Хмельницький національний університет
Факультет інформаційних технологій
Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

Тімоша Владислава Леонідовича

на здобуття ступеня вищої освіти Бакалавра

Система двофакторної аутентифікації користувачів мобільних пристроїв

Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека
Освітня програма Кібербезпека

Шифр КРБКБ.220254.22.02.35 ПЗ

Виконав студент 4 курсу група КБ-22-2 Владислав ТІМОШ
Керівник канд. техн. наук, доцент Віра ТІТОВА
Нормоконтролер д-р філософії Наталія ПЕТЛЯК

До захисту допускаю:
Завідувач кафедри кібербезпеки Юрій КЛЬОЦ

8 06 2026 р.

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Кібербезпеки

Рівень вищої освіти Бакалавр

Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека

Освітня програма Кібербезпека

ЗАТВЕРДЖУЮ

Завідувач кафедри кібербезпеки

Юрій КЛЬОЦ

9 січня 2026 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Тімошу Владиславу Леонідовичу

1 Тема роботи Система двофакторної аутентифікації користувачів мобільних пристроїв.

Керівник роботи канд. техн. наук, доцент, Віра Тітова

Затверджено наказом ректора університету від 8 січня 2026 р. № 7

2 Строк подання студентом кваліфікаційної роботи на кафедру 27 травня 2026р.

3 Вихідні дані до роботи Проаналізувати методи захисту мобільних систем та існуючі рішення двофакторної аутентифікації. Визначити вимоги та спроєктувати клієнт-серверну архітектуру комплексу. Обгрунтувати вибір криптографічних алгоритмів. Розробити серверну частину для перевірки одноразових паролів. Створити автономний мобільний застосунок під Android із захищеним апаратним зберіганням ключів. Провести тестування функціональності та безпеки розробленої системи.

4 Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Вступ. Аналіз методів забезпечення безпеки мобільних пристроїв та огляд рішень 2FA. Постановка задачі. Проєктування архітектури системи двофакторної аутентифікації та вибір алгоритмів захисту. Програмна реалізація серверної частини та мобільного клієнта. Тестування та оцінка ефективності роботи розробленої системи. Висновки.

5 Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Схема програмної моделі системи 2FA. UML-діаграма послідовності процесу авторизації. ER-схема реляційної бази даних.

6 Консультанти розділів кваліфікаційної роботи

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

7 Дата видачі завдання 12 січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

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

Студент _____ Владислав ТІМОШ

Керівник кваліфікаційної роботи _____ Віра ТІТОВА

АНОТАЦІЯ

Тема кваліфікаційної роботи: Система двофакторної аутентифікації користувачів мобільних пристроїв.

Автор роботи: Тімош Владислав Леонідович.

Керівник роботи: канд. техн. наук, доц. Тітова Віра Юріївна.

Пояснювальна записка: 67 сторінок, 3 додатки, 17 рисунків, 47 джерел.

Графічна частина: 3 плакати.

Ключові слова: автентифікація, безпека, двофакторна автентифікація, криптографія, мобільний додаток, система контролю доступу, aes, totp.

Кваліфікаційна робота присвячена дослідженню, проєктуванню та розробці системи двофакторної аутентифікації для підвищення рівня безпеки доступу до конфіденційних даних на мобільних пристроях. У роботі проаналізовано сучасні вектори кібератак на системи авторизації, вразливості традиційних паролів, методів перехоплення SMS та проблеми біометричної ідентифікації. Визначено критичну необхідність створення автономних ізольованих систем генерації паролів на основі часу.

У результаті розроблено та програмно реалізовано комплексну клієнт-серверну архітектуру. Серверна частина, побудована на базі FastAPI та SQLite, забезпечує криптографічний захист секретних ключів за допомогою алгоритмів AES-GCM та хешування паролів за стандартом Bcrypt. Клієнтська частина реалізована у вигляді мобільного застосунку для ОС Android, особливістю якого є повна мережева автономність та апаратний захист криптографічних ключів у довіреному середовищі виконання за допомогою Android Keystore System. Здійснено тестування стійкості системи до розсинхронізації часу та автоматизованих Brute-force атак.

01.06.2026

ABSTRACT

Theme of the qualification work: System of two-factor authentication for mobile device users.

Author of the work: Timosh Vladyslav Leonidovych

Supervisor: Ph.D., Assoc. Prof. Titova Vira Yuriivna.

Explanatory note: 67 pages, 3 appendices, 17 figures, 47 references.

Graphic part: 3 posters.

Keywords: authentication, security, two-factor authentication, cryptography, mobile application, access control system, aes, totp.

The bachelor's qualification thesis is devoted to the research, design, and development of a two-factor authentication system to increase the security level of access to confidential data on mobile devices. The study analyzes modern cyberattack vectors on authorization systems, vulnerabilities of traditional passwords, SMS interception methods, and problems of biometric identification. The critical need for creating autonomous, isolated time-based password generation systems is identified.

As a result, a comprehensive client-server architecture was developed and software-implemented. The server part, built on FastAPI and SQLite, provides cryptographic protection of secret keys using AES-GCM algorithms and password hashing according to the Bcrypt standard. The client part is implemented as a mobile application for Android OS, featuring full network autonomy and hardware protection of cryptographic keys within a Trusted Execution Environment using the Android Keystore System. Testing was conducted to evaluate the system's resilience to time desynchronization and automated Brute-force attacks.

01.06.2026 

ЗМІСТ

ВСТУП

Тотальна цифровізація призвела до того, що більшість повсякденних, фінансових та робочих процесів остаточно перейшли у смартфон. Сьогодні мобільний телефон – це головний інструмент для взаємодії з банківськими рахунками, доступу до корпоративних баз даних та зберігання особистого листування. Але саме через таку шалену концентрацію конфіденційної інформації в одному кишеньковому пристрої, смартфони стали мішенню номер один для зловмисників. Кібератаки постійно еволюціонують, тому забезпечення надійного захисту мобільних сервісів перетворилося на один із найголовніших викликів для сучасної IT-індустрії [1].

Класичні методи авторизації, які тримаються лише на зв’язці «логін-пароль», вже давно довели свою неефективність. Також зрозуміло, що більшості користувачів просто ліньки вигадувати і запам’ятовувати складні криптографічні паролі. Люди створюють примітивні комбінації і використовують їх повторно на десятках різних сайтів, що неминуче призводить до масових зломів облікових записів. Здавалося б, ситуацію має врятувати біометрія, але й вона має критичний недолік, адже якщо ваш звичайний пароль вкрали, ви можете його змінити за хвилину, а от замінити скомпрометований відбиток пальця чи скан обличчя вже майже неможливо.

Тому єдиним дійсно надійним інженерним рішенням залишається перехід на багатофакторні моделі перевірки [2]. Впровадження двофакторної аутентифікації створює той самий додатковий рівень безпеки, який зводить нанівець більшість віддалених атак. Розробка власної системи, яка б поєднувала високий рівень криптографічної стійкості та не дратувала користувача незручним інтерфейсом, є вкрай актуальним практичним завданням.

Мета кваліфікаційної роботи – це глибоке дослідження принципів роботи сучасних систем контролю доступу. Тому на основі цих досліджень буде спроєктована та програмна реалізація двофакторної аутентифікації для мобільних пристроїв. Для досягнення цієї мети в роботі послідовно вирішується комплекс завдань: від аналізу актуальних векторів загроз і порівняння існуючих факторів

перевірки до формування строгих архітектурних вимог та вибору математичних алгоритмів генерації часових кодів.

Завершальним етапом роботи є написання програмного коду для клієнтської та серверної частин системи, а також їх комплексне тестування. Практична цінність цього проєкту полягає в тому, що це не просто теоретичний опис відомих алгоритмів безпеки, а готове, працююче інженерне рішення. Створення власного аутентифікатора дозволяє на практиці зрозуміти, як криптографія працює «під капотом». Адже завдяки цьому формується глибоке розуміння того, як правильно захищати ключі в апаратній пам’яті смартфона та як налаштовувати безпечний обмін даними. Розроблений програмний модуль у майбутньому можна легко масштабувати або інтегрувати в реальні комерційні чи корпоративні продукти, де вимагається безкомпромісний рівень захисту облікових записів.

Більше того, варто звернути увагу на ще один критичний нюанс. Масове використання готових сторонніх додатків змушує компанії та їхніх співробітників беззаперечно довіряти зовнішнім вендорам. Для закритих корпоративних екосистем це часто є неприпустимим ризиком. Розробка ж власного, повністю ізольованого та автономного застосунку гарантує стовідсотковий контроль над інфраструктурою. Завдяки цьому система точно знатиме, де локально лежать секретні ключі, і гарантуватиме, що вони не синхронізуються з чужими хмарними серверами без відома адміністраторів.

Цей проєкт є ефективним способом закріплення здобутих за роки навчання знань. Це чудова можливість продемонструвати не просто вміння писати програмний код, а здатність мислити як інженер-архітектор: вміти передбачати можливі дії хакера, закривати логічні вразливості ще на етапі проєктування та створювати продукт, який реально працює і надійно захищає конфіденційні дані.

1 АНАЛІЗ ПРОБЛЕМИ ЗАХИСТУ ДОСТУПУ КОРИСТУВАЧІВ МОБІЛЬНИХ ПРИСТРОЇВ

1.1 Особливості забезпечення безпеки мобільних пристроїв та сервісів

Сучасні мобільні пристрої функціонують зовсім інакше, ніж традиційні стаціонарні комп'ютери, демонструючи новий тип кіберзагроз. Їх головною специфікою є постійна змінюваність умов використання. На відміну від серверів чи офісних комп'ютерів мобільні пристрої супроводжують власника всюди. Це спричиняє постійний ризик фізичної втрати чи викрадення.

Іншою суттєвою характеристикою є специфіка операційних систем мобільних пристроїв та спосіб поширення програмного забезпечення. У роботі мобільних платформ використовується ізоляція процесів, відома як «Sandboxing». Адже воно забезпечує відокремлення функцій різних додатків [3]. Проте цей механізм безпеки часто втрачає ефективність через взаємодію із самим користувачем. Встановлюючи програми з ненадійних джерел користувач у більшості випадків надає небезпечним утилітам надмірні дозволи, такі як доступ до повідомлень, контактів, файлової системи, відстеження геопозиції та надавання доступу до інших програм. Часто шкідливі програми маскуються під корисні застосунки, які непомітно збирають конфіденційну інформацію під час роботи у фоновому режимі.

Необхідно також враховувати мережеву динаміку та особливості передачі даних. Мобільні сервіси створені для постійного та безперебійного зв'язку, тому пристрої автоматично та безперервно перемикаються між різними базовими станціями стільникових операторів і випадковими бездротовими мережами. Водночас відбувається постійна фонова синхронізація, а саме месенджери, банківські програми, поштові клієнти та соціальні мережі регулярно обмінюються токенами з серверами. Такий безперервний і часто неконтрольований користувачем обмін даними робить смартфон вразливою мішенню для атак, які спрямовані на перехоплення незашифрованого трафіку або підміну комунікаційних вузлів [4].

Особливо важливими є ергономічні та психологічні аспекти взаємодії з

мобільними сервісами. Швидкий ритм життя змушує людей шукати найефективніші способи досягнення бажаного результату. Процес тривалого введення складних паролів на невеликій клавіатурі викликає дискомфорт, тому користувачі часто спрощують бар'єри безпеки. Одним із найрозповсюдженіших типів паролів є введення дати свого народження або якоїсь близької людини і це дуже небезпечно адже зловмисники за лічені хвилини можуть знайти всю інформацію, яка їм потрібна про власника девайсу. Також вони встановлюють короткі пін-коди, активно використовують функцію автозаповнення або іноді взагалі відмовляються від базового блокування екрана.

Враховуючи всі ці фактори від високої ймовірності фізичної втрати пристрою до специфіки дозволів мобільних операційних систем і звичок користувачів – стає зрозуміло, що традиційний підхід до захисту інформації є недостатнім. Забезпечення безпеки мобільних сервісів вимагає створення комплексних механізмів, які не лише надійно перевіряють права доступу, але й компенсують нестабільність фізичного середовища та мінімізують ризики, пов'язані з людською недбалістю.

Для кращого аналізу загроз, що впливають на сучасні мобільні пристрої, експерти з кібербезпеки спираються на міжнародні стандарти та нові дослідницькі матеріали. Одним із найбільш авторитетних джерел у цьому напрямку є список глобальних вразливостей OWASP Mobile Top 10. Він був створений у 2014 році міжнародною організацією Open Worldwide Application Security Project [5]. Цей документ детально описує найпоширеніші шляхи атак на мобільні програми, звертаючи увагу на тому, що захист не може обмежуватися лише елементарною перевіркою пароля на сервері.

Одна з найсерйозніших та найбільш розповсюдженіших проблем, які були виявлені у класифікації OWASP – це незахищене зберігання даних. Під час розробки мобільних додатків часто хибно припускають уявлення про файлову систему самого смартфону як про безпечне середовище для зберігання даних. У результаті конфіденційні дані, а саме такі як сесійні токени, локальні паролі, записи історії транзакцій і навіть криптографічні ключі, можуть зберігатися в незашифрованому вигляді у базах даних SQLite, конфігураційних файлах XML чи

спільних теках пам’яті. Якщо все ж таки зловмисник таки отримав фізичний доступ до розблокованого пристрою або використав зловмисне програмне забезпечення для привласнення привілеїв, то він має можливість безперешкодно скопіювати та зберегти подібну інформацію. У таких ситуаціях відсутність додаткових механізмів аутентифікації може дозволити атакуючому легко дублювати активну сесію легітимного користувача на власний пристрій [6].

Однією з важливих загроз залишається небезпека передача даних через мережу. Не зважаючи на те, що більшість сучасних сервісів за замовчуванням використовують протокол HTTPS, його базова реалізація у мобільних додатках часто має архітектурні слабкі місця. У більшості випадків зловмисники використовують розповсюджену атаку типу «людина посередині» [7]. Вони створюють фальшиві точки доступу Wi-Fi у громадських місцях або компрометують маршрутизатори. У випадках, коли мобільний додаток не реалізує механізм жорсткого закріплення сертифікатів, то хакер може замінити цифровий сертифікат сервера та направити весь трафік пристрою через власний контрольований вузол. І тому це відкриває можливість для перехоплення логінів, паролів та іншої конфіденційної інформації у відкритому тексті прямо в режимі реального часу.

Дослідження сучасних загроз інформаційної безпеки демонструють, що значну частину успішних кібератак спричиняють вразливості, пов’язані з недостатньою або некоректно реалізованою процедурою аутентифікації (Insecure Authentication). У контексті мобільних додатків проєктування часто орієнтується на забезпечення тривалості авторизованих сесій користувачів. Це було створено аби мінімізувати необхідність повторного введення пароля під час переходів між активними та фоновими станами програми. Але механізми перевірки терміну дії токенів нерідко характеризуються недоліками або помилками в реалізації. Тим паче багато додатків здійснюють перевірку лише локально на пристрої, без належної синхронізації з серверною інфраструктурою. В таких умовах обхід захисту локального блокування екрану смартфона може автоматично надати ворожим стороннім особам доступ до привілейованих функцій додатка.

Окрему і не мало важливу увагу слід приділити ризикам, які пов’язані із

реверс-інжинірингом та прямим втручанням у програмний код додатка. Також варто зазначити, що мобільні додатки є скомпільованими архівами, які після завантаження розміщуються безпосередньо на пристрої кінцевого користувача. Зловмисники можуть застосовувати спеціалізовані засоби, такі як декомпілятори та дизасемблери, для перетворення програми у вихідний код, аналізу її логічного функціонування і виявлення захищених алгоритмів шифрування чи жорстко закодованих ключів доступу до серверних API. Після отримання такої інформації існує великий ризик, що буде застосовано несанкціонована модифікація програмного коду із наступних шляхів, а саме примусове відключення вбудованих засобів захисту, перепакування додатка та його розповсюдження через сторонні платформи під виглядом офіційного оновлення або нової модифікації.

Тому такий аналіз зазначених мобільних загроз демонструє, що сучасні кіберзлочинці застосовують багатогранний підхід, водночас націлюючись на різні аспекти системи безпеки. Їхні дії охоплюють широкий спектр методів, по-типу перехоплення мережевого трафіку, фізичне отримання даних із мікросхем пам’яті пристрою та декомпіляцію програмного забезпечення. З огляду на це все, можна зрозуміти, що використання лише одного традиційного засобу захисту (логін + пароль) стає явно недостатньо. Це підтверджує важливість розробки архітектури безпеки мобільних сервісів на основі принципів багаторівневого захисту. Також, навіть у ситуації компрометації локальної бази даних пристрою або успішного перехоплення статичного пароля, додаткові механізми динамічної аутентифікації можуть виконувати роль критичного бар’єру, що ефективно стримує подальше просування атаки та забезпечує збереження конфіденційності даних.

1.2 Методи аутентифікації користувачів та їх порівняльний аналіз

Розв’язання проблеми захисту доступу до мобільних пристроїв і сервісів є неможливим без розуміння основних принципів перевірки користувачів. У теорії інформаційної безпеки процес надання доступу включає три послідовні етапи:

– ідентифікацію, коли користувач повідомляє про себе, вводячи логін;

– аутентифікацію, перевірка достовірності цієї інформації;

– авторизацію, надання певних прав доступу.

Щодо етапу аутентифікації то він є найбільш вразливим тому, що вимагає компромісу між надійністю криптографічного захисту та зручністю для користувача. Всі існуючі механізми перевірки особи можна класифікувати за трьома основними критеріями, які в літературі називають «факторами аутентифікації». До них відносяться такі критерії як:

– фактор знання – це інформація, відома лише легітимному користувачу;

– фактор володіння – це фізичний об’єкт, що знаходиться у користувача;

– фактор властивості – це унікальні біометричні характеристики особи.

Кожен з цих підходів має свої етапи розвитку, а також специфічні переваги і недоліки в мобільному середовищі. Найбільш поширеним та історично першим підходом до забезпечення інформаційної безпеки є фактора знання. До цієї категорії належать класичні буквено-цифрові паролі, PIN-коди та графічні ключі, які здобули популярність завдяки операційній системі Android, адже саме у жовтні 2008 року вперше було представлено графічний ключ компанією Google при релізі ОС Android 1.0 [8]. З технічної точки зору даний підхід вирізняється простотою програмної реалізації та відсутністю необхідності впровадження додаткового апаратного забезпечення. Але практичне застосування методів парольного захисту в контексті мобільних пристроїв виявляє значні труднощі, аргументовано це особливостями людської психології та ергономічними обмеженнями.

Введення складного криптографічно стійкого пароля, що містить великі і малі літери, цифри та спеціальні символи на телефоні є незручним процесом. Тому багато користувачів обирають створення простих комбінацій або використовують один і той же пароль для численних сервісів. Це відкриває значні можливості для зловмисників, які можуть проводити словникові атаки або використовувати вже скомпрометовані бази даних з інших джерел. PIN-коди та графічні ключі також мають свої вразливості. До прикладу, дослідження показують, що більшість графічних ключів створюються за передбачуваними шаблонами, такими як початок з верхнього лівого кута та схожість на літери алфавіту. Крім того, існує специфічна загроза для смартфонів – так звана «Smudge attack», коли зловмисник може

відновити правильний графічний ключ або PIN-код, спостерігаючи за жировими слідами на екрані [9].

Наступною значною групою є фактор володіння. Цей підхід вимагає, щоб для успішного входу користувач пред'явив певний унікальний предмет. У корпоративному секторі часто використовують фізичні смарт-картки, USB-токени або автономні генератори одноразових паролів у вигляді брелків. У мобільних додатках роль такого фактора часто виконує SIM-карта або сам телефон, на якому встановлено спеціальний додаток-аутентифікатор. Фактор володіння значно підвищує стійкість системи до віддалених кібератак. Навіть якщо хакер дізнається пароль, він не зможе отримати доступ до системи без фізичного ключа або смартфона жертви. Проте цей метод має свої недоліки. Апаратні токени можуть бути незручними для щоденного використання і потребують додаткових фінансових витрат для їх придбання та обслуговування. Щодо використання мобільного телефону як фактора володіння, виникає проблема залежності від заряду акумулятора та якості покриття мобільної мережі. Тому, що смартфон є портативним пристроєм, ризик його крадіжки чи втрати разом з усіма прив'язаними токенами залишається високим.

