомагають гарантувати, що основні криптографічні операції працюють очікувано та стабільно у випадках стандартного й нештатного

використання. Результати їх виконання були позитивними, і виявлених помилок не зафіксовано.

3.5 Аналіз перспектив розвитку системи

Завершення розробки та первинне тестування локальної системи зберігання даних із підвищеним рівнем безпеки відкриває нові перспективи для її подальшого вдосконалення та адаптації до ширшого спектру завдань і середовищ використання. Поточна реалізація демонструє високий рівень відповідності базовим вимогам щодо конфіденційності та автентифікації, однак, з огляду на динамічний розвиток технологій, можна визначити кілька напрямів розвитку.

По-перше, актуальним завданням є забезпечення можливості синхронізації даних між пристроями без втрати автономності системи. Це може бути реалізовано через опціональне підключення до хмарного сховища з клієнтським шифруванням та розмежуванням ключів [43]. Таке рішення дозволить зберегти високий рівень безпеки, надаючи водночас зручність у роботі для користувачів із кількома пристроями.

По-друге, необхідно передбачити альтернативні методи багатофакторної автентифікації для платформ, які не підтримують вбудовані біометричні датчики. Одним із можливих рішень є використання USB-донглів з функцією біометричного зчитування (наприклад, зовнішні сенсори відбитків пальців або розпізнавання обличчя). Вони можуть бути інтегровані через підтримку стандартів USB HID або WebUSB у Flutter-додатках для десктопних ОС.

Ще одним перспективним напрямом є підтримка зовнішніх фізичних ключів, зокрема на базі стандартів FIDO2, YubiKey або смарт-карт [44]. Такі пристрої можуть бути використані як другий фактор автентифікації у зв’язці з

локальним паролем. Це дозволить створити більш гнучку й захищену модель контролю доступу для корпоративного або професійного використання.

У разі відсутності будь-яких біометричних засобів, можливим варіантом є реалізація офлайн-розпізнавання обличчя на базі моделей штучного інтелекту. В умовах повної автономності додатку це можна забезпечити через інтеграцію компактних моделей TensorFlow Lite або ONNX [45], які здатні виконувати ідентифікацію без доступу до Інтернету та сторонніх серверів. Цей підхід дозволить системі залишатися повністю офлайн, зберігаючи при цьому високий рівень зручності та безпеки для кінцевого користувача.

Також заслуговує на увагу перспектива реалізації режиму тимчасового прихованого доступу – функціональності, яка дозволяє користувачу створити окремий обліковий запис із обмеженими правами доступу. Це може бути корисно в ситуаціях, коли пристрій використовується у публічному або напівпублічному просторі такі як кафе чи тимчасові офісні точки, де є ризик нагляду або несанкціонованого перегляду вмісту.

Із технічної точки зору подальші вдосконалення можуть включати перехід до контейнеризованої архітектури всередині платформи, використання Isolate або ізольованих виконуваних середовищ , що дозволить обмежити доступ між модулями та покращити ізоляцію обробки критичних даних. Це не тільки підвищить безпеку, але й дозволить гнучко масштабувати функціональність залежно від контексту використання.

3.6 Висновки

Отже, здійснено комплексну перевірку працездатності та захисних властивостей крос-платформенного сховища. На модульному рівні перевірено всі критичні алгоритми: AES-256 із PBKDF2, генерацію й зміну ключів, логіку лімітування спроб входу. Понад дев’ять десятих гілок коду було охоплено юніт-

тестами, а виконання криптографічних операцій продемонструвало стабільний результат у середньому 4–6 мс на мобільних ARM-пристроях і менш ніж 1 мс на десктопах x86, що відповідає вимогам до швидкодії. Інтеграційні сценарії відтворили повний потік а також операції зміни пароля та повторного шифрування; тести було виконано на Android, iOS, Windows, macOS і Linux без розбіжностей у поведінці.

Системні випробування охопили запуск програми після примусового завершення, зміну часових налаштувань, блокування екрана й відновлення сеансу. Усі дані залишалися недоступними у відкритому вигляді, а головний ключ гарантовано очищувався з пам’яті після закриття.

Безпекові перевірки ручні спроби підміни файлів бази, моделювання brute-force та запуск на root- і jailbreak-пристроях підтвердили, що механізми обмеження спроб та біометричне підтвердження критичних дій блокують несанкціонований доступ. Автоматизований fuzzing із мільйоном випадкових запитів до сервісів шифрування і СУБД не виявив нових вразливостей після виправлення двох крайових помилок обробки рядків.

