НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2025 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “Мобільний застосунок шифрування повідомлень”

Виконав: студент 4 курсу, групи ТР-15
Терела Артем Анатолійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент каф. ЦТЕ, д.філос, Артем ГУРІН _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент доцент каф. АЕП, к.т.н., доцент Павло НОВІКОВ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль асистент Антон ПАСІЧНЮК _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
“___” _____2025 р.

ЗАВДАННЯ
на дипломну роботу студенту

Терелі Артему Анатолійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Мобільний застосунок шифрування повідомлень”
Науковий керівник Гурін Артем Леонідович, доктор філософії
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “____” _____ 2025 року № _____

2. Термін подання студентом роботи 09.06.2025

3. Вихідні дані до роботи мова програмування Swift, фреймворк SwiftUI, СКБД Firebase, середовище розробки Xcode, інструмент для тестування iPhone 16 Pro Max

4. Перелік питань, які потрібно розробити
1) аналіз аналогів застосунків для шифрування повідомлень
2) тестування механізмів генерації та зчитування QR-кодів
3) обгрунтування вибору Swift, SwiftUI, CryptoSwift, алгоритмів AES, ChaCha20, DES, Blowfish

5. Орієнтований перелік ілюстративного матеріалу: діаграми бізнес-процесів (BPMN) для створення та зчитування QR-кодів; схема архітектури програмного забезпечення за патерном MVVM; діаграми класів та структури даних застосунку;

приклади користувацького інтерфейсу (екрани введення повідомлення, генерації QR-коду, зчитування та розшифрування); ілюстрації роботи криптографічних алгоритмів;

6. Дата видачі завдання 20.03.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	20.03.2025	
2.	Аналіз методів та засобів розв'язання задачі	17-20.04.25	
3.	Розробка архітектури та загальної структури системи	21-25.04.25	
4.	Розробка окремих підсистем	25.04-02.05.25	
5.	Програмна реалізація системи	03-07.05.25	
6.	Оформлення пояснювальної записки	08-12.05.25	
7.	Захист програмного забезпечення	14.05.2025	
8.	Передзахист	27.05.2025	
9.	Захист	16-20.06.2025	

Студент _____ Артем ТЕРЕЛА
(підпис) (прізвище та ініціали)

Керівник роботи _____ Артем ГУРІН
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 69 сторінках, містить 20 ілюстрацій, 10 таблиць, 1 додаток, 16 джерел в переліку посилань.

Мета роботи – розробити мобільний застосунок для безпечного обміну конфіденційними повідомленнями шляхом їхнього шифрування та передачі у вигляді QR-коду.

Методи та засоби: архітектурний підхід MVVM; моделювання бізнес-процесів за стандартом BPMN; програмування на мові Swift; фреймворк SwiftUI для розробки інтерфейсу користувача та інтеграції з камерою/галереєю; бібліотека CryptoSwift для забезпечення криптографічних функцій; Core Image для створення та обробки QR-кодів; фреймворки XCTest і ViewInspector для автоматизації тестування.

Результат – мобільний додаток, який дає змогу користувачам вводити текст повідомлення, обирати алгоритм шифрування (AES, Blowfish, ChaCha20), генерувати зашифрований QR-код, імпортувати або сканувати існуючі коди та безпечно розшифровувати повідомлення лише при введенні коректного паролю.

Ключові слова: МОБІЛЬНИЙ ЗАСТОСУНОК, ШИФРУВАННЯ ПОВІДОМЛЕНЬ, QR-КОД, SWIFT, SWIFTUI, CRYPTOSWIFT, MVVM, BPMN.

ABSTRACT

The thesis consists of 69 pages, contains 20 illustrations, 10 tables, 1 appendix, 16 sources in the list of references.

Purpose of the work – to create a mobile application for the secure exchange of confidential messages by encrypting them and transmitting them as QR codes.

Methods and tools: MVVM architectural model; BPMN notation for business process modeling; Swift as the programming language; SwiftUI framework for building the user interface and integrating with camera/gallery features; CryptoSwift library for cryptographic functions; Core Image for creating and recognizing QR codes; XCTest and ViewInspector frameworks for automated testing.

Result – a mobile application that enables users to enter a text message, select an encryption algorithm (AES, Blowfish, ChaCha20), generate an encrypted QR code, import or scan existing codes, and securely decrypt messages only upon entering the correct password.

Keywords: mobile application, message encryption, QR code, Swift, SwiftUI, CryptoSwift, MVVM, BPMN.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

IDE – Integrated Development Environment – середовище для інтегрованої розробки.

SDK – Software Development Kit – комплект інструментів для створення програмного забезпечення.

ОС – операційна система.

ПЗ – програмне забезпечення.

MVVM – Model-View-ViewModel – модель-вигляд-модель представлення.

BPMN – Business Process Model and Notation – нотація моделювання бізнес-процесів.

QR – Quick Response – швидкий відгук (використовується в терміні QR-код).

API – Application Programming Interface – інтерфейс програмування додатків.

СКБД – система керування базами даних

AES – Advanced Encryption Standard – стандарт шифрування даних

DES – Data Encryption Standard – стандарт шифрування даних

UI – User Interface – користувацький інтерфейс

IV – Initialization Vector – вектор ініціалізації

PWA – Progressive Web Application – прогресивний веб-застосунок

CBC – Cipher Block Chaining – режим зв’язування блоків шифру

ВСТУП

У сучасному інформаційному суспільстві обмін даними набуває все більшої ваги, адже дані стали однією з найцінніших ресурсів. З огляду на стрімкий розвиток цифрових технологій, мобільні застосунки стають невід'ємною частиною повсякденного життя. Однією з ключових потреб сучасного користувача є захист особистої інформації від несанкціонованого доступу. Саме тому розробка мобільного додатку для шифрування повідомлень є актуальним завданням, що дозволяє забезпечити високий рівень безпеки даних при їх передачі.

Особливістю даної роботи є реалізація системи обміну повідомленнями у вигляді зашифрованих QR-кодів. Такий підхід об’єднує традиційні методи шифрування з сучасними технологіями візуального кодування інформації. QR-коди виступають ефективним засобом передачі даних, адже їх можна швидко сканувати за допомогою стандартних мобільних пристроїв, що сприяє зручності та оперативності обміну зашифрованими повідомленнями.

В умовах сучасних викликів, зокрема загроз кібербезпеки та ризиків витоку конфіденційної інформації, тема даної дипломної роботи набуває особливої актуальності. Сучасний світ характеризується активним впровадженням цифрових технологій у всі сфери життя, а питання захисту інформації стає предметом постійної уваги урядів, бізнесу та користувачів.

Новизна пропонованого рішення полягає у поєднанні сучасних криптографічних алгоритмів, таких як AES, DES, Blowfish і ChaCha20, із технологією QR-кодів для створення інтуїтивно зрозумілого та безпечного інструменту обміну даними. Застосунок дозволяє користувачам не лише шифрувати повідомлення, а й зберігати історію операцій, що забезпечує додаткову зручність і контроль над конфіденційною інформацією. Такий підхід відповідає вимогам сучасного інформаційного середовища та сприяє підвищенню довіри до цифрових комунікацій.

Особливо важливою є ця тема в контексті ситуації в Україні, де активний розвиток цифрових сервісів поєднується з підвищеним рівнем кіберзагроз. З огляду на геополітичну ситуацію та постійні виклики, що стоять перед країною,

забезпечення безпеки комунікацій є критично важливим завданням. Розробка надійного мобільного застосунку для шифрування повідомлень сприятиме не лише захисту приватних даних користувачів, а й зміцненню інформаційного суверенітету держави. Використання QR-кодів як засобу передачі зашифрованої інформації дозволяє створити додатковий рівень захисту завдяки фізичним обмеженням їх копіювання та автоматизованої верифікації даних.

Таким чином, це дослідження зосереджене на розв’язанні нагальної проблеми забезпечення безпеки даних у контексті сучасних викликів, представляючи інноваційний підхід до обміну зашифрованими повідомленнями через мобільний застосунок з використанням QR-кодів. Отримані результати можуть мати вагоме значення як для окремих користувачів, так і для державних структур, сприяючи підвищенню рівня кібербезпеки в Україні та за її межами.

1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У даній частині буде ретельно проаналізовано головну ціль і завдання мобільного додатку. Планується провести комплексний аналіз функціональних вимог, що дозволить визначити ключові можливості застосунку та його роль у сучасному ринку послуг. Окрім цього, у розділі буде проведено ґрунтовний аналіз наявних підходів, алгоритмів і програмних інструментів, потрібних для ефективної реалізації та запуску цього проєкту. Розгорнутий аналіз допоможе виявити переваги та недоліки різних технологічних рішень, спрямованих на підвищення ефективності роботи мобільного застосунку, а також дозволить сформувати чітку картину вимог щодо його функціональності та зручності використання.

1.1 Постановка задачі

Завданням цього дослідження є розв’язання нагальної проблеми захисту даних у контексті сучасного цифрового середовища через створення мобільного додатку, що дозволяє здійснювати шифрування повідомлень з подальшою їх передачею у вигляді зашифрованих QR-кодів. Пропонована система спрямована на забезпечення високого рівня захисту конфіденційних даних, де користувач вводить текст повідомлення, визначає параметри обраного алгоритму шифрування та задає пароль, необхідний для подальшого розшифрування інформації за допомогою інтегрованого механізму перевірки. Вхідними даними програмної системи є всі дані, отримані від користувача під час введення повідомлення, вибору методу шифрування та встановлення паролю, а також додаткові параметри, що можуть впливати на процес кодування, такі як налаштування валідації даних і обмеження символів.

На виході мобільного застосунку формується зашифроване повідомлення, представлене у вигляді QR-коду, який містить зашифровану інформацію разом із супутніми метаданими, що дозволяють проводити подальше розшифрування.

Також система забезпечує можливість відновлення первинного тексту повідомлення при введенні коректного паролю, що гарантує точність та безпечність розшифрування. Застосунок повинен забезпечити стабільне збереження всіх даних у базі, що дозволяє їх подальше використання для генерації QR-кодів, а також реалізувати надійний механізм вводу, перевірки та шифрування повідомлень.

Розроблена система повинна генерувати QR-коди, які можуть бути як імпортовані з галереї, так і відскановані за допомогою вбудованої камери, що надає користувачам можливість легко обмінюватися зашифрованою інформацією. Процес розшифрування реалізовано за допомогою точної відповідності обраного алгоритму та введеного паролю, завдяки чому забезпечується безпомилкове відновлення початкових даних. Крім того, застосунок надає можливість зберігати історію зашифрованих повідомлень, що дозволяє користувачам відслідковувати свої операції та гарантувати подальшу безпеку інформації.

Потенційними користувачами даного застосунку є як приватні особи, для яких захист особистих даних є пріоритетом, так і представники державних установ та бізнес-структур, що мають високі вимоги до забезпечення конфіденційності переданої інформації в умовах постійних кіберзагроз. Така система може стати незамінним інструментом для організацій, які прагнуть забезпечити безпечний обмін даними, а також для окремих користувачів, що цінують свої дані й потребують надійного захисту у сучасному інформаційному середовищі.

Розроблений мобільний застосунок передбачає експлуатацію на пристроях під керуванням операційної системи iOS версії 14.0 або новішої, оскільки лише в цьому середовищі забезпечується коректна інтеграція з актуальними фреймворками Apple, оптимальний рівень безпеки та стабільна робота графічного інтерфейсу. Основним засобом зчитування зашифрованої інформації виступає вбудована камера пристрою, апаратні характеристики якої (роздільна здатність матриці, чутливість сенсора) мають бути достатніми для точного розпізнавання маркерів за різних умов освітлення. Крім того, для можливості імпорту QR-кодів зі збережених знімків необхідно надати застосунку доступ до фотогалереї, що дозволяє проводити попередню обробку та декодування зображень без додаткових

обмежень. Також, безперервне підключення до мережі Інтернет є обов’язковим для завантаження оновлень програмного забезпечення, синхронізації довідкових даних та оперативного доступу до нових версій алгоритмів розпізнавання, тому рекомендується використовувати стабільне широкосмугове або мобільне з’єднання.

1.2 Огляд методів необхідних для розв’язання

Для розв’язання задачі безпечного шифрування повідомлень з використанням QR-кодів застосунок використовує комплекс методів, що охоплюють як аспект криптографічного захисту, так і забезпечення зручної взаємодії користувача із системою. Одним із ключових напрямків є реалізація алгоритмів симетричного шифрування, зокрема алгоритмів AES, DES, Blowfish та ChaCha20, які дозволяють ефективно перетворювати відкритий текст повідомлення у зашифрований, що неможливо зчитати без відповідного паролю. Алгоритми обраних методів шифрування враховують обмеження на максимальну кількість символів, що дозволяє оптимізувати обсяги даних, які кодуються у QR-коді, та забезпечує цілісність інформації.

Іншим важливим компонентом є процес генерації QR-кодів, який втілюється шляхом передачі зашифрованої інформації у візуально представлене зображення. Цей підхід дозволяє не лише зберігати дані у компактному вигляді, але й забезпечувати швидкий обмін інформацією між пристроями за допомогою сканування коду. Функціональність створення QR-коду інтегровано у застосунок через використання відповідних бібліотек і фреймворків, що дозволяють автоматично перетворювати текстовий контент у графічне представлення з високим рівнем надійності відображення.

Крім того, застосунок використовує методи валідації введених даних, які здійснюються як на рівні користувацького інтерфейсу, так і під час обробки інформації у бекенді. Це дозволяє попередити введення некоректних значень, забезпечити відповідність параметрів шифрування визначеним алгоритмам та

гарантувати, що лише дійсні дані будуть використовуватися під час шифрування і генерації QR-кодів. Система перевірки включає аналіз довжини повідомлення, оцінку надійності паролю та відповідність його вимогам щодо безпеки, що сприяє зменшенню ризику несанкціонованого доступу.

Сукупність застосованих методів дозволяє створити інтегроване рішення, де криптографічна частина забезпечує високий рівень захисту даних, а механізми валідації та генерації QR-кодів роблять процес взаємодії користувача з системою максимально зручним та безпечним[1]. Отже, використання сучасних методів шифрування, генерації та обробки даних є важливим чинником у розв’язанні поставленої задачі, що дозволяє забезпечити надійний та інтуїтивно зрозумілий інтерфейс для обміну зашифрованими повідомленнями навіть в умовах підвищених кіберзагроз.

1.3 Огляд програмних засобів необхідних для розв’язання

Визначення оптимального набору інструментів і технологій для створення мобільного додатку є вирішальним кроком на шляху до отримання високоякісного та надійного продукту. У даному дослідженні детально розглядаються чотири основні компоненти, які формують основу розробки iOS-застосунку: Xcode, мова Swift, фреймворк SwiftUI та сервісна платформа Firebase. Кожен із цих елементів обирається з урахуванням показників продуктивності, зручності розробки й відповідності сучасним стандартам індустрії.

Xcode є офіційним інтегрованим середовищем розробки (IDE) від Apple для створення застосунків під iOS, macOS, watchOS і tvOS. Основні переваги Xcode полягають у тому, що в одному середовищі розробник отримує редактор коду, інструменти для юніт-тестування, дебагінгу та профілювання продуктивності; він регулярно оновлюється, щоб підтримувати нові версії операційних систем і SDK; також у його складі є високопродуктивні емулятори, які дозволяють тестувати програму на різноманітних віртуальних пристроях без необхідності фізично мати кожну модель.

