ПРИВАТНИЙ ЗАКЛАД ВИЩОЇ ОСВІТИ «ІТ СТЕП Університет»

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи бакалавра

Андросов Нікіта Сергійович

студент групи 4CS-42, спеціальності 122 Комп'ютерні науки

Тема:

РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБЕРЕЖЕННЯ ТА ШИФРУВАННЯ ДАНИХ ІЗ ВИКОРИСТАННЯМ БЛОКЧЕЙН-ТЕХНОЛОГІЙ

Науковий керівник: Райтер О.К., PhD _____

Консультант: Слобода Л.Я., доцент, к.е.н. _____

Кваліфікаційна робота захищена з оцінкою «_____»

Секретар ЕК _____

«____» _____20__ р.

Львів – 2025

ПРИВАТНИЙ ЗАКЛАД ВИЩОЇ ОСВІТИ «ІТ СТЕП Університет»

ЗАТВЕРДЖУЮ
Проректор з науково-педагогічної роботи

_____
“____” _____20___року

З А В Д А Н Н Я
НА ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Освітній рівень – бакалавр
Спеціальність - 122 Комп’ютерні науки

Андросов Нікіта Сергійович
(прізвище, ім’я, по батькові)

1. Тема роботи *Розроблення програмного забезпечення для збереження та шифрування даних із використанням блокчейн-технологій*
Науковий керівник *Райтер Орест Костянтинович, PhD*
2. Вихідні дані до роботи: *тестові файли різних розмірів та різних форматів для шифрування і дешифрування; тестові мережі блокчейн; матеріали про блокчейн-технології, методи шифрування; технічна документація.*
3. Мета та завдання дослідження:
Мета роботи *розроблення програмного забезпечення для надійного зберігання та шифрування даних із використанням блокчейн-технологій для розподілу та управління шифрувальними ключами, що забезпечить захист конфіденційної інформації, контроль доступу та інтеграцію сучасних методів криптографії в процес управління персональними даними.*
Завдання дослідження: *аналіз сучасних рішень; формування вимог до системи; розроблення архітектури системи; розроблення смарт-контракту; реалізація шифрування даних; розроблення інтерфейсу користувача; тестування функціоналу програмного забезпечення.*
4. Об’єкт і предмет дослідження:
Об’єкт дослідження: *системи зберігання та захисту конфіденційної інформації за допомогою шифрування й блокчейн-технологій.*
Предмет дослідження: *реалізація шифрування файлів, генерація, розподіл і управління шифрувальними ключами через блокчейн, контроль доступу, а також інтеграція цих технологій у програмне забезпечення для забезпечення надійного захисту конфіденційних даних.*

5. Методи дослідження: *метод аналізу та порівняння; метод прототипування; метод криптографічного моделювання; емпіричні методи тестування: метод інфраструктурне моделювання; економічний аналіз.*

6. Очікувані результати: *програмне забезпечення для зберігання, шифрування та дешифрування даних, смарт-контракт для розподілу та управління шифрувальними ключами в блокчейні.*

7. Інструментарій: *Python, Solidity, FastAPI, PyCryptodome, Web3.py, Tkinter, pytest, AWS Nitro Enclaves, Terraform, AWS EC2, SQLite, Polygon Amoy, SRP, Remix.*

8. Терміни виконання:

Вид діяльності	Термін виконання, до	Відмітка про виконання	Підпис керівника
Затвердження теми роботи	01.12.2024	Виконано	
Підготовка огляду літератури	20.12.2024	Виконано	
Розробка методології	01.01.2025	Виконано	
Тестування та впровадження	01.02.2025	Виконано	
Написання та оформлення роботи	10.03.2025	Виконано	

9. Додаткові вимоги:_____
(За необхідності додаються вимоги від наукового керівника чи керівника освітньої програми).

_____

_____

Графік узгоджено «___» _____ 20__ р.

Здобувач _____
(Підпис)

Науковий керівник _____
(Підпис)

АНОТАЦІЯ

Дана робота полягає у створенні безпечного програмного забезпечення для локального шифрування, дешифрування та зберігання даних користувача, в основі якого лежить поєднання хмарних сервісів, блокчейн-технологій та сучасних методів криптографії. Основною метою є усунення недоліків централізованих систем зберігання секретної інформації шляхом використання децентралізованої архітектури з високим рівнем ізоляції.

Для реалізації проєкту використано Python як основну мову для серверної та клієнтської частин, Bash і HCL для налаштування інфраструктури, Solidity для розроблення смарт-контракту. Архітектура побудована на основі сервісів AWS (зокрема EC2 з підтримкою Nitro Enclaves) для забезпечення апаратної ізоляції та підвищеного рівня безпеки. Інфраструктура як код реалізована через Terraform. Для побудови API використано FastAPI, а для взаємодії з блокчейном – Web3.py. Криптографічні операції виконуються за допомогою AES-256 і Shamir’s Secret Sharing Scheme.