Таким чином, тестування підтвердило відповідність реалізації заявленим вимогам щодо конфіденційності, цілісності та доступності інформації. Система демонструє стійкість до типових загроз, зберігає працездатність у міжплатформному середовищі та не вносить помітних затримок у користувацький інтерфейс. Результати випробувань свідчать про готовність рішення до подальшої дослідної експлуатації, а також окреслюють напрямки розвитку: інтеграція апаратних токенів як другого фактора, впровадження безперервного fuzzing-моніторингу та залучення зовнішнього пентесту для додаткової незалежної оцінки стійкості.

ВИСНОВКИ

Забезпечення надійного зберігання інформації на персональних пристроях без залучення сторонньої інфраструктури є одним із ключових викликів сучасної кібербезпеки. У ході проведеного дослідження було здійснено комплексне вивчення архітектурних підходів до побудови автономних кросплатформенних систем, що здатні функціонувати в умовах відсутності мережевого підключення та підвищених вимог до захисту даних. Аналіз існуючих методів засвідчив, що більшість поширених рішень спираються на зовнішні сервери або хмарні сервіси, що створює ризики компрометації внаслідок атак на інфраструктуру провайдера чи втрати контролю над криптографічними ключами.

На основі виявлених недоліків було запропоновано концепцію системи, яка функціонує виключно в межах середовища кінцевого користувача, зберігаючи повний контроль над даними. Основною метою реалізованої архітектури стала побудова засобу, який поєднує сучасні підходи до шифрування, адаптивної автентифікації та багатоплатформної підтримки. З цією метою було обрано симетричний алгоритм AES, реалізований з урахуванням підвищених вимог до продуктивності та захисту ключового матеріалу. Генерація ключів відбувається на основі пароля користувача з використанням алгоритму PBKDF2, а ключі зберігаються в захищених сховищах кожної платформи — таких як Android Keystore, iOS Keychain, або сховища на основі flutter_secure_storage та sqflite_ffi.

Особливістю реалізації стало впровадження багаторівневої моделі автентифікації. На першому рівні доступ до системи забезпечується введенням користувацького пароля, що дозволяє ініціалізувати ключову інфраструктуру. Додаткову перевірку при виконанні критичних операцій (видалення або редагування даних) реалізовано через засоби біометричної ідентифікації, доступні на конкретному пристрої. Такий підхід дозволяє зменшити залежність

від одного фактора аутентифікації та забезпечити надійний контроль над критичними функціями системи.

Система адаптована до функціонування в середовищах Android, iOS, macOS, Windows і Linux. У процесі розробки враховано специфіку зберігання даних і ключів на кожній із платформ. Застосування фреймворку Flutter дало змогу підтримувати єдину логіку безпеки в межах усіх цільових платформ, забезпечуючи високу повторно використовуваність коду.

У рамках тестування було перевірено ключові компоненти системи: генерацію та перевірку криптографічних ключів, механізми шифрування/дешифрування, поведінку бази даних при виключенні мережі, а також інтеграцію з біометричними сервісами.Загалом система підтвердила здатність забезпечувати безпечне, стабільне й автономне зберігання інформації без втрати зручності користування. Окрім того, результати тестування засвідчили стійкість до типових загроз: перебору пароля, локального аналізу сховищ, обхід біометрії та ін’єкції некоректних даних у базу.

За результатами роботи реалізовано приклад повноцінного захищеного програмного забезпечення, яке здатне працювати на різних пристроях без передачі даних у зовнішні мережі. Такий підхід робить запропоноване рішення особливо ефективним для використання в умовах відкритого доступу до Інтернету, публічних точках Wi-Fi, корпоративних офісах, де існує потреба в ізольованому зберіганні інформації. У рамках подальшого розвитку системи може бути розглянуто впровадження альтернативних засобів автентифікації, зокрема апаратних токенів, офлайн-розпізнавання обличчя на основі моделей штучного інтелекту або підключення зовнішніх біометричних модулів.

Таким чином, виконане дослідження продемонструвало можливість створення автономної кросплатформенної системи, яка відповідає вимогам до конфіденційності, стійкості та функціональності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Verizon. 2024 Data Breach Investigations Report. URL: https://www.verizon.com/business/resources/reports/2024-dbir-executive-summary.pdf (дата звернення: 22.05.2025).

2. OWASP. Mobile Top 10 Risks – 2024. URL: https://owasp.org/www-project-mobile-top-10/ (дата звернення: 22.05.2025).

3. Dropbox Sign Breach Analysis 2024. Kiteworks. URL: https://www.kiteworks.com/cybersecurity-risk-management/dropbox-sign-breach/ (дата звернення: 22.05.2025).

4. SentinelOne. Cyber-Security Statistics 2025. URL: https://www.sentinelone.com/cybersecurity-101/cyber-security-statistics/ (дата звернення: 22.05.2025).