Проте останнім часом, завдяки впровадженню високоточних сенсорів у споживчу електроніку, значно зросла популярність третього фактора – біометричної аутентифікації [10]. Технології, такі як розпізнавання відбитків пальців, геометрії обличчя та сканування сітківки ока, забезпечують користувачам майже ідеальний рівень зручності. Тепер людям не потрібно запам'ятовувати паролі чи носити додаткові предмети, а процес підтвердження особи займає всього кілька секунд і є інтуїтивно зрозумілим. На даний момент майже у кожному телефоні є функція біометрики. Але, незважаючи на високу ергономічність, біометрія має критичні концептуальні вразливості. Головна проблема полягає в незмінності біометричних характеристик. Якщо традиційний пароль можна змінити за кілька хвилин після компрометації, то змінити відбиток пальця або структуру обличчя неможливо. У разі викрадення бази цифрових зліпків біометрії з сервера, користувач назавжди втратить можливість безпечно використовувати дані. Надійність біометричних систем зазвичай оцінюється двома показниками:

– FAR – коефіцієнт помилкового пропуску сторонньої особи;

– FRR – коефіцієнт помилкової відмови легітимному користувачеві [11].

Щоб система залишалася зручною і не викликала незадоволення через часті відмови при вологих руках або поганому освітленні, виробники часто знижують чутливість датчиків, що збільшує ризик успішного обману системи за допомогою якісних муляжів або фотографій з високою роздільною здатністю.

У контексті мобільних пристроїв також варто виділити відносно новий напрямок – поведінкову біометрію [12]. Сучасні смартфони оснащені гіроскопами, акселерометрами та чутливими сенсорними екранами, що дозволяє аналізувати не лише те, ким є людина, а й те, як вона взаємодіє з пристроєм. Аналізуються швидкість друку, сила натискання на екран, кут нахилу телефону під час роботи та характерні свайпи. Перевагою цього підходу є можливість безперервної фонової аутентифікації без активних дій з боку користувача. Однак на даному етапі розвитку технологій поведінкова біометрія потребує значних обчислювальних ресурсів, що швидко розряджає акумулятор мобільного пристрою, а також має високий відсоток помилкових спрацьовувань при зміні фізичного стану користувача.

Для повноцінного аналізу надійності біометричних методів необхідно вивчити їхню архітектуру та апаратну складову для реалізації у смартфонах. Рівень безпеки цьому типі технології залежить не лише від якості сканера, але й від того, як операційна система обробляє та зберігає дані. У теперішніх мобільних пристроях використовують для цього спеціалізовані апаратні платформи, а саме технологію ARM TrustZone в пристроях на базі Android,, що створює довірене середовище виконання [13] та мікропроцесор Secure Enclave у пристроях від компанії Apple [14]. Основний принцип функціонування цих архітектур полягає у забезпеченні суворої фізичної та логічної ізоляції. Операційна система смартфона, яка відповідає за запуск усіх користувацьких додатків, включно з банківськими застосунками, не має прямого доступу до біометричної інформації. Саме у процесі реєстрації відбитка пальця або обличчя користувача створюється математичний шаблон, який шифрується у за допомогою унікального апаратного ключа та зберігається лише в ізольованій пам’яті TEE або Secure Enclave. Навіть у разі

повного компрометування основної операційної системи отримати ці біометричні дані в нерозшифрованому вигляді неможливо.

Сама процедура аутентифікації в застосунку проходить за таким алгоритмом. Спочатку застосунок звертається до системного API, наприклад як BiometricPrompt на Android, для запиту підтвердження особи користувача [15]. Далі сканер зчитує біометричні дані, які одразу пересилаються у захищене середовище, минаючи основну операційну систему. У цьому захищеному середовищі проводяться порівняння нових даних із попередньо збереженим шаблоном. Якщо результати таки співпадають то TEE не передає застосунку самі біометричні дані, а створює криптографічний токен або цифровий підпис. Цей токен, що підтверджує успішне проходження локальної аутентифікації, потім надходить до основної операційної системи.

Проте основна вразливість локальної біометрії при використанні для віддаленого доступу до серверів прихована саме у механізмі передачі токена. Серверна сторона додатка змушена покластись на результат перевірок, які виконуються на клієнтському девайсі. Тому це відкриває можливість зловмисникам використовувати методи обходу системних API, зокрема Biometric API Hooking. На пристроях із підвищеними привілеями хакери можуть встановлювати спеціальні фреймворки, які здатні перехоплювати звернення додатка до системи. Такими фреймворками можуть бути такі як Frida або Xposed [16]. У разі запиту програми на біометричну ідентифікацію шкідливий код може зімітувати позитивну відповідь системи й передати цей хибний сигнал про успішну авторизацію, навіть якщо сканер фактично не використовувався.

Нажаль використання локальної біометрії має суттєвий недолік, який пов’язаний із ризиком делегування довіри. До прикладу, якщо зловмисник отримає доступ до PIN-COD для розблокування викраденого девайсу, то мобільна операційна система дозволити йому додати власний відбиток пальця у налаштуваннях пристрою. У результаті всі додатки, що спираються виключно на локальну біометрію, автоматично вважатимуть його законним власником.

Ці архітектурні обмеження свідчать про те, що локальна біометрія, навіть попри високий рівень зручності та апаратний захист шаблонів, не може бути також

єдиним засобом забезпечення безпеки, особливо в контексті критично важливих сервісів. Для справжнього захисту від компрометації системи необхідно використовувати фактори, які створюються на пристрої, але перевіряються за допомогою криптографічних методів на віддаленому сервері. Це унеможливлює втручання зловмисників у виконання коду. Такий підхід підкреслює потребу у впровадженні механізмів генерації одноразових часових паролів, а саме ТОТР, який доповнює локальні методи захисту.

Комплексний аналіз досліджених методів показує, що вибір певного підходу завжди пов'язаний із необхідністю знаходження оптимального балансу між криптографічною стійкістю, витратами на впровадження та зручністю для користувачів. Традиційні системи авторизації, побудовані на використанні паролів, хоча й залишаються найдоступнішими з економічної точки зору, мають найнижчий рівень захисту від атак на основі соціальної інженерії та помилок користувачів. Біометричні методи забезпечують високий рівень зручності та локальної безпеки, але створюють значні ризики у випадках віддаленої автентифікації, оскільки компрометовані біометричні дані не підлягають заміні. Застосування фізичних носіїв або апаратних рішень для автентифікації ефективно протидіє дистанційним кібератакам, проте їх масове впровадження ускладнюється логістичними проблемами і зниженням загального рівня зручності використання системи.

1.3 Двофакторна аутентифікація як засіб підвищення рівня захисту

Як показав порівняльний аналіз основних методів перевірки користувачів, ізольоване використання будь-якого з існуючих факторів аутентифікації залишає систему вразливою до специфічних атак. Статичні паролі легко викрадаються або підбираються, апаратні носії можуть губитися, а біометричні дані можуть бути скомпрометовані без можливості їх відновлення. Розуміння цих фундаментальних недоліків призвело до розвитку систем контролю доступу та формування концепції багатофакторної аутентифікації, найпоширенішим варіантом якої є двофакторна аутентифікація (2FA).

Основний принцип двофакторної аутентифікації полягає в створенні додаткового та незалежного шару захисту шляхом комбінування двох різних факторів перевірки. Найчастіше у житті використовують поєднання фактора знання з фактором володіння. Головна ідея цього підходу ґрунтується на математичній ймовірності: шанси зловмисника одночасно дізнатися секретний пароль користувача та фізично отримати його особистий пристрій під час входу є вкрай мізерними. Впровадження другого фактора кардинально змінює модель загроз. Якщо раніше витік бази даних із хешами паролів або успішна фішингова атака означали повну втрату контролю над обліковим записом, то за наявності 2FA ці загрози частково нівелюються. Навіть якщо зловмисник отримує правильну комбінацію логіна та пароля, система не надасть йому доступу, бо вимагатиме введення тимчасового коду підтвердження або виконання дії на довіреному пристрої. Це робить масові автоматизовані атаки, такі як підстановка вкрадених облікових даних, економічно невигідними та технічно неможливими для реалізації в масштабах.

Проте важливо розуміти, що не всі методи реалізації двофакторної аутентифікації забезпечують однаковий рівень безпеки. Історично першим і досі найпоширенішим способом доставки другого фактора стали SMS-повідомлення з одноразовими паролями (OTP). Цей метод здобув популярність завдяки простоті використання: користувачу не потрібно встановлювати додаткове програмне забезпечення, достатньо лише мати мобільний зв’язок. Але з розвитком кіберзлочинності SMS-аутентифікація виявила свою критичну вразливість. Передача кодів у відкритому тексті через стільникові мережі робить їх легкою здобиччю для атак, пов’язаних із вразливостями протоколу SS7 [17]. Також поширилася соціальна інженерія у вигляді підміни SIM-карт – це SIM-swapping, коли шахраї обманом змушують оператора перевипустити картку жертви на свій носій і водночас перехоплюючи всі повідомлення [18]. Через ці ризики міжнародні організації стандартизації, такі як NIST, визнали SMS-канали небезпечними для передачі авторизаційних даних.

Більш надійним і сучасним підходом є використання Push-сповіщень. У цьому випадку на довірений пристрій користувача надходить інтерактивне

повідомлення з проханням підтвердити або відхилити спробу входу. Цей метод забезпечує відмінний користувацький досвід, оскільки зводить дії до одного натискання на екрані. Канал зв'язку при цьому є зашифрованим і не підлягає вразливостям стільникових мереж. Однак цей метод сильно залежить від стабільного інтернет-з'єднання на пристрої. Крім того, зловмисники адаптувалися і до цього захисту, використовуючи так звану атаку «MFA Fatigue» [19]. Хакер генерує десятки запитів на вхід поспіль, сподіваючись, що роздратований або неуважний користувач врешті-решт випадково натисне кнопку «Підтвердити», щоб зупинити потік сповіщень на своєму смартфоні.

Найбільш ефективним рішенням з точки зору криптографічної стійкості, автономності та витрат на інтеграцію є алгоритми генерації одноразових паролів на основі часу (TOTP). У цьому випадку мобільний додаток-аутентифікатор і сервер мають спільний секретний ключ, за допомогою якого кожні 30 секунд генерується новий шестизначний код. Перевага такої архітектури в тому, що телефон користувача не потребує підключення до Інтернету або мобільного зв'язку для створення коду. Увесь процес відбувається офлайн у межах ізольованого середовища додатка, що унеможливлює перехоплення коду, оскільки він не передається нікуди.

Щоб краще зрозуміти та дізнатись, чому цей метод забезпечує такий високий рівень надійності, потрібно розглянути сам алгоритм Time-Based One-Time Password з технічної точки зору. Цей протокол був стандартизований Інженерною радою Інтернету (IETF) у документі RFC 6238 [20] і по суті є вдосконаленою версією старішого алгоритму HOTP (RFC 4226) [21]. Різниця між цими алгоритмами заключаєтсья в тому, що старий алгоритм спирався на лічильник подій, який дуже часто розсинхронізовувався. TOTP вирішує цю проблему значно ефективніше: замість лічильника він використовує поточний час. Щоб мобільний додаток і сервер могли синхронно згенерували однаковий пароль то їм потрібно дві складові. Перша – це спільний секретний ключ, який створюється сервером під час налаштування 2FA і надійно зберігається на обох пристроях. Друга – це поточний час у форматі Unix Time [22]. Оскільки час іде безперервно, а системі потрібен код, який буде працювати певний проміжок то цей час перетворюють на цілочисельний

лічильник кроків за такою формулою:

$$T = \frac{t - T_0}{X}, \quad (1.1)$$

де t – поточний системний час; T_0 – точка відліку (зазвичай приймається за нуль); X – тривалість самого часового вікна (стандартно 30 секунд). Результат ділення просто округлюється в меншу сторону. Завдяки цьому цілих 30 секунд значення лічильника T не змінюється, і користувач спокійно встигає ввести код на сайті.

Далі система бере цей часовий лічильник і секретний ключ та криптографічно їх змішує. Для цього використовується такий алгоритм як HMAC [23]. Програма виконує серію математичних операцій, зокрема побітове XOR і на виході видає криптографічний хеш. У випадку з алгоритмом SHA-1 це 20-байтовий масив. І так зрозуміло, що змушувати користувача переписувати 20 байт символів з екрана смартфона нераціонально. Тому стандарт RFC 4226 описує спеціальний алгоритм динамічного усічення. Він акуратно вирізає цей довгий хеш до звичних шести цифр. Працює це в кілька етапів: спочатку з усього масиву хешу за допомогою бітової маски визначається індекс зміщення. Починаючи від цього індексу, програма відрізає фрагмент довжиною 4 байти. Старший біт цього шматка примусово зануляють, щоб уникнути помилок при обробці знакових чисел на різних архітектурах процесорів. У результаті залишається 31-бітне ціле число. Щоб отримати з нього класичний код, до цього числа застосовують математичну операцію ділення по модулю:

$$TOTP = (BinaryCode)\ (mod\ 10^6). \quad (1.2)$$

Якщо ж після ділення результат виходить коротшим за шість цифр, програма просто дописує необхідну кількість нулів на початку.

Щодо практичної сторони інтеграції, то найцікавішим є момент першого налаштування. Щоб користувачу не доводилось вводити свій секретний ключ вручну, його кодують у формат Base32 і генерують з нього QR-код [24]. Цей код

містить звичайний рядок URI. Камера телефону його сканує, і додаток-аутентифікатор миттєво імпортує всі налаштування. Окрема інженерна проблема, з якою доводиться стикатися під час розробки таких систем – це розсинхронізація часу. Навіть попри те, що годинник на телефоні відстає на декілька секунд то код згенерується неправильно, тому і сервер його відхилить. Для вирішення цієї проблеми розробники використовують на сервері механізм Look-ahead window. Бекенд обчислює і перевіряє не лише поточний код для кроку T, але й коди для частин T - 1 та T + 1. Це допомагає компенсувати мережеві затримки або дрібні похибки системного часу смартфонів, роблячи роботу системи стабільною.

Passkeys – це інноваційна технологія входу, яка дала великий крок уперед в розвитку багатофакторної аутентифікації. Ця система базується на WebAuthn та FIDO2. Apple, Google та Microsoft дають велику підтримку саме цій системі і на теперішній час вважається, що Passkeys є прогресивніша альтернатива традиційним паролям та навіть TOTP-додаткам. Ідея полягає в тому, що введення пароля пристрій користувача генерує унікальну пару криптографічних ключів, а саме публічний ключ, який зберігається на сервері, та приватний ключ, який лишається на самому пристрої та захищається біометричними даними. На перший погляд, ця схема зможе повністю замінити потребу у більшості додатках захисту, але після детального вивчення архітектури цієї системи можна зрозуміти, що є досить велика низка обмежень, які можуть виникнути у корпоративних чи державних середовищах. Проблема заключаєтсья в тому, що механізм синхронізації приватних ключів за замовчуванням синхронізує через хмарні сервіси. Тобто, це означає, що доступ до приватних ключів виходять за межі фізичного пристрою користувача. Для таких організацій така модель не відповідає дійсності щодо забезпечення конфіденційності даних. Аналіз показує, що автономні генератори одноразових паролів на основі алгоритму TOTP залишається досі найбільш надійним способом безпеки у двофакторній аутентифікації. Адже він забезпечує ізоляцію ключів та унеможливлює їх спотворення або копіювання через сторонні сервіси.

Виходячи з вище написаного можна зрозуміти, що двофакторна аутентифікація є не просто додатковою опцією, а критично важливим

інструментом для протидії сучасним загрозам інформаційній безпеці. Вона дозволяє перервати ланцюг атаки ще до того, як зловмисник отримає доступ до конфіденційних даних. Враховуючи архітектурні обмеження мобільних операційних систем і постійне зростання обсягів чутливої інформації на смартфонах, відмова від статичних паролів на користь динамічної двофакторної перевірки є єдиним дієвим шляхом до створення надійної системи захисту. Впровадження таких архітектурних рішень дозволяє мінімізувати ризики перехоплення конфіденційних даних та гарантує стабільний захист облікових записів у будь-яких умовах.

1.4 Постановка задачі

На основі проведеного аналізу в попередніх розділах можна зробити висновок, що сучасна екосистема мобільних додатків вимагає нових підходів до забезпечення інформаційної безпеки. Швидке зростання обсягів конфіденційних даних, які обробляються на смартфонах, у поєднанні з вразливістю традиційних систем паролів та ризиками фізичної втрати пристроїв роблять використання однофакторної аутентифікації критично недостатнім. Інтеграція додаткових рівнів безпеки часто призводить до погіршення користувацького досвіду, що вимагає пошуку оптимального балансу між криптографічною стійкістю та зручністю мобільного інтерфейсу.

Більшість наявних рішень, до прикладу підтвердження через SMS або Push-сповіщення, вже не так відповідають сучасним стандартам безпеки через вразливість стільникових мереж і ризики атак соціальної інженерії. У зв’язку з цим зростає інтерес до автономних алгоритмів генерації одноразових паролів на основі часу. Якщо зловмисник отримає доступ до розблокованого девайсу то він зможе з легкістю скористатись додатком-аутентифікатором і отримати доступ до паролів. Така ситуаціє точно дає знати про те, що важливість розробки нового інженерного рішення, яка б забезпечувала захист процесу генерації кодів за допомогою

апаратної біометрії девайсу. Це дозволить створити гібридну трифакторну модель аутентифікації, що суттєво підвищить рівень безпеки.

Враховуючи все те, що було вище описано, метою цієї кваліфікаційної роботи є дослідження, проєктування та програмна реалізація системи двофакторної аутентифікації для користувачів мобільних пристроїв, яка базуватиметься на алгоритмах генерації одноразових паролів на основі часу і забезпечуватиме надійний захист облікових записів від несанкціонованого доступу.

Для досягнення поставленої мети та успішної реалізації розроблюваної системи необхідно вирішити такий комплекс взаємопов’язаних завдань:

– сформувати чіткі технічні та ергономічні вимоги до системи безпеки, враховуючи особливості взаємодії користувача з сенсорним екраном мобільного пристрою;

– розробити логічну архітектуру взаємодії між клієнтським мобільним додатком і серверною частиною, визначивши безпечні протоколи для передачі даних;

– здійснити обґрунтований вибір алгоритмів для генерації одноразових кодів та механізмів безпечного локального зберігання криптографічних ключів на рівні мобільної операційної системи;

– здійснити практичну програмну реалізацію спроектованих модулів та інтегрувати механізм двофакторної аутентифікації в тестовий мобільний застосунок;

– виконати комплексне тестування розробленої системи для оцінки її продуктивності, стійкості до потенційних атак та зручності для кінцевих користувачів.

Розв’язання зазначених завдань дозволить створити надійний, автономний і масштабований модуль безпеки, який можна інтегрувати в сучасні мобільні продукти для захисту конфіденційної інформації та підвищення загального рівня довіри користувачів до цифрових сервісів. Поміж цього саме відмова від прив’язки до сторонніх застосунків забезпечить стовідсотковий контроль щодо зберігання секретних ключів. Такий підхід повністю виключає ризики прихованої хмарної синхронізації даних на сервери третіх осіб.

2 ПРОЄКТУВАННЯ СИСТЕМИ ДВОФАКТОРНОЇ АУТЕНТИФІКАЦІЇ

2.1 Формування вимог до системи

Розробка будь-якого програмного забезпечення, а тим більше системи, яка безпосередньо розроблена для захисту конфіденційних даних, завжди повинна починатися з чіткого формулювання вимог. На основі аналізу, який проведено у першому розділі, стає очевидним, що використання лише готових рішень не завжди дозволяє потреби гнучкої інтеграції. Тому виникає необхідність створення власного програмного комплексу. Для того, щоб розробка була послідовною, а кінцевий результат працював без збоїв, то неодмінно потрібно детально описати всі функціональні, нефункціональні і безпекові вимоги до системи. З архітектурної точки зору розроблювана система буде розділена на два незалежні модулі:

– серверна частина, яка імітує сервіс, що потребує захисту;

– клієнтська частина, яка є самим мобільним додатком аутентифікатором.

В такому випадку вимоги до кожного з цих модулів варто формувати окремо, але з урахуванням їхньої взаємодії.