На клієнтській стороні використано Tkinter для побудови графічного інтерфейсу та SQLite для локального зберігання даних. Таким чином, робота об’єднує апаратні засоби захисту, хмарні та блокчейн-технології, сучасну криптографію.

ABSTRACT

This work is aimed at creating secure software for local encryption, decryption, and storage of user data based on a combination of cloud services, blockchain technologies, and modern cryptography methods. The main goal is to eliminate the disadvantages of centralized systems for storing sensitive information by using a decentralized architecture with a high level of isolation.

To implement the project, Python was used as the main language for the server and client parts, Bash and HCL for infrastructure configuration, and Solidity for smart contract development. The architecture is based on AWS services (in particular, EC2 with Nitro Enclaves support) to provide hardware isolation and increased security. The infrastructure as code is implemented through Terraform. FastAPI is used to build the API, and Web3.py is used to interact with the blockchain. Cryptographic operations are performed using AES-256 and Shamir's Secret Sharing Scheme.

On the client side, Tkinter is used to build a graphical interface and SQLite for local data storage. Thus, the work combines hardware security, cloud and blockchain technologies, and modern cryptography.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

AES – Advanced Encryption Standard (стандарт симетричного шифрування).

SSSS – Shamir's Secret Sharing Scheme (схема розподілу секретів Шаміра).

RSA – Rivest–Shamir–Adleman (асиметричний криптографічний алгоритм).

ECC – Elliptic Curve Cryptography (криптографія на еліптичних кривих).

SHA – Secure Hash Algorithm (хеш-функції).

TLS – Transport Layer Security (протокол захисту переданих даних).

FIPS – Federal Information Processing Standards (федеральні стандарти обробки інформації).

HSM – Hardware Security Module (апаратний модуль безпеки).

AWS – Amazon Web Services (хмарні сервіси Amazon).

EC2 — Elastic Compute Cloud (віртуальні сервери AWS).

vsock – Virtual Socket (віртуальні сокети для комунікації).

Nitro Enclaves – технологія апаратної ізоляції AWS.

Zero Trust – архітектурний принцип нульової довіри.

Defense in Depth – багатошарова оборона (захист у глибину).

CTR – Counter Mode (режим лічильника, спосіб роботи блочного шифру).

ZKP – Zero-Knowledge Proof (доказ з нульовим розголошенням).

SRP – Secure Remote Password protocol (протокол безпечного підтвердження паролю).

PoS – Proof of Stake (механізм консенсусу в блокчейнах).

API – Application Programming Interface (інтерфейс прикладного програмування).

REST – Representational State Transfer (архітектурний стиль для API).

CID – Context ID (ідентифікатор анклаву Nitro Enclaves).

JSON – JavaScript Object Notation (формат передачі даних).

HCL – HashiCorp Configuration Language (мова для конфігурації Terraform).

ВСТУП

За останні роки кількість інформації в діджитал просторі стрімко зростає, майже кожна людина має телефон або персональний комп'ютер. В бази даних по всьому світу постійно йдуть мільйони запросів на зберігання паролів, приватних даних або клієнтських баз. Це, в свою чергу, означає, що ця інформація є дуже цінною і може спонукати тих самих хакерів на спроби дістати ці данні.

Звісно, для запобігання втручання в особисте життя людей та компаній було створено та створюються різні системи захисту даних – локальні, хмарні, з відкритим кодом або закритим, платні та безкоштовні, більшість з них доволі зручні та надійні, але у своєму традиційному вигляді вони створюють слабкі місця (single point of failure), при зламі яких, уся система є скомпрометованою, дані викрадені та репутація зруйнована, якщо проблема сталась на рівні компанії. У випадку ж якщо рішення для захисту надає високий рівень безпеки, все рівно користувачі мають сліпо довіряти провайдеру або розробнику із закритим кодом без можливості власної перевірки.

Саме тому за останні роки активно розвиваються технології з різних галузей IT-індустрії, які надають нові підходи до захисту даних з переосмисленням традиційних систем, надаючи безпеку навіть у закритих системах. Серед них – блокчейн, хмарні технології, апаратна ізоляція або навіть алгоритми, які постійно вдосконалюються та перевіряються на дієздатність навіть проти квантових комп'ютерів. Таким чином

Мета роботи: розроблення програмного забезпечення для надійного зберігання та шифрування даних із використанням блокчейн-технологій для розподілу та управління шифрувальними ключами, для надання захисту конфіденційної інформації, контролю доступу та інтеграцію сучасних методів криптографії в процес управління персональними даними.

Об'єкт дослідження: системи зберігання та захисту конфіденційної інформації за допомогою шифрування й блокчейн-технологій.