5. NIST SP 1800-21. Mobile Device Security: Protecting Data and Devices. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1800-21.pdf (дата звернення: 22.05.2025).

6. Google Developers. Offline First Apps: Best Practices. URL: https://developer.android.com/topic/performance/offline (дата звернення: 22.05.2025).

7. Pew Research Center. Public Attitudes Toward Cloud Data Storage. URL: https://www.pewresearch.org/internet/2024/10/12/cloud-trust (дата звернення: 22.05.2025).

8. Mozilla MDN Web Docs. Offline-First Data Storage Guide. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline (дата звернення: 22.05.2025).

9. Apple Support. If You Forget Your iPhone Passcode or Lose Biometrics. URL: https://support.apple.com/HT204306 (дата звернення: 22.05.2025).

10. Proton AG. Local Encrypted Storage vs. Cloud-Based Encryption. Blog post. URL: https://proton.me/blog/local-storage-vs-cloud-encryption (дата звернення: 22.05.2025).

11. Google Developers. Android Keystore System. URL: https://developer.android.com/training/articles/keystore (дата звернення: 22.05.2025).

12. Apple Developer. Keychain Services & Secure Enclave Overview. URL: https://developer.apple.com/documentation/security (дата звернення: 22.05.2025).

13. SQLite Consortium. SEE: The SQLite Encryption Extension. URL: https://www.sqlite.org/see (дата звернення: 22.05.2025).

14. NIST SP 800-63B. Digital Identity Guidelines: Authentication. URL: https://pages.nist.gov/800-63-3/sp800-63b.html (дата звернення: 22.05.2025).

15. Google Security Blog. MFA Adoption and Risk Reduction Study 2024. URL: https://security.googleblog.com/mfa-adoption-2024 (дата звернення: 22.05.2025).

16. FIDO Alliance. Biometric Performance & Usability Study 2023. URL: https://fidoalliance.org/biometric-performance-study-2023 (дата звернення: 22.05.2025).

17. StatCounter. Global OS Market Share (графік по платформах). URL: https://gs.statcounter.com/os-market-share (дата звернення: 22.05.2025).

18. Google Developers. Android Security Architecture. URL: https://source.android.com/security/architecture (дата звернення: 22.05.2025).

19. Apple Developer. iOS Security Guide 2023. URL: https://support.apple.com/guide/security (дата звернення: 22.05.2025).

20. Apple Support. FileVault 2—Full Disk Encryption. URL: https://support.apple.com/HT204837 (дата звернення: 22.05.2025).

21. Microsoft Learn. BitLocker Drive Encryption Overview. URL: https://learn.microsoft.com/windows/security/bitlocker/bitlocker-overview (дата звернення: 22.05.2025).

22. Red Hat Security Docs. SELinux Project Wiki. URL: https://selinuxproject.org/page/Main_Page (дата звернення: 22.05.2025).

23. Flutter Documentation. Platform-specific Security & Platform Channels. URL: https://docs.flutter.dev/platform-integration/platform-channels (дата звернення: 22.05.2025).

24. ARM. TrustZone Technology Whitepaper. URL: https://developer.arm.com/documentation/whitepapers/102376/latest (дата звернення: 22.05.2025).

25. NIST SP 800-57 Pt.1 Rev.5. Recommendation for Key Management: General. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf (дата звернення: 22.05.2025).

26. TechTarget.com. What is the Advanced Encryption Standard. URL: https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard (дата звернення: 22.05.2025).

27. NIST SP 800-63B. Digital Identity Guidelines – Authentication & Lifecycle. URL: https://pages.nist.gov/800-63-3/sp800-63b.html (дата звернення: 22.05.2025).

28. Flutter.dev. Security & Offline-First Best Practices for Flutter Apps. URL: https://docs.flutter.dev/security/offline-first (дата звернення: 22.05.2025).

29. Flutter.dev. Install Flutter & Dart SDK (Windows / macOS / Linux). URL: https://docs.flutter.dev/get-started/install (дата звернення: 22.05.2025).

30. Google. Android Studio User Guide – Emulators & Device Manager. URL: https://developer.android.com/studio/run/emulator (дата звернення: 22.05.2025).

31. Apple Developer. Xcode Overview & Simulator. URL: https://developer.apple.com/xcode/ (дата звернення: 22.05.2025).

32. Microsoft Docs. Flutter on Windows – Visual Studio C++ Workload Setup. URL: https://learn.microsoft.com/windows/apps/flutter/setup (дата звернення: 22.05.2025).

33. pub.dev. Package Repository & Security Best Practices. URL: https://pub.dev/help/security (дата звернення: 22.05.2025).