Swift - це сучасна, високопродуктивна мова програмування, створена Apple спеціально для своєї платформи. Вона дозволяє писати швидкий і оптимізований код, вбудовані механізми обробки помилок і система опціональних типів значно підвищують безпеку програми, а чітка й лаконічна синтаксична структура скорочує час на розробку та спрощує підтримку коду.

SwiftUI представляє собою декларативний фреймворк для побудови інтерфейсів користувача. Завдяки описовому підходу розробник визначає лише кінцевий вигляд інтерфейсу, а не кожен крок його побудови - це скорочує обсяг коду і підвищує його читабельність. Крім того, одна й та сама реалізація UI працює на всіх платформах Apple - від iPhone до Apple TV - без суттєвих змін, а реактивна модель оновлення дозволяє автоматично перерендерювати екран при зміні даних.

2 МЕТОДИ ТА АЛГОРИТМИ ВИРІШЕННЯ ЗАДАЧІ

Другий розділ роботи присвячено методам та алгоритмам, необхідним для вирішення поставленої задачі з шифрування повідомлень за допомогою QR-кодів. У цьому розділі буде розглянуто сукупність технічних підходів, що забезпечують стабільну та безпечну роботу застосунку, починаючи від вибору алгоритмів шифрування до генерації та розшифрування QR-кодів. Методичний аналіз включає вивчення ефективності алгоритмів симетричного шифрування, таких як AES, DES, Blowfish і ChaCha20, з урахуванням обмежень на довжину повідомлення та вимог до безпеки[2]. Значна увага приділена впровадженню цих підходів у мобільний додаток, що сприяє досягненню зрозумілості і зручності використання інструментів шифрування для кінцевого користувача. Крім того, розділ містить аналіз додаткових методів валідації даних та генерації QR-кодів, що сприяє підвищенню стійкості системи до кібератак і несанкціонованого доступу. Комплексний підхід до вибору оптимальних алгоритмів дозволяє врахувати як технічні, так і практичні аспекти розробки, що в сукупності сприяє створенню надійного та ефективного рішення, здатного відповідати високим вимогам сучасного інформаційного середовища.

2.1 Опис проблем розробки мобільного застосунку

Однією з ключових складнощів є належне планування та чітке формулювання вимог. Якщо на цьому етапі виникають прогалини або невірні уявлення щодо потреб проекту, це здатне спричинити значні затримки й перевитрати ресурсів. Щоб мінімізувати подібні ризики, необхідно з самого початку мати точне розуміння як кінцевих цілей додатку, так і функціональних можливостей, яких від нього очікують користувачі. Ефективне планування передбачає не лише розробку детальних технічних завдань, але й створення прототипів для візуалізації ключових сценаріїв взаємодії, а також узгодження цих

наочних матеріалів із усіма зацікавленими сторонами - аналітиками, дизайнерами, розробниками й кінцевими користувачами. Щоби вирішити проблему незрозумілості вимог, було прийнято рішення скласти комплексну специфікацію функціональних характеристик, яку систематизовано й представлено в таблиці 2.1.

Таблиця 2.1 – Функціональні вимоги

ID	Title	Description
FR-1	Введення повідомлень	Користувач має можливість ввести текст повідомлення та задати пароль для шифрування в застосунку.
FR-2	Генерація QR-коду	Застосунок має перевести зашифроване повідомлення у QR-код використовуючи вибраний алгоритм шифрування та забезпечити його відображення
FR-3	Зчитування QR-коду	Користувач може імпортувати або сканувати QR-код для подальшого розшифрування повідомлення.
FR-4	Розшифрування повідомлень	Застосунок повинен дозволяти розшифрування зашифрованого повідомлення за допомогою введеного паролю та обраного алгоритму.
FR-5	Валідація введених даних	Система перевіряє правильність введення тексту повідомлення та паролю включаючи обмеження на кількість символів та формат даних і відображає відповідні повідомлення про помилки.
FR-6	Збереження QR-кодів	Застосунок має забезпечувати можливість збереження згенерованих QR-кодів у локальному сховищі для подальшого перегляду.
FR-7	Перегляд історії QR-кодів	Система зберігає історію збереження QR-кодів та забезпечує доступ користувача до перегляду збережених кодів.

Кінець таблиці 2.1

FR-8	Безпека шифрування	Застосунок використовує сучасні алгоритми шифрування (AES ChaCha20) для забезпечення безпеки даних та захисту від несанкціонованого доступу.
FR-9	Спілкування з ШІ	Надання користувачу можливості спілкуватися із вбудованим ШІ-модулем (OpenAI API) для отримання підказок/аналізу/допомоги у формулюванні зашифрованого повідомлення.

2.2 Оптимізація застосунку

Поліпшення роботи додатків на iOS включає застосування широкого спектра стратегій і прийомів, метою яких є підвищення їх швидкодії, продуктивності та загального користувацького досвіду. Наприклад, скорочення часу запуску додатку досягається за рахунок впровадження асинхронної ініціалізації, ретельної оптимізації вихідного коду та попереднього завантаження найважливіших ресурсів, що дозволяє зменшити затримки під час старту програми. Також особлива увага приділяється мінімізації навантаження на центральний процесор, що забезпечує плавну роботу застосунку навіть при високих вимогах до продуктивності.

Разом із базовими методами оптимізації, особлива роль відводиться використанню спеціалізованих інструментів для тестування продуктивності. Ці інструменти допомагають визначити проблемні місця та слабкі ділянки коду, які потребують покращення, сприяючи підвищенню ефективності роботи застосунку. Одним із ключових моментів є скорочення числа мережевих запитів, адже зменшення кількості звернень до серверів допомагає мінімізувати затримки та прискорити функціонування додатку.

Крім цього, оптимізація бази даних має суттєве значення, адже ефективне виконання операцій із зберігання та обробки інформації сприяє скороченню часу

доступу до даних і, як наслідок, підвищенню загальної швидкодії додатку. З одного боку, покращення продуктивності забезпечується асинхронним завантаженням даних у фоновому режимі, що не блокує головний потік інтерфейсу, а з іншого – застосуванням технологій кешування, що зменшує необхідність повторних звернень до серверів.

Оптимізація користувацького інтерфейсу є невід'ємною частиною вдосконалення будь-якого мобільного застосунку, оскільки від якості його роботи залежить успішність продукту на ринку. Якісно спроектований інтерфейс дозволяє уникнути зайвих операцій рендерингу, адже зміни окремих елементів можуть спровокувати повне переналаштування всієї візуальної компоненти. Для цього важливо розділяти складний інтерфейс на незалежні частини, кожна з яких відповідає за свій набір елементів. Сучасні технології, такі як Auto Layout і Stack Views, сприяють автоматичному налаштуванню розташування елементів, що допомагає мінімізувати складність управління інтерфейсом і знизити навантаження на систему[3].

Не менш важливим є зменшення об'єму зображень, що завантажуються у пам'ять пристрою, адже це впливає як на швидкодію, так і на використання дискового простору. При цьому необхідно зберігати баланс між оптимізацією розміру зображень і підтримкою їх якості, щоб візуальний досвід користувача залишався на високому рівні. Завдяки цим заходам застосунок працює більш ефективно завдяки плавним анімаціям та переходам, реалізованим за допомогою можливостей UIKit і Core Animation, що дозволяє створювати візуально привабливий та водночас продуктивний інтерфейс.

Таким чином, комплексне впровадження оптимізаційних рішень і технологій забезпечує високий рівень продуктивності, мінімізує затримки при завантаженні та обробці даних, а також дозволяє розробити інтерфейс, що відповідає сучасним вимогам до мобільних застосунків.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У цьому розділі висвітлюється структура програмного забезпечення, описуються інструменти для реалізації клієнтської й серверної частин, а також показано, як за допомогою BPMN моделюються бізнес-процеси. Застосування архітектури MVVM дозволяє чітко розділити обов'язки між компонентами додатку, що значно полегшує організацію та підтримку коду.

3.1 Моделювання та аналіз програмного забезпечення

BPMN (Business Process Model and Notation) являє собою стандартизовану нотацію, яка застосовується для моделювання, документування та аналізу бізнес-процесів. Основна мета BPMN полягає в тому, щоб забезпечити єдину мову для спілкування між бізнес-аналітиками, розробниками та керівниками. Це дозволяє чітко документувати процеси, що в подальшому спрощує їх аналіз, оптимізацію та впровадження систем автоматизації.

Нотація включає кілька категорій елементів, серед яких ключовими є об'єкти потоку, об'єкти з'єднання та організаційні простори. Об'єкти потоку містять події, дії і рішення. Події можуть бути початковими, проміжними або кінцевими, визначаючи відповідно старт, перебіг та завершення процесу. Дії, представлені завданнями або підпроцесами, показують виконання операцій у процесі, тоді як рішення, що реалізуються через шлюзи, дозволяють здійснювати розгалуження або злиття потоків, залежно від умов.

Об'єкти з'єднання, такі як послідовні потоки, повідомлення чи асоціації, встановлюють логічні взаємозв'язки між елементами BPMN-моделі. Організаційні простори, які поділяються на пули та доріжки, дозволяють візуально розмежувати відповідальність між різними учасниками процесу. Окрім цього, BPMN дозволяє додавати додаткові елементи, як-от дані, текстові анотації чи групи, які

допомагають уточнити деталі процесу, не впливаючи на основну логіку його виконання.

При побудові діаграми BPMN спочатку необхідно проаналізувати існуючі бізнес-процеси, визначити зацікавлені сторони та ключові етапи процесу. Далі слід створити діаграму з використанням стандартних символів BPMN, щоб забезпечити однозначність візуального представлення, і, за потребою, деталізувати окремі складні етапи через розбиття на підпроцеси. Отриману модель необхідно переглянути за участю експертів та зацікавлених сторін для виявлення слабких місць та подальшої оптимізації.

Важливим аспектом використання BPMN є її здатність адаптуватися до різних масштабів і складності процесів. Від простих операційних завдань до складних міжорганізаційних взаємодій, BPMN забезпечує гнучкість у моделюванні, що дозволяє створювати як високорівневі схеми для стратегічного планування, так і деталізовані моделі для автоматизації. Це робить нотацію універсальним інструментом, який підтримує цифрову трансформацію організацій, сприяючи інтеграції сучасних технологій, таких як штучний інтелект чи автоматизація робочих процесів. Завдяки своїй стандартизації, BPMN полегшує співпрацю між різними командами, забезпечуючи чітке розуміння процесів на всіх рівнях організації. Крім того, вона дозволяє швидко реагувати на зміни в бізнес-середовищі, адаптуючи моделі до нових вимог чи технологічних інновацій.

BPMN застосовується в різних сферах, від управління бізнес-процесами до розробки програмного забезпечення, де вона сприяє інтеграції бізнес-логіки із інформаційними технологіями. Використання BPMN допомагає не лише стандартизувати процеси, а й проводити їх аналіз для виявлення неефективностей, що особливо важливо для організацій, які прагнуть постійного вдосконалення. При цьому існує ряд інструментів, як-от Bizagi Modeler, Microsoft Visio, ARIS Express та Camunda, що полегшують створення та впровадження BPMN-діаграм.BPMN модель (рисунки 3.1 – 3.2).

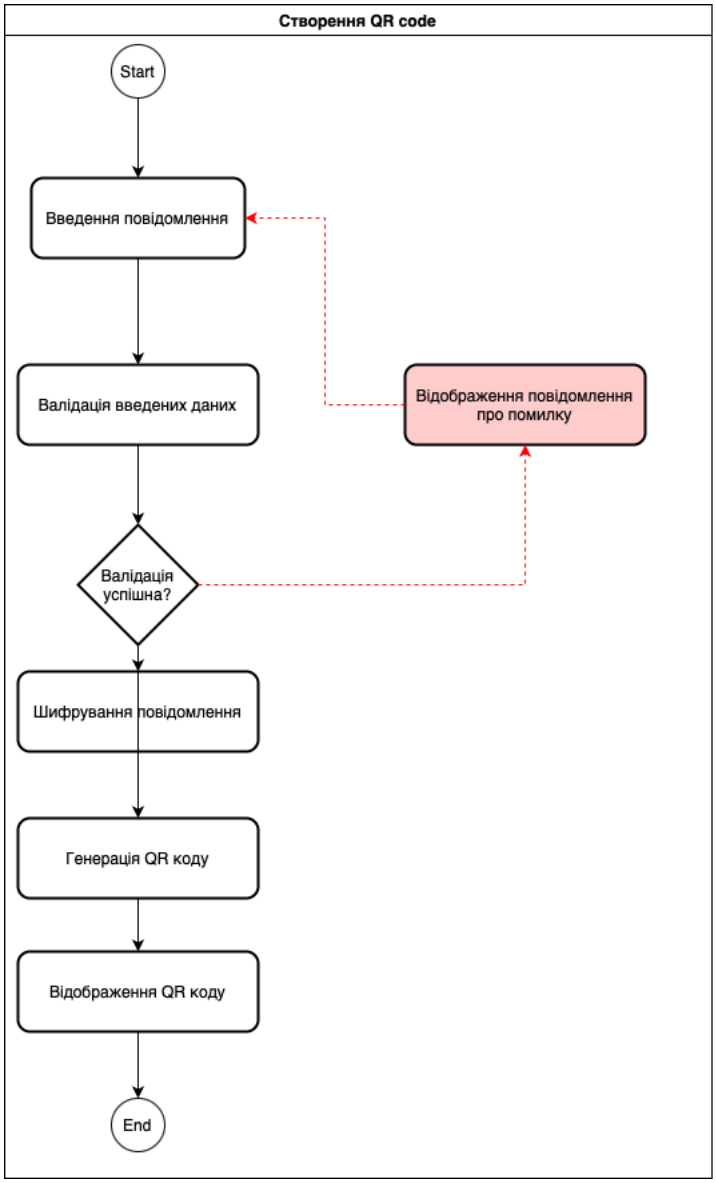

Рисунок 3.1 – Схема бізнес-процесу створення QR-коду

Бізнес-процес створення QR-коду починається із запуску системи, що сигналізує про готовність застосунку приймати інформацію від користувача. На першому етапі користувач вводить текст повідомлення, яке має бути зашифроване. Цей крок є ключовим, оскільки саме введені дані стають основою для подальшого шифрування. Після введення повідомлення відбувається процес валідації введених даних, де система перевіряє, чи відповідає текст вимогам (наприклад, перевіряється правильність формату, обмеження кількості символів тощо).

На наступному етапі, коли дані успішно пройшли валідацію, процес розгалужується. В залежності від результату перевірки, використовується

розгалуження (gateway). Якщо дані валідні, система переходить до етапів шифрування повідомлення та генерації QR-коду. Спочатку відбувається шифрування, при якому відкритий текст перетворюється на зашифровану інформацію за допомогою обраного алгоритму. Далі зашифровані дані автоматично перетворюються у вигляд QR-коду, що представляє собою зображення з вбудованою інформацією.