Предмет дослідження: реалізація сучасних криптографічних підходів для шифрування файлів, генерації, розподілу й управління ключами, а також інтеграція блокчейн- та хмарних технологій у програмне забезпечення.

Методи дослідження: метод аналізу та порівняння; метод прототипування; метод криптографічного моделювання; емпіричні методи тестування: метод інфраструктурного моделювання; економічний аналіз.

Основні результати кваліфікаційної роботи було апробовано на IV студентській науково-практичній конференції «Прикладні комп'ютерні науки», 25 квітня 2025, IT СТЕП Університет (м. Львів) [17].

РОЗДІЛ 1 АНАЛІТИЧНИЙ ОГЛЯД ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Огляд систем захисту конфіденційних даних

Сьогодні, у цифрову епоху, кількість конфіденційної інформації, яку обробляють і зберігають, дуже швидко зростає і це стосується не лише особистих даних користувачів, таких як файли, паролі, медичні чи фінансові записи, а й корпоративної інформації – наприклад, документів, даних про клієнтів і звітів. І захист цих даних стає головним пріоритетом як для компаній так і для окремих користувачів.

1.1.1 Проблеми централізованого зберігання конфіденційних даних

Більшість сучасних систем захисту даних працюють на основі централізованих сховищ. Це означає, що всі паролі або ключі для шифрування, від яких залежить безпека всієї системи, зберігаються на окремих серверах або в дата-центрах. Такий підхід має свої плюси: легше керувати системою, простіше робити резервні копії і контролювати доступ, а також це дешевше з точки зору технічного обслуговування.

Але у такого способу зберігання є й серйозний мінус – це одна точка, через яку може статися витік або злом. За даними звіту IBM Cost of a Data Breach Report 2024, середня вартість витоку даних склала 4,88 мільйона доларів США. У дослідженні було проаналізовано 604 компанії, які пережили витік даних у період з березня 2023 до лютого 2024 року [1]. Навіть якщо сервери добре захищені, вони все одно можуть постраждати через вразливості в програмному забезпеченні, фішингові атаки або неправильне використання прав адміністраторів.

Ще одна проблема централізованого зберігання – це те, що користувачі змушені повністю довіряти постачальнику послуг, бо вони не можуть перевірити, як саме їхні дані зберігаються. Це суперечить принципу мінімізації довіри, який дуже важливий для безпеки.

1.1.2 Переваги використання блокчейн-технологій

За останні десять років блокчейн-технології значно розвинулися і вже давно перестали бути просто основою для криптовалют. Сьогодні їх використовують у різних сферах: управління даними, голосування, логістиці, фінансах, а також для захисту конфіденційної інформації. Головна перевага блокчейну – це його децентралізованість: дані зберігаються не на одному сервері, а розподіляються між багатьма вузлами, тому зламати всю систему через один сервер практично неможливо.

Що стосується зберігання конфіденційних даних, це означає, що навіть якщо зламають один або кілька вузлів, безпека всієї системи не постраждає. Крім того, блокчейн гарантує прозорість і незмінність записів, тобто ніхто не може просто так змінити або видалити інформацію.

Серед популярних платформ для таких рішень варто згадати Ethereum і Polygon, які підтримують смарт-контракти – маленькі програми, що автоматично виконують певні дії з управління даними.

1.1.3 Огляд хмарних рішень для ізольованої генерації ключів шифрування

Сьогодні, коли говоримо про захист даних, важливо розуміти не лише, де вони зберігаються, але й де й як генеруються ключі для шифрування. Якщо робити це на звичайних серверах, є ризик, що хтось отримає доступ до операційної системи й перехопить ці ключі. Тому виникла потреба в спеціальних рішеннях, які дозволяють робити такі операції в ізольованих середовищах, куди ніхто не має доступу.

Серед найпопулярніших рішень – Intel SGX (Software Guard Extensions). Це спеціальна технологія, яка дозволяє створювати всередині процесора так звані анклави – маленькі ізольовані частини пам’яті, де можна безпечно виконувати обчислення. Навіть операційна система комп’ютера не може побачити, що відбувається всередині. SGX дуже популярний у фінансових компаніях та стартапах, які працюють із чутливими даними.

Ще одна відома технологія – AMD (Secure Encrypted Virtualization). Вона дозволяє шифрувати пам’ять віртуальних машин, щоб навіть адміністратори хост-машини не мали доступу до даних, що обробляються всередині VM. Це зручно, якщо компанія використовує сервери AMD і хоче бути впевненою, що її віртуальні середовища залишаються захищеними.

Для тих, хто використовує хмарні сервіси, є AWS Nitro Enclaves – це рішення від Amazon, яке дозволяє створювати спеціальні ізольовані середовища всередині EC2-інстансів. Дані можна передавати в ці анклави через особливий захищений канал vsock, а ключі, які генеруються всередині, ніколи не покидають межі анклаву.