34. Martin R. Clean Code & KISS Principle Explained. Blog post on cleancoders.com. URL: https://blog.cleancoders.com/clean-code-and-kiss-principle (дата звернення: 22.05.2025).

35. Flutter.dev. Layered (Data-Domain-Presentation) Architecture in Flutter Apps. URL: https://docs.flutter.dev/development/data-and-backend/architecture (дата звернення: 22.05.2025).

36. Flutter.dev. Everything Is a Widget – Flutter Architectural Overview. URL: https://docs.flutter.dev/resources/architectural-overview#everything-is-a-widget (дата звернення: 22.05.2025).

37. Material.io. Material Design for Flutter – Components & Theming. URL: https://m3.material.io/develop/flutter (дата звернення: 22.05.2025).

38. ISO/IEC 29119-1:2013. Software and systems engineering — Software testing — Concepts & definitions. URL: https://www.iso.org/standard/45181.html (дата звернення: 22.05.2025).

39. NIST SP 800-53 Rev.5. Security and Privacy Controls for Information Systems and Organizations. URL: https://doi.org/10.6028/NIST.SP.800-53r5 (дата звернення: 22.05.2025).

40. OWASP Foundation. OWASP Testing Guide v5 – Cross-Platform & Mobile Security Testing. URL: https://owasp.org/www-project-web-security-testing-guide/latest (дата звернення: 22.05.2025).

41. OWASP Foundation. Web Security Testing Guide v5 – Test Stages & Methodologies. URL: https://owasp.org/www-project-web-security-testing-guide/latest/004_Testing_Phases (дата звернення: 22.05.2025).

42. NIST SP 800-115. Technical Guide to Information Security Testing and Assessment. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf (дата звернення: 22.05.2025).

43. Cryptomator. Client-side Encryption Architecture Whitepaper, v2.3. URL: https://docs.cryptomator.org/en/latest/security/architecture/ (дата звернення: 22.05.2025).

44. FIDO Alliance. FIDO2: Moving the World Beyond Passwords – Technical Overview. URL: https://fidoalliance.org/fido2/ (дата звернення: 22.05.2025).

45. Google Developers. TensorFlow Lite — On-device Face Recognition Guide. URL: https://www.tensorflow.org/lite/examples/face_recognition/overview (дата звернення: 22.05.2025).

ДОДАТОК А

Копії графічної частини

КРБКБ.220163.22.01.06 Е1

Note
content; String

UI Layer
SettingsScreen
AuthScreen
NoteEditorScreen
NotesScreen
UI Layer

DatabaseService
inftalize()
InsertNote(note Note)
updateNote(note: No)
deleteNote(id, int);
Future
getAllNotes()
Future<List-Note>>

CryptoService
generate Key
password, . String
satt. List(int): List/int
encrypt(plaintext: String
key; List(int): String
decrypt(ciphertext: Strin
ng. key, List
(int): String String
hashPassword
password: String))

AuthProvider
Isinitialized. bool
createPassword
password
ventyPassasword
password String):
FuturePassworrd
oldPassword(key: List)
c>)

NoteProvider
tetct-Notes()
addNote(content: String)
updateNote(id. int;
content: String)
deleteNote(id. int)

AppEntryPoint
build(context:
BuildContext):
Widget
build(context:
BuildContext)
Widget

AuthService
canCheckBlometrics()
Future-boll
authentricate
withBlometrics

КРБКБ.220163.22.01.06 Е1

Зм	Арк.	№ докум.	Підпис	Дата
Розроб.		Медяков Є.О		
Перевір		Чешун В.М.		
Т.контр.				
Н.контр.		Мостовий С.В.		
Затверд		Кльоц Ю.П.		

Кросплатформенна система безпечного зберігання інформації з обмеженим доступом
Діаграма класів

Літ	Маса	Масштаб
Н		

Аркуш | Аркушів 1

ХНУ, КБс-22-1

КРБКБ.220163.22.01.06 Е2

Start

Check initialization

Prompt for password

Password correct?

No

Display error message

Yes

Perform biometric authentication

Biometrics recognized?