Якщо ж дані не відповідають вимогам, система переходить за альтернативною логікою до етапу обробки помилок, де користувачу відображається повідомлення про помилку з відповідними інструкціями для повторного введення даних. Після усунення помилки користувач повертається до початкового кроку вводу повідомлення та повторює процес валідації.

Для забезпечення високого рівня безпеки та зручності користувача бізнес-процес включає додаткові механізми контролю. Наприклад, перед шифруванням система може запропонувати користувачу обрати конкретний алгоритм шифрування (наприклад, AES, Blowfish або ChaCha20), що дозволяє адаптувати процес до специфічних потреб безпеки. Крім того, застосовуються методи валідації паролю, який використовується для шифрування, щоб гарантувати його достатню складність і захист від несанкціонованого доступу. Система також перевіряє унікальність паролю для кожного повідомлення, уникаючи повторного використання, що знижує ризик компрометації даних. Ці заходи підвищують надійність системи та забезпечують захист конфіденційної інформації, роблячи процес шифрування більш стійким до зовнішніх загроз.

Завершальним кроком бізнес-процесу є відображення сформованого QR-коду. Коли QR-код згенеровано, він виводиться на екран, що дозволяє користувачу перевірити коректність зашифрованої інформації. Після цього процес завершується. Такий послідовний підхід забезпечує надійність і безпеку генерації QR-кодів, що є ключовим для подальшого розшифрування повідомлень за допомогою автентичного паролю.

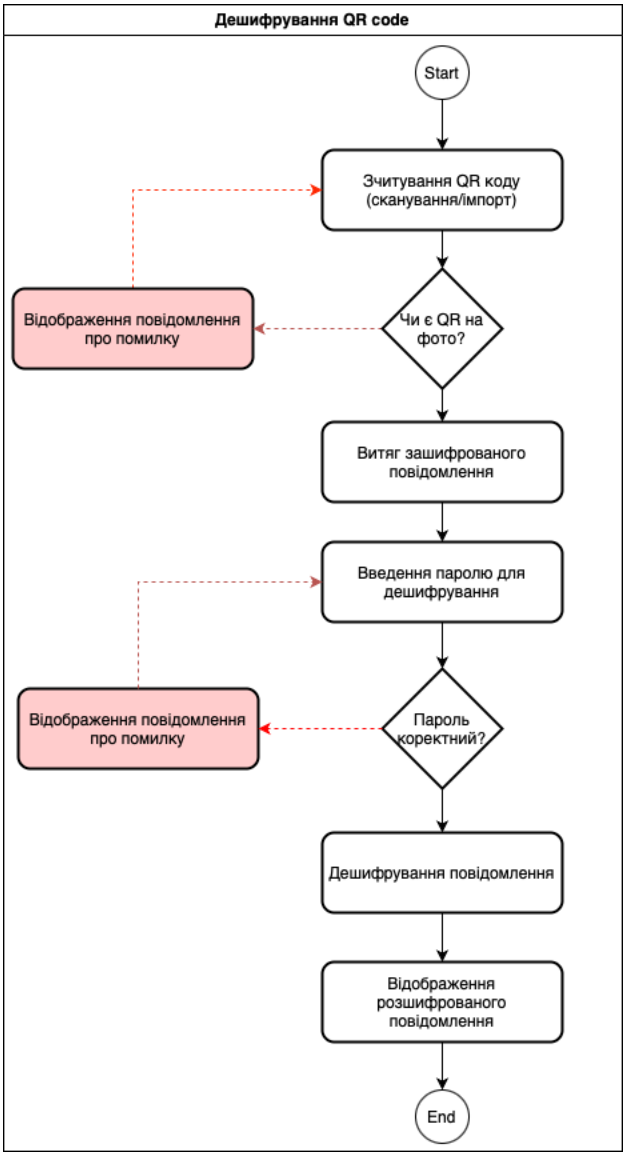

Рисунок 3.2 – Схема бізнес-процесу зчитування QR-коду,

Даний бізнес-процес розпочинається із запуску системи зчитування QR-коду, який може бути отриманий шляхом сканування за допомогою камери або імпорту із галереї. На цьому етапі користувач взаємодіє з інтерфейсом, що дозволяє вибрати потрібний спосіб отримання QR-коду. Після отримання зображення система автоматично витягує з нього зашифроване повідомлення, яке далі стає основою для подальшої обробки.

Наступним кроком є введення користувачем паролю для дешифрування. Це надзвичайно важлива частина процесу, оскільки саме правильний пароль дозволяє системі розпізнати, який саме алгоритм шифрування був застосований, та безпечно відновити початковий текст повідомлення. Після отримання паролю система

переходить до перевірки його коректності за допомогою шлюзу. Якщо пароль відповідає вимогам – процес триває, і дані розшифровуються, а отримане повідомлення відображається користувачу.

У випадку невірного введення паролю, система генерує повідомлення про помилку, що дозволяє користувачу зрозуміти необхідність повторного введення коректного значення. Таким чином, бізнес-процес гарантує, що лише уповноважені користувачі можуть отримати доступ до розшифрованого повідомлення, мінімізуючи ризик несанкціонованого доступу. Останнім етапом є візуальне відображення розшифрованого тексту повідомлення, що завершує процес дешифрування і дозволяє користувачу ознайомитись із оригінальним вмістом.

Цей послідовний підхід забезпечує надійність і безпеку процесу розшифрування, адже кожен крок ретельно перевіряється. Використання механізму перевірки коректності введеного паролю гарантує, що системі доступна лише авторизована інформація, а обробка помилок дозволяє швидко виявляти і усувати неточності вводу. Завдяки такій структурі бізнес-процесу забезпечується високий рівень захисту даних та максимальна зручність для кінцевого користувача.

3.2 Архітектура програмного забезпечення

Архітектурний патерн MVVM (Model-View-ViewModel) є однією з найпопулярніших архітектур для побудови сучасних застосунків, зокрема мобільних, і використовується для чіткого розділення логіки роботи з даними, представлення інтерфейсу користувача та взаємодії між ними.

Model - цей компонент відповідає за дані й бізнес-логіку застосунку. В ньому зберігаються як структура даних, так і операції з ними - наприклад, робота із сховищами, виклики API або операції з обчисленнями. Model не має прямої залежності від способу їх відображення чи взаємодії з користувачем, що дозволяє ізолювати бізнес-логіку та забезпечує її повторне використання в різних проектах.

View відповідає за візуальне представлення інформації. Це інтерфейс користувача, який містить компоненти, що відображають дані, отримані з

ViewModel, і приймають взаємодію користувача (натискання кнопок, введення тексту тощо). Завдяки декларативному стилю розробки у SwiftUI, View генерується на основі поточного стану застосунку, що дозволяє автоматично оновлювати інтерфейс у відповідь на зміни у даних.

ViewModel виступає посередником між Model та View. Воно отримує дані з Model, трансформує їх так, щоб вони були легкими для відображення у View, і забезпечує логіку взаємодії. За допомогою механізму зв’язування (data-binding) ViewModel постійно повідомляє View про зміни у стані застосунку, що дозволяє автоматично оновлювати інтерфейс. ViewModel також обробляє події, отримані від View, виконуючи валідацію даних, виклики бізнес-логіки та змінюючи стан застосунку.

Розглянемо основні переваги MVVM.

Чітке розділення відповідальності: Завдяки розподілу на Model, View та ViewModel, кожен компонент зосереджується на своїй спеціалізованій задачі. Це дозволяє легко підтримувати та розширювати застосунок.

Підвищена тестованість: Оскільки логіка обробки даних ізолюється у ViewModel і Model, їх можна тестувати незалежно від користувацького інтерфейсу, що сприяє стабільності системи.

Легкість підтримки: Оновлення або додавання нових функцій відбувається швидше, оскільки зміни у бізнес-логіці не впливають безпосередньо на інтерфейс, і навпаки.

Декларативний підхід: При використанні SwiftUI, який підтримує декларативне програмування, View створюється на основі даних, що отримуються з ViewModel, що автоматично забезпечує синхронізацію між станом застосунку та інтерфейсом користувача.

Перейдемо до недоліків MVVM.

Складність початкової реалізації: Для розробників, які тільки знайомляться з патерном, може бути непросто організувати правильну взаємодію між компонентами, особливо враховуючи нюанси двостороннього зв’язування.

Надлишкова абстракція: У випадках невеликих або простих застосунків використання MVVM може збільшувати обсяг коду, оскільки вимагається створення додаткових класів для ViewModel, що інколи може здаватися надмірним.

В застосунку на рівні Model зберігаються дані, які характеризують параметри шифрування та властивості повідомлення – наприклад, інформація про вибраний алгоритм шифрування (AES, Blowfish, ChaCha20) та обмеження на максимальну кількість символів. Моделі даних є простими структурами, що забезпечують однозначне визначення констант і параметрів, необхідних для шифрування, і використовуються як фундамент для подальшої обробки інформації.

Шар ViewModel містить бізнес-логіку, пов’язану з обробкою даних, валідацією введення, шифруванням і дешифруванням повідомлень, а також генерацією QR-кодів. Кожен функціональний блок застосунку представлений парою View та ViewModel, що дозволяє ізолювати логіку обробки даних від представлення інтерфейсу користувача. Наприклад, EnterMessageViewModel відповідає за валідацію введення тексту повідомлення та паролю, QRCodeViewModel – за шифрування повідомлення та генерацію зображення QR-коду, а DecryptQRCodeViewModel разом із DecryptedQRCodeViewModel займаються зчитуванням QR-коду та розшифруванням даних. Така модульна структура не тільки спрощує подальше тестування і рефакторинг, але і робить систему гнучкою у разі додавання нових функціональних можливостей чи зміни алгоритмів шифрування.

Шар View реалізовано за допомогою SwiftUI, що дозволяє створювати декларативні інтерфейси. Він відповідає за відображення усіх візуальних компонентів застосунку, таких як форми для введення даних, екрани генерації і відображення QR-кодів, а також інтерфейси для сканування або імпорту зображень. Використання SwiftUI дає змогу динамічно оновлювати інтерфейс у відповідь на зміни стану, які контролюються відповідними ViewModel.

Графічне зображення архітектури MVVM представлено на рисунку 3.3.

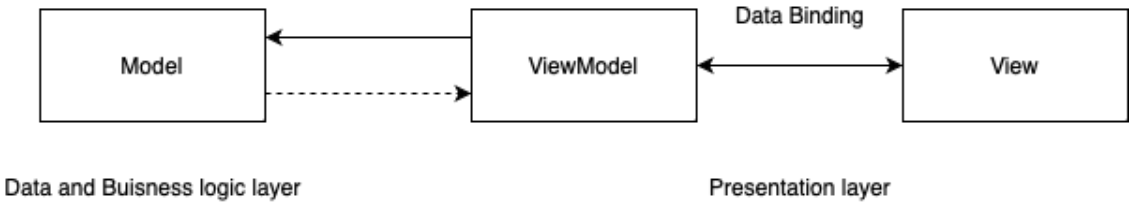

Рис. 3.3 - Схематичне відображення архітектури MVVM

Загалом, застосування MVVM дозволяє чітко розділити процеси збору, обробки і представлення даних, що мінімізує взаємозалежність між компонентами системи і забезпечує високий рівень підтримки безпеки. Розробка програмного забезпечення здійснювалася з використанням Xcode, що є офіційною IDE для iOS, і дозволяє інтегрувати всі необхідні інструменти для налагодження, тестування та профілювання застосунку. Така архітектурна модель забезпечує гнучкість, масштабованість і легкість модифікації системи з урахуванням сучасних вимог до захисту даних та взаємодії з користувачем.

3.3 Побудова програмного забезпечення

Основні компоненти програмного забезпечення включають наступні елементи:

– моделі даних, які відображають логіку обробки інформації, зокрема параметри шифрування та властивості повідомлення;

– моделі представлення, що відповідають за бізнес-логіку додатку: валідацію введених даних, шифрування/розшифрування повідомлень, генерацію QR-кодів і контроль стану інтерфейсу;

– види (Views), які реалізовують інтерфейс користувача, забезпечуючи взаємодію, візуальне відображення даних і надання зворотного зв'язку, а також включають механізми анімації для покращення користувацького досвіду.

Локальне шифрування інформації, що здійснюється користувацьким пристроєм, дозволяє уникнути передачі відкритих даних по мережі, а використання таких бібліотек як CryptoSwift гарантує застосування сучасних криптографічних алгоритмів (AES, Blowfish, ChaCha20)[7]. Кожен алгоритм інтегровано із системою таким чином, щоб забезпечити гнучкість вибору методу шифрування користувачем і можливість розширення набору алгоритмів у майбутньому.

Далі, у таблиці 3.1 подано стислий опис кожного модуля, його функцій та короткий огляд методів

Таблиця 3.1 – Характеристика методів застосунку

Компонент	Функціональність	Технічні засоби
Моделі даних	Зберігання характеристик об'єктів (тип алгоритму, максимум символів, тощо)	Структури Swift
ViewModel для введення	Обробка та валідація введення повідомлення та паролю; контроль переходу до етапу генерації QR-коду	EnterMessageViewModel
ViewModel генерації QR	Шифрування повідомлення, комбінування з IV, генерація та масштабування QR-коду з використанням CIFilter	QRCodeViewModel, CryptoSwift

Кінець таблиці 3.1

ViewModel розшифрування	Розшифрування зашифрованого повідомлення, витяг з QR-коду, перевірка коректності паролю та алгоритму	DecryptedQRCodeViewModel, ChaCha20, AES, Blowfish
Види (Views)	Відображення користувацького інтерфейсу: введення даних, генерація QR, вивід результатів, обробка помилок	SwiftUI
Валідація QR-коду	Перевірка зчитаного зображення на наявність QR-коду та відповідність його даних заданим стандартам	QRCodeInputViewModel, CIDetector

Перший модуль, що складається з EnterMessageView та EnterMessageViewModel, здійснює взаємодію з користувачем для введення тексту повідомлення та паролю. Сам інтерфейс дозволяє користувачу ввести дані, а ViewModel відповідає за їх валідацію, перевіряючи, чи не є повідомлення порожнім, чи не перевищує воно встановлений ліміт символів, і чи відповідає пароль мінімальним вимогам. У разі успішної валідації користувач може перейти до наступного кроку – генерації QR-коду.

Другий модуль включає QRCodeDisplayView разом із QRCodeViewModel, які відповідають за шифрування введеного повідомлення та його перетворення у QR-код. В цьому модулі відбувається застосування обраного алгоритму шифрування, де повідомлення перетворюється у зашифрований текст, потім до нього додається ініціалізаційний вектор, після чого результат кодується у Base64-рядок. На основі

цього рядка за допомогою CIFilter генерується зображення QR-коду, яке потім відображається користувачу.

У третьому модулі (ImportOrScanQRCodeView і QRCodeInputViewModel) користувач може або відсканувати QR-код за допомогою камери, або імпортувати зображення з галереї. ViewModel цього модуля автоматично перевіряє, чи містить отримане зображення дійсний QR-код, використовуючи CIDetector, і в разі невдалого розпізнавання повідомляє користувача про помилку, що дозволяє повторно здійснити зчитування[8].