Спочатку необхідно визначити базові функціональні вимоги до серверної частини спроєктованої системи. Головним завданням сервера є керування життєвим циклом самої двофакторної аутентифікації для кожного користувача. Наприклад, коли людина вирішує захистити свій обліковий запис та вмикає 2FA, то сервер повинен вміти створити для неї унікальний криптографічний секретний ключ. Цей ключ є початком для усієї подальшої роботи, тому система вимагає, щоб він створювався за допомогою надійних генераторів випадкових чисел. Після того, як було згенеровано ключ, серверна частина повинна сформувати з нього спеціальний стандартизований рядок у форматі URI, в якому буде зашифровано сам секрет, назва цільового сервісу та електронна пошта користувача. Також не мало важливим є факт того, що користувачеві було зручно перенести цей ключ на свій девайс, тому сервер має автоматично перетворити цей рядок на графічний QR-код і вивести його на екран.

Ще однією критично важливою функціональною вимогою до сервера є здатність правильно перевіряти ті коди, які буде вводити сам користувач під час

спроби входу. Сервер не просто повинен звірити шість цифр, а має самостійно в режимі реального часу взяти секретний ключ цього користувача зі своєї бази даних, взяти поточний системний час та провести обчислення за алгоритмом TOTP. Також видається вимога до реалізації так званого «вікна допуску». Тобто сервер повинен враховувати можливі затримки при передачі даних через інтернет або невеликі відхилення годинника та телефоні у людини. Ось чому система повинна успішно приймати код, навіть попри те, якщо він був згенерований на один часовий крок раніше або пізніше від поточного серверного часу.

У той же період клієнтська частина також має чітко визначений набір функціональних вимог. Головна взаємодія користувача з програмою починається вже з самого додавання нового облікового запису. Програмне забезпечення повинне мати доступ до камери девайсу для швидкого сканування QR-коду, який показує сам сервер. Після успішного сканування програма повинна розпізнати зашифрований текст, витягти з нього секретний ключ і зберегти його. Потрібно також передбачити і ручний режим. Тобто, якщо у користувача не працює камера на телефоні, то він може мати можливість ввести цей секретний ключ за допомогою клавіатури. Після того, як ключ буде успішно збережено, головною функцією мобільного додатка стає безперервна генерація часових кодів. Програма повинна брати збережений ключ і системний час девайсу і після чого кожні 30 секунд створювати новий шестизначний пароль. Важливою вимогою до користувацького інтерфейсу є наявність візуального індикатора часу. Людина має чітко розуміти та бачити таймер зворотного відліку або кругову діаграму, яка показує скільки секунд лишилось до того моменту, коли поточний код стане недійсним і зміниться на новий. Також додаток повинен мати дозвіл до керування записами. Це означає, що користувач має право у будь-який момент видалити збережений токен зі свого пристрою. Це зроблено для того, якщо він вирішив відключити двофакторну аутентифікацію на сервері.

Також окрім чистого функціоналу, система повинна відповідати низці нефункціональних вимог, які визначають її зручність, надійність та стабільність роботи. Найважливішою такою вимогою є повна автономність. Процес генерації кодів має відбуватись виключно в офлайн-режимі. Часто виникають такі ситуації,

наприклад, коли людина намагається увійти в систему з комп’ютера, але на її телефоні в цей момент немає доступу до мобільно інтернету або Wi-Fi. Оскільки алгоритм TOTP базується лише на математиці та системному часі пристрою, то цей мобільний додаток взагалі може не мати жодних мережевих запитів після первинного створення облікового запису та сканування QR-коду. Це не лише робить використання програми зручним у будь-яких умовах, але й суттєво підвищує безпеку, оскільки перехопити трафік додатка стає неможливим через його відсутність.

Значна увага має приділятись також ергономіці та користувацькому досвіду. Інтерфейс програми має бути мінімалістичним і не мати зайві деталі. Згенеровані шестизначні коди повинні відображатись великим шрифтом, щоб їх було легко зчитувати. Також для того, щоб людині було простіше запам’ятати цифри під час переписування їх з екрану телефона на екран комп’ютера, то логічно додати візуальний пробіл посередині. Це водночас і для кращого прийняття коду і вписання, наприклад, йде три цифри, тоді пробіл і знову три цифри. Сама програма має бути оптимізованою, не споживати багато ресурсів акумулятора у фоновому режимі та швидко запускатися.

Так як кваліфікаційна робота присвячена кібербезпеці, то ключовий акцент під час формування вимог ми робимо на безпекову складову. Перша і головна проблема, яку потрібно першочергово вирішити – це захист секретних ключів на самому мобільному пристрої. Зберігати такі чутливі дані у звичайному текстовому файлі або у відкритих локальних базах даних категорично заборонено. Якщо зловмисник якимось чином отримає доступ до файлової системи смартфона, то він зможе просто з легкістю скопіювати цей ключ і генерувати коди вже на своєму пристрої. Тому висувається така сувора вимога: додаток повинен використовувати спеціалізовані апаратні або системні сховища ключів. У такому випадку це унеможливлює крадіжку ключа, навіть якщо мати права суперкористувача. Це все за допомогою того, що секретний ключ буде зашифрований на системному рівні. Цікавою, але водночас і дуже ефективною вимогою безпеки є захист інтерфейсу додатка від перехоплення зображення. Сучасні шкідливі програми часто не намагаються саме зламати базу даних, а вони просто роблять приховані знімки

екрана або записують відео з екрана девайсу в той момент, коли користувач відриває додаток-аутентифікатор. Щоб протидіяти цьому, мобільний додаток повинен мати на рівні операційної системи заборону створення скріншотів. Якщо вірус спробує зробити знімок, він отримає лише чорний екран.

Ну і зрозуміло, що безпекові вимоги також висуваються і до серверної частини. Найпоширенішими векторами атак на етапі введення другого фактора є звичайний брутфорс. Оскільки код складається лише із шести цифр, то загальна кількість комбінацій становить один мільйон. Для сучасних комп'ютерів перебрати таку кількість варіантів займає лічені секунди. Щоб унеможливити такий сценарій, сервер повинен мати чіткий механізм обмеження швидкості запитів – Rate Limiting. Система повинна вимагати блокування IP-адреси або тимчасового заморожування облікового запису після, наприклад, п'яти або десяти невдалих спроб введення TOTP-коду поспіль.

2.2 Розробка архітектури системи

Після детального формування всіх функціональних, нефункціональних й безпекових вимог до майбутнього програмного забезпечення, наступним логічним і найбільш масштабним кроком є проєктування його базової архітектури. В інженерії програмного забезпечення архітектура системи визначає фундаментальну структуру комплексу. Тому, що описується з яких саме логічних та фізичних блоків складатиметься сама програма, як ці блоки будуть взаємодіяти між собою та як маршрутизуватимуться потоки даних і де саме будуть зберігатись найбільш чутливі елементи інформації. Зважаючи також на головну мету розробки – це забезпечення захисту конфіденційних даних користувачів. Усі без винятку архітектурні рішення повинні будуватися на концептуальному принципі мінімізації довіри, відомому в індустрії кібербезпеки як Zero Trust [25]. Цей принцип означає, що жоден компонент системи не повинен довіряти іншому по замовчуванню. Більше того, мобільний пристрій користувача взагалі повинен розглядатися архітектурою як потенційно скомпрометоване або вороже

середовище. Тільки за допомогою цього вдасться здобути ту саму безпеку для конфіденційних даних.

Глобально спроєктована система двофакторної аутентифікації базується лише на класичній розподіленій клієнт-серверній архітектурі. Цей підхід абсолютним стандартом для побудови сучасних безпечних інформаційних систем тому, що він дозволяє фізично і логічно відокремити середовище прийняття рішень від середовища користувацького інтерфейсу. У розробленій моделі чітко виділяються два абсолютно незалежні вузли. Кожен з цих вузлів має сувору регламентовану роль. Перший – це серверна частина. У системі він буде виступати єдиним джерелом істини та центром обробки всіх даних. Саме сервер має приймати остаточне рішення щодо того, чи є введений код правильний і чи варто пускати користувача до його облікового запису. Другий – це клієнтська частина, а саме мобільний додаток. З архітектурної точки зору роль мобільного додатка зводиться до функції ізольованого портативного апаратного токена. Він ніби є як архів, який лише надійно зберігає секретний ключ і за допомогою командою самого користувача генерує часові коди, проте самостійно він не виконує жодної перевірки прав доступу і не приймає жодних рішень. Такий суворий поділ обов'язків зможе з точністю дати гарантії того, що навіть повний злам мобільного додатка хакером за допомогою реверс-інжинірингу не дасть йому доступу до серверної логіки.

Архітектурний стиль REST – це те за допомогою чого взаємодія між мобільним додатком та сервером буде проєктуватись [26]. Вибір саме REST API серед інших можливих варіантів обґрунтовується його високою масштабністю, простотою інтеграції та саме головною вимогою – це відсутністю збереження стану сесії між запитами. При такому підході кожен мережевий запит від самого мобільного додатка до сервера містить абсолютно всю інформацію, яка необхідна для його обробки. Сервер не зберігає в оперативній пам'яті жодних даних про попередні кроки клієнта. Тому це критично важливо для безпеки і стабільності. У випадку, якщо зв'язок на девайсі раптово обірветься, або якщо зловмисник спробує відправити пакет даних поза чергою, то сервер просто обробить запит як незалежну одиницю або відхилить його.

Форматом обміну даними між цими двома вузлами буде слугувати JSON [27].

Це текстовий формат, який легко читається як і машиною, так і людино. Його використання в мобільній розробці продиктоване необхідністю мінімізувати розмір мережевих пакетів. Так як мобільні мережі часто характеризуються нестабільним покриттям та втратою пакетів, то використання легковагового формату дозволяє пришвидшити обмін інформацією і зменшити навантаження на мережевий модуль самого телефону. Таким чином мережевий рівень архітектури висуває одну жорстку і безальтернативну вимогу. Ця вимога базується на тому, що абсолютно всі транзакції, тобто передача логінів та пароля до відправки одноразового коду підтвердження, повинні переходити тільки через зашифрований криптографічний тунель. Зв'язок по відкритому протоколу HTTP блокується на рівні архітектури сервера, проте натомість використовується виключно протокол HTTPS із налаштованим транспортним шифруванням. Це єдиний спосіб гарантувати захист від атак типу «людина по середині». Сама ж логічна структура представлена на рисунку 2.1.

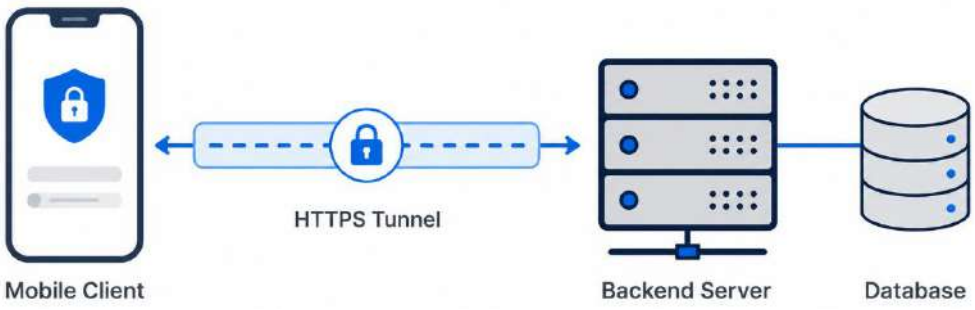


Рисунок 2.1 – Логічна архітектура взаємодії компонентів системи

Основою для серверної частини є маршрутизація та база даних. Так як проєктується систем, яка працює з найбільш конфіденційними даними, а саме паролі та секретні ключі, структура бази даних не може бути хаотичною. Її логічна структура проєктується у вигляді суворо нормалізованої реляційної моделі. Нормалізація дозволяє уникнути дублювання інформації та забезпечує її цілісність у разі непередбачуваних збоїв. Базова схема бази даних складається з трьох ключових таблиць, які пов'язані між собою логічними зв'язками.

Першою і водночас найважливішою частиною бази даних є таблиця облікових записів. Тут зберігається базова інформація про користувачів. Дивлячись

саме з позиції безпеки, то робити звичайні порядкові ідентифікатори, до прикладу користувач з ID 1, наступний з ID 2 і так далі, – це є величезна помилка. Такий підхід робить усю систему вразливою до Insecure Direct Object Reference [28], адже хакер може просто підставляти наступні цифри і вгадувати ID інших людей. Тому в цій архітектурі не використовуються звичайні числа. Замість них для ідентифікації генеруються унікальні 128-бітні рядки формату Universally Unique Identifier [29]. Адже їх просто нереально передбачити або підібрати. У цій же таблиці лежать і паролі. Але зрозуміло, що відповідно до базових вимог безпеки, вони не зберігаються відкритим текстом. Вони записуються у базу тільки після криптографічного хешування з обов'язковим додаванням унікального випадкового набору символів для кожного запису.

Друга таблиця – це конфігурації двофакторної аутентифікації. Вона має строгий зв'язок з таблицею користувачів по типу «один до одного». Це означає, що за одним акаунтом в один і той самий момент часу може бути закріплений лише один активний набір ключів 2FA. Тут зберігаються ті самі секретні ключі, з яких потім генеруються часові коди. Загалом, зберігання цих ключів на сервері – це мабуть найслабше місце будь-якої подібної системи. Якщо хакер зламає базу і вкраде ці ключі у відкритому вигляді, то вся ця двофакторна система перестане мати любий сенс. Тому було закладено у архітектуру бази даних механізм шифрування Data at Rest [30]. Тобто перед тим як попасти у таблицю, всі секрети проходять симетричне блочне шифрування. А розшифровуються вони виключно в оперативній пам'яті сервера і тільки в ту долю секунди, коли треба перевірити код користувача. Після перевірки ці дані відразу стираються з пам'яті.

Третя таблиця – це журнал авторизації. Будь-який розробник повинен розуміти, що система безпеки буде не завжди повноцінною, якщо все ж таки вона не логує події. Ця таблиця потрібна для моніторингу. Сюди будуть записуватись кожна спроба входу в систему, незалежно від того, чи вона була вдалою, чи ні. Тут фіксується UUID самого користувача, точний час, IP-адреса та сам результат перевірки. Завдяки такому журналу система може не тільки збирати статистику, але й автоматично виявляти щось підозріле. До прикладу, блокувати атаки повного перебору, якщо з однієї IP-адреси йде багато помилок.

Тепер щодо самого коду серверної частини, то він будується за модульним принципом. Це не один суцільний монолітний файл, а набір ізольованих сервісів. Перший модуль – це аутентифікація. Він обробляє перші запити, далі бере логін з паролем і звіряє їх з базою. Другий – це криптографічний модуль. Це найбільш захищена частина бекенду. Саме він створює нові секретні ключі високої ентропії, шифрує їх перед записом у базу та виконує всю складну математику під час перевірки шестизначних кодів від користувача. Третій – це модуль управління сесіями. Проте є один важливий момент: цей модуль створює тимчасові токени доступу тільки після того, як криптографічний модуль дасть добро і підтвердить, що пароль і тимчасовий код повністю збігаються. Ніяких проміжних чи часткових сесій архітектурою не передбачено.

Далі варто перейти до архітектури клієнтської частини, а саме самого мобільного застосунку. При розробці під мобільні платформи потрібно зробити вибір правильного патерну, адже це і є ключовою ціллю для стабільності. Традиційні підходи часто призводять до того, що код інтерфейсу змішується з логікою, а у даному випадку це є неприпустимим порушенням. Тому мобільний додаток проєктується на базі архітектурного патерну MVVM [31]. Model-View-ViewModel чудово дозволить чітко і жорстко розділити те, що бачить користувач та від того, що відбувається у самих процесорах програми.

Рівень View тут зроблений максимально пасивним. Він взагалі не містить жодних алгоритмів генерації паролів, також не має доступу до системний час і не має доступу до бази даних самого телефону. Головне і єдине його завдання – це зображати на екрані цифри, які йому передали з інших рівнів, та показувати кнопку для сканування QR-коду і анімувати круговий таймер, який показує час до оновлення коду.

Рівень ViewModel працює по-типу диспетчера. Саме він розуміє, що користувач відкрив додаток та запускає внутрішній цикл відділку секунд. Коли минає 30 секунд, то цей рівень не намагається згенерувати код самостійно, а сигналізує нижчому рівні, що потрібно створити новий пароль. Отриманий вже результат відправляє на рівень View, щоб користувач його побачив.

Наступний рівень вже являється справжнім ядром мобільно додатку – це

Model. Саме тут прописана вся математика обчислень кодів і алгоритмів роботи з пам’яттю девайсу. Найбільшою проблемою при розробці цього шару буде те, як безпечно зберігати секретні ключі. Якщо просто записати їх у звичайний текстовий файл чи стандартні налаштування телефону, то будь-який вірус, який має root-права, легко їх викраде. Тому найоптимальнішим варіантом для цього унеможливлення – це використання у архітектурі апаратних сховищ операційної системи. Цей додаток не зберігає ключ у відкритому вигляді. Він передає його операційній системі і шифрує його за допомогою ізольованого апаратного середовища процесора. У кінцевому варіанті на файловій системі лежить звичайна криптограма. Тому таке рішення робить із звичайного смартфона надійний апаратний крипто-гаманець.

Розібравшись із тим, як влаштовані бази та патерни, потрібно описати і динаміку – тобто послідовність дій системи під час роботи. До прикладу, потік прив’язки нового пристрою працює як ланцюжок асинхронних запитів. Спочатку людина через сайт просить увімкнути двофакторну аутентифікацію, далі сервер миттєво генерує унікальний ключ, ховає його зашифровану версію у базу і створює конфігураційний рядок. Цей рядок перетворюється на QR-код на екрані. Далі користувач бере смартфон, наводить камеру і додаток розпізнає цей код. Він витягує з тексту ключ і передає на рівень Model, який ховає його в апаратне сховище. На цьому етап налаштування закінчується. Далі мобільний телефон може взагалі бути без інтернету, бо для генерації кодів він йому не потрібен.

Потік авторизації також має чіткий алгоритм. Людина вводить логін та пароль на сайті і ці дані йдуть на сервер. Сервер шукає користувача в базі, хешує пароль і перевіряє його. Якщо є розбіжності, то з’єднання розривається з помилкою. А якщо пароль вірний, сервер реалізує важливий механізм безпеки. Тобто він не пускає користувача і не створює сесію. Він просто переводить цей конкретний запит у статус очікування другого фактора. У цей же момент користувачу потрібно відкрити сам додаток на телефоні. Це програмне забезпечення бере системний час, далі ділить його на рівні 30-секундні шматки і дістає зашифрований ключ із самої пам’яті. Після чого розшифровує його через процесор і генерує 6 цифр. Користувач вводить їх на сайті і вони відправляються

на сервер. Тепер сервер бере свій годинник, дістає з бази ключ цього користувача, розшифровує його в оперативці і робить таку ж саму генерацію коду.

Тому саме на цьому етапі працює одна дуже важлива деталь – це Look-ahead window. Так як годинник сервера і мільйонів телефонів не можуть йти ідеально синхронно то у багатьох користувачів були б невдалі спроби вводу. Тому сервер генерує цілий масив кодів: для поточних 30 секунд, для попередніх і для наступних. Код від користувача звіряється з усім масивом. Якщо є таки збіг, то другий фактор підтверджено. І тільки після цього система генерує токен доступу і пускає людину в акаунт. Щоб цей процес не можна було зламати звичайним перебором кодів, то на сервері є механізм обмеження запитів. Він працює як фільтр перед криптографічним модулем. Сервер просто запам’ятовує кожну помилкову спробу і прив’язує її до IP-адреси. Щоб це працювало швидко і не тормозило основну базу, дані зберігаються в пам’яті. Якщо хтось неправильно вводить код кілька разів поспіль, фільтр просто блокує запити з цієї адреси і навіть не пускає їх до криптографічного ядра перевірки.

2.3 Вибір алгоритмів та механізмів генерації одноразових кодів

Після успішно проєктування логічної архітектури системи настав найбільш відповідальний та технічно складний етап роботи – це безпосередньо вибір криптографічних алгоритмів та механізмів захисту інформації. Будь-яка, навіть ідеально розроблена на папері архітектура, не має жодного практичного сенсу, якщо на рівні програмного коду використовуються застарілі, вразливі або просто неправильно підібрані математичні моделі. Оскільки розроблена система позиціонується як надійний інструмент кібербезпеки, то кожне рішення щодо вибору алгоритмів автор чітко обґрунтовує. Головним завданням є досягнення оптимального балансу між високою криптографічною стійкістю, швидкістю роботи на мобільних пристроях та сумісністю із загальноприйнятими світовими стандартами безпеки. Особлива увага приділяється тому, як ці інструменти вписуються в реальну роботу смартфона, щоб захист не сповільнював пристрій і не

виснажував акумулятор. Такий підхід дозволяє перетворити суху теорію на дієвий механізм, який справді здатен захистити конфіденційні дані користувача. У цьому підрозділі детально розглядається кожен алгоритм, який забезпечує життєдіяльність системи: від генерації самих кодів до апаратного шифрування ключів у пам'яті смартфона.