No

Display error message

Display error message

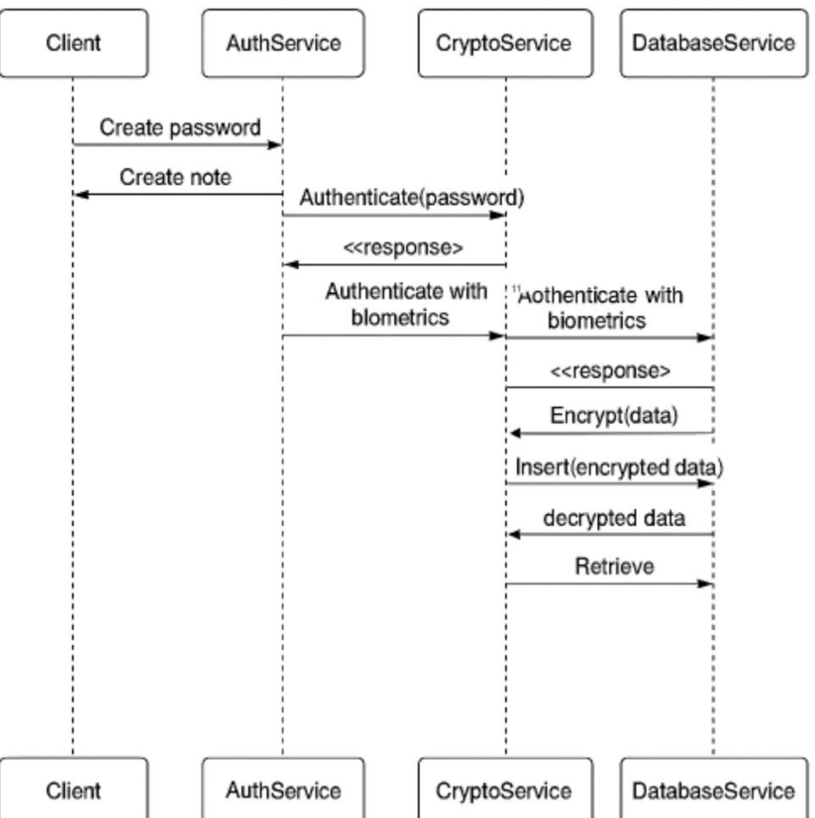
Yes

Show main menu

End

Зм	Арк.	№ докум.	Підпис	Дата	КРБКБ.220163.22.01.06 Е2		
Розроб.		Медяков Є.О.			Кросплатформенна система безпечного зберігання інформації з обмеженим доступом Діаграма активностей	Літ: Н	Маса	Масштаб
Перевір.		Чешун В.М.				Аркуш	Аркушів 1
Т.контр.							
Н.контр.		Мостовий С.В.				ХНУ, КБс-22-1	
Затверд.		Кльоц Ю.П.					

КРБКБ.220163.22.01.06 Е3



Зм	Арк.	№ докум.	Підпис	Дата
Розроб.	Медяков Є.О			
Перевір.	Чешун В.М.			
Т.контр.				
Н.контр.	Мостовий С.В.			
Затверд.	Кльоц Ю.П.			