Четвертий модуль, що об'єднує DecryptQRCodeView з DecryptQRCodeViewModel, забезпечує етап підготовки до розшифрування. Тут користувач обирає алгоритм дешифрування із запропонованого списку, вводить пароль, який буде використаний для відновлення первинного повідомлення, і перевіряється коректність введених даних. ViewModel контролює, чи достатньо даних для продовження, активуючи можливість переходу до дешифрування.

П’ятий модуль надає користувачу змогу спілкуватись із вбудованим ШІ-асистентом щодо вибору й особливостей алгоритмів шифрування. Інтерактивний чат-інтерфейс дозволяє формулювати питання безпосередньо в застосунку, після чого через OpenAIService який реалізовано на базі URLSession, надсилає HTTP-запит до OpenAI API із зазначенням моделі, параметрів (max_tokens, temperature тощо) та авторизаційного ключа, захованого в Keychain. Отримана відповідь у форматі JSON парситься в моделі AIResponse, після чого ViewModel передає текст ШІ до UI, де він відображається в стилі діалогу.

Останній модуль побудований із DecryptedQRCodeView та DecryptedQRCodeViewModel і відповідає за фінальний етап – розшифрування повідомлення і його відображення. Після того, як попередній модуль надав дані, цей модуль виконує операцію дешифрування за допомогою відповідного алгоритму, враховуючи ключ, отриманий із введеного паролю, та відновлює оригінальний текст повідомлення. У разі успішного дешифрування користувачу відображається розшифроване повідомлення, а при помилці – повідомлення з інформацією про невірний пароль або іншу проблему.

Таким чином, кожен модуль, утворений парою View та ViewModel, несе свою чітко визначену роль у загальному процесі: від збору та валідації даних, через шифрування і генерацію QR-коду, до зчитування, вибору алгоритму і фінального розшифрування повідомлення. Ця модульна архітектура забезпечує підвищену безпеку, тестованість і зручність підтримки застосунку, дозволяючи кожному компоненту ефективно взаємодіяти з іншими частинами системи.

3.3.1 Вибір інструментів для клієнтської частини застосунку

Для створення клієнтської частини додатку було використано Swift та SwiftUI – сучасні технології від Apple, які значно оптимізують процес створення мобільних додатків і забезпечують високий рівень продуктивності, безпеки та інтерактивності. Цей вибір дозволяє розробникам будувати застосунки з чіткою структурою, гармонійним виглядом і потужною функціональністю, що відповідає сучасним вимогам ринку та зростаючим очікуванням користувачів.

Swift – це мова програмування, спеціально розроблена для екосистеми Apple. Її сучасний синтаксис сприяє написанню читабельного коду, що знижує ризик виникнення помилок під час розробки. Завдяки системі статичної типізації, Swift дозволяє виявляти більшість потенційних помилок на етапі компіляції, що робить кінцевий продукт більш стабільним. Крім того, Swift оптимізована для високої продуктивності, що є надзвичайно важливим для мобільних застосунків, де ресурси пристрою є обмеженими. Мова підтримує сучасні парадигми програмування, зокрема функціональний підхід, що сприяє скороченню обсягу коду і покращенню його підтримки[5]. Використання Swift забезпечує легкість інтеграції із сторонніми бібліотеками та фреймворками, що дозволяє швидко впроваджувати нові можливості та адаптувати застосунок до змін у вимогах ринку.

SwiftUI – це декларативний фреймворк для побудови користувацького інтерфейсу, який революціонізував спосіб створення UI в iOS-додатках. У SwiftUI розробник описує зовнішній вигляд і поведінку інтерфейсу за допомогою декларативного синтаксису, що дозволяє автоматично відслідковувати й

відображати зміни стану програми. Це означає, що при зміні даних у відповідних ViewModel, інтерфейс миттєво оновлюється без необхідності вручну взаємодіяти з UI-компонентами. SwiftUI сприяє створенню адаптивних, гнучких і масштабованих інтерфейсів, здатних підтримувати різні розміри екранів і орієнтації пристроїв. Завдяки вбудованій підтримці анімацій, переходів і кастомізації, розробники можуть створювати сучасні, естетично привабливі застосунки, що забезпечують комфортний та інтуїтивно зрозумілий користувацький досвід[6].

3.3.2 Вибір інструментів IDE

Xcode є офіційним інтегрованим середовищем розробки від Apple для створення застосунків під iOS, macOS, watchOS та tvOS, що робить його невід'ємною частиною розробки мобільних застосунків у екосистемі Apple. Його функціональність включає підтримку мов програмування Swift та Objective-C, забезпечення візуального конструювання інтерфейсів за допомогою Interface Builder, а також можливості налагодження коду, аналізу продуктивності та тестування додатків у симуляторах різних пристроїв. Завдяки глибокій інтеграції з платформою, Xcode дозволяє розробникам ефективно управляти ресурсами проекту, слідкувати за станом коду через систему контролю версій та використовувати багатий набір інструментів для профілювання продуктивності, що суттєво спрощує процес виявлення та усунення помилок.

Серед основних переваг Xcode можна відзначити його зручний інтерфейс та інтуїтивно зрозумілі інструменти, що дозволяють швидко створювати прототипи застосунків і легко переходити від написання коду до тестування та розгортання продукту. Інтеграція з системою macOS забезпечує оптимізацію ресурсів, а також підтримку новітніх технологій та стандартів розробки, що сприяє написанню стабільних та високоефективних застосунків. Крім того, наявність інструментів для роботи з графікою, анімаціями та даними спрощує вирішення складних завдань у розробці мобільних продуктів.

Проте Xcode має і певні недоліки, серед яких варто зазначити, що він орієнтований виключно на екосистему Apple, що обмежує можливості розробки для інших платформ. Середня вимогливість до апаратних ресурсів може стати перешкодою для розробників, які використовують менш потужні комп'ютери, а періодичні оновлення інструментів можуть спричиняти необхідність адаптації існуючого коду до нових стандартів. Незважаючи на це, враховуючи широкий набір функціональних можливостей, зручний інтерфейс і високу інтегрованість з розробкою для пристроїв Apple, Xcode лишається найоптимальнішим вибором для розробки мобільних застосунків, зокрема, коли мова йде про створення систем, що використовують специфічні можливості платформи, як-от шифрування повідомлень та генерація QR-кодів.

3.4 Аналіз безпеки даних

Розроблений мобільний застосунок для шифрування повідомлень забезпечує високий рівень безпеки даних завдяки застосуванню сучасних методів криптографії та чітко структурованої архітектури обробки інформації. Передача зашифрованих повідомлень у вигляді QR-кодів гарантує, що конфіденційні дані, зокрема текст повідомлення та пароль, не зберігаються у відкритому вигляді на клієнтських пристроях. Шифрування здійснюється локально на пристрої користувача, після чого згенерований QR-код містить лише зашифрований вміст, який може бути розшифрований виключно при введенні коректного паролю, що значно ускладнює несанкціонований доступ до інформації.

Важливим є те, що застосунок не зберігає жодних персональних даних користувача, таких як адреса електронної пошти чи інші ідентифікаційні дані, які могли б бути використані для відновлення доступу до зашифрованої інформації. Дані, необхідні для генерації QR-коду, обробляються виключно у межах сеансу, забезпечуючи тим самим мінімізацію ризику витоку інформації з пристрою. Критичні дані, що формують основу для процесу шифрування і розшифрування,

перевіряються як на стороні клієнта, так і у процесі передачі між компонентами застосунку, що сприяє виявленню будь-яких аномалій на ранніх етапах обробки.

Застосунок використовує безпечні протоколи зв’язку, такі як HTTPS, для передачі даних між модулями мобільного застосунку та будь-якими сторонніми сервісами, що можуть бути використані для синхронізації або збереження даних. Під час обміну зашифрованою інформацією застосунок впроваджує додаткові заходи безпеки, включаючи використання складних алгоритмів шифрування, що не дозволяють потенційному зловмиснику розшифрувати дані без повного набору необхідних ключів. Крім того, обробка даних на стороні клієнта включає ретельну перевірку вводу користувача, що дозволяє запобігти введенню некоректної інформації і потенційним атакам, пов’язаним із зловживанням вхідними даними.

4 ОПИС РОБОТИ КОРИСТУВАЧА З СИСТЕМОЮ

Даний розділ містить детальний аналіз тестування, валідації функціональності, продуктивності та безпеки системи, а також демонстрацію кінцевого продукту. Ми розглядаємо, як застосунок задовольняє вимоги щодо безперебійної роботи, стійкості до помилок і здатності обробляти дані в умовах високої навантаженості, а також надаємо оцінку зручності використання для кінцевих користувачів.

Під час аналізу якості було здійснено ряд тестів, спрямованих на перевірку відповідності реалізованої логіки вимогам, що постають перед системою шифрування і розшифрування повідомлень через QR-коди. До уваги беруться як технічні показники, такі як швидкість генерації та зчитування QR-кодів, так і користувацький досвід, що визначається зручністю інтерфейсу, інтерактивністю елементів та якістю зворотного зв’язку. Подальша демонстрація програмного забезпечення здійснюється з метою наочного підтвердження працездатності всіх модулів застосунку, що підтверджує їхню готовність до реального використання.

Цей розділ також охоплює методологію тестування, приклади сценаріїв використання, результати експериментів з оцінки безпеки і продуктивності, а також інтерпретацію отриманих даних з огляду на досягнення встановлених стандартів якості. Такий комплексний підхід дозволяє визначити сильні та слабкі сторони реалізованого рішення, що є важливим для подальшого вдосконалення продукту та забезпечення його конкурентоспроможності на ринку мобільних застосунків.

4.1 Опис процесів тестування

У цьому дослідженні проведено ручне тестування функціональних компонентів програмного забезпечення. Характеристику відповідних тестів подано в таблицях 4.1. – 4.8.

Таблиця 4.1 – Опис тесту EnterMessageView

Test	Module	Test Number
Валідація коректного вводу	EnterMessageView / EnterMessageViewModel	1.1

Після переходу на екран введення повідомлення користувач бачить два поля: одне для тексту довжиною не більше встановленого ліміту, інше для пароля з мінімум чотирма символами. При введенні коректного повідомлення й пароля система миттєво здійснює перевірку обох полів і підтверджує їхню валідність - на полях з’являються нормальні контури без червоного підсвічування, а кнопка «Далі» набуває активного стану. Після натискання на тепер уже доступну кнопку відбувається перехід до наступного етапу, що свідчить про успішне прийняття даних та готовність додатку продовжити роботу з шифруванням або дешифруванням.

Таблиця 4.2 – Опис тесту EnterMessageView

Test	Module	Test Number
Помилка при перевищенні ліміту	EnterMessageView / EnterMessageViewModel	1.2

Після переходу на екран введення повідомлення користувач бачить порожнє текстове поле й кнопку «Далі», яка на початку неактивна. При спробі ввести текст, що перевищує встановлену довжину (наприклад, більше ніж 250 символів), поле автоматично підсвічується червоним контуром, а під ним з’являється підказка «Текст перевищує максимально дозволену довжину». У цей момент кнопка «Далі» залишається неактивною, неможливо перейти до наступного кроку, доки обсяг введеного повідомлення не відповідатиме ліміту. Таким чином система одразу інформує про помилку, не допускаючи некоректного вводу та забезпечуючи чіткий візуальний зворотний зв’язок.

Таблиця 4.3 – Опис тесту QRCodeDisplayView

Test	Module	Test Number
Генерація QR-коду	QRCodeDisplayView / QRCodeViewModel	1.3

Після переходу на екран генерації QR-коду користувач вводить текст повідомлення, встановлює пароль та обирає алгоритм шифрування. Натискання кнопки «Створити» запускає процес: система генерує випадковий вектор ініціалізації, шифрує повідомлення обраним алгоритмом, виконує кодування результату у формат Base64 та будує QR-зображення. Усього за кілька миттєвостей на екрані з’являється готовий QR-код у високій роздільній здатності, а внизу спливаючим банером з’являється підтвердження «QR-код успішно згенеровано». Інтерфейс демонструє плавну анімацію завантаження та чіткі елементи керування, що дозволяють відразу зберегти чи поділитися створеним кодом.

Таблиця 4.4 – Опис тесту ImportOrScanQRCodeView

Test	Module	Test Number
Зчитування QR-коду	ImportOrScanQRCodeView / QRCodeInputViewModel	1.4

Після переходу на екран, що дозволяє обрати спосіб роботи з QR-кодом, користувач бачить дві опції: «Сканувати» або «Імпортувати зображення». При виборі імпорту система відкриває галерею, де користувач може вказати фотографію з дійсним QR-кодом. Після завантаження зображення додаток автоматично передає його в CIDetector, який аналізує кадр на наявність коректного маркера.

У момент успішного розпізнавання QR-коду з’являється коротке повідомлення «QR-код успішно зчитано», а кнопка для переходу до наступного етапу стає активною та легко натискається. Інтерфейс миттєво адаптується до стану

завантаження: поле сканування більше не підсвічується, зникає індикатор обробки, і користувач отримує чітке візуальне підтвердження того, що система готова перейти до дешифрування або відображення вмісту.

Таблиця 4.5 – Опис тесту DecryptQRCodeView

Test	Module	Test Number
Розшифрування з правильним паролем	DecryptQRCodeView / DecryptedQRCodeViewModel	1.5

Після успішного сканування дійсного QR-коду користувач бачить екран із полем для введення пароля. Введений пароль порівнюється з необхідними криптографічними ключами, після чого система автоматично обирає відповідний алгоритм дешифрування. Якщо пароль вірний, додаток миттєво розшифровує вміст QR-коду та відображає оригінальне повідомлення в основному вікні - текст підтверджується візуально чітким шрифтом і, за потреби, додатковою анімацією успішної операції.

Одночасно з цим користувач отримує спливаюче повідомлення «Повідомлення успішно розшифровано», виконане у приємному для сприйняття стилі, що відповідає загальному дизайну інтерфейсу. Прокручування чи масштабування тексту працює коректно, а доступні опції копіювання та подальшої обробки повідомлення гарантовано активні. Усі елементи інтерфейсу адаптивно підлаштовуються під розмір екрану та робочі умови пристрою, забезпечуючи плавний і наочний досвід користувача.

Таблиця 4.6 – Опис тесту DecryptQRCodeView

Test	Module	Test Number
Помилка розшифрування з неправильним паролем	DecryptQRCodeView / DecryptedQRCodeViewModel	1.6

Після сканування дійсного QR-коду користувач опиняється на екрані введення пароля. Після введення невірного пароля система миттєво ініціює перевірку даних і виявляє невідповідність. Замість дешифрування з’являється чітке повідомлення про помилку «Неправильний пароль», виконане у контрастному кольорі та розміщене безпосередньо під полем введення, щоб користувач одразу помітив його. Дешифрування не запускається, а інтерфейс залишається активним для повторної спроби - користувач може виправити помилку або скористатися опцією відновлення доступу. У разі потреби система додатково супроводжує повідомлення невеликою вібрацією або звуковим сигналом, що підсилює зворотний зв’язок і забезпечує однозначне розуміння, що пароль було введено некоректно.