Центральним елементом розробки є алгоритм генерації одноразових паролів. Сама система спирається на стандарті TOTP, який офіційно описаний у специфікації Інженерної ради Інтернету IETF RFC 6238. Проте важливо розуміти, що сам по собі TOTP – це лише концептуальна модель, яка визначає використання поточного системного часу замість послідовного лічильника подій. Справжнім ядром цього процесу є алгоритм хешування HMAC. Сучасні криптографічні стандарти дозволяють інтегрувати в цю конструкцію різні хеш-функції, найпопулярнішими з яких сьогодні є SHA-1, SHA-256 та SHA-512. Такий механізм гарантує унікальність кожного значення та унеможливлює прогнозування наступних паролів навіть за умови перехоплення попередніх кодів зловмисником. Процес перетворення секретного ключа та поточного часу на фінальний шестизначний пароль (разом з етапами обчислення HMAC-хешу та його динамічним усіченням) наочно зображено на рисунку 2.2.

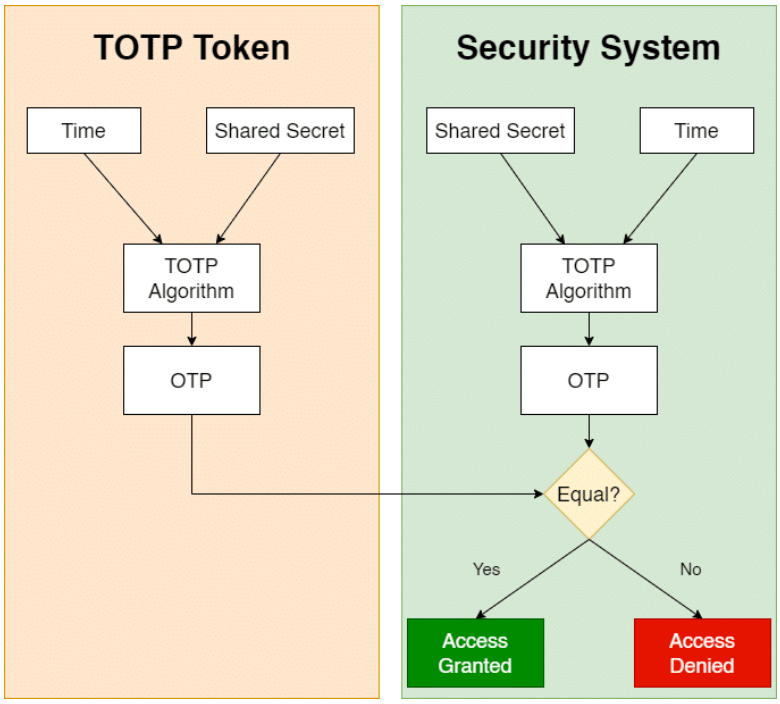


Рисунок 2.2 – Схема математичного алгоритму генерації TOTP-пароля

З академічної точки зору для забезпечення максимальної безпеки, то логічно було б обрати алгоритм SHA-512 тому, що це є найвідоміший і найміцніший алгоритм з сімейства Secure Hash Algorithms. Але проводячи аналіз специфіки роботи автономних мобільних аутентифікаторів, то самий припустимий був алгоритм HMAC-SHA1. Цей вибір обґрунтовується низкою вагомих інженерних причин, які на практиці переважають звичайну криптографічну стійкість:

– стійкість до колізій у конструкції HMAC: хоча чиста функція SHA-1 має вразливості, алгоритм HMAC вимагає обов'язкового додавання випадкового секретного ключа та виконання побітових операцій. Це повністю унеможливлює ризики колізій і робить алгоритм безпечним для генерації 6-значних паролів [32];

– стовідсоткова зворотна сумісність: абсолютна більшість світових платформ історично використовують протоколи, які очікують від клієнта саме 160-бітний вихідний масив від SHA-1. Це дозволить додатку працювати з будь-якими сторонніми сервісами;

– енергоефективність та оптимізація ресурсів: алгоритм SHA-1 вимагає значно меншої кількості обчислювальних тактів процесора і менше оперативної пам'яті смартфона порівняно з SHA-256 або SHA-512. Для фонової генерації кодів кожні 30 секунд це критично важливо для збереження заряду акумулятора.

Наступним механізмом став спосіб безпечної передачі вже згенерованого секретного ключа від сервера до мобільного додатка під час першого налаштування системи. Створений сервером секретний ключ пристає перед користувачем масивом випадкових байтів із високим рівнем ентропії. Передавати та відображати його у сирому бінарному вигляді технічно неможливо, тому виникає необхідність перетворення у зручний текстовий формат. Найпопулярнішим алгоритмом кодування бінарних даних у текст є Base64. Але для систем двофакторної аутентифікації використання Base64 є серйозною ергономічною помилкою. Тому після порівняльного аналізу як стандарт кодування обирається алгоритм Base32.

З метою максимальної мінімізації людського фактора, то якраз саме Base32 для цього чудово підходить. Цей алгоритм використовується для кодування вкрай обмежений словникових слів, лише великі літери англійського алфавіту та цифри. З його алфавіту були виключені цифри «0», «1» та літери «O», «I». Це пояснюється

тим, що у більшості сучасних шрифтів ці символи візуально ідентичні, що дуже часто призводило до плутанини. Тому ця особливість стає критично важливою, якщо у користувача пошкоджена або не працює камера на девайсі і йому доводиться вводити 32-значний секретний ключ вручну. Якби таки було застосовано стандарте кодування Base64, то ручне введення неминуче закінчився би помилкою. Алгоритм Base32 ефективно вирішує цю проблему, формуючи безпечний та зручний для сприйняття рядок.

Далі закодований рядок підставляється у спеціальний формат URI, який набуває вигляду: otpauth://totp/SystemName:UserEmail?secret=BASE32KEY&issuer=SystemName. Саме цей довгий рядок серверна частина по далі перетворює на QR-код. Після генерації QR-коду також було передбачено застосування алгоритму корекції помилок Ріда-Соломона [33]. Для роботи застосовується саме середній рівень корекції «M» тому, що він дозволяє мобільному додатку успішно розпізнати та відновити секретний ключ. Навіть, якщо приблизно 15% площі всього графічного ключа буде пошкоджено, закрито бліком від екрана до монітора або частково не увійде в об’єктив камери. Нижче на рисунку 2.3 буде зображено як буде виглядати QR-код.

Рисунок 2.3 – Приклад візуального представлення TOTP у вигляді QR-коду

Поміж математичної генерації кодів, архітектура вимагає стовідсоткового надійного захисту конфіденційних даних у Data at Rest на стороні сервера. У

спроєктованій базі даних зберігаються дві найбільш критичні сутності – це паролі користувачів та їхні унікальні секретні ключі 2FA. Оскільки природа цих даних принципово різна, методи їх криптографічного захисту також відрізняються. Для збереження паролів користувачів категорично заборонено використовувати симетричне чи асиметричне шифрування тому, що серверу об'єктивно не потрібно знати оригінальний пароль, так як йому достатньо лише перевірити математичний факт збігу під час спроби входу. Тому у цій підсистемі використовується хешування. Детальний аналіз демонструє, що звичайні алгоритми хешування, по типу MD5 та SHA-256, для цієї задачі абсолютно не підходять. Вони розроблялися для забезпечення максимальної швидкості обчислень. Якщо зловмисник вкраде базу даних, то він зможе використати потужності графічних процесорів і перебирати десятки мільярдів варіантів SHA-256 за одну секунду, миттєво відновивши прості паролі.

Дивлячись на всі ці загрози, для захисту облікових записів було обрано спеціалізований алгоритм хешування паролів Bcrypt [34]. Його головні переваги полягають у таких пунктах як:

– вбудований механізм розтягування ключа: алгоритм має так званий «фактор вартості», який штучно уповільнює хешування. Рівень складності налаштовано на 212 (4096 ітерацій), через що перевірка одного пароля займає близько 250-300 мілісекунд;

– стійкість до атак повного перебору: завдяки штучному уповільненню, хакер, який викраде базу даних, зможе перебирати лише кілька варіантів на секунду, що робить підбір паролів на відеокартах економічно недоцільним;

– автоматична генерація криптографічного випадкового набору символі: Bcrypt самостійно генерує унікальний криптографічний модифікатор для кожного пароля. Це гарантує різні хеші навіть для однакових паролів і робить хакерські райдужні таблиці абсолютно марними.

Щодо ситуації із захистом секретних ключів TOTP у базі даних вимагала іншого підходу. Порівнюючи з паролями, то ці ключі не можна хешувати. Серверу необхідно знати оригінальне значення секретного ключа у відкритому вигляді, щоб мати змогу обчислити поточний пароль і порівняти його з тим, що ввів користувач.

Тому для їх захисту інтегровано алгоритм симетричного шифрування. Найбільш надійним та визнаним стандартом є AES [35].

Для системи було обрано алгоритм AES з максимальною довжиною ключа у 256 біт. Але вибір самого алгоритму то це лише перший крок. Саме важливо було обрати правильний режим роботи цього блочного шифру. Традиційні режими, до прикладу як ECB або CBC, мають серйозні математичні недоліки. З огляду на це, застосовується режим GCM. Перевага AES-GCM полягає в тому, що він забезпечує автентифіковане шифрування з приєднаними даними. Також він не лише надійно шифрує секретний ключ, а й паралельно обчислює спеціальний автентифікаційний тег за математики полів Галуа. Тобто, якщо зловмисник отримає доступ до бази і спробує змінити хоча б один біт у зашифрованому масиві, то сервер під час спроби розшифрування миттєво виявить порушення цілісності криптограми і заблокує операцію. Усі ключі шифруються за допомогою єдиного потужного майстер-ключа сервера, який задля безпеки винесено за межі вихідного коду програми у захищені змінні оточення. Такий вибір режиму GCM зумовлений не лише безпекою, а й високою швидкістю обробки даних завдяки можливості паралельного обчислення блоків. Використання AES-GCM робить систему стійкою до атак типу «Padding Oracle», які часто стають проблемою для застарілих методів шифрування. Поміж цього, зберігання майстер-ключа у змінних оточення дозволяє повністю ізолювати секрети від самого коду, що відповідає кращим практикам сучасної розробки. Це створює додатковий рівень захисту: навіть якщо вихідний код проєкту потрапить до сторонніх осіб, дані все одно залишаться під надійним замком. У результаті формується сучасна криптографічна база, яка не просто закриває формальні вимоги, а реально захищає систему від професійних втручань.

Після того як було вирішено завдання безпеки на серверному боці, далі було необхідно реалізувати надійний механізм на стороні клієнта. Тобто, як безпечно зберігати розшифрований секретний ключ у пам’яті мобільного телефону. Збереження цього секрету у стандартних файлах конфігурації або у відкритій базі даних SQLite є дуже великою помилкою. Будь-який шкідливий додаток із розширеними привілеями міг би з легкістю прочитати цей файл і згенерувати валідні коди доступу на сторонньому пристрої.

Серед величезного вибору механізмів захисту мобільних операційних систем демонструє, що найкращим рішенням є інтеграція в додаток системний API – Android Keystore System [36]. На рисунку 2.4 наочно показано, як мобільний застосунок взаємодіє з апаратним захистом через сервіс Keystore. Це дозволяє повністю ізолювати ключі від операційної системи, щоб до них не було прямого доступу.

Рисунок 2.4 – Архітектура системи Android Keystore та механізм апаратної ізоляції ключів

Цей механізм дозволяє створювати та використовувати криптографічні ключі в ізольованому апаратному середовищі центрального процесора девайсу (TEE). Логіка роботи спроєктованого додатку виглядає наступним чином: коли камера зчитує QR-код із секретним ключем, програма звертається до Keystore із запитом на апаратну генерацію нового ключа шифрування. Далі цей ключ генерується всередині мікрочіпа і не залишає його межі у відкритому вигляді. Потім додаток передає отриманий секретний TOTP-ключ у це апаратне середовище, мікропроцесор шифрує його і повертає додатку лише криптограму – набір зашифрованих байтів, які вже безпечно зберігаються в локальній базі. Коли настає

час генерації 6-значного коду, додаток відправляє криптограму назад у Keystore. Апаратний модуль розшифровує ключ у своїй ізольованій пам'яті, самостійно обчислює функцію HMAC-SHA1 і повертає додатку готові цифри. Таким чином, навіть у разі крадіжки пристрою та повного фізичного дампу пам'яті, зловмисник отримає лише марний зашифрований текст, так як ключ для його розшифрування апаратно прив'язаний до процесора конкретного смартфона.

Так не мало важливим комплексом механізмів стали алгоритми протидії атакам у реальному часі та вирішення проблем синхронізації. Так як головна технічна проблема у роботі протоколу TOTP є розсинхронізація часу і алгоритм суворо математично прив'язаний до системного часу, то якщо годинник на телефоні користувача відстає або поспішає всього на кілька десятків секунд, сервер неминуче відхилятиме всі спроби авторизації. Щоб усунути розсинхронізацію, було реалізовано алгоритм компенсації Look-ahead window. Суть його дуже просто: сервер генерує не один код для перевірки, а масив із трьох кодів. Ці три коди є для поточного 30-секундного інтервалу, для попереднього та для майбутнього. Введений користувачем пароль порівнюється з усім масивом. Тому введення цього алгоритму у систему ефективно згладжує мережеві затримки та похибки вбудованих кварцових генераторів смартфонів, але не залишає зловмиснику достатньо часу для перехоплення та використання старого коду.

Але у цьому розширені вікна доступу дає свій же побічний ефект. Так як тепер правильними є три комбінації з мільйона, що полегшує атаку повного перебору. Для повного усунення цієї загрози в архітектуру сервера було інтегровано алгоритм жорсткого обмеження частоти запитів на базі математичної моделі Sliding Window Log. На відміну від примітивних лічильників, цей алгоритм фіксує точний час кожної невдалої спроби введення коду у швидку In-Memory базу даних. Якщо система фіксує понад 5 помилок за останні 10 хвилин з однієї IP-адреси або для одного профілю, то вона активує захисне блокування і відхиляє будь-які подальші спроби. Блокування знімається автоматично, щойно найстаріші спроби виходять за межі 10-хвилинного вікна.

Після реалізації всіх методів наступним кроком стало створення алгоритму генерації резервних кодів відновлення. Сувора реалізація двофакторки має один

недолік. Якщо у разі втрати або пошкодження девайсу, то користувач фізично втрачає секретний ключ, який зашифрований в самому апаратному Keystore. У результаті це призводить до втрати доступу до акаунту. Щоб запобігти цього, то під час первинного налаштування сервер генерує масив із 10 випадкових резервних цифро-літерних кодів. Ці коди зберігаються в базі даних сервера за тими ж суворими правилами, що й звичайні паролі у вигляді Bcrypt-хешів із випадковими модифікаторами. Кожен код є суворо одноразовим. Під час використання резервного коду сервер хешує його, перевіряє збіг із базою, надає доступ і миттєво видаляє цей запис. Це гарантує, що навіть перехоплений код неможливо буде використати повторно.

2.4 Висновки

Підсумовуючи роботу, виконану у другому розділі, можна стверджувати, що етап проєктування є найважливішим фундаментом для створення будь-якої системи безпеки. Так як у цьому розділі було здійснено перехід від теоретичного аналіз проблем двофакторної аутентифікації до розробки цілеспрямованої математично та архітектурно обґрунтованої програмної моделі.

Спочатку було сформовано чіткий перехід функціональних, нефункціональних та безпекових вимог до майбутньої системи. Головним акцентом стала вимога повної автономності клієнтського додатка. Тому це не лише користувацький досвід, але й повністю відрізняє зловмисникам можливість перехоплення трафіку. Також, що не мало важливо, було закладено жорсткі вимоги щодо захисту пам’яті телефону та протидії автоматизованим атакам на сервер. Також було спроєктовано логічну клієнт-серверну архітектуру, яка базується на концепції Zero Trust. Система була розділена на два ізольовані вузли такі, як серверна частина, яка бере на себе всю вагу обчислень, зберігання бази даних та управління сесіями та клієнтська частина, яка виконує роль виключно безпечного генератора кодів. Для мобільного клієнта було обрано архітектурний патерн MVVM, що дозволило повністю відокремити графічний інтерфейс від складної

криптографічної бізнес-логіки. Взаємодія між вузлами під час налаштування системи була побудована на базі легковагового REST API з обов'язковим транспортним шифруванням HTTPS.

Проте найскладнішим завданням став вибір конкретних криптографічних алгоритмів. Для генерація TOTP-кодів обирається алгоритм HMAC-SHA1 так, як він забезпечує ідеальний баланс між криптографічною стійкістю, швидкістю обчислень на мобільних пристроях та 100% сумісністю зі світовими стандартами. А для безпечної передачі секретних ключів через QR-коду було застосовано кодування Base32. Окрема увага приділяється захисту даних у стані спокою. Тому на серверній стороні паролі користувачів захищаються за допомогою алгоритму Bcrypt, адже він унеможливлює атаки за допомогою райдужних таблиць. Секретні ключі TOTP шифруються симетричним алгоритмом AES-256 у режимі GCM. На стороні мобільного додатка для захисту секретів інтегровано апаратне сховище Android Keystore, що перетворює звичайний смартфон на надійний апаратний токен і перешкоджає викрадення ключів навіть вірусам з root-правами.

Створена архітектура – це не просто навчальний макет, а міцна основа для реального незалежного продукту. Відмова від використання готових сторонніх сервісів аутентифікації дає величезну перевагу, адже отримується тотальний контроль над усім життєвим циклом безпеки. Система гарантує, що жоден секретний ключ або резервний код не синхронізується з невідомими хмарними серверами без відома користувача чи адміністратора. Така автономність та ізольованість робить розроблену модель ідеально придатною для інтеграції в закриті корпоративні екосистеми, фінансові установи або державні структури, де вимоги до приватності є безкомпромісними. Це є прямим доказом того, що надійний захист потрібно будувати з нуля, контролюючи кожен байт інформації.

Тому у результаті отримано продуману, цілісну та безпечну архітектурну модель системи двофакторної аутентифікації. Запропоновані інженерні рішення повністю задовольняють поставлені вимоги і створюють надійну базу для фінального етапу кваліфікаційної роботи – це написання програмного коду, тестування системи та її впровадження.

3 РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ СИСТЕМИ

3.1 Програмна реалізація системи

На самому етапі проєктування програмного комплексу двофакторної автентифікації було прийнято рішення щодо використання класичної клієнт-серверної архітектури. Адже такий підхід дозволяє чітко логіку збереження важливих даних та логіку генерації одноразових паролів. У даному ж підрозділі детально розглядається процес самої реалізації серверної частини, вибір інструментів та проєктування бази даних.

Щоб одразу дати чітке розуміння того, з яких саме логічних блоків складається розроблений програмний комплекс, загальна архітектура візуалізується перед тим, як переходити до описуваної частини реалізації. На рисунку 3.1 наведено схему програмної моделі системи.