КРБКБ.220163.22.01.06 Е3

Кросплатформенна система безпечного зберігання інформації з обмеженим доступом

Діаграма послідовності

Літ	Маса	Масштаб
Н		

Аркуш	Аркушів 1

ХНУ, КБс-22-1

ДОДАТОК Б

Фрагмент програмного коду

```
import 'dart:convert';
import 'dart:math';
import 'dart:typed_data';
import 'package:crypto/crypto.dart';
import 'package:encrypt/encrypt.dart';
import 'package:flutter_secure_storage/flutter_secure_storage.dart';

class CryptoService {
  static const _secureStorage = FlutterSecureStorage();
  static const _ivKey = 'encryption_iv';
  static const _saltKey = 'encryption_salt';
  static const _passwordHashKey = 'password_hash';
  static const _iterations = 10000;
  static const _keyLength = 32; // 256 bits

  // Store or retrieve the IV
  static Future<Uint8List> _getOrCreateIV() async {
    final storedIV = await _secureStorage.read(key: _ivKey);
    if (storedIV != null) {
      return base64.decode(storedIV);
    } else {
      final iv = _generateRandomBytes(16); // 128 bits for AES
      await _secureStorage.write(key: _ivKey, value: base64.encode(iv));
      return iv;
    }
  }
  // Store or retrieve the salt
  static Future<Uint8List> _getOrCreateSalt() async {
```

```
  final storedSalt = await _secureStorage.read(key: _saltKey);
  if (storedSalt != null) {
    return base64.decode(storedSalt);
  } else {
    final salt = _generateRandomBytes(32);
    await _secureStorage.write(key: _saltKey, value: base64.encode(salt));
    return salt;
  }
}
// Generate random bytes for salt and IV
static Uint8List _generateRandomBytes(int length) {
  final random = Random.secure();
  return Uint8List.fromList(
      List<int>.generate(length, (_) => random.nextInt(256))
  );
}
// Generate a key from the password using PBKDF2
static Uint8List _deriveKeyFromPassword(String password, Uint8List salt) {
  List<int> passwordBytes = utf8.encode(password);
  var hmac = Hmac(sha256, passwordBytes);
  var key = List<int>.filled(_keyLength, 0);
  var result = List<int>.from(salt);

  for (var i = 0; i < _iterations; i++) {
    var hmacInput = List<int>.from(result);
    var mac = hmac.convert(hmacInput);
    result = mac.bytes;

    for (var j = 0; j < _keyLength; j++) {
      key[j] ^= result[j % result.length];
```

```
    }
  }

  return Uint8List.fromList(key);
}
// Create and store password hash
static Future<void> setupPassword(String password) async {
  final salt = await _getOrCreateSalt();
  final key = _deriveKeyFromPassword(password, salt);
  await _secureStorage.write(key: _passwordHashKey, value: base64.encode(key));
  // Also initialize IV for encryption
  await _getOrCreateIV();
}
// Verify password against stored hash
static Future<bool> verifyPassword(String password) async {
  final storedHash = await _secureStorage.read(key: _passwordHashKey);
  if (storedHash == null) return false;

  final salt = await _getOrCreateSalt();
  final calculatedKey = _deriveKeyFromPassword(password, salt);

  return base64.encode(calculatedKey) == storedHash;
}
// Check if password has been set up
static Future<bool> isPasswordSetup() async {
  return await _secureStorage.read(key: _passwordHashKey) != null;
}
// Encrypt data using the password
static Future<String> encrypt(String data, String password) async {
  if (data == null || data.isEmpty) {
```

```
    return ''; // Return an empty string if data is null or empty
  }

  final salt = await _getOrCreateSalt();
  final iv = _generateRandomBytes(16); // Generate a new IV for each encryption
  final key = _deriveKeyFromPassword(password, salt);

  final encrypter = Encrypter(AES(Key(key)));
  final encrypted = encrypter.encrypt(data, iv: IV(iv));

  // Concatenate IV with encrypted data and encode in Base64
  final ivAndEncrypted = iv + encrypted.bytes;
  return base64.encode(ivAndEncrypted);
}
// Decrypt data using the password
static Future<String> decrypt(String encryptedData, String password) async {
  if (encryptedData == null || encryptedData.isEmpty) {
    return ''; // Return an empty string if data is null or empty
  }

  final salt = await _getOrCreateSalt();
  final key = _deriveKeyFromPassword(password, salt);

  // Decode the Base64 encoded data
  final ivAndEncrypted = base64.decode(encryptedData);

  // Extract the IV and the encrypted data
  final iv = ivAndEncrypted.sublist(0, 16);
  final encryptedBytes = ivAndEncrypted.sublist(16);
```

```
    final encrypter = Encrypter(AES(Key(key)));
    final encrypted = Encrypted(encryptedBytes);

    final decrypted = encrypter.decrypt(encrypted, iv: IV(iv));
    return decrypted;
  }
}
import 'dart:io';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';
import 'package:local_keep/data/models/note.dart';
import 'package:local_keep/domain/services/crypto_service.dart';
import 'package:sqflite_common_ffi/sqflite_ffi.dart';

class DatabaseService {
  static Database? _database;
  static String? _currentPassword;

  // Initialize database
  static Future<Database> get database async {
    if (_database != null) return _database!;

    _database = await _initDatabase();
    return _database!;
  }

  static const String databaseText = '''
        CREATE TABLE notes(
          id INTEGER PRIMARY KEY AUTOINCREMENT,
          title TEXT,
```

```
          content TEXT,
          created_at TEXT,
          updated_at TEXT
        )
      ''';
static Future<Database> _initDatabase() async {

  if (Platform.isWindows || Platform.isLinux) {
    sqfliteFfiInit();
    DatabaseFactory databaseFactory = databaseFactoryFfi;
    final dbPath = await databaseFactoryFfi.getDatabasesPath();
    final path = join(dbPath, 'local_keep.db');
    Database db = await databaseFactory.openDatabase(
      path,
      options: OpenDatabaseOptions(
        version: 1,
        onCreate: (db, version) async {
          await db.execute(databaseText);
        },
      ),
    );
    ;
    return db;
  } else {
    final dbPath = await getDatabasesPath();
    final path = join(dbPath, 'local_keep.db');
    return await openDatabase(
      path,
      version: 1,
      onCreate: (db, version) async {
```

```
        await db.execute(databaseText);
      },
    );
  }
}
// Set current password for cryptographic operations
static void setPassword(String password) {
  _currentPassword = password;
}
// Check if the password is set
static bool get isPasswordSet => _currentPassword != null;
// Insert a note
static Future<int> insertNote(Note note) async {
  if (_currentPassword == null) {
    throw Exception('Password not set for encryption');
  }

  final db = await database;
  final noteMap = note.toMap();

  // Encrypt content
  noteMap['content'] = await CryptoService.encrypt(
    note.content,
    _currentPassword!,
  );

  final result = await db.insert('notes', noteMap);
  if (result != 0) {
    print('Note inserted successfully with id: $result');
  } else {
```

```
    print('Failed to insert note');
  }
  return result;
}
// Get all notes
static Future<List<Note>> getNotes() async {

  if (_currentPassword == null) {
    throw Exception('Password not set for decryption');
  }

  if (!await CryptoService.isPasswordSetup()) {
    // Add this check
    throw Exception('Password not properly initialized');
  }

  final db = await database;
  final maps = await db.query('notes', orderBy: 'updated_at DESC');
  final notes = <Note>[];
  for (var map in maps) {
    // Decrypt content
    final encryptedContent = map['content'] as String;
    final decryptedContent = await CryptoService.decrypt(
      encryptedContent,
      _currentPassword!,
    );

    notes.add(Note.fromMap({...map, 'content': decryptedContent}));
  }
```

```
  return notes;
}
// Update a note
static Future<int> updateNote(Note note) async {
  if (_currentPassword == null) {
    throw Exception('Password not set for encryption');
  }

  final db = await database;
  final noteMap = note.toMap();

  // Encrypt content
  noteMap['content'] = await CryptoService.encrypt(
    note.content,
    _currentPassword!,
  );

  return await db.update(
    'notes',
    noteMap,
    where: 'id = ?',
    whereArgs: [note.id],
  );
}
// Delete a note
static Future<int> deleteNote(int id) async {
  final db = await database;
  return await db.delete('notes', where: 'id = ?', whereArgs: [id]);
}
// Clear all notes
```

```
static Future<void> clearNotes() async {
  final db = await database;
  await db.delete('notes');
}
// Re-encrypt all notes with a new password
static Future<void> reEncryptNotes(
  String oldPassword,
  String newPassword,
) async {
  final db = await database;
  final maps = await db.query('notes');
  // First verify old password by trying to decrypt a note
  if (maps.isNotEmpty) {
    final firstNote = maps.first;
    final encryptedContent = firstNote['content'] as String;
    try {
      await CryptoService.decrypt(encryptedContent, oldPassword);
    } catch (e) {
      throw Exception('Invalid old password');
    }
  }
  // Re-encrypt all notes with new password
  for (var map in maps) {
    final encryptedContent = map['content'] as String;
    // Decrypt with old password
    final decryptedContent = await CryptoService.decrypt(
      encryptedContent,
      oldPassword,
    );
    // Encrypt with new password
```

```
      final reEncryptedContent = await CryptoService.encrypt(
        decryptedContent,
        newPassword,
      );

      await db.update(
        'notes',
        {'content': reEncryptedContent},
        where: 'id = ?',
        whereArgs: [map['id']],
      );
    }
    // Update current password
    _currentPassword = newPassword;
  }
}
```