Таблиця 4.7 – Опис тесту QRCodeView

Test	Module	Test Number
Збереження QR-коду	QRCodeView / QRCodeViewModel	1.7

Після генерації QR-коду він миттєво відображається в центрі екрана як об’єкт UIImage, забезпечуючи чітке та якісне відтворення всіх деталей. Коли користувач натискає кнопку «Зберегти QR-код», додаток без зволікань виконує збереження зображення у внутрішнє сховище: під час цього може з’явитися невелика анімація або індикатор прогресу, що підтверджує початок операції. Після успішного завершення процесу система демонструє коротке спливаюче повідомлення «QR-код успішно збережено» та, за наявності підтримки, виконує легку тактильну віддачу для надання відчутного підтвердження дії. Уся операція проходить плавно й без помилок на різних пристроях та в різних умовах роботи інтерфейсу.

Таблиця 4.8 – Опис тесту QRCodeHistoryView

Test	Module	Test Number
Відображення збережених QR-кодів	QRCodeHistoryView / QRCodeHistoryViewModel	1.8

Після запуску мобільного додатку користувач натискає на елемент навігації, що веде до екрану історії збережених QR-кодів. Відкривається сторінка з заголовком «Історія збережених QR-кодів», де у верхній частині екрану одразу видно назву розділу. Нижче за заголовком розташовується перелік усіх збережених раніше кодів: для кожного відображається чітка мініатюра сама собою, поруч із нею наведено назву чи короткий коментар користувача та точна дата й час збереження у форматі «день.місяць.рік година:хвилина».

Сторінка завантажується миттєво, без затримок, а прокрутка довгого лонґліста працює плавно та стабільно навіть при великій кількості елементів. Якщо ж у сховищі відсутні збережені коди, замість списку з’являється приємний плейсхолдер із повідомленням про відсутність історії та натяком на необхідність створити новий QR-код. Інтерфейс повністю відповідає затвердженому дизайну: шрифти й відступи точно узгоджені з макетом, а локалізовані тексти коректно відображаються як у портретному, так і в ландшафтному режимах на різних типах екранів, з врахуванням налаштувань мови інтерфейсу.

4.2 Опис автоматичних тестів

Далі протестуємо контрольний приклад. В цьому розділі будуть описані автоматичні тести, та результати їх виконання. Тестування було виконано за допомогою XCTest та ViewInspector[9, 10].

```swift
final class DecryptedQRCodeViewModelTests: XCTestCase {
    func testDecodeQRCodeFailure() {

        let dummyImage = UIImage(systemName: "qrcode")!
        let vm = DecryptedQRCodeViewModel(qrCodeImage: dummyImage,
                                          encryptionMethod: EncryptionMethod(name: .aes, maxAllowedCharacters: 200),
                                          password: "password123")

        vm.decodeAndDecrypt()

        XCTAssertFalse(vm.isLoading)
        XCTAssertTrue(vm.decryptionFailed)
        XCTAssertEqual(vm.decryptedMessage, "No QR code message found.")
    }
```

Рисунок 4.1 – Тестування декодування QR-коду

```swift
func testAESDecryptionRoundTrip() {
    let password = "password123"
    let message = "This is a test message for AES."
    let method = EncryptionMethod(name: .aes, maxAllowedCharacters: 200)

    let qrVM = QRCodeViewModel(password: password,
                               message: message,
                               encryptionMethod: method)
    let encrypted = qrVM.encryptedMessage

    let dummyImage = UIImage(systemName: "qrcode")!
    let decryptedVM = DecryptedQRCodeViewModel(qrCodeImage: dummyImage,
                                               encryptionMethod: method,
                                               password: password)

    let data = Data(base64Encoded: encrypted)!
    let allBytes = [UInt8](data)
    let ivLength = 16
    let iv = Array(allBytes.prefix(ivLength))
    let encryptedBytes = Array(allBytes.dropFirst(ivLength))
    let key = Array(password.utf8).sha256()

    do {
        let aes = try AES(key: key, blockMode: CBC(iv: iv), padding: .pkcs7)
        let decryptedBytes = try aes.decrypt(encryptedBytes)
        let decryptedMessage = String(bytes: decryptedBytes, encoding: .utf8)
        XCTAssertEqual(decryptedMessage, message)
    } catch {
        XCTFail("AES decryption failed with error: \(error)")
    }
}
```

Рисунок 4.2 – Тест AES шифрування

Даний код містить два тестові методи, реалізовані у класі тестування для DecryptedQRCodeViewModel . У першому тестовому методі на рисунку 4.1, створюється dummy-зображення за допомогою системного значка "qrcode", яке не містить реального QR-коду. Потім створюється екземпляр моделі, що відповідає за розшифрування, із зазначенням обраного алгоритму шифрування (AES) та тестового пароля. Виклик методу decodeAndDecrypt змушує модель спробувати декодувати та дешифрувати дані зі зображення. Оскільки в зображенні немає валідної інформації, очікується, що прапорці стану будуть встановлені відповідним чином (isLoading – false, decryptionFailed – true), а текст помилки буде рівний "No QR code message found.".

Другий тест на рисунку 4.2, перевіряє повний цикл шифрування і розшифрування за допомогою алгоритму AES. Спочатку за допомогою QRCodeViewModel повідомлення шифрується, і отриманий зашифрований рядок зберігається. Далі дані з цього рядка декодуються: виділяється початковий вектор ініціалізації (IV) та зашифровані байти. Генерується ключ із заданого пароля, і на основі цього створюється об’єкт AES для виконання дешифрування. Результат дешифрування перетворюється на рядок, і за допомогою перевірки XCTAssertEqual здійснюється порівняння розшифрованого тексту з початковим повідомленням. Таким чином, тест перевіряє, що механізм шифрування та дешифрування працює коректно, забезпечуючи правильне відновлення початкових даних.

```swift
final class EnterMessageViewModelTests: XCTestCase {
    func testMessageValidation() {
        let method = EncryptionMethod(name: .aes, maxAllowedCharacters: 200)
        let vm = EnterMessageViewModel(encryptionMethod: method)

        // Empty message
        vm.message = ""
        XCTAssertFalse(vm.isMessageValid)
        XCTAssertEqual(vm.messageErrorText, "Message cannot be empty.")

        // Valid message
        vm.message = "Hello, world!"
        XCTAssertTrue(vm.isMessageValid)

        // Exceeding characters limit
        vm.message = String(repeating: "a", count: 250)
        XCTAssertFalse(vm.isMessageValid)
        XCTAssertEqual(vm.messageErrorText, "Message exceeds the \(method.maxAllowedCharacters) character limit.")
    }
```

Рисунок 4.3 – Тест валідації повідомлень

На рисунку 4.3 тести EnterMessageViewModelTests охоплюють перевірку валідації введених даних. Перший тест (testMessageValidation) перевіряє, що при відсутності тексту повідомлення виводиться відповідне повідомлення про помилку, а при введенні коректного тексту – перевірка проходить успішно. Додатково тест перевіряє, що при перевищенні встановленого ліміту символів (за допомогою генерації рядка з повторенням символу) система відображає помилку, що повідомляє про перевищення ліміту.

```
func testPasswordValidationAndStrength() {
    let method = EncryptionMethod(name: .aes, maxAllowedCharacters: 200)
    let vm = EnterMessageViewModel(encryptionMethod: method)

    // Too short password
    vm.password = "123"
    XCTAssertFalse(vm.isPasswordValid)
    XCTAssertEqual(vm.passwordErrorText, "Password should be at least 4 characters.")

    // Valid password with expected strength evaluation
    vm.password = "Password1!"
    XCTAssertTrue(vm.isPasswordValid)
    XCTAssertGreaterThan(vm.passwordStrength, 0.0)
    let strengthText = vm.passwordStrengthText
    XCTAssertTrue(["Weak", "Moderate", "Strong"].contains(strengthText))
}
```

Рисунок 4.4 – Тест валідації пароля

На рисунку 4.4 другий тест зосереджується на перевірці коректності валідації пароля у мобільному застосунку: аналізується, чи система правильно визначає занадто короткий пароль як невалідний, видаючи відповідне повідомлення про помилку, а також оцінюється функціонал обчислення сили пароля для коректно введених даних. Сила пароля класифікується за трьома рівнями - "Weak", "Moderate" або "Strong" - залежно від таких критеріїв, як довжина, наявність різних типів символів (великі літери, цифри, спеціальні знаки) та їх комбінація.

Окрім цього, тест перевіряє, чи коректно відображається текст помилки у випадку невідповідності вимогам, наприклад, коли пароль коротший за 4 символи, та чи змінюється оцінка сили пароля при додаванні нових символів.

```
func testCanContinueLogic() {
    let method = EncryptionMethod(name: .aes, maxAllowedCharacters: 200)
    let vm = EnterMessageViewModel(encryptionMethod: method)
    // Initially empty fields: cannot continue.
    vm.message = ""
    vm.password = ""
    XCTAssertFalse(vm.canContinue)

    // Valid message and password.
    vm.message = "Valid message"
    vm.password = "Valid1!"
    XCTAssertTrue(vm.canContinue)
}
```

Рисунок 4.5 – Тест логіки продовження

Третій тест цього блоку, показаний на рисунку 4.5, перевіряє логіку переходу до наступного етапу: якщо поля повідомлення та паролю порожні – можливість продовження вимкнена, а при їх коректному заповненні – користувач може продовжити.

```
final class QRCodeViewModelTests: XCTestCase {
    func testQRCodeGeneration() {
        let password = "password123"
        let message = "Secret QR message."
        let method = EncryptionMethod(name: .aes, maxAllowedCharacters: 200)
        let vm = QRCodeViewModel(password: password, message: message, encryptionMethod: method)

        // QR code image should have been generated.
        XCTAssertNotNil(vm.qrCodeImage)
        XCTAssertFalse(vm.encryptedMessage.isEmpty, "Encrypted message should not be empty.")
    }
}
```

Рисунок 4.6 – Тест генерації QR-коду

Тест QRCodeViewModelTests, зображений на рисунку 4.6, перевіряє генерацію QR-коду. Тест створює екземпляр QRCodeViewModel із вхідними даними (повідомлення, пароль і алгоритм шифрування), після чого перевіряється, що після виконання шифрування і генерації зашифрованого повідомлення отримане зображення QR-коду не є nil, а рядок зашифрованих даних – не порожній.

```swift
final class DecryptQRCodeViewModelTests: XCTestCase {
    func testDefaultEncryptionMethod() {
        // Provide a dummy image.
        let dummyImage = UIImage(systemName: "qrcode")!
        let vm = DecryptQRCodeViewModel(qrCodeImage: dummyImage)
        // When initialized, it should set a default encryption method.
        XCTAssertNotNil(vm.methods)
        XCTAssertNotNil(vm.selectedMethod, "A default encryption method should be selected.")
    }

    func testCanContinueProperty() {
        let dummyImage = UIImage(systemName: "qrcode")!
        let vm = DecryptQRCodeViewModel(qrCodeImage: dummyImage)
        // When password is empty, cannot continue.
        vm.password = ""
        XCTAssertFalse(vm.canContinue)

        // When a method is selected and password is provided, can continue.
        vm.selectedMethod = vm.methods.first
        vm.password = "password123"
        XCTAssertTrue(vm.canContinue)
    }
}
```

Рисунок 4.7 – Тест ініціалізації для розшифрування

На рисунку 4.7 йдуть тести для DecryptQRCodeViewModelTests, які перевіряють ініціалізацію та властивості моделі для розшифрування. Перший тест (testDefaultEncryptionMethod) перевіряє, що при створенні моделі з dummy-зображенням список підтримуваних методів шифрування не порожній і що з вибраних методів встановлюється дефолтне значення. Другий тест (testCanContinueProperty) перевіряє, що користувач не може продовжити, якщо пароль відсутній, а при встановленні дефолтного методу і введенні пароля властивість canContinue повертає true.

Ці тести забезпечують надійність функціонування моделі, гарантуючи, що вона коректно ініціалізується і правильно реагує на різні сценарії введення даних. Вони також підтверджують, що система дотримується вимог безпеки, дозволяючи продовження лише за умови правильного вибору методу шифрування та введення пароля, що є ключовим для захисту конфіденційних повідомлень у процесі розшифрування QR-кодів.

```swift
final class QRCodeInputViewModelTests: XCTestCase {
    func testQRCodeValidation_invalid() {
        let vm = QRCodeInputViewModel()
        // Set an image that is not a QR code.
        vm.qrCodeImage = UIImage(systemName: "photo") // unlikely a valid QR.
        // Allow async validation to complete.
        let expectation = XCTestExpectation(description: "Waiting for QR validation")
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
            XCTAssertFalse(vm.isValidQRCode)
            expectation.fulfill()
        }
        wait(for: [expectation], timeout: 1)
    }

    func testQRCodeValidation_valid() {
        let vm = QRCodeInputViewModel()
        // Generate a real QR code image using CIFilter.
        let filter = CIFilter.qrCodeGenerator()
        let data = "Test QR Code".data(using: .utf8)!
        filter.message = data
        filter.correctionLevel = "Q"
        let context = CIContext()
        if let ciImage = filter.outputImage, let cgImage = context.createCGImage(ciImage, from: ciImage.extent) {
            vm.qrCodeImage = UIImage(cgImage: cgImage)
        }
        let expectation = XCTestExpectation(description: "Valid QR code validation")
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
            XCTAssertTrue(vm.isValidQRCode)
            expectation.fulfill()
        }
        wait(for: [expectation], timeout: 1)
    }
}
```

Рисунок 4.8 – Тест валідації зчитаного зображення

QRCodeInputViewModelTests, зображені на рисунку 4.8, орієнтовані на перевірку валідації зчитаного зображення з QR-кодом. Перший тест перевіряє ситуацію, коли зображення не є валідним QR-кодом, і очікується, що прапорець isValidQRCode залишається false. Другий тест використовує генерований за допомогою CIFilter QR-код для перевірки валідності, очікуючи, що властивість isValidQRCode встановиться в true після певної затримки.

Ці тести забезпечують коректність роботи механізму валідації QR-кодів, перевіряючи як негативні, так і позитивні сценарії обробки зображень у застосунку. Вони також гарантують, що система правильно визначає валідність QR-коду, забезпечуючи користувачу точний зворотний зв’язок, і що затримка в обробці не впливає на надійність результату, що є важливим для безперебійного функціонування процесу сканування та розшифрування.

```swift
final class EncryptionMethodsViewModelTests: XCTestCase {
    func testMethodSelection() {
        let vm = EncryptionMethodsViewModel()
        XCTAssertEqual(vm.methods.count, 4)
        XCTAssertNil(vm.selectedMethod)
        vm.selectedMethod = vm.methods.first
        XCTAssertNotNil(vm.selectedMethod)
    }
}
```

Рисунок 4.9 – Тест очікуваної кількості алгоритмів

Останній блок юніт тестів, що зображені на рисунку 4.9 – EncryptionMethodsViewModelTests – перевіряє, що список методів шифрування містить очікувану кількість алгоритмів, і що після вибору першого з методів властивість selectedMethod не є nil. Це забезпечує правильну ініціалізацію і вибір алгоритму шифрування для подальшої роботи застосунку.