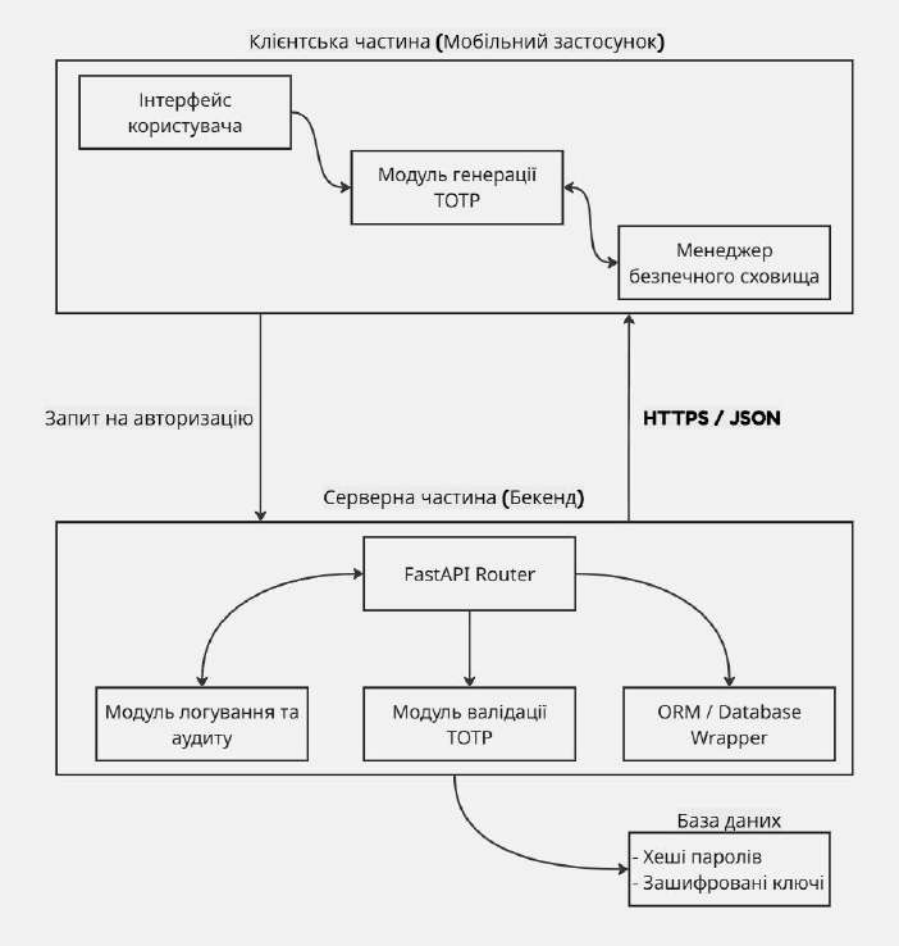


Рисунок 3.1 – Схема програмної моделі системи двофакторної автентифікації

Дивлячись на світлину вище, можна побачити комплекс, який чітко розділено на два великі незалежні блоки. На стороні мобільного застосунку локалізовано модулі інтерфейсу, генерації TOTP-кодів та менеджер безпечного апаратного сховища. Натомість серверна частина бере на себе прийом API-запитів через FastAPI Router, валідацію кодів, логування подій та взаємодію з базою даних. Такий модульний поділ ідеально демонструє принцип ізоляції. Тобто, клієнт лише генерує коди, а сервер приймає остаточне рішення щодо авторизації.

Основна мова програмування для реалізації бекенду використовувалась Python 3, адже має досить високу читабельність, швидкість розробки прототипів та головне для кваліфікаційної роботи – це наявність потужних бібліотек криптографічних обчислень. Для побудови ж самого веб-сервера та маршрутизації API-запитів було використано мікрофреймворк FastAPI [37]. Адже під час дослідження інших альтернатив, такі як Django та Flask, то саме FastAPI показав найкращий результат у категоріях як:

– висока продуктивність: завдяки асинхронній архітектурі FastAPI здатен обробляти велику кількість одночасних запитів, що є критично важливим для системи авторизації;

– автоматична документація: фреймворк ще з першого використання генерує інтерактивну документацію за стандартом OpenAPI [38], що значно спрощує процес тестування API та взаємодію з клієнтським додатком;

– строга типізація: використання саме бібліотеки Pydantic [39] дозволяє автоматично перевіряти вхідні дані, що захищає сервер від некоректних запитів.

Проте для забезпечення цілісності та довготривалого зберігання облікових даних користувачів було використано саме базу даних. Оскільки розроблена система на даному етапі позиціонується як Proof of Concept та мікросервіс, використання важких систем керування базами даних на кшталт PostgreSQL чи MySQL було б не раціонально. Тому замість цього було обрано саме SQLite [40] – це вбудована реляційна база даних. Вона зберігає всі дані у єдиному локальному файлі «user.db» на самому ж сервері, тому і не потребує додаткового налаштування портів та є ідеальним рішенням для локального зберігання зв’язок користувачів та їх секретних ключів.

У самій базі даних під час ініціалізації сервера автоматично створюється таблиця під назвою «users» з двома полями:

– email – унікальний ідентифікатор користувача;

– secret_key – згенерований сервером секретний ключ.

Нижче на рисунку 3.2 буде відображено саму функцію init_db() з SQL-запитом CREATE TABLE:

```
def init_db():
    with sqlite3.connect("users.db") as conn:
        cursor = conn.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS users (
                email TEXT PRIMARY KEY,
                secret_key TEXT NOT NULL
            )
        """)
```

Рисунок 3.2 – Ініціалізація бази даних SQLite та створення таблиці користувачів

Так, на етапі базового прототипу можна було б і обійтись лише однією простою таблицею users. Але розроблена система проєктувалася з урахуванням реальних вимог інформаційної безпеки, тому структура розширюється до повноцінної нормалізованої моделі. Для кращого і наочного відображення всіх сутностей та зв'язків між ними було побудовано ER-схему бази даних, яка показана на рисунку 3.3. База складається з головно таблиці «USERS», яка пов'язана відношенням «один-до-одного» з таблицею зашифрованих секретних ключів «TOTP_CONFIGS». Також було реалізовано зв'язки «один-до-багатьох» із журналом авторизації «AUTH_LOGS» і таблицею одноразових кодів відновлення «RECOVERY_CODES». Така графічна модель чудово показує системність підходу до безпечного та надійного зберігання даних.

Ядро розробленої серверної частини є бібліотека pyotp [41], яка повністю реалізовує весь математичний алгоритм TOTP згідно зі стандартом RFC 6238. Також важливим етапом під час розробки стало правильна генерація секретних ключів. Саме алгоритм TOTP строго вимагає, щоб секретний ключ був закодований у форматі Base32.

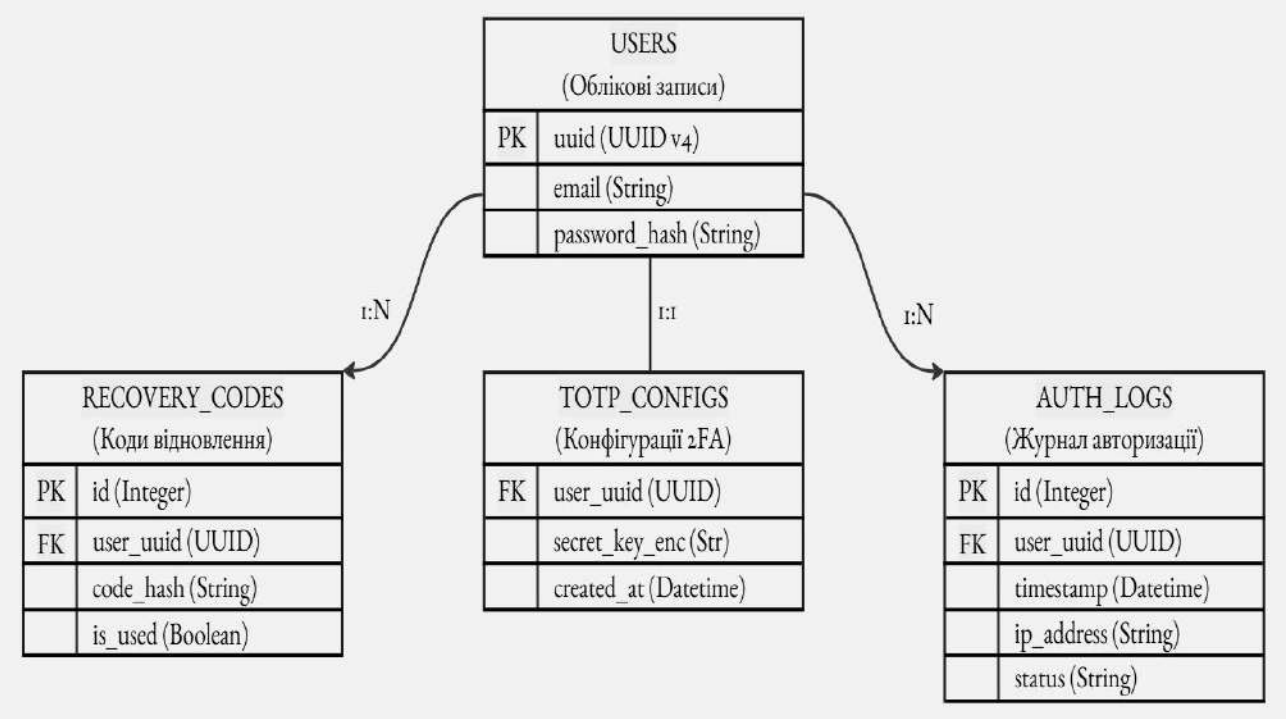


Рисунок 3.3 – ER-діаграма реляційної бази даних програмного комплексу

При розробці було виявлено досить типову помилку, а саме використання довільних символів по типу «0», «1» і це призводить до критичних помилок ще на етапі розшифрування. Це пов’язано з тим, що стандарт спеціально виключає ці символи зі свого алфавіту, щоб користувачі не плутали їх з літерами «O», «I» при ручному введені. Саме тому для генерації ключів було використано функція «pyotp.random_base32()», яка максимально гарантує криптографічну стійкість та сумісність формату. Також сервер надає два основні API точки доступу:

– POST /api/v1/generate-key: приймає email користувача. Перевіряє, чи існує такий користувач у базі. Якщо користувач новий то генерує унікальний секретний ключ, зберігає його в БД та формує спеціальне URI-посилання;

– POST /api/v1/verify-code: приймає email та 6-значний код, який вводиться користувачем. Сервер дістає з бази даних секретний ключ для цього email, бере поточний системний час і самостійно генерує код. Якщо згенерований сервером код збігається з тим, що надіслав користувач, то авторизація підтверджується. Нижче на рисунку 3.4 показано сторінку Swagger UI.

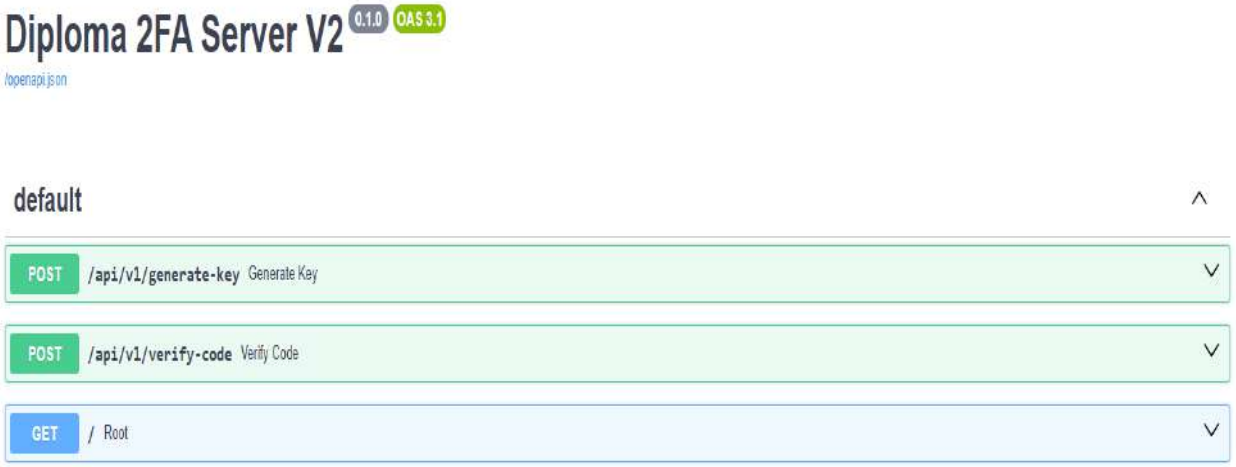


Рисунок 3.4 – Автоматично згенеровано OpenAPI документація маршрутів сервера

Зважаючи на те, що основна взаємодія із сервером передбачається через мобільний додаток, то для повноцінної демонстрації роботи системи та процесу прив’язки нового пристрою було розроблено адаптивний веб-інтерфейс.

Інтерфейс було реалізовано у вигляді Single Page Application за допомогою файлу «index.html». Стилізація виконана з використанням чистого CSS3 у темній темі, адже воно і краще виглядає і практичніше. Логіка взаємодії з сервером написана на JavaScript, що дозволяє відправляти запити та отримувати відповіді без перевантаження сторінки. Такий підхід забезпечує швидкий відгук елементів керування під час створення нового облікового запису. Завдяки цьому згенерований сервером QR-код та резервний текстовий ключ миттєво виводяться на екран, дозволяючи одразу відсканувати їх камерою смартфона.

Веб-інтерфейс було розділено на дві логічні вкладки:

– Реєстрація ключа: дозволяє користувачу ввести свою електронну пошту та отримати згенерований сервером секретний ключ. Також для зручності користувача ключ дублюється у двох виглядах: у вигляді QR-коду та у текстовому вигляді з кнопкою швидкого копіювання в буфер обміну. Це продемонстровано на рисунку 3.5;

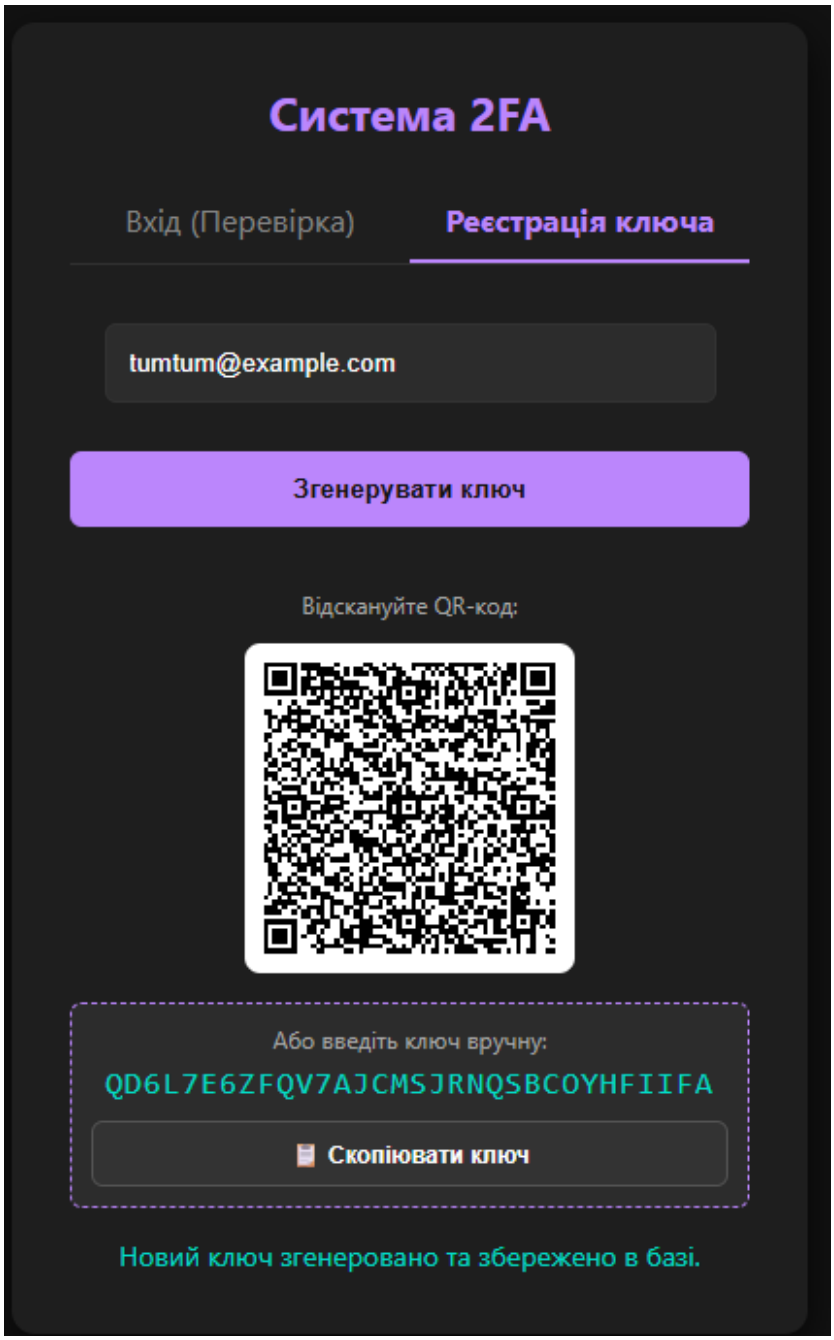


Рисунок 3.5 – Веб-інтерфейсу системи реєстрації ключа

– Вхід (Перевірка): імітує авторизацію як у реальному веб-сайті. Користувач вводить свій email 6-значний код, який користувач бачить у програмі автентифікації. JavaScript відправляє ці дані на сервер і у разі правильному введені коду то виводить повідомлення про успішну авторизацію. На рисунку 3.6 це дуже добре відображено.

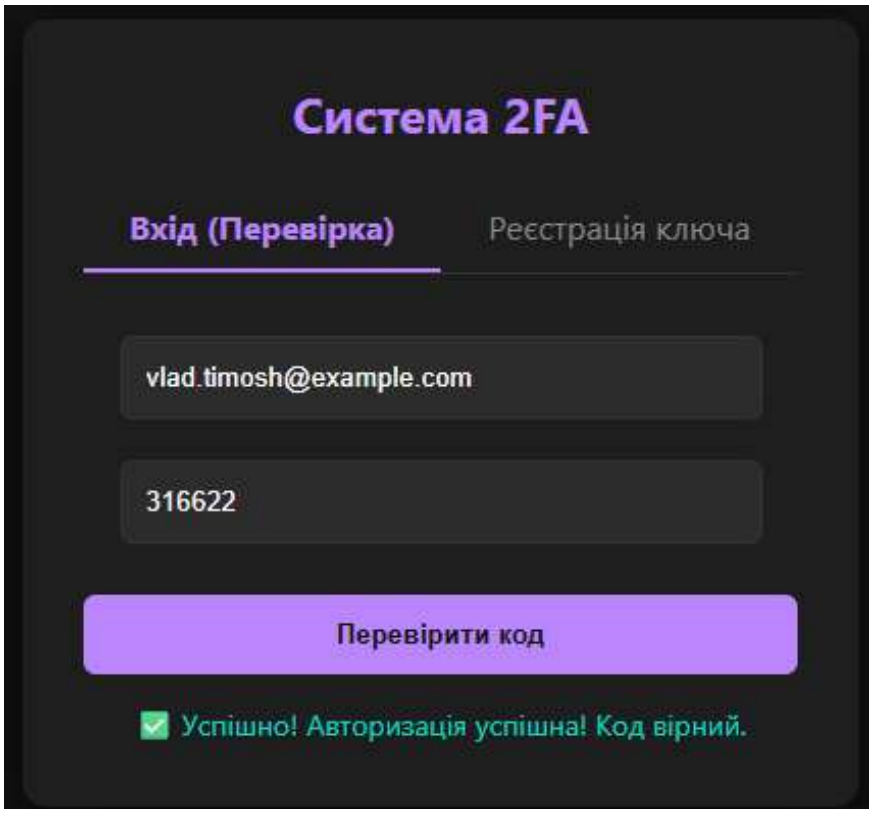


Рисунок 3.6 – Успішна валідація одноразового паролю через веб-інтерфейс входу

Таким чином, серверна частина програмного комплексу є повністю автономним, який здатен безпечно генерувати криптографічні ключі, зберігати їх у базі даних та перевіряти легітимність запитів від клієнтських пристроїв. Наступним етапом розробки стала реалізація захищеного мобільного клієнта, який виступає в ролі апаратного генератора кодів.

3.2 Інтеграція механізму двофакторної аутентифікації у мобільний застосунок

Після успішного проєктування, реалізації та комплексного тестування серверної частини системи, логічним і найважливішим наступним кроком стала розробка клієнтського програмного забезпечення. У класичній ієрархічній структурі двофакторної аутентифікації першим фактором є знання, а другим фактором є володіння фізичним об’єктом. На сьогодення найзручнішим фізичним об’єктом, який людина завжди має при собі, є сам смартфон. Тому по цій причині,

що клієнт виконує роль токена для генерації цих одноразових 6-цифрових паролів, було вирішено створити мобільний застосунок. У цьому ж підрозділі буде детально розписано самі етапи проєктування архітектури, вибір інструментів, реалізація алгоритмів шифрування та побудови інтерфейсу.