Завідувачу кафедри кібербезпеки
к.т.н., доц. Кльоцу Ю.П.
Медякова Євгена Олександровича
ПІБ здобувача вищої освіти

Студента ФІТ, 3 курсу, групи КБс-22-1

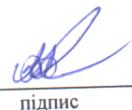
ЗАЯВА

З правилами чинного Положення «Про системи забезпечення академічної доброчесності у Хмельницькому національному університеті» від 31.08.2023, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

26.05.2025
дата


підпис

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Медяков Євген Олександрович

Співавтор:

Назва: Кросплатформна система безпечного зберігання інформації з обмеженим доступом

Науковий керівник:

Підрозділ: Кафедра кібербезпеки

Коефіцієнт подібності 1:1.3%

Коефіцієнт подібності 2:0%

Мікропробіли: 0

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2025-05-29 03:29:23.0

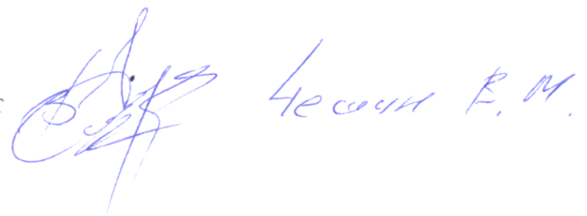
Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 29.05.2025 експерт 

Anti-Plagiarism v-15.274 Educational

The maximum coincidence with one document 1.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 9%**

ID: 242332 Title: Кросплатформна система безпечного зберігання інформації з обмеженим доступом Added in a DB: 2025-05-28 Authors: Медяков Євген Олександрович Heads: Чешун В.М. Consultants: Opponents:	Document		Sum coincidence on the DB	
	Symbols	Lexemes	Symbols	Lexemes
	60218	891	508 (1%)	8 (1%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ КІБЕРБЕЗПЕКИ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Кросплатформенна система безпечного зберігання інформації з обмеженим доступом