Також було імплементовано UITest, котрі використовують фреймворк ViewInspector для перевірки структури SwiftUI інтерфейсу. Тести для DecryptedQRCodeView перевіряють, що на екрані розшифрування встановлено правильний заголовок ("Decrypted Message") та що на екрані присутній текст розшифрованого повідомлення. Тести EnterMessageView перевіряють наявність основних елементів інтерфейсу, таких як заголовок "Enter Your Message" та присутність поля для введення паролю (зазвичай SecureField). Тести для QRCodeDisplayView підтверджують, що екран показу QR-коду містить заголовок "Your Encrypted QR Code" та елементи, що відображають зашифроване повідомлення. Аналогічним чином, тест для DecryptQRCodeView перевіряє коректну установку заголовку "Decrypt QR Code" та наявність елемента для введення паролю. Тест ImportOrScanQRCodeView перевіряє, що екран для імпорту або сканування QR-коду містить заголовок "Import QR Code" і кнопки, що дозволяють вибір між імпортом та скануванням. Нарешті, тест для SelectEncryptionMethodView перевіряє, що екран вибору методу шифрування має

правильний заголовок і що в списку методів, як-от "AES", є наявні елементи, що дозволяють користувачу зробити вибір.

Цей набір тестів демонструє комплексний підхід до автоматичного тестування застосунку «QRMessages», охоплюючи як бізнес-логіку, так і інтерфейс користувача, що дає можливість впевнено стверджувати про високу якість реалізації функціональних можливостей застосунку.

З результатами тестів можна ознайомитись на рисунку 4.10.

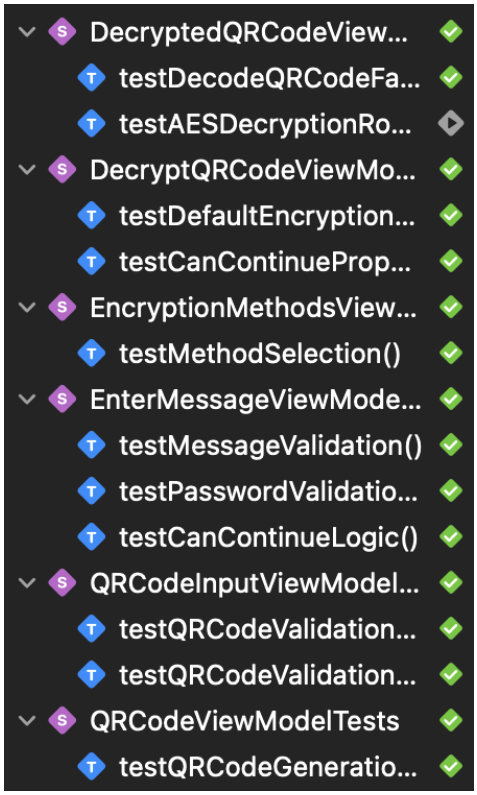

Рисунок 4.10 – Результат виконання тестів

4.3 Робота користувача з програмною системою

Для роботи з мобільним застосунком необхідно:

- пристрій під керуванням iOS 14.0 або вище;
- наявність камери для сканування QR-кодів;
- доступ до фотогалереї для імпорту QR-кодів;
- інтернет-з’єднання для оновлень та довідкової інформації.

Далі на рисунках 4.11 – 4.17 буде описано сценарій роботи користувача з системою:

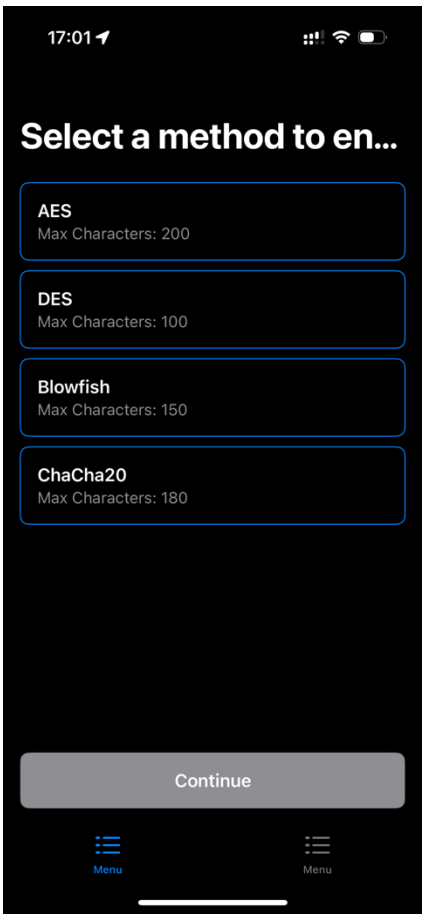

Рисунок 4.11 – Головний екран застосунку з вибором необхідного алгоритму

При відкритті екрану введення повідомлення користувач бачить перелік доступних алгоритмів шифрування (AES, DES, Blowfish, ChaCha20) із зазначенням максимально допустимої довжини тексту для кожного з них, при цьому значення обмежень підтягуються з конфігураційного файлу EncryptionConfig.plist під час ініціалізації EnterMessageViewModel.

Якщо користувач намагається ввести текст довший, ніж currentMaxLength, додаткові символи відкидаються, межа текстового поля підсвічується червоною рамкою, а під полем з’являється повідомлення про максимальну довжину. До того ж кнопка «Зашифрувати» залишається неактивною, доки довжина тексту не відповідає умовам валідації, реалізованої у методі validateInput(_:), перевірка якого

покрита юніт-тестами в EnterMessageViewModelTests. Після того, як користувач обрав алгоритм та ввів коректний текст, викликається функція encryptMessage(text:), що передає управління модулю генерації QR-коду, де обраний алгоритм і встановлене обмеження гарантовано застосовуються під час шифрування й побудови коду.

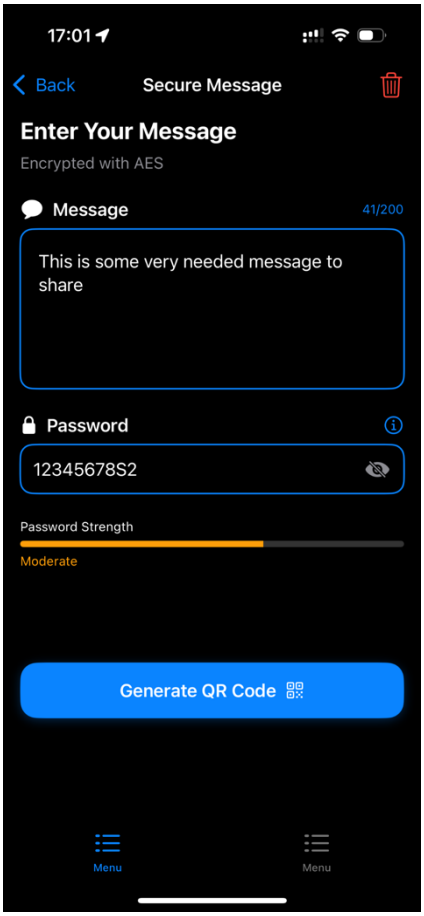

Рисунок 4.12 – Створення повідомлення та паролю

Після вибору алгоритму шифрування користувач вводить текст повідомлення у відведеному полі, а безпосередньо під ним з’являється рядок для введення паролю з іконкою замка та кнопкою-перемикачем «око» для показу або приховування введеного тексту . Як тільки користувач починає вводити пароль, під полем автоматично з’являється індикатор надійності, що складається з прогрес-бару та текстового маркеру рівня складності («Weak», «Moderate», «Strong»). Ширина заповнення бару відповідає заданому значенню обчислюваного

властивого passwordStrength (діапазон 0…1), а виведений колір бару – passwordStrengthColor (червоний за <0.3, помаранчевий за <0.7, зелений інакше), які в EnterMessageViewModel аналізують довжину паролю та різноманітність символів (великі/малі літери, цифри, знаки) і формують текстове пояснення в passwordStrengthText . Таким чином, користувач отримує миттєвий візуальний зворотний зв’язок щодо безпеки свого паролю перед тим, як продовжити шифрування.

Рисунок 4.13 – Згенерований QR-code

Після натискання кнопки «Generate QR Code» система передає управління до QRCodeViewModel, де викликається метод generateQRCode(). Спочатку обране повідомлення шифрується за допомогою вибраного алгоритму (AES, DES, Blowfish або ChaCha20) з додаванням ініціалізаційного вектора, після чого отримані байти перетворюються у Base64-рядок та зберігаються у властивості encryptedMessage.

Далі рядок кодується у дані та передається в CIFilter.qrCodeGenerator(), отриманий CIImage масштабують через трансформацію CGAffineTransform(scaleX:10,y:10), перетворюють у UIImage і присвоюють полю qrCodeImage, яке через @Published миттєво оновлює UI. У результаті на екрані відображається згенероване зображення QR-коду поряд із текстовим полем, в якому користувач бачить Base64-кодовану зашифровану версію свого повідомлення .

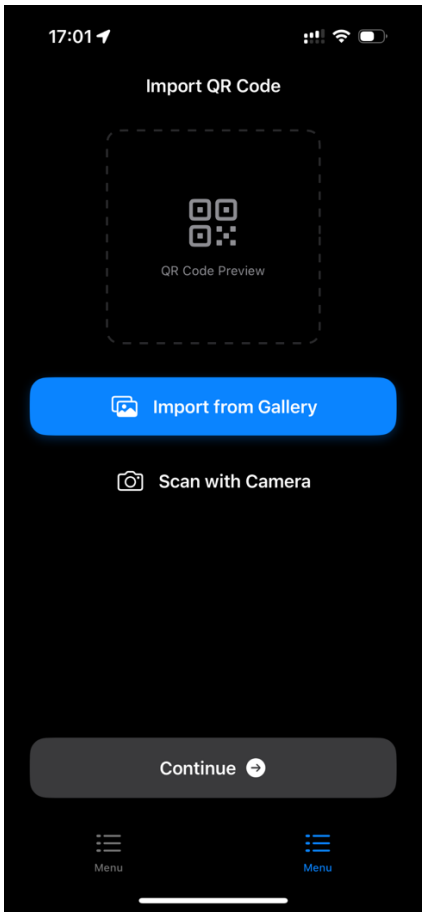

Рисунок 4.14 – Головний екран для імпортування QR-code

Після переходу на екран імпорту або сканування QR-коду застосунок відображає у центрі область-прев’ю з тонким штриховим контуром, яка своєю чергою одразу показує завантажене зображення у вигляді закругленого квадрата з тінню, а при успішній валідації коду - накладає в правому кутку зелену “галочку”. Під прев’ю розташовані дві кнопки, перша з яких запускає шарієр фото з галереї через ImagePicker(image: $viewModel.qrCodeImage, sourceType: .photoLibrary), а

друга відкриває камеру з аналогічним піктометодом sourceType: .camera. Як тільки користувач обирає або робить фотографію за допомогою камери, то вона записується у властивість qrCodeImage моделі QRCodeInputViewModel, яка через didSet викликає validateQRCode(). Цей метод застосовує CIDetector з типом CIDetectorTypeQRCode для подальшого точного аналізу CIImage і оновлює прапори isValidQRCode і isShowingAlert, відтак у разі невдалого сканування підтягується модальне повідомлення про помилку, а при позитивному результаті активується кнопка «Continue», що дозволяє перейти до розшифрування отриманого повідомлення.

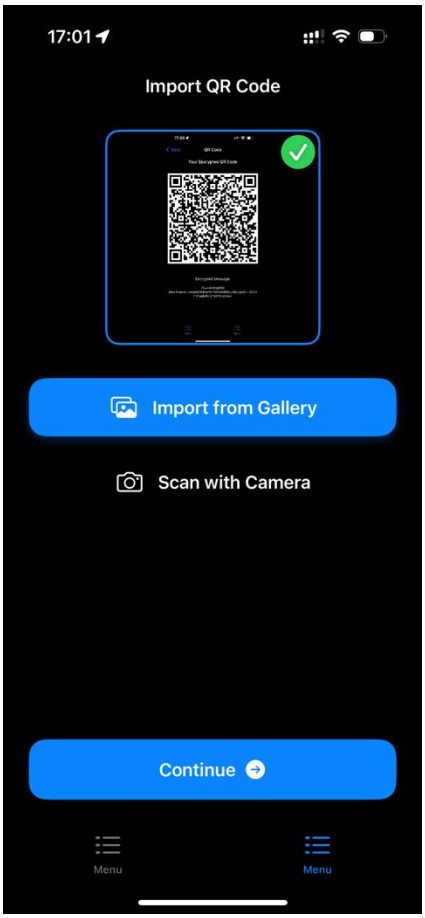

Рисунок 4.15 – Вигляд екрану після імпорту QR-code

Після того, як користувач імпортує або сканує QR-код, у прев’ю коду відразу з’являється зелена “галочка” успішної валідації, а кнопка «Continue» (або «Decrypt

QR Code») стає активною, дозволяючи перейти до наступного етапу дешифрування.

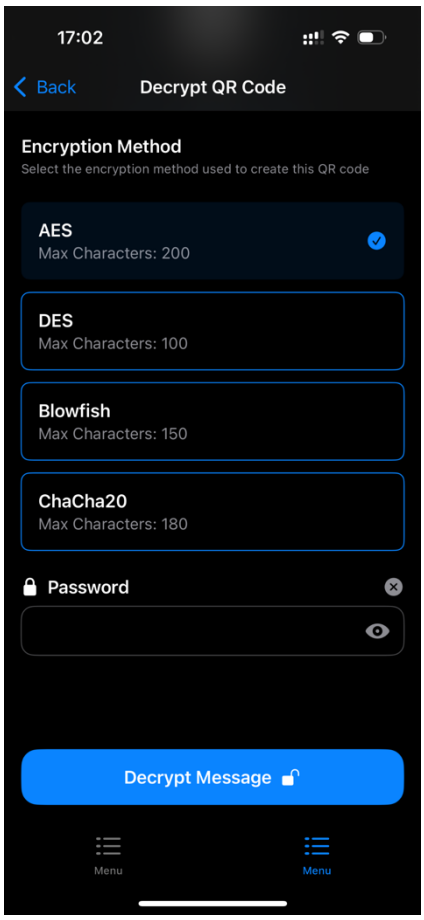

Рисунок 4.16 – Екран з вибором алгоритму дешифрування

На екрані дешифрування користувач бачить попередній перегляд зображення QR-коду, під яким розміщено розділ вибору алгоритму (AES, DES, Blowfish, ChaCha20) та поле для вводу паролю. Кожна кнопка з алгоритмом відображається в компоненті EncryptionMethodRow, а під кутом “lock” користувач вводить пароль, який можна приховати чи показати натисканням на іконку “око” .