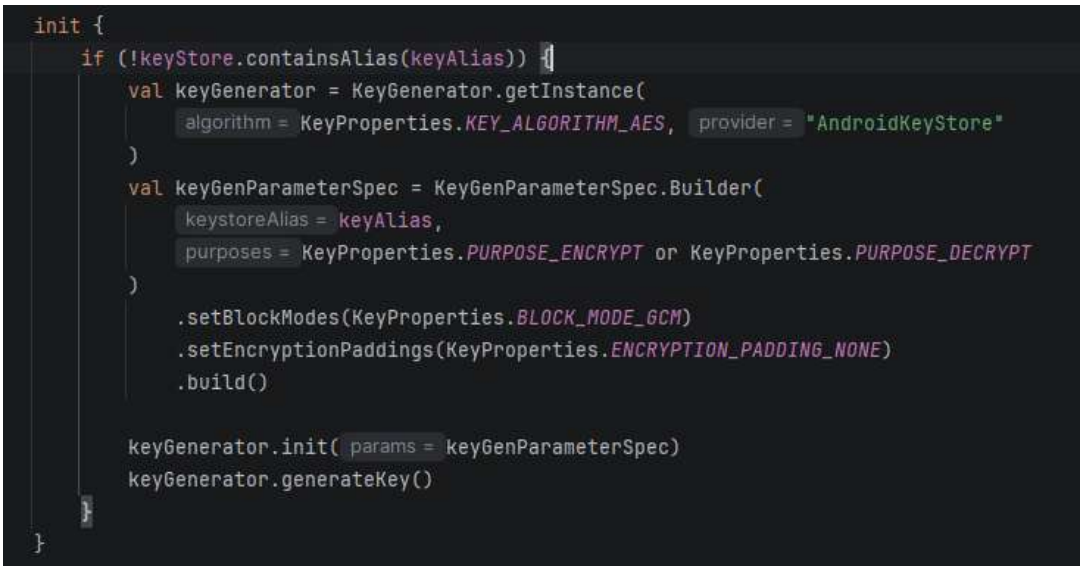
Під час вибору платформи для створення мобільного застосунку було найбільше приділено увагу саме на операційну систему Android. Це рішення аргументується тим, що ця операційна система є відкрита, величезна ринкова частка припадає саме на ці мобільні пристрої, наявність гнучких інструментів для розробки, які надає компанія Google. Якщо беручи їх головна конкурента IOS то там набагато тяжче розробити та впровадити мобільний застосунок у маси. Для розробки використовується офіційне інтегроване середовище Android Studio, яке забезпечує зручну роботу з макетами екранів, вбудований аналізатор коду та потужні засоби для тестування застосунку на віртуальних емуляторах смартфонів. Основна мова програмування використовувалась Kotlin [42] тому, що якщо зрівнюючи з Java, то саме Kotlin є сучасним стандартом для всіє екосистеми Android. Також саме ця мова пропонує набагато логічний синтаксис, що зменшує обсяг коду та вбудовану систему захисту від помилок нульового вказівника. Отже, завдяки цьому вдалось уникнути найпоширеніші причини раптового закриття програма, що робить застосунок набагато стабільнішим та надійним у повсякденному використанні.

Головним пріоритетом під проєктування архітектури застосунку була визначена абсолютна безпека даних користувачів. Так як мобільний телефон може бути випадково втрачений, поцуплений зловмисниками або навіть заражений якимось шкідливим програмним забезпеченням, зберігання секретних ключів у відкритому вигляді категорично заборонено та неприпустимо. Спочатку розглядалися варіанти збереження ключів у SQLite або у SharedPreferences, але ці методи є досить вразливими. До прикладу, якщо зловмисник отримає root-права до файлової системи пристрою, то він зможе з легкістю прочитати ці файли та скопіювати ключі на свій пристрій, тим самим повністю скомпрометувавши всю систему двофакторної аутентифікації.

Для вирішення цієї критичної проблеми було вирішено інтегрувати

застосунок із Android Keystore System – це спеціальний системний механізм, розроблений на рівні самої операційної системи, який дозволяє створювати та зберігати криптографічні ключі у спеціальному ізольованому апаратному середовищі пристрою під назвою «Trusted Execution Environment». Головна перевага полягає у тому, що ключі шифрування ніколи не покидають межі цього захищеного процесора і не потрапляють у звичайну оперативну пам’ять у відкритому вигляді. Навіть маючи повний контроль над операційною системою, хакер не зможе програмно витягти сам ключ.

У практичній реалізації цієї логіки був створений окремий клас «KeystoreManager». Він оснований на використанні симетричного алгоритму шифрування AES. Сам захист даних відбувається у декілька послідовних етапів. При першому запуску програми клас перевіряє наявність майстер-ключа у сховищі «Keystore», якщо його немає то автоматично генерує новий унікальний ключ шифрування та прив’язуючи його до застосунку. Коли користувач додає новий обліковий запис у систему, оригінальний секретний ключ формату Base 32 передається до цього менеджера. Натомість менеджер використовує режим шифрування AES/GCM/NoPadding [43], який не тільки надійно зашифровує текстовий рядок, а й перетворює його на хаотичний набір байтів. Також не мало важливо, що він додає криптографічний підпис для перевірки цілісності. Це захищає дані від будь-яких непомітних модифікацій [44]. Лише після того, як було успішно пройдено всі процедури апаратного шифрування то цей набір байтів кодується у зручний формат Base 64 і вже в такому безпечному вигляді зберігається у звичайній пам’яті мобільного девайсу. Проте, коли програмі потрібно згенерувати одноразовий пароль для користувача, відбувається взагалі зворотній процес. Застосунок зчитує зашифрований рядок Base 64, передає його в KeystoreManager і апаратний процес розшифровує його. Після чого він віддає оригінальний ключ прямо в оперативну пам’ять тільки на декілька мілісекунд, які потрібні для математичних обчислень. На рисунку 3.7 чітко відображено використання KeyProperties.KEY_ALGORITHM_AES.



```
init {
    if (!keyStore.containsAlias(keyAlias)) {
        val keyGenerator = KeyGenerator.getInstance(
            algorithm = KeyProperties.KEY_ALGORITHM_AES, provider = "AndroidKeyStore"
        )
        val keyGenParameterSpec = KeyGenParameterSpec.Builder(
            keystoreAlias = keyAlias,
            purposes = KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT
        )
            .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
            .build()

        keyGenerator.init( params = keyGenParameterSpec)
        keyGenerator.generateKey()
    }
}
```

Рисунок 3.7 – Програмна реалізація шифрування даних

Наступний етап стає реалізація самого математичного алгоритму генерації одноразових паролів TOTP. Одна з найбільших архітектурних особливостей та найбільших переваг цієї системи є те, що клієнтська частина працює повністю в автономному офлайн-режимі. Тому завдяки цьому програма взагалі не спілкується із сервером у роботі і це автоматично унеможливлює проведення атаки «людини по середині». Повна синхронізація між сервером та застосунком відбувається лише завдяки спільному секретному ключу та спирання на єдиний світовий час.

У мобільному застосунку генерація коду базується на використанні бібліотеки kotlin-onetimepassword. Сама логіка генерації працює безперервно у фоновому режимі. Програма бере розшифрований секретний ключ користувача та отримує від операційної системи телефону поточний час у мілісекундах. Далі цей час ділиться на фіксовані інтервали по 30 секунд. За допомогою цій математичній операції гарантується те, що користувач відкрив програму на перших чи останніх секундах цього інтервалу то формула отримає абсолютно однакове вхідне значення часу. Далі секретний ключ і це значення пропускається через хеш-функцію HMAC-SHA1 та створює довгий хеш-код, який обрізається до звичайного 6-значного числового коду. Також для зручності користувачеві було зроблено зворотній відлік часу, який показує скільки часу лишилося до наступного коду. За допомогою класу Timer та методу runOnUiThread програма кожну секунду оновлює графічний інтерфейс. Тобто, він перераховує залишок секунд у поточному вікні, оновлює

текстове поле та зменшує пропорційно шкалу прогресу. Коли час закінчується то цикл миттєво перезапускається і генерує нову комбінацію. Нижче на рисунку 3.8 відображено сам застосунок з 6-значним кодом та таймером для наступного циклу.

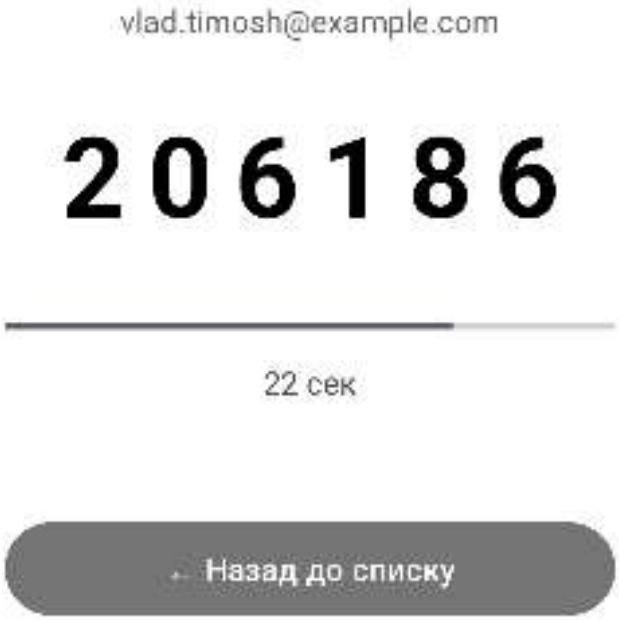


Рисунок 3.8 – Інтерфейс генерації одноразового TOTP-паролю

Також не мало важливою частиною розробки стало проєктування користувацького інтерфейсу та логіки взаємодії. Також для зручності користування цим застосунком було зроблено підтримка одразу декількох облікових записів. Користувач маю можливість зберігати в одному місці ключі від різних робочих сервісів, соціальних мереж або електронних пошт. Для виконання цієї функції використовувалась архітектура єдиного вікна [45]. Весь інтерфейс розташований у файлі XML, проте він розділений на два великі блоки. Перехід між екранами імітується шляхом програмної зміни видимості цих блоків за допомогою View.VISIBLE та View.GONE.

Перший екран, який зустрічає користувача при запуску, є головне меню зі списком підключених облікових записів. При підключенні нових пошт, застосунок формує список динамічно. Програма звертається до локальної пам’яті зчитує збережений текстовий рядок з усіма адресами, розбиває його на окремі елементи і прямо в програмному коді генерує для кожної пошти красиву кнопку, яку додає у вертикальний список на екрані. Коли користувач натискає на яку небудь пошту,

цей список ховається, а замість нього появляється другий блок – це інтерфейс генерації коду. На рисунку 3.9 відображено головне меню програми.

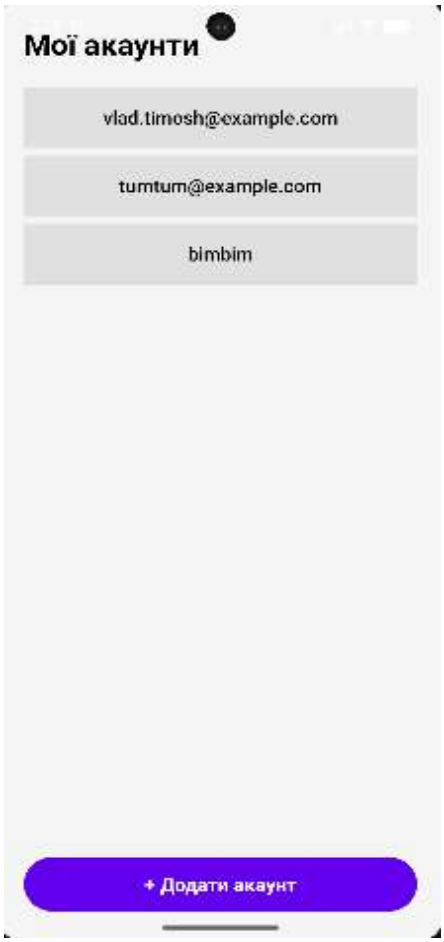


Рисунок 3.9 – Головний екран мобільного клієнта

Під час проєктування, тестування та інтеграції виникла важлива проблема щодо способу додавання нових акаунтів через QR-код. Під час роботи стало очевидним те, що такий підхід не є достатнім надійним. До прикладу, камера пристрою може бути пошкоджена, не працює або люба інша проблемо, то користувач може налаштувати на тому ж пристрої, де відкрито сам сайт. Тому функціонал підключення був значно розширений. При натисканні на кнопку додавання акаунта застосунок викликає системне діалогове вікно, яке пропонує користувачеві альтернативних шлях. Тобто, користувач може вручну ввести свою електронну адресу та вставити секретний ключ з буфера обміну.

Для захисту від людського фактора у вікні передбачена базова валідація даних. Це працює так, що програма перевіряє, щоб поля не були порожніми та

автоматично видаляє випадкові пробіли, які користувач міг випадково захопити при копіювання ключа. Якщо ж ключ отримується через сканування QR-коду, то застосунок використовує вбудований парсер, який аналізує стандартизоване посилання формату otpauth://totp/... . Автоматично тоді знаходить у ньому параметри пошти та самого ключа і миттєво передає їх до модуля шифрування та збереження. Таким чином система гарантує максимальну зручність і гнучкість для користування за будь-яких умов використання. На рисунку 3.10 показано методи додавання нових електронних скриньок у програму.

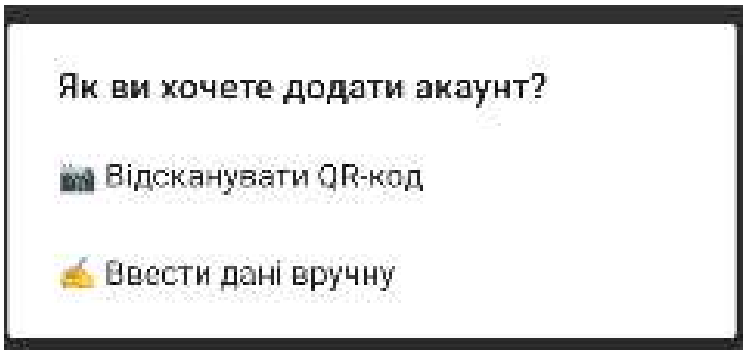


Рисунок 3.10 – Реалізація діалогового вікна методів підключення облікових записів

Так як однією з головних архітектурних фішок застосунку є повна автономність, процес загальної авторизації може здаватися дещо незвичайним. Для найкращої демонстрації взаємодій між користувачем, смартфоном, веб-сайтом, базою даних у реальному часу, було побудовано UML-діаграму послідовності (рисунок 3.11). На нижній світлині детально показано життєвий цикл передачі даних. Тут чітко видно, що після введення пароля сервер не одразу пускає користувача, а затримує процес і вимагає другий фактор автентифікації. Проте у цей час мобільний застосунок локально розшифровує ключ і генерує 6 цифр. І лише тоді після цих дій, коли користувач вручну вводить цей одноразовий пароль на сервер, бекенд дістає свій ключ із бази, робить паралельне обчислення і порівнює результати.

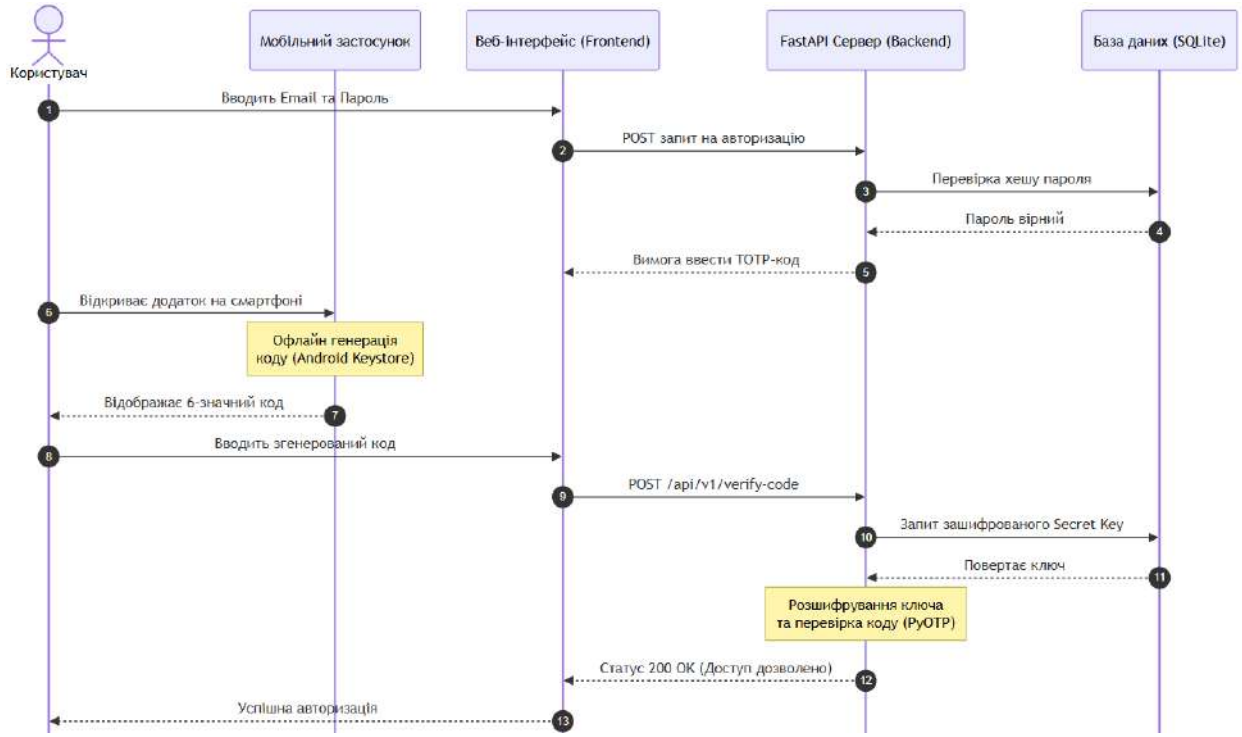


Рисунок 3.11 – UML-діаграма послідовності інтеграції компонентів системи під час авторизації

Тому дивлячись на пророблену роботу можна повноцінно стверджувати те, що розроблений мобільний застосунок повноцінним, безпечним та ергономічним інструментом. Він успішно реалізує складні криптографічні операції, ефективно використовує апаратні можливості пристрою для захисту даних від викрадення та забезпечує повну автономність процесу автентифікації без необхідності підключення до мережі.

3.3 Тестування та аналіз результатів роботи системи

Настав критично важливий етап тестування після завершення написання коду для серверної та клієнтської частини. Так як головним завданням розробленого комплексу є забезпечення безпеки та захисту облікових користувачів, то система не має права на любу помилку. Навіть найменша помилка у базовій архітектурі може призвести до того, що користувач втратить доступ до свого акаунту, або ще гірше це отримання доступу зловмисником. Тому було

розроблено та проведено комплексний план тестування. В нього входить перевірка окремих модулів, тестування на стійкість до збоїв, аналіз безпеки збереження даних та наскрізне інтеграційне тестування всієї системи в умовах реального використання.

Першим кроком стало тестування серверної частини та її взаємодія з базою даних SQLite. За для цього локальний сервер був запущений за допомогою Unicorn, а всі API-запити тестувались через фреймворк FastAPI інтерфейс Swagger UI. Початковий тест можна вважати базова реєстрація нового користувача. Було введено вигадану тестову електронну адресу у відповідне поле маршруту генерації ключа і виконав запит. Сервер відповідає статусом 200 «Успіх» та повернув згенерований секретний ключ і посилання формату otpauth для створення QR-коду. Але просто отримати ключ це недостатньо. Потрібно перевірити його на здатність системи пам’ятати дані після критичного збою. Тому зупинено роботу сервера у терміналі Visual Studio Code. Це робить імітацію раптового відключення електроенергії, що зараз є не рідкістю, або перезавантаження серверного обладнання. Після цього повторно було запущено сервер і спроба пройти авторизацію з тим самим ключем. SQLite успішно зберегла файл і сервер миттєво знайшов тестовий обліковий запис і підтвердив вхід. Тому ця перевірка довела, що архітектура готова до довготривалої експлуатації.

Під час тестування модуля генерації криптографічних ключів виявляється програмний баг. Спочатку перевірино, чи зможе система працювати з кастомними заданими ключами. Тому здійснюється спроба реєстрації користувача з секретним ключем «DIPLOMA2FA2026SECURITYKEY». На перший погляд можна побачити, що це досить надійний та довгий рядок, але як тільки спробував відправити запит на перевірку, то сервер перервав роботу і видав критичну помилку 500 Internal Server Error. Це означає, що бібліотека обчислення TOTP-кодів не змогла розшифрувати цей рядок. Причина цього є цифра «0». Алгоритм TOTP вимагає кодування ключа виключно формату Base32. Світовий стандарт Base32 категорично забороняє цифри «0», «1», «8», «9». Це зроблено для того, щоб людина під час ручного введення ключа не плутала їх з літерами «O», «I», «B», «g». Тому після цього було оновлено логіку сервера, щоб тепер він ігнорував будь-які ручні

втручання і завжди генерував правильний Base32 рядок самостійно за допомогою функції pyotp.random_base32(). Нижче на рисунку 3.12 зображено відкритий інтерфейс Swagger UI.