Автор: Медяков Євген Олександрович

Спеціальність: 125 – Кібербезпека

Освітня програма: Кібербезпека

Науковий керівник: Віктор ЧЕШУН, канд. техн. наук, доцент

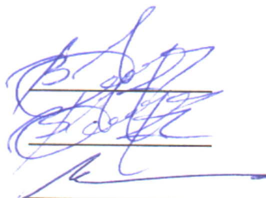
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних). Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи.	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Оригінальність тексту роботи за результатами перевірки системою StrikePlagiarism складає 98,7%, оригінальність тексту роботи за результатами перевірки системою Anti-Plagiarism складає 99%.

Згідно з правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 24.09.2024, авторська робота, обсяг оригінального тексту у відсотках до загального обсягу матеріалу в якій складає 90-100%, визначається роботою з високою унікальністю тексту і допускається до захисту.



Керівник роботи ____________ Віктор ЧЕШУН

Гарант ОП ____________ Віктор ЧЕШУН

Завідувач кафедри кібербезпеки ____________ Юрій КЛЬОЦ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітнього ступеня «бакалавр»

Студен Медяков Євген Олександрович

Тема : Кросплатформна система безпечного зберігання інформації з обмеженим доступом

Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека та захист інформації

Обсяг кваліфікаційної роботи освітнього ступеня «магістр»:

кількість листів креслень 3 ; кількість сторінок записки 78

1. Короткий зміст роботи та прийнятих рішень Кваліфікаційна робота присвячена розробці системи безпечного зберігання інформації з обмеженим доступом. В роботі здійснено аналіз загроз інформації, проблематику автономних рішень, запропоновано і надано опис створення системи безпечного зберігання інформації, розроблено шифрування інформації використовуючи різні методи шифрування, та виклик нативних біометричних методів перевірки, та описано тестування системи на доступ та цілісність до інформації.

2. Висновок про відповідність кваліфікаційної роботи завданню Кваліфікаційна робота відповідає поставленому завданню як в теоретичній, так і в практичній частині.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі викладено задачі та обґрунтовано вибір Flutter/Dart для створення автономної системи безпечного зберігання даних. У першому розділі виконано огляд загальної інформації про зберігання, операційні системи які потребують автономне рішення захисту інформації та методи локального шифрування такі як AES-256, PBKDF2, а також визначено вимоги до кросплатформенності та автономності системи. Другий розділ присвячено реалізації трирівневої архітектури: Data Layer на основі SQLite sqflite/sqflite_ffi, Domain Layer із CryptoService AES-256, унікальні IV, зберігання ключів і Presentation Layer на Material 3 із підтримкою local_auth. У третьому розділі проведено модульні тести CryptoService, NoteProvider і AuthProvider, інтеграційні сценарії CRUD і зміни пароля, а також системні перевірки на ARM системах і x86 системах, підтвердивши відповідність реалізації заявленим вимогам щодо продуктивності та безпеки.

4. Позитивні сторони роботи Розроблена в роботі кросплатформна система безпечного зберігання інформації з обмеженим доступом має орієнтованість на велику кількість користувачів найбільш популярних операційних систем та реалізацію програмним кодом.

5. Негативні сторони роботи В роботі відсутня інформація про обмеження системи по відношенню до операційних систем різної версії та конфігурацій, а також про мінімально необхідні характеристики пристрою для використання системи .

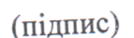
6. Оцінка графічного оформлення та пояснювальної записки роботи Оформлення всіх матеріалів кваліфікаційної роботи є якісним, здійснене з дотриманням актуальних стандартів та інституційних положень ХНУ. Пояснювальна записка відповідає нормам щодо її оформлення як за структурою, так і за представленням і форматуванням матеріалу.

7. Відгук про роботу в цілому Кваліфікаційна робота заслуговує позитивної оцінки. Весь матеріал кваліфікаційної роботи структурований, чіткий та наскрізно пов'язаний. Усі розділи роботи послідовні та логічні, що дозволяє чітко розуміти викладений матеріал в рамках тематики кваліфікаційної роботи. Графічний та ілюстративний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для досягнення поставленої мети.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Враховуючи всі позитивні та негативні сторони представленої кваліфікаційної роботи, можна зробити висновок, що вона заслуговує оцінку «добре».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Підченко Сергій Костянтинович
завідувач кафедри ТМІТ, доктор технічних наук, професор

« 3 » червня 2025.  (підпис)