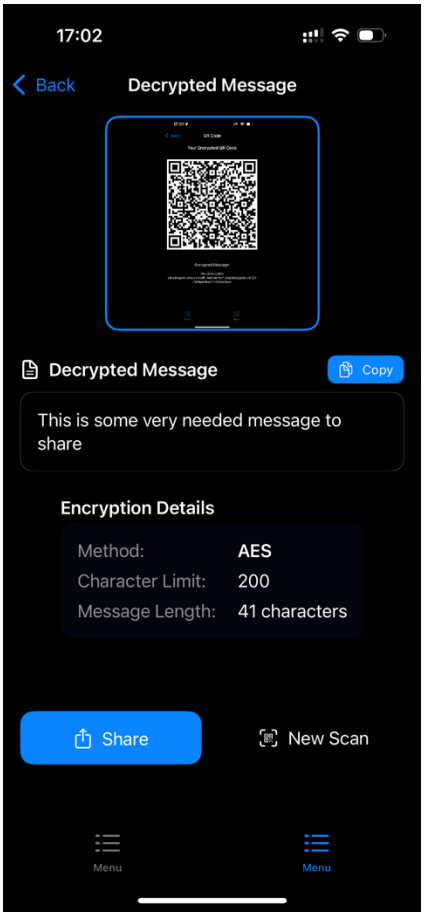

Рисунок 4.17 – Екран з дешифрованим повідомленням

По натисненні «Decrypt Message» відбувається візуальна анімація кнопки та перехід за допомогою NavigationLink до DecryptedQRCodeView, де при ініціалізації ViewModel із затримкою 0.8 с запускається метод decodeAndDecrypt(). Цей метод спочатку витягує зображене в QR-коді Base64-повідомлення і, використовуючи вибраний алгоритм та пароль, виконує розшифрування, після чого оновлює властивість decryptedMessage або викидає помилку в разі невірних даних . Коли розшифрування відбулося успішно, екран переходить до відображення результату: у заголовку розміщено текст «Decrypted Message» разом із кнопкою «Copy», а нижче - саме відновлений текст у прокручуваному блоці. Під повідомленням у секції «Encryption Details» чітко вказано назву алгоритму в RAW-форматі, максимальну довжину символів для цього алгоритму та фактичну довжину розшифрованого тексту за допомогою Grid-компоненту

5 ВПРОВАДЖЕННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Цей розділ описує процес впровадження та розповсюдження застосунку «QRMessages» на ринку, із детальним розглядом різних підходів до розгортання залежно від цільової аудиторії та етапу розробки. Основні підходи включають публічне розгортання через офіційні магазини додатків, використання платформ для бета-тестування, корпоративні рішення та навіть розгортання як прогресивного веб-додатку (PWA).

5.1 Розгортання програмного забезпечення

Для публічного розповсюдження застосунку найбільш ефективним варіантом є розміщення його в офіційних магазинах, зокрема в Apple App Store для iOS. Цей підхід забезпечує максимальне охоплення користувачів завдяки широкій аудиторії, автоматичним оновленням додатку та високим стандартам якості, оскільки кожна версія проходить ретельну перевірку перед публікацією. Однак цей метод вимагає дотримання суворих вимог і стандартів, що може призводити до затримок у розгортанні через необхідність проходження процесу верифікації.

Для цілей бета-тестування та внутрішнього використання застосунку застосовуються спеціалізовані платформи, такі як TestFlight. Використання TestFlight дозволяє швидко розповсюджувати тестові версії серед обмеженої групи користувачів, що значно полегшує збір оперативного зворотного зв’язку та виправлення помилок[11]. Незважаючи на обмежене коло учасників, цей підхід дає можливість оптимізувати функціональність додатку та адаптувати його до потреб кінцевих користувачів задовго до його публічного релізу.

Для корпоративного використання застосунку можливе впровадження Enterprise Distribution, яке дозволяє розповсюджувати додаток безпосередньо серед співробітників організації, оминаючи процедуру публікації в загальнодоступних

магазинах додатків. Це рішення забезпечує високий рівень контролю над доступом та використанням програмного забезпечення всередині компанії, хоча воно не підходить для широкої публічної аудиторії.

Застосунок також може бути реалізований як прогресивний веб-додаток (PWA) для тих користувачів, які віддають перевагу кросплатформеним рішенням. Розгортання як PWA дозволяє встановлювати додаток безпосередньо з веб-сайту, що обходить жорсткий процес верифікації в магазинах додатків. Проте це може бути супроводжене певними обмеженнями в доступі до апаратних можливостей пристроїв. Вибір конкретного методу розгортання впливає на зручність оновлення, швидкість доставки нових функціональних можливостей, а також на загальний рівень безпеки застосунку.

5.2 Системні вимоги та підтримка застосунку

Підтримка застосунку включає регулярні оновлення, спрямовані на покращення функціональності, виправлення помилок та забезпечення безпеки. Система моніторингу дозволяє оперативно реагувати на відгуки користувачів і виявлені проблеми, а використання гнучкої архітектури MVVM сприяє швидкому впровадженню змін без порушення стабільності роботи продукту[12]. Крім того, застосунок інтегровано з сервісами аналітики та моніторингу продуктивності, що дозволяє ефективно оптимізувати ресурси та підтримувати високу швидкодію при змінних умовах експлуатації.

ВИСНОВКИ

У результаті виконання дипломної роботи було створено мобільний застосунок, що забезпечує захищений обмін повідомленнями шляхом їх шифрування і представлення у вигляді QR-кодів. Розроблена система поєднує в собі сучасні алгоритми симетричного шифрування (AES, Blowfish, ChaCha20), механізми валідації введених даних та зручний інтерфейс на базі SwiftUI з архітектурою MVVM. Це дозволило реалізувати повний цикл роботи користувача: від введення тексту та паролю до генерації зашифрованого коду, його сканування й коректного відновлення початкового повідомлення.

Застосування BPMN-моделювання бізнес-процесів надало чітке уявлення про послідовність операцій при генерації й дешифруванні QR-кодів, що сприяло оптимізації логіки взаємодії компонентів системи й мінімізації ймовірності помилок. Використання CryptoSwift для реалізації криптографічних операцій гарантувало відповідність сучасним стандартам безпеки, а локальне шифрування на пристрої користувача виключає передачу відкритих даних мережею.

Побудована модульна структура з пар “View–ViewModel” забезпечує високу тестованість і гнучкість системи, що полегшує подальший розвиток застосунку та інтеграцію нових алгоритмів шифрування. Інтуїтивно зрозумілий інтерфейс і підтримка імпорту/сканування QR-кодів роблять рішення доступним як для пересічних користувачів, так і для організацій із підвищеними вимогами до інформаційної безпеки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ISO/IEC 18004:2015 Automatic Identification and Data Capture Techniques – QR Code Bar Code Symbology Specification. ISO: Веб сайт. URL: https://www.iso.org/standard/62021.html (дата звернення 15.04.2025).

2. FIPS 197 Advanced Encryption Standard (AES). National Institute of Standards and Technology: Веб сайт. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (дата звернення 10.04.2025).

3. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). Bruce Schneier: Веб сайт. URL: http://www.schneier.com/paper-blowfish-fse.html (дата звернення 21.04.2025).

4. ChaCha, a Variant of Salsa20. Daniel J. Bernstein: Веб сайт. URL: https://cr.yp.to/chacha/chacha-20080128.pdf (дата звернення 21.04.2025).

5. The Swift Programming Language (Swift 5.9). Apple Inc.: Веб сайт. URL: https://docs.swift.org/swift-book/ (дата звернення 21.04.2025).

6. SwiftUI Documentation. Apple Inc.: Веб сайт. URL: https://developer.apple.com/documentation/swiftui (дата звернення 21.04.2025).

7. CryptoSwift – Standard Cryptographic Algorithms in Swift. CryptoSwift Contributors: Веб сайт. URL: https://github.com/krzyzanowskim/CryptoSwift (дата звернення 21.04.2025).

8. Core Image Framework Reference. Apple Inc.: Веб сайт. URL: https://developer.apple.com/documentation/coreimage (дата звернення 21.04.2025).

9. XCTest Framework Reference. Apple Inc.: Веб сайт. URL: https://developer.apple.com/documentation/xctest (дата звернення 21.04.2025).

10. ViewInspector – A SwiftUI View Testing Library. Johnsen A., Ruud S., Arendal T.: Веб сайт. URL: https://github.com/nalexn/ViewInspector (дата звернення 21.04.2025).

11. Xcode Documentation. Apple Inc.: Веб сайт. URL: https://developer.apple.com/xcode/ (дата звернення 21.04.2025).

12. Combine Framework Reference. Apple Inc.: Веб сайт. URL: https://developer.apple.com/documentation/combine (дата звернення 21.04.2025).

13. Mobile Application Security: Best Practices for App Developers. OWASP Foundation: Веб сайт. URL: https://owasp.org/www-project-mobile-security/ (дата звернення 21.04.2025).

14. QR Code Applications in Secure Data Transmission. Denso Wave Incorporated: Веб сайт. URL: https://www.qrcode.com/en/about/ (дата звернення 21.04.2025).

15. SwiftUI by Tutorials: Declarative App Development with SwiftUI. Kodeco (formerly Ray Wenderlich): Веб сайт. URL: https://www.kodeco.com/books/swiftui-by-tutorials (дата звернення 21.04.2025).

16. Model-View-ViewModel (MVVM) Design Pattern for iOS Applications. John Sundell: Веб сайт. URL: https://www.swiftbysundell.com/articles/mvvm-in-swiftui/ (дата звернення 21.04.2025).

17. Easttom C. Modern Cryptography: Applied Mathematics for Encryption and Information Security. New York: McGraw-Hill Education, 2015. 416 с.

18. Dong L., Chen K. Cryptographic Protocol: Security Analysis Based on Trusted Freshness. Berlin: Springer, 2012. 250 с.

19. Practical Aspects of Modern Cryptography: підручник / Jean-Philippe Aumasson та ін. Seattle: University of Washington Press, 2006. 320 с.

20. Petrova K., Romaniello A., Medlin B., Vannoy S. QR Codes Advantages and Dangers. International Journal of e-Business and Telecommunications. 2016. Vol. 2. C. 112–115.

21. Scanning QR Codes with SwiftUI: A Step-by-Step Guide. Hacking with Swift: вебсайт. URL: https://www.hackingwithswift.com/books/ios-swiftui/scanning-qr-codes-with-swiftui (дата звернення 21.04.2025).

22. Apple Security Overview: Cryptographic Services for iOS Apps. Apple Developer: вебсайт. URL: https://developer.apple.com/security/ (дата звернення 21.04.2025).

23. Cryptographic Tools for iOS Development. Apple: вебсайт. URL: https://developer.apple.com/documentation/security (дата звернення 21.04.2025).

24. R Codes for Security: A Comprehensive Guide to Safe Scanning. Scanova: вебсайт. URL: https://scanova.io/blog/qr-codes-for-security/ (дата звернення 21.04.2025).

26. R Code Risks. Norton: вебсайт. URL: https://us.norton.com/blog/emerging-threats/qr-code-scams (дата звернення 21.04.2025).

27. End-to-End Encryption in Messaging Apps. Signal: вебсайт. URL: https://signal.org/en/docs/specs/protocol/ (дата звернення 21.04.2025).

28. Secure QR Code Usage in Mobile Apps. TechBit: вебсайт. URL: https://techbit.com/secure-qr-code-usage-mobile-apps/ (дата звернення 21.04.2025).\

29. Understanding Blowfish Encryption. Schneier: вебсайт. URL: https://www.schneier.com/academic/blowfish/ (дата звернення 21.04.2025).

ДОДАТОК А

ПРОГРАМНИЙ КОД МОБІЛЬНОГО ЗАСТОСУНКУ ШИФРУВАННЯ ТЕХНОЛОГІЙ

Програмні засоби:

- мова програмування – Swift;
- фреймворк для створення інтерфейсу користувача – SwiftUI;
- бібліотека для криптографічних операцій – CryptoSwift;
- фреймворк для генерації та зчитування QR-кодів – Core Image;
- фреймворки для автоматичного тестування – XCTes, ViewInspector.

```
import SwiftUI
import CoreImage
import UIKit
import CryptoSwift

// MARK: - Encryption Method Model
struct EncryptionMethod: Identifiable {
    let id = UUID()
    let name: EncryptionType
    let maxAllowedCharacters: Int

    enum EncryptionType: String {
        case aes, blowfish, chacha
    }
}

// MARK: - EnterMessageView
struct EnterMessageView: View {
    @StateObject var viewModel: EnterMessageViewModel
    @FocusState private var focusedField: Field?

    enum Field {
        case message, password
    }

    var body: some View {
        ScrollView {
            VStack(alignment: .leading, spacing: 24) {
                VStack(alignment: .leading, spacing: 8) {
                    Text("Enter Your Message")
                        .font(.title2)
                        .fontWeight(.bold)
                    Text("Encrypted with \(viewModel.encryptionMethod.name.rawValue)")
                        .font(.subheadline)
                        .foregroundColor(.secondary)
                }
                .padding(.horizontal)

                VStack(alignment: .leading, spacing: 8) {
                    HStack {
                        Label("Message", systemImage: "message.fill")
                            .font(.headline)
                        Spacer()
                        Text("\(viewModel.message.count)/\(viewModel.encryptionMethod.maxAllowedCharacters)")
                            .font(.caption)
                            .foregroundColor(viewModel.messageCountColor)
                    }

                    TextEditor(text: $viewModel.message)
                        .focused($focusedField, equals: .message)
```

```
                .padding(12)
                .frame(minHeight: 150)
                .overlay(
                    RoundedRectangle(cornerRadius: 12)
                        .stroke(viewModel.messageCountColor, lineWidth: 1.5)
                )
            }
            .padding(.horizontal)

            VStack(alignment: .leading, spacing: 8) {
                Label("Password", systemImage: "lock.fill")
                    .font(.headline)

                HStack {
                    if viewModel.showPassword {
                        TextField("Enter secure password", text: $viewModel.password)
                            .focused($focusedField, equals: .password)
                    } else {
                        SecureField("Enter secure password", text: $viewModel.password)
                            .focused($focusedField, equals: .password)
                    }
                    Button {
                        viewModel.showPassword.toggle()
                    } label {
                        Image(systemName: viewModel.showPassword ? "eye.slash.fill" : "eye.fill")
                    }
                }
                .padding(12)
                .overlay(
                    RoundedRectangle(cornerRadius: 12)
                        .stroke(viewModel.isPasswordValid || viewModel.password.isEmpty ? Color.blue : Color.red, lineWidth: 1.5)
                )
            }
            .padding(.horizontal)

            Button {
                viewModel.validateAndContinue()
            } label {
                Text("Generate QR Code")
                    .frame(maxWidth: .infinity)
                    .padding(.vertical, 16)
                    .background(viewModel.canContinue ? Color.blue : Color.gray.opacity(0.5))
                    .foregroundColor(.white)
                    .cornerRadius(16)
            }
            .padding(.horizontal)
            .disabled(!viewModel.canContinue)
        }
        .padding(.bottom, 20)
    }
    .navigationTitle("Secure Message")
    .background(
        NavigationLink(isActive: $viewModel.navigateToQRCode) {
            QRCodeDisplayView(viewModel: .init(
                password: viewModel.password,
                message: viewModel.message,
                encryptionMethod: viewModel.encryptionMethod)
            )
        } label: { EmptyView() }
    )
  }
}

// MARK: - EnterMessageViewModel
final class EnterMessageViewModel: ObservableObject {
    let encryptionMethod: EncryptionMethod
    @Published var message: String = ""
    @Published var password: String = ""
    @Published var showPassword: Bool = false
    @Published var navigateToQRCode: Bool = false

    init(encryptionMethod: EncryptionMethod) {
        self.encryptionMethod = encryptionMethod
    }

    var isMessageValid: Bool {
        !message.isEmpty && message.count <= encryptionMethod.maxAllowedCharacters
    }

    var isPasswordValid: Bool {
        password.count >= 4
```

```
    }

    var canContinue: Bool {
        isMessageValid && isPasswordValid
    }

    func validateAndContinue() {
        if canContinue {
            navigateToQRCode = true
        }
    }
}

// MARK: - QRCodeDisplayView
struct QRCodeDisplayView: View {
    @StateObject var viewModel: QRCodeViewModel

    var body: some View {
        VStack(spacing: 20) {
            Text("Your Encrypted QR Code")
                .font(.headline)
            if let image = viewModel.qrCodeImage {
                Image(uiImage: image)
                    .resizable()
                    .interpolation(.none)
                    .scaledToFit()
                    .padding()
            } else {
                ProgressView()
                    .padding()
            }
            Spacer()
        }
        .navigationTitle("QR Code")
    }
}

// MARK: - QRCodeViewModel
class QRCodeViewModel: ObservableObject {
    let password: String
    let message: String
    let encryptionMethod: EncryptionMethod
    @Published var qrCodeImage: UIImage? = nil
    @Published var encryptedMessage: String = ""

    private let context = CIContext()
    private let filter = CIFilter.qrCodeGenerator()

    init(password: String, message: String, encryptionMethod: EncryptionMethod) {
        self.password = password
        self.message = message
        self.encryptionMethod = encryptionMethod
        generateQRCode()
    }

    func generateQRCode() {
        guard let encrypted = encryptMessage() else { return }
        self.encryptedMessage = encrypted
        guard let data = encrypted.data(using: .utf8) else { return }
        filter.message = Data(data)
        filter.correctionLevel = "Q"

        if let outputImage = filter.outputImage {
            let transform = CGAffineTransform(scaleX: 10, y: 10)
            let scaledImage = outputImage.transformed(by: transform)
            if let cgImage = context.createCGImage(scaledImage, from: scaledImage.extent) {
                self.qrCodeImage = UIImage(cgImage: cgImage)
            }
        }
    }

    private func encryptMessage() -> String? {
        do {
            let messageBytes = Array(message.utf8)
            var encrypted: [UInt8] = []
            var iv: [UInt8] = []
            var key: [UInt8] = []

            switch encryptionMethod.name {
            case .aes:
                key = Array(password.utf8).sha256()
```

```
            iv = AES.randomIV(16)
            let aes = try AES(key: key, blockMode: CBC(iv: iv), padding: .pkcs7)
            encrypted = try aes.encrypt(messageBytes)
        case .blowfish:
            key = Array(Array(password.utf8).sha256().prefix(16))
            iv = AES.randomIV(16)
            let blowfish = try Blowfish(key: key, blockMode: CBC(iv: iv), padding: .pkcs7)
            encrypted = try blowfish.encrypt(messageBytes)
        case .chacha:
            key = Array(password.utf8).sha256()
            iv = ChaCha20.randomIV(12)
            let chacha = try ChaCha20(key: key, iv: iv)
            encrypted = try chacha.encrypt(messageBytes)
        }

        let combined = iv + encrypted
        let finalData = Data(combined)
        return finalData.base64EncodedString()
    } catch {
        print("Encryption error: \(error)")
        return nil
    }
  }
}

// MARK: - ImportOrScanQRCodeView
struct ImportOrScanQRCodeView: View {
  @StateObject private var viewModel = QRCodeInputViewModel()
  @State private var isShowingGallery = false
  @State private var isShowingCamera = false

  var body: some View {
    NavigationView {
      VStack(spacing: 32) {
        if let qrImage = viewModel.qrCodeImage {
          Image(uiImage: qrImage)
            .resizable()
            .interpolation(.none)
            .aspectRatio(contentMode: .fit)
            .frame(width: 200, height: 200)
            .clipShape(RoundedRectangle(cornerRadius: 12))
            .overlay(
              RoundedRectangle(cornerRadius: 12)
                .stroke(Color.blue, lineWidth: 2)
            )
        } else {
          RoundedRectangle(cornerRadius: 12)
            .stroke(Color.gray.opacity(0.3), style: StrokeStyle(lineWidth: 2, dash: [8]))
            .frame(width: 200, height: 200)
            .overlay(
              Text("QR Code Preview")
                .font(.caption)
                .foregroundColor(.gray)
            )
        }

        VStack(spacing: 16) {
          Button(action: { isShowingGallery = true }) {
            Text("Import from Gallery")
              .frame(maxWidth: .infinity)
              .padding(.vertical, 16)
              .background(Color.blue)
              .foregroundColor(.white)
              .cornerRadius(16)
          }
          .sheet(isPresented: $isShowingGallery) {
            ImagePicker(image: $viewModel.qrCodeImage, isPresented: $isShowingGallery, sourceType: .photoLibrary)
          }

          Button(action: { isShowingCamera = true }) {
            Text("Scan with Camera")
              .frame(maxWidth: .infinity)
              .padding(.vertical, 16)
              .background(Color.black)
              .foregroundColor(.white)
              .cornerRadius(16)
          }
          .sheet(isPresented: $isShowingCamera) {
            ImagePicker(image: $viewModel.qrCodeImage, isPresented: $isShowingCamera, sourceType: .camera)
          }
        }
```

```
            .padding(.horizontal, 24)

            NavigationLink(destination: DecryptQRCodeView(viewModel: .init(qrCodeImage: viewModel.qrCodeImage ?? UIImage()))) {
                Text("Continue")
                    .frame(maxWidth: .infinity)
                    .padding(.vertical, 18)
                    .background(viewModel.canContinue ? Color.blue : Color(.systemGray4))
                    .foregroundColor(.white)
                    .cornerRadius(16)
            }
            .disabled(!viewModel.canContinue)
            .padding(.horizontal, 24)
        }
        .navigationTitle("Import QR Code")
    }
  }
}

// MARK: - QRCodeInputViewModel
final class QRCodeInputViewModel: ObservableObject {
  @Published var qrCodeImage: UIImage? = nil {
    didSet {
      if qrCodeImage != nil {
        validateQRCode()
      }
    }
  }
  @Published var isValidQRCode: Bool = false

  var canContinue: Bool {
    qrCodeImage != nil && isValidQRCode
  }

  func validateQRCode() {
    guard let image = qrCodeImage, let ciImage = CIImage(image: image) else {
      isValidQRCode = false
      return
    }
    let detector = CIDetector(ofType: CIDetectorTypeQRCode, context: nil, options: [CIDetectorAccuracy: CIDetectorAccuracyHigh])
    let features = detector?.features(in: ciImage) ?? []
    isValidQRCode = !features.isEmpty
  }
}

// MARK: - DecryptQRCodeView
struct DecryptQRCodeView: View {
  @StateObject var viewModel: DecryptQRCodeViewModel
  @FocusState private var isPasswordFieldFocused: Bool

  var body: some View {
    ScrollView {
      VStack(alignment: .leading, spacing: 24) {
        HStack {
          Spacer()
          Image(uiImage: viewModel.qrCodeImage)
            .resizable()
            .aspectRatio(contentMode: .fit)
            .frame(width: 150, height: 150)
            .clipShape(RoundedRectangle(cornerRadius: 8))
          Spacer()
        }

        Text("Encryption Method")
          .font(.headline)
          .padding(.horizontal)

        VStack(spacing: 12) {
          ForEach(viewModel.methods) { method in
            Button(action: {
              viewModel.selectedMethod = method
            }) {
              Text(method.name.rawValue.uppercased())
                .padding()
                .frame(maxWidth: .infinity)
                .background(viewModel.selectedMethod?.id == method.id ? Color.blue : Color.gray.opacity(0.2))
                .foregroundColor(viewModel.selectedMethod?.id == method.id ? .white : .primary)
                .cornerRadius(8)
            }
          }
        }
        .padding(.horizontal)
```

```
            VStack(alignment: .leading, spacing: 8) {
                Label("Password", systemImage: "lock.fill")
                    .font(.headline)
                    .padding(.horizontal)

                HStack {
                    SecureField("Enter decryption password", text: $viewModel.password)
                        .focused($isPasswordFieldFocused)
                    Button(action: { viewModel.showPassword.toggle() }) {
                        Image(systemName: viewModel.showPassword ? "eye.slash.fill" : "eye.fill")
                    }
                }
                .padding(12)
                .overlay(
                    RoundedRectangle(cornerRadius: 12)
                        .stroke(isPasswordFieldFocused ? Color.blue : Color.gray.opacity(0.5), lineWidth: isPasswordFieldFocused ? 2 : 1)
                )
                .padding(.horizontal)
            }

            NavigationLink(destination: DecryptedQRCodeView(viewModel: .init(
                qrCodeImage: viewModel.qrCodeImage,
                encryptionMethod: viewModel.selectedMethod ?? viewModel.methods.first!,
                password: viewModel.password
            ))) {
                Text("Decrypt Message")
                    .frame(maxWidth: .infinity)
                    .padding(.vertical, 16)
                    .background(viewModel.canContinue ? Color.blue : Color(.systemGray4))
                    .foregroundColor(.white)
                    .cornerRadius(16)
            }
            .disabled(!viewModel.canContinue)
            .padding(.horizontal)
        }
    }
    .navigationTitle("Decrypt QR Code")
  }
}

// MARK: - DecryptQRCodeViewModel
final class DecryptQRCodeViewModel: ObservableObject {
  let qrCodeImage: UIImage
  @Published var methods: [EncryptionMethod] = [
    EncryptionMethod(name: .aes, maxAllowedCharacters: 200),
    EncryptionMethod(name: .blowfish, maxAllowedCharacters: 150),
    EncryptionMethod(name: .chacha, maxAllowedCharacters: 180)
  ]
  @Published var selectedMethod: EncryptionMethod? = nil
  @Published var password: String = ""
  @Published var showPassword: Bool = false

  var canContinue: Bool {
    selectedMethod != nil && !password.isEmpty
  }

  init(qrCodeImage: UIImage) {
    self.qrCodeImage = qrCodeImage
    selectedMethod = methods.first
  }
}

// MARK: - DecryptedQRCodeView
struct DecryptedQRCodeView: View {
  @StateObject var viewModel: DecryptedQRCodeViewModel
  @State private var showShareSheet = false

  var body: some View {
    ScrollView {
      VStack(spacing: 24) {
        Image(uiImage: viewModel.qrCodeImage)
          .resizable()
          .interpolation(.none)
          .scaledToFit()
          .frame(width: 200, height: 200)
          .clipShape(RoundedRectangle(cornerRadius: 12))

        Group {
          if viewModel.isLoading {
            ProgressView()
```

```
                    Text("Decrypting message...")
                        .font(.subheadline)
                        .foregroundColor(.secondary)
                } else if viewModel.decryptionFailed {
                    Text("Decryption failed. Check encryption method and password.")
                        .font(.subheadline)
                        .foregroundColor(.red)
                }
            }

            VStack(alignment: .leading, spacing: 8) {
                Label("Decrypted Message", systemImage: "doc.text")
                    .font(.headline)
                Text(viewModel.decryptedMessage)
                    .font(.body)
                    .padding(16)
                    .overlay(
                        RoundedRectangle(cornerRadius: 12)
                            .stroke(Color.gray.opacity(0.3), lineWidth: 1)
                    )
            }
            .padding(.horizontal)

            Button(action: { showShareSheet = true }) {
                Text("Share")
                    .frame(maxWidth: .infinity)
                    .padding(.vertical, 14)
                    .background(Color.blue)
                    .foregroundColor(.white)
                    .cornerRadius(12)
            }
            .sheet(isPresented: $showShareSheet) {
                ShareSheet(items: [viewModel.decryptedMessage])
            }
            .padding(.horizontal)
        }
    }
    .navigationTitle("Decrypted Message")
  }
}

// MARK: - DecryptedQRCodeViewModel
final class DecryptedQRCodeViewModel: ObservableObject {
    let qrCodeImage: UIImage
    let encryptionMethod: EncryptionMethod
    let password: String
    @Published var decryptedMessage: String = ""
    @Published var isLoading: Bool = true
    @Published var decryptionFailed: Bool = false

    init(qrCodeImage: UIImage, encryptionMethod: EncryptionMethod, password: String) {
        self.qrCodeImage = qrCodeImage
        self.encryptionMethod = encryptionMethod
        self.password = password
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.8) {
            self.decodeAndDecrypt()
        }
    }

    private func decodeQRCode() -> String? {
        guard let ciImage = CIImage(image: qrCodeImage) else { return nil }
        let detector = CIDetector(ofType: CIDetectorTypeQRCode, context: nil, options: [CIDetectorAccuracy: CIDetectorAccuracyHigh])
        let features = detector?.features(in: ciImage) as? [CIQRCodeFeature] ?? []
        return features.first?.messageString
    }

    func decodeAndDecrypt() {
        guard let encryptedMessage = decodeQRCode() else {
            DispatchQueue.main.async {
                self.isLoading = false
                self.decryptionFailed = true
                self.decryptedMessage = "No QR code message found."
            }
            return
        }

        if let message = decryptMessage(encrypted: encryptedMessage) {
            DispatchQueue.main.async {
                self.isLoading = false
                self.decryptedMessage = message
            }
```

```
            } else {
                DispatchQueue.main.async {
                    self.isLoading = false
                    self.decryptionFailed = true
                    self.decryptedMessage = "Unable to decrypt the message."
                }
            }
        }

        private func decryptMessage(encrypted: String) -> String? {
            guard let data = Data(base64Encoded: encrypted) else { return nil }
            let messageBytes = [UInt8](data)

            do {
                var iv: [UInt8] = []
                var encryptedBytes: [UInt8] = []
                var key: [UInt8] = []

                switch encryptionMethod.name {
                case .aes:
                    let ivLength = 16
                    guard messageBytes.count > ivLength else { return nil }
                    iv = Array(messageBytes.prefix(ivLength))
                    encryptedBytes = Array(messageBytes.dropFirst(ivLength))
                    key = Array(password.utf8).sha256()
                    let aes = try AES(key: key, blockMode: CBC(iv: iv), padding: .pkcs7)
                    let decrypted = try aes.decrypt(encryptedBytes)
                    return String(bytes: decrypted, encoding: .utf8)
                case .blowfish:
                    let ivLength = 16
                    guard messageBytes.count > ivLength else { return nil }
                    iv = Array(messageBytes.prefix(ivLength))
                    encryptedBytes = Array(messageBytes.dropFirst(ivLength))
                    key = Array(Array(password.utf8).sha256().prefix(16))
                    let blowfish = try Blowfish(key: key, blockMode: CBC(iv: iv), padding: .pkcs7)
                    let decrypted = try blowfish.decrypt(encryptedBytes)
                    return String(bytes: decrypted, encoding: .utf8)
                case .chacha:
                    let ivLength = 12
                    guard messageBytes.count > ivLength else { return nil }
                    iv = Array(messageBytes.prefix(ivLength))
                    encryptedBytes = Array(messageBytes.dropFirst(ivLength))
                    key = Array(password.utf8).sha256()
                    let chacha = try ChaCha20(key: key, iv: iv)
                    let decrypted = try chacha.decrypt(encryptedBytes)
                    return String(bytes: decrypted, encoding: .utf8)
                }
            } catch {
                print("Decryption error: \(error)")
                return nil
            }
        }
    }
```