Рисунок 3.12 – Тестування REST API

Наступний колосальний етап є тестування клієнтського мобільного застосунку. Так як розробка і первинний запуск проводився на віртуальному емуляторі Android-пристрою, виникає важливе обмеження – це не здатність сканування QR-коду за допомогою камери. Тому, щоб не шукати якій складні шляхи обходу цієї проблеми, було додано альтернативний шлях – це ручне введення ключа. При створенні нової електронної скриньки у серверній частині, потрібно скопіювати сам секретний ключ і у самому застосунку вставити його у відповідне поле при додаванні нового акаунту. Застосунок успішно прийняв цей ключ, закрив діалогове вікно і на головному меню миттєво зв’явилася нова кнопка з назвою тестової пошти. Саме цей тест дав зрозуміти, що наявність два шляхи додавання акаунтів є досить практичним та зручним методи для користувача.

Далі є тестування безпеки зберігання ключів та пристрої. За допомогою інструменту Device File Explorer в Android Studio імітується поведінка зловмисника, який отримує доступ до всієї внутрішньої пам’яті смартфона. Далі здійснюється доступ до прихованої системної папки застосунка «data/data/com.example.diploma2faauth/shared_prefs» та відкривається конфігураційний файл SecurePrefs.xml, де фізично зберігаються всі дані. Замість звичайного секретного ключа відображається лише хаотичний набір символів, що складається з букв, цифр та знаків рівності. Це був зашифрований шифротекст. Спроба скопіювати цей шифротекст і розшифрувати його іншими онлайн-інструментами в результаті нічого не дало, адже оригінальний майстер-ключ AES

знаходиться глибоко у захищеному процесорі емулятора і був прив’язаний тільки до цифрового підпису додатка. Завдяки цьому тесту було доведено, що архітектура відповідає стандартам комерційної безпеки. На рисунку 3.13 показано як виглядає зашифрований шифротекс.

Рисунок 3.13 – Вигляд зашифрованого секретного ключа у системній пам’яті пристрою

Тепер настав час найголовнішого тестування цього застосунку – це тест алгоритму генерації кодів на основі часу. Так як клієнт і сервер не мають постійного з’єднання між собою, вони мусять створювати ідентичні 6 цифрові коди, які спираються лише на системний годинник. Для цього у застосунку відкривається тестова електронна пошта та паралельно відкривається для перевірки веб-інтерфейс сервера. Коли таймер коду в застосунку дійшов до кінця, цифри одразу мінились. Після очікування ще 10 секунд, щоб перевірити, що код стабільний і ввів його на сайті. Сервер відповів повідомленням «Авторизація успішна». Тобто, сервер генерує не лише код для поточних 30 секунд, але й перевіряє код з попередніх 30 секунд. Проте якщо ввести той самий код, який був хвилину тому, то сервер відхилив його з помилкою «Невірний код або час його дії минув». Це доводить те, що вікно доступу не створює серйозних вразливостей для атаки перехоплення та є чітко контрольованою.

Окрему увагу слід приділити користувацькому інтерфейсу та логіці роботи з кількома акаунтами. Так як користувачі використовують десятки сервісів, тому програма повинна легко витримувати масштабування. Для цього у програму додається шість різних тестових електронних скриньок з різними секретними ключами. Головне меню одразу адаптувалось і перетворилось на список, який можна зручно гортати свайпом. При натисканні на любу пошту програма генерує один код, а при натисканні на іншу пошту то показує абсолютно інший. Також це

перевіряє логіку перемикання екранів і воно працює ідеально. Перехід відбувається за долі секунд без жодних візуальних затримок, а споживання оперативної пам’яті взагалі мінімальне [46].

Аналізуючи всі результати тестів, можна з упевненістю зробити висновок, що розроблений застосунок повністю виконує поставленні завдання. Система демонструє високий рівень відмовостійкості, надійно захищає криптографічні дані від несанкціонованого доступу [47], забезпечує плавну та зрозумілу взаємодію з користувачем.

3.4. Висновки

Робота на третім розділом кваліфікаційної роботи став найважливішим та вирішальним етап усієї роботи. Адже саме тут теоретичні концепції, математичні формули та спроєктована архітектура була перетворена у реальний, працюючий застосунок. Головне завдання цього етапу було безпечну систему двофакторної автентифікації, відмовостійкий комплекс і дивлячись на всю виконану роботу можна з упевненістю сказати, що це завдання було успішно виконано.

Під час реалізації було успішно створена та розділена два незалежні компоненти: серверна частина та клієнтський мобільний застосунок. Для сервера було обрано мову програмування Python та фреймворк FastAPI. Також інтеграція бази даних SQLite дозволила надійно зберігати облікові записи користувачів та їх секретні ключі. Такий базовий набір інструментів дозволив зосередитись саме на логіці безпеки, а не на складному налаштуванні серверної інфраструктури.

Для клієнтського мобільного застосунку використовувалась мова програмування Kotlin. Використання технології Android Keystore System стало ключовим рішенням у цій роботі. Для того, щоб реалізувати шифрування для оригінальних ключів було використано KeystoreManager, адже якраз він розроблений за стандартом AES/GCM. Це гарантує, що навіть при отриманні фізичного доступу до файлової системи смартфона розшифрувати ці ключі без авторизації буде неможливо.

Окрема увага була приділена ергономіці та користувацькому досвіду. Щоб програма була дійсно корисним у повсякденному використанні була реалізована гнучка система роботи з декількома обліковими записами через єдине вікно. Із-за цього програма працює швидко і економно споживає ємність батареї. Також було додано для зручнішого використання два види додавання нових електронних пошт: через сканування QR-коду та ручне введення пошти та секретного ключа. Крім того, для зручного візуального контролю кожен згенерований пароль супроводжується динамічним таймером, який показує час до створення нового коду.

Проведення тестування готового програмного забезпечення підтвердило повну стабільність взаємодії клієнтської та серверної частини під час генерації та валідації токенів. Завдяки оптимізації алгоритмів, перевірка одноразових паролів на стороні бекенду відбувається майже миттєво, не створюючи жодних помітних затримок при вході. Окремо в ході випробувань було змодельовано спроби прямого витягування секретних ключів із системних папок операційної системи Android. Успішне блокування цих дій на практиці підтвердило високу надійність архітектури та вибраного криптографічного підходу із використанням ізольованого середовища Keystore.

Отже, можна стверджувати, що створений програмний комплекс не є просто макетом, а повноцінний Minimum Viable Product, який доводить ефективність використання TOTP та апаратне шифрування мобільних пристроїв. Розробка даного проєкту надає величезний практичний досвід у проєктуванні архітектури, роботі з криптографічними стандартами та глибокому розумінні принципів захисту інформації в сучасних інформаційних системах.

ВИСНОВКИ

Кваліфікаційна робота присвячена розробці та практичному впровадженні комплексної архітектури для систем двофакторної автентифікації. У сучасному світі, де щогодини проводяться десятки тисяч кібератак, звичайний захист за допомогою логіна та пароля остаточно втратив свою ефективність та актуальність. Щодня відбуваються масові витоки баз даних, а хакери активно використовують автоматизовані системи перебору паролів та соціальну інженерію. Навіть одна з самих розповсюдженіших типів підтвердження входу через SMS-повідомлення більше не є безпечною. Це завдяки атаки із підміною SIM-карти. Саме з цієї причини ставиться за мету спроєктувати та створити власну, повноцінно автономну систему 2FA.

Дослідження розпочинається із глибокого аналізу міжнародних стандартів. За основу береться світовий протокол RFC 6238, який описує алгоритм генерації одноразових паролів, які основані на поточному часу – TOTP. Розуміння того, що вразливість більшості систем лежить у зберіганні секретного ключа та як передаються дані між сервером та клієнтом, забезпечує основу для побудови архітектури. Потрібно розділити систему на два рівні – бекенд-сервер для створення ключів та мобільний застосунок.

Практична частина стартувала з розробки серверної частини. В якості основного інструменту було обрано мову програмування Python та фреймворк FastAPI. Також було інтегровано SQLite для забезпечення надійного зберігання зв’язку «користувач – криптографічний ключ». Для того, щоб виключити людський факто при створенні криптографічних пар довелось впровадити формат Base32 для суворого автоматичного генерації ключів.

Серцем кваліфікаційної роботи є клієнтська частина – це мобільний застосунок на мові Kotlin у середовищі Android Studio. Основна мета була розробка системи, яка вміє рахувати секунди та хешувати дані через алгоритм HMAC-SHA1. Основною проблемою у мобільних операційних системах є зберігання секретних ключів у звичайних текстових файлах. Для вирішення цієї проблеми застосовується підхід багаторівневого захисту. Було розроблено модуль на основі Android Keystore

System. Замість відкритого зберігання, він автоматично шифрує всі секретні ключі за допомогою AES/GCM. Оригінальні ключі натомість генеруються та назавжди залишаються в Trusted Execution Environment.

Коли код був написаний, почався етап тестування, під час якого проводяться спроби зламати розроблену систему. Спеціальне введення паролів з невеликою затримкою, відключення інтернету, щоб перевірити автономну роботу. Також намагання витягнути збережені ключі з системних папок телефону. Проте програма витримала все: він ідеально генерує коди в офлайн режимі, дає користувачеві ввести протягом 30 секунд ввести старий пароль та надійно ховає всі дані від несанкціонованого отримання. У підсумку отримується досить гнучке рішення. Серверна частина та мобільний додаток працюють абсолютно незалежно один від одного.

Поміж поточних результатів, розроблений комплекс має хороший простір для майбутнього масштабування. Завдяки REST API серверну частину можна без проблем підключити до вже існуючих корпоративних мереж або інших вебсервісів. При цьому ж архітектура не вимагатиме переписування базової логіки. Що стосовно мобільного додатку, то можна розширити його функціонал за допомогою біометричної автентифікації, щоб запитувати відбиток пальця або скан лиця перед показом самих одноразових кодів. І за допомогою цього це стане не двофакторною, а трьохфакторною системою автентифікації. Для серверної частини залишається відкритим напрямок створення повноцінної вебпанелі адміністратора, де фіксуватимуться підозрілі спроби несанкціонованого доступу. У загальному підсумку, поставлене на початку роботи завдання виконано в повному обсязі. Спроєктована система двофакторної автентифікації надійно перекриває вразливості звичайних паролів і гарантує збереження конфіденційних даних користувача.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. World Economic Forum. Global Cybersecurity Outlook 2024. URL: https://surl.li/ixqpzm (дата звернення: 05.03.2026).

2. NIST Special Publication 800-63B. Digital Identity Guidelines: Authentication and Lifecycle Management. National Institute of Standards and Technology. URL: https://pages.nist.gov/800-63-3/sp800-63b.html (дата звернення: 05.03.2026).

3. Android Application Sandbox. Android Developers Documentation. URL: https://source.android.com/docs/security/app-sandbox (дата звернення: 06.03.2026).

4. Сталлінгс В. Криптографія та безпека мереж. Принципи та практика. 7-е вид. К. : Діалектика, 2022. 944 с.

5. OWASP Mobile Top 10. Open Worldwide Application Security Project. URL: https://owasp.org/www-project-mobile-top-10/ (дата звернення: 06.03.2026).

6. Session Hijacking in Mobile Apps. OWASP Foundation. URL: https://surl.li/tiosln (дата звернення: 06.03.2026).

7. Шнайєр Б. Прикладна криптографія. Протоколи, алгоритми та вихідні тексти на мові С. К. : Діалектика, 2023. 1024 с.

8. Android Developers. App security best practices. URL: https://developer.android.com/topic/security/best-practices (дата звернення: 10.03.2026).

9. Aviv A. J. et al. Smudge Attacks on Smartphone Touch Screens. 4th USENIX Workshop on Offensive Technologies (WOOT 10). URL: https://surl.lt/khoilm (дата звернення: 10.03.2026).

10. FIDO Alliance. FIDO2: Web Authentication (WebAuthn). URL: https://fidoalliance.org/fido2/ (дата звернення: 10.03.2026).

11. ISO/IEC 19795-1:2021. Information technology — Biometric performance testing and reporting. URL: https://www.iso.org/standard/73515.html (дата звернення: 10.03.2026).

12. Болтов С. В., Сидоров І. П. Біометричні системи ідентифікації та автентифікації. К. : Наукова думка, 2021. 320 с.

13. ARM Developer. TrustZone for Cortex-A. URL: https://surl.li/vboynw (дата звернення: 10.03.2026).

14. Secure Enclave overview. Apple Platform Security. URL: https://surl.li/hienjq (дата звернення: 11.03.2026).

15. Android BiometricPrompt API. Android Developers. URL: https://developer.android.com/training/sign-in/biometric-auth (дата звернення: 11.03.2026).

16. Frida: Dynamic instrumentation toolkit for developers and reverse-engineers. URL: https://frida.re/ (дата звернення: 11.03.2026).

17. Cyble. US Telecom Networks Security: SS7 Vulnerabilities. URL: https://cyble.com/blog/us-telecom-networks-security/ (дата звернення: 13.03.2026).

18. FTC Consumer Advice. SIM Swap Scams: How to Protect Yourself. URL: https://surl.li/upekhe (дата звернення: 13.03.2026).

19. Microsoft Tech Community. Defend your users from MFA fatigue attacks. URL: https://surl.li/gpegrt (дата звернення: 13.03.2026).

20. RFC 6238: TOTP: Time-Based One-Time Password Algorithm. URL: https://datatracker.ietf.org/doc/html/rfc6238 (дата звернення: 14.03.2026).

21. RFC 4226: HOTP: An HMAC-Based One-Time Password Algorithm. URL: https://datatracker.ietf.org/doc/html/rfc4226 (дата звернення: 14.03.2026).

22. The Open Group Base Specifications. Epoch definition (Unix Time). URL: https://surl.li/xruzys (дата звернення: 14.03.2026).

23. RFC 2104: HMAC: Keyed-Hashing for Message Authentication. URL: https://datatracker.ietf.org/doc/html/rfc2104 (дата звернення: 14.03.2026).

24. RFC 4648: The Base16, Base32, and Base64 Data Encodings. URL: https://datatracker.ietf.org/doc/html/rfc4648 (дата звернення: 14.03.2026).

25. Rose S., Borchert O., Mitchell S., Connelly S. Zero Trust Architecture (NIST SP 800-207). URL: https://csrc.nist.gov/publications/detail/sp/800-207/final (дата звернення: 20.03.2026).

26. Ричардсон Л., Рубі С. RESTful Web Services. Київ, 2020. 448 с.

27. RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format. URL: https://datatracker.ietf.org/doc/html/rfc8259 (дата звернення: 20.03.2026).

28. Insecure Direct Object Reference (IDOR). OWASP Cheat Sheet Series. URL: https://surl.li/mbykwi (дата звернення: 21.03.2026).

29. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. URL: https://datatracker.ietf.org/doc/html/rfc4122 (дата звернення: 21.03.2026).

30. OWASP Mobile Application Security. MASVS-STORAGE. URL: https://mas.owasp.org/MASVS/05-MASVS-STORAGE/ (дата звернення: 21.03.2026).

31. Мартін Р. Чиста архітектура. Мистецтво розробки програмного забезпечення. К. : Фабула, 2022. 432 с.

32. NIST SP 800-131A Rev. 2: Transitioning the Use of Cryptographic Algorithms and Key Lengths. URL: https://surl.li/wrwhjn (дата звернення: 23.03.2026).

33. QR Code Standard. Denso Wave Incorporated. URL: https://www.qrcode.com/en/about/version.html (дата звернення: 23.03.2026).

34. Provos N., Mazières D. A Future-Adaptable Password Scheme (Bcrypt). USENIX Annual Technical Conference. URL: https://surl.li/noghps (дата звернення: 23.03.2026).

35.FIPS 197: Advanced Encryption Standard (AES). URL: https://surl.li/hqvvkq (дата звернення: 24.03.2026).

36. Android Keystore System. Android Developers Documentation. URL: https://developer.android.com/training/articles/keystore (дата звернення: 24.03.2026).

37. Лутц М. Вивчаємо Python. 5-е вид. К. : Діалектика, 2021. 1160 с.

38. Models Concepts. Pydantic Official Documentation. URL: https://pydantic.dev/docs/validation/latest/concepts/models/ (дата звернення: 17.04.2026).

39. SQLite Official Documentation. Database File Format and Security. URL: https://www.sqlite.org/docs.html (дата звернення: 17.04.2026).

40. PyOTP – The Python One-Time Password Library. GitHub Repository. URL: https://github.com/pyauth/pyotp (дата звернення: 18.04.2026).

41. OpenAPI Specification (OAS). Swagger. URL: https://swagger.io/specification/ (дата звернення: 18.04.2026).

42. Kotlin Programming Language. Official Documentation. URL: https://kotlinlang.org/docs/home.html (дата звернення: 23.04.2026).

43. NIST Special Publication 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. URL: https://surl.lt/twkcjn (дата звернення: 23.04.2026).

44. McGrew D. A., Viega J. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. IACR Cryptology ePrint Archive. URL: https://eprint.iacr.org/2004/193.pdf (дата звернення: 23.04.2026).

45. Single Activity Architecture and Navigation Component. Android Developers. URL: https://surl.li/fidxbd (дата звернення: 24.04.2026).

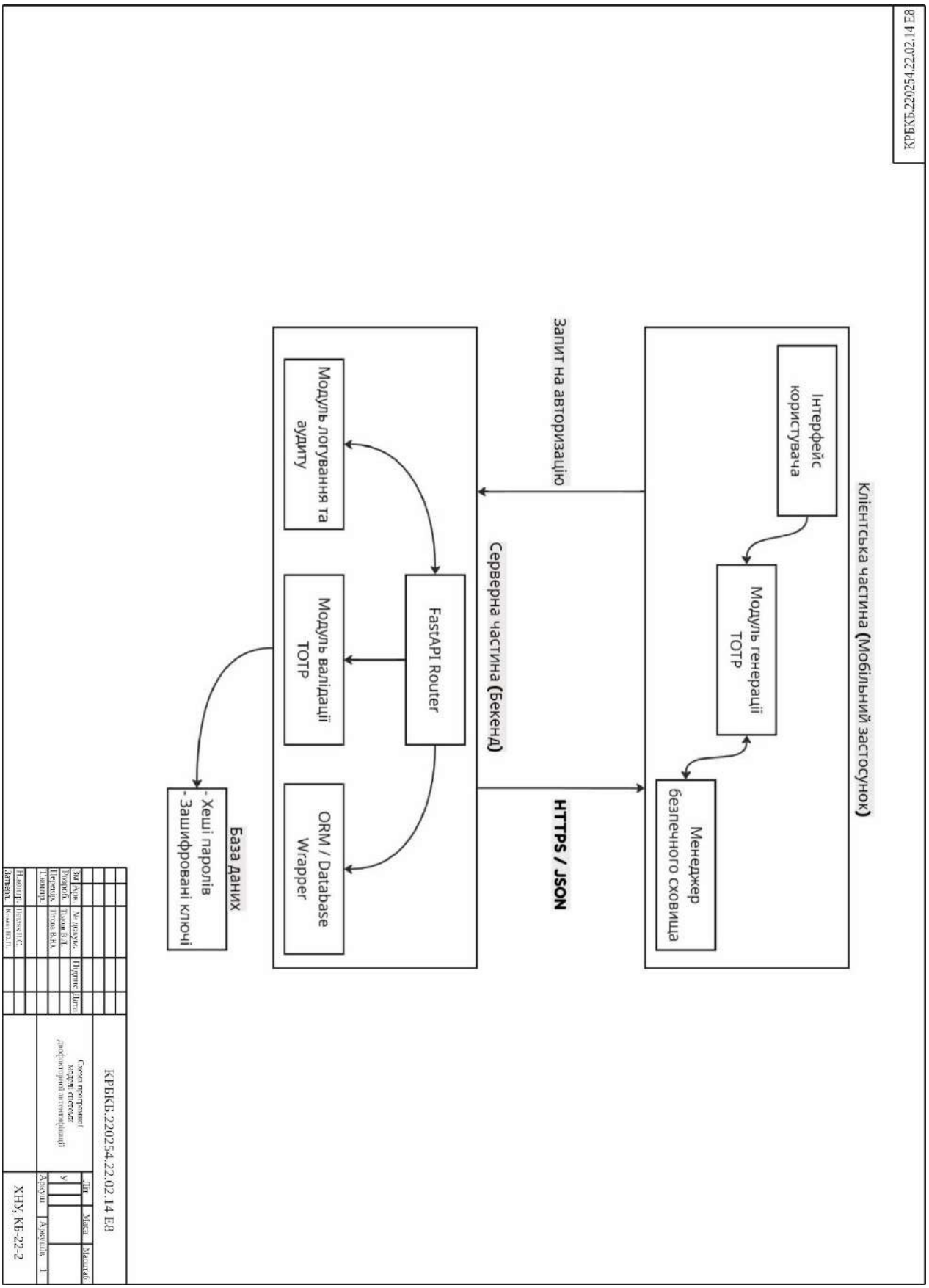
46. Купер А. Інтерфейс. Основи проектування взаємодії. 4-е вид. К. : Фабула, 2022. 720 с.

47. Rate-Limiting Strategies and Techniques. Google Cloud Architecture Center. URL: https://surl.li/ipkovy (дата звернення: 24.04.2026).

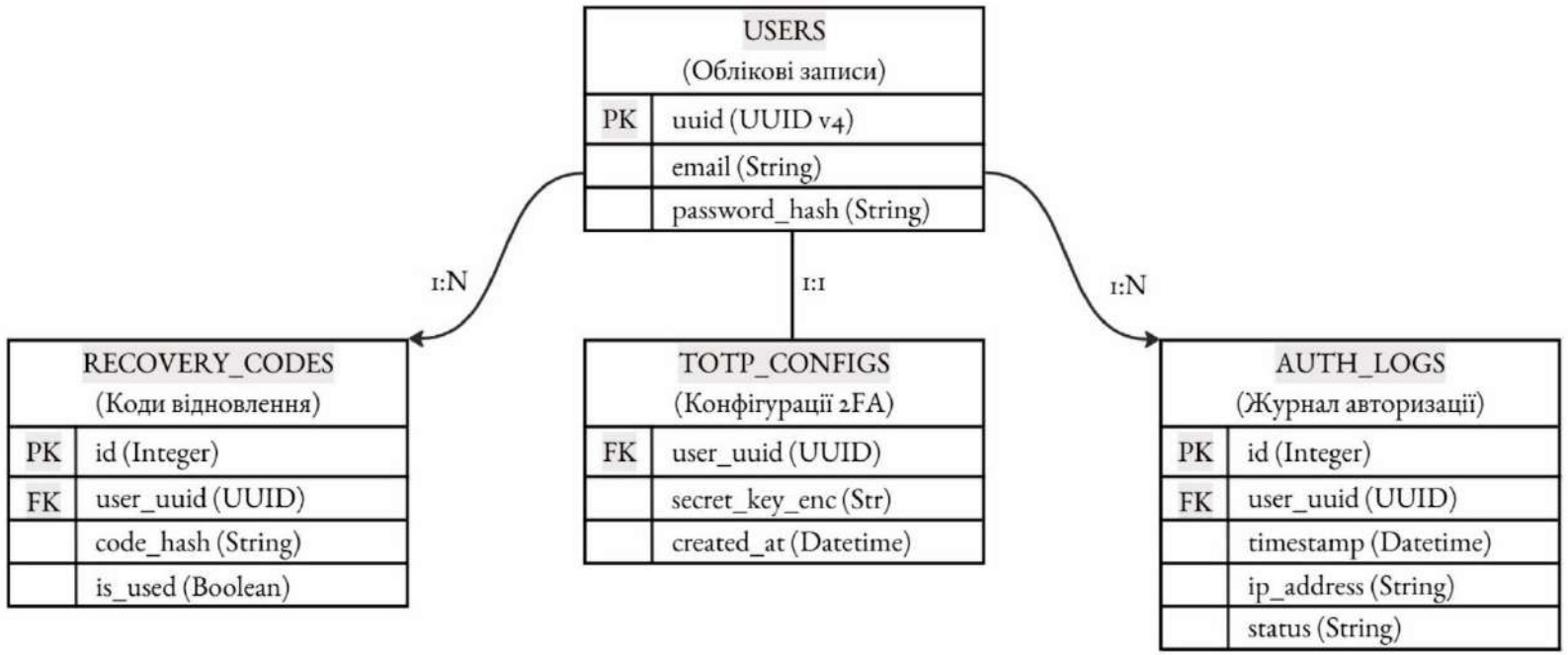
ДОДАТОК А

(обов’язковий)

Копії графічної частини

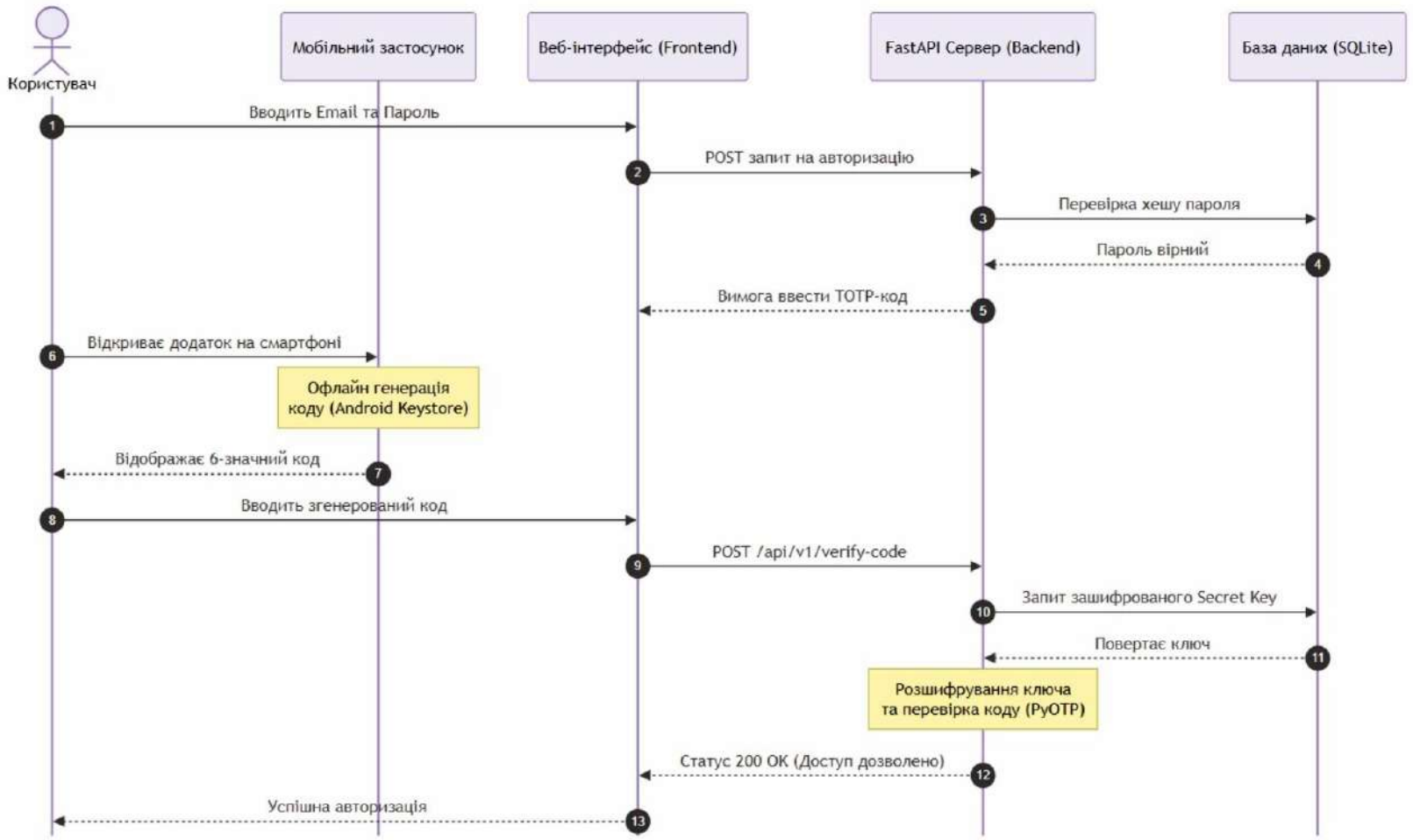


КРБКБ.220254.22.02.14 Е8



					КРБКБ.220254.22.02.14 Е8		
Зм.	Арк.	№ докум.	Підпис	Дата	ER-діаграма реляційної бази даних програмного комплексу	Літ.	Маса	Масштаб
Розроб.		Тимко В.Д.				У		
Перевір.		Гнова В.Ю.				Аркуш	Аркушів 1	
Т.контр.								
Н.контр.		Петлюк Н.С.				ХНУ, КБ-22-2		
Затверд.		Ільєв Ю.П.						

КРБКБ.220254.22.02.14 Е8



					КРБКБ.220254.22.02.14 Е8		
Зм.	Арк.	№ докум.	Підпис	Дата	UML-діаграма послідовності взаємодії компонентів системи під час авторизації	Літ.	Маса	Масштаб
Розроб.		Іванов В.І.				У		
Перевір.		Петров В.О.						
Т.контр.						Аркуш	Аркушів 1	
Н.контр.		Петров В.С.				ХНУ, КБ-22-2		
Затверд.		Коваль Ю.П.						

ДОДАТОК Б

Код застосунку

```
package com.example.diploma2faauth

import android.content.Context
import android.graphics.Color
import android.os.Bundle
import android.util.Base64
import android.view.View
import android.widget.Button
import android.widget.LinearLayout
import android.widget.ProgressBar
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.journeyapps.barcodescanner.ScanContract
import com.journeyapps.barcodescanner.ScanOptions
import dev.turingcomplete.kotlinonetimepassword.HmacAlgorithm
import dev.turingcomplete.kotlinonetimepassword.TimeBasedOneTimePasswordConfig
import dev.turingcomplete.kotlinonetimepassword.TimeBasedOneTimePasswordGenerator
import org.apache.commons.codec.binary.Base32
import java.net.URI
import java.util.Timer
import java.util.concurrent.TimeUnit
import kotlin.concurrent.timerTask
import android.app.AlertDialog
import android.widget.EditText

class MainActivity : AppCompatActivity() {

    private lateinit var layoutList: View
    private lateinit var layoutCode: View
    private lateinit var accountContainer: LinearLayout
    private lateinit var btnAddAccount: Button
    private lateinit var btnBack: Button
    private lateinit var tvActiveAccount: TextView
    private lateinit var tvCode: TextView
    private lateinit var tvTimer: TextView
    private lateinit var progressBar: ProgressBar
    private val keystoreManager = KeystoreManager()
```

```
private var totpTimer: Timer? = null
private val barcodeLauncher = registerForActivityResult(ScanContract()) { result ->
    if (result.contents != null) {
        processScannedQR(result.contents)
    }
}
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    layoutList = findViewById(R.id.layoutList)
    layoutCode = findViewById(R.id.layoutCode)
    accountContainer = findViewById(R.id.accountContainer)
    btnAddAccount = findViewById(R.id.btnAddAccount)
    btnBack = findViewById(R.id.btnBack)
    tvActiveAccount = findViewById(R.id.tvActiveAccount)
    tvCode = findViewById(R.id.tvCode)
    tvTimer = findViewById(R.id.tvTimer)
    progressBar = findViewById(R.id.progressBar)
    btnAddAccount.setOnClickListener {
        val options = arrayOf("📷 Відсканувати QR-код", "✍️ Ввести дані вручну")
        val builder = AlertDialog.Builder(this)
        builder.setTitle("Як ви хочете додати акаунт?")
        builder.setItems(options) { dialog, which ->
            when (which) {
                0 -> {
                    val scanOptions = ScanOptions()
                  scanOptions.setDesiredBarcodeFormats(ScanOptions.QR_CODE)
                    scanOptions.setPrompt("Наведіть камеру на QR-код")
                    scanOptions.setBeepEnabled(false)
                    barcodeLauncher.launch(scanOptions)
                }
                1 -> {
                    val manualBuilder = AlertDialog.Builder(this)
                    manualBuilder.setTitle("Додати новий акаунт")
                    val layout = LinearLayout(this)
                    layout.orientation = LinearLayout.VERTICAL
                    layout.setPadding(50, 40, 50, 10)
                    val emailInput = EditText(this)
                    emailInput.hint = "Email (напр. user@mail.com)"
                    layout.addView(emailInput)
                    val secretInput = EditText(this)
                    secretInput.hint = "Секретний ключ (Base32)"
```

```
                layout.addView(secretInput)

                manualBuilder.setView(layout)
                manualBuilder.setPositiveButton("Зберегти") { manualDialog, _ ->
                    val email = emailInput.text.toString().trim()
                    val secret = secretInput.text.toString().trim().replace(" ", "")
                    if (email.isNotEmpty() && secret.isNotEmpty()) {
                        saveAccount(email, secret)
                        refreshAccountList()
                        Toast.makeText(this, "Акаунт успішно додано!", Toast.LENGTH_SHORT).show()
                    } else {
                        Toast.makeText(this, "Помилка: Заповніть всі поля", Toast.LENGTH_SHORT).show()
                    }
                }
                manualBuilder.setNegativeButton("Скасувати") { manualDialog, _ ->
                    manualDialog.cancel()
                }
                manualBuilder.show()
            }
        }
    }
    builder.show()
  }
  btnBack.setOnClickListener {
    stopCodeGeneration()
    layoutCode.visibility = View.GONE
    layoutList.visibility = View.VISIBLE
  }
  refreshAccountList()
}
private fun processScannedQR(qrContent: String) {
  try {
    val uri = URI(qrContent)
    var accountEmail = uri.path.removePrefix("/")
    if (accountEmail.contains(":")) {
      accountEmail = accountEmail.substringAfter(":")
    }
    val query = uri.query
    val secretPart = query.split("&").find { it.startsWith("secret=") }
    if (secretPart != null && accountEmail.isNotEmpty()) {
      val secretKey = secretPart.substringAfter("secret=")
      saveAccount(accountEmail, secretKey)
```

```
            refreshAccountList()
            Toast.makeText(this, "Акаунт додано!", Toast.LENGTH_SHORT).show()
        }
    } catch (e: Exception) {
        Toast.makeText(this, "Помилка формату QR", Toast.LENGTH_SHORT).show()
    }
}
private fun saveAccount(email: String, secret: String) {
    val sharedPref = getSharedPreferences("SecurePrefs", Context.MODE_PRIVATE)
    val accountsString = sharedPref.getString("account_list", "") ?: ""
    val accountsSet = accountsString.split(",").filter { it.isNotEmpty() }.toMutableSet()
    accountsSet.add(email)
    sharedPref.edit().putString("account_list", accountsSet.joinToString(",")).apply()
    val encryptedData = keystoreManager.encryptData(secret)
    val base64String = Base64.encodeToString(encryptedData, Base64.DEFAULT)
    sharedPref.edit().putString("secret_$email", base64String).apply()
}
private fun refreshAccountList() {
    accountContainer.removeAllViews()
    val sharedPref = getSharedPreferences("SecurePrefs", Context.MODE_PRIVATE)
    val accountsString = sharedPref.getString("account_list", "") ?: ""
    val accountsList = accountsString.split(",").filter { it.isNotEmpty() }
    if (accountsList.isEmpty()) {
        val emptyText = TextView(this)
        emptyText.text = "У вас ще немає доданих акаунтів"
        emptyText.textSize = 16f
        emptyText.setPadding(0, 50, 0, 0)
        accountContainer.addView(emptyText)
        return
    }
    for (email in accountsList) {
        val btn = Button(this)
        btn.text = email
        btn.textSize = 18f
        btn.isAllCaps = false
        btn.setBackgroundColor(Color.parseColor("#E0E0E0"))
        btn.setTextColor(Color.BLACK)

        val params = LinearLayout.LayoutParams(
            LinearLayout.LayoutParams.MATCH_PARENT,
            150
        )
```

```
            params.setMargins(0, 0, 0, 24)
            btn.layoutParams = params
            btn.setOnClickListener {
                openCodeScreen(email)
            }
            accountContainer.addView(btn)
        }
    }
    private fun openCodeScreen(email: String) {
        tvActiveAccount.text = email
        layoutList.visibility = View.GONE
        layoutCode.visibility = View.VISIBLE
        val sharedPref = getSharedPreferences("SecurePrefs", Context.MODE_PRIVATE)
        val base64String = sharedPref.getString("secret_$email", null) ?: return
        val encryptedData = Base64.decode(base64String, Base64.DEFAULT)
        val decryptedSecret = keystoreManager.decryptData(encryptedData) ?: return
        startCodeGeneration(decryptedSecret)
    }
    private fun startCodeGeneration(secretString: String) {
        stopCodeGeneration()
        val base32 = Base32()
        val decodedSecret = base32.decode(secretString)
        val config = TimeBasedOneTimePasswordConfig(
            timeStep = 30,
            timeStepUnit = TimeUnit.SECONDS,
            codeDigits = 6,
            hmacAlgorithm = HmacAlgorithm.SHA1
        )
        val totpGenerator = TimeBasedOneTimePasswordGenerator(decodedSecret, config)
        totpTimer = Timer()
        totpTimer?.scheduleAtFixedRate(timerTask {
            val currentTimeMillis = System.currentTimeMillis()
            val timeStepMillis = 30 * 1000L
            val timeRemaining = 30 - ((currentTimeMillis % timeStepMillis) / 1000).toInt()
            val currentCode = totpGenerator.generate(currentTimeMillis)
            runOnUiThread {
                tvCode.text = currentCode
                tvTimer.text = "$timeRemaining сек"
                progressBar.progress = timeRemaining
            }
        }, 0, 1000)
    }
```

```
        private fun stopCodeGeneration() {
            totpTimer?.cancel()
        }
        override fun onDestroy() {
            super.onDestroy()
            stopCodeGeneration()
        }
}
```

ДОДАТОК В

Код сервера

```
import time
import sqlite3
import pyotp
from fastapi import FastAPI, HTTPException
from fastapi.responses import FileResponse
from pydantic import BaseModel

app = FastAPI(title="Diploma 2FA Server V2")

def init_db():
    with sqlite3.connect("users.db") as conn:
        cursor = conn.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS users (
                email TEXT PRIMARY KEY,
                secret_key TEXT NOT NULL
            )
        """)
        cursor.execute(
            "INSERT OR REPLACE INTO users VALUES (?, ?)",
            ("vlad.timosh@example.com", "FMNBLKQ7CTEHH7GXDUWGE22G7GKDDRGN")
        )
        conn.commit()
init_db()

def save_user_key(email: str, secret: str):
    with sqlite3.connect("users.db") as conn:
        cursor = conn.cursor()
        cursor.execute("INSERT OR REPLACE INTO users VALUES (?, ?)", (email, secret))
        conn.commit()

def get_user_key(email: str):
    with sqlite3.connect("users.db") as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT secret_key FROM users WHERE email = ?", (email,))
        result = cursor.fetchone()
        return result[0] if result else None
```

```
class RegisterRequest(BaseModel):
    email: str

class VerifyRequest(BaseModel):
    email: str
    code: str
@app.post("/api/v1/generate-key")
async def generate_key(request: RegisterRequest):
    existing_secret = get_user_key(request.email)

    if existing_secret:
        secret = existing_secret
        message = "Цей користувач уже зареєстрований. Повертаємо існуючий ключ."
    else:
        secret = pyotp.random_base32()
        save_user_key(request.email, secret)
        message = "Новий ключ згенеровано та збережено в базі."
    totp = pyotp.TOTP(secret)
    qr_uri = totp.provisioning_uri(name=request.email, issuer_name="Diploma_Security_System")
    return {
        "status": "success",
        "message": message,
        "secret_key": secret,
        "qr_uri": qr_uri
    }
@app.post("/api/v1/verify-code")
async def verify_code(request: VerifyRequest):
    secret = get_user_key(request.email)

    if not secret:
        raise HTTPException(status_code=404, detail="Користувача не знайдено")

    totp = pyotp.TOTP(secret)

    if totp.verify(request.code, valid_window=1):
        return {"status": "success", "message": "Авторизація успішна! Код вірний."}
    else:
        raise HTTPException(status_code=400, detail="Невірний код або час його дії минув.")

@app.get("/")
async def root():
return FileResponse("index.html")
```