Такі рішення використовують там, де потрібна максимальна безпека: фінтех, охорона здоров’я, державні проєкти, а також у стартапах, що працюють з криптовалютами або блокчейн-рішеннями.

1.1.4 Аналіз конкурентних рішень

На ринку існує кілька категорій конкурентів, які вирішують завдання захисту даних.

По-перше, це локальні програми для шифрування, такі як VeraCrypt або Windows BitLocker. Вони забезпечують високий рівень захисту даних на диску, але при цьому користувачі самостійно відповідають за зберігання ключів, що потенційно створює ризики втрати доступу. Також, мають обмежені можливості синхронізації між пристроями та відсутній централізований аудит безпеки.

По-друге, існують хмарні сервіси, наприклад, Google Drive чи Dropbox, які забезпечують синхронізацію й резервне копіювання, але не гарантують шифрування даних. Їх структура зберігання є повністю централізованою, вимагають довіри до провайдера.

Третя категорія – рішення з управління ключами, такі як AWS KMS або HashiCorp Vault. З плюсів – вони надають корпоративні рішення для управління секретними даними та дуже високу безпеку, але з мінусів – знову ж таки, повністю централізоване сховище, значні витрати на інфраструктуру та

обслуговування разом із високою складністю налаштування та подальшої підтримки та адміністрування.

Підводячи підсумок до розглянутих рішень, можна виділити такі загальні недоліки:

- відсутній комплексний підхід до безпеки даних;
- недостатня прозорість процесів шифрування та зберігання;
- компроміс між зручністю та рівнем захисту;
- вразливість до компрометації серверної інфраструктури;

Таким чином, комбінований підхід, що використовує локальне шифрування, повністю апаратно-ізольовані середовища для роботи з конфіденційною інформацією та блокчейн для розподіленого зберігання, забезпечуючи багаторівневий захист, вигідно виділяється на фоні аналогів.

1.2 Вимоги до безпечної архітектури шифрування даних

Щоб система була надійною, потрібно використовувати сучасні криптографічні алгоритми, які справді захищають дані. Не можна застосовувати застарілі методи, такі як DES або MD5, бо вони вже давно не вважаються безпечними. Замість них краще використовувати перевірені стандарти: AES з ключем мінімум 128 біт, а краще 256 біт для симетричного шифрування; RSA з ключем не менше 2048 біт або алгоритми на еліптичних кривих, наприклад Curve25519, для асиметричного шифрування; а для хешування – SHA-2 або SHA-3. Також важливо використовувати сучасні протоколи, наприклад TLS 1.3, щоб безпечно передавати дані.

Особливу увагу варто приділяти режимам шифрування. Наприклад, AES-GCM або AES-CCM - це режими, які не тільки шифрують дані, а й перевіряють їх цілісність, тобто гарантують, що інформація не була змінена.

Більш того, важливо не вигадувати власні алгоритми шифрування, бо вони зазвичай виявляються небезпечними без тривалого тестування. Краще дотримуватися стандартів, як FIPS 140-3, які підтверджують, що алгоритми відповідають сучасним вимогам безпеки.

1.2.1 Управління ключами

Навіть найкраща криптографія не допоможе, якщо ключі шифрування будуть скомпрометовані, через це з'являється необхідність у наступних методологіях:

- генерувати ключі за допомогою надійних генераторів випадкових чисел;
- зберігати ключі тільки в захищених місцях;
- мінімізувати кількість копій ключів і контролювати, хто до них має доступ;
- регулярно оновлювати ключі;
- робити резервні копії ключів.

1.2.2 Архітектурні принципи

У сучасних системах захисту даних принцип Zero Trust став справжнім стандартом. Сьогодні безпека починається з припущення, що навіть внутрішнім компонентам не можна автоматично довіряти – кожен запит має бути перевірений, кожна дія має бути обґрунтованою, а всі дані – зашифрованими.

Що означає Zero Trust на практиці? Передусім, це перевірка ідентичності кожного користувача, пристрою або мікросервісу. Тут важливу роль грає багатофакторна аутентифікація, перевірка сертифікатів і цифрових підписів, а також аналіз контексту доступу – наприклад, з якого пристрою чи локації надходить запит. Крім того, діє правило мінімальних прав (principle of least privilege) – кожен компонент системи отримує лише той рівень доступу, який потрібен йому для конкретної задачі, що дозволяє значно знизити ризики у разі атаки або помилки.

Ще одна важлива річ – сегментація системи. Ми розділяємо її на окремі сегменти і встановлюємо між ними чіткі бар’єри доступу. Наприклад, навіть якщо одна частина скомпрометована, це не означає, що зловмисники автоматично отримають доступ до решти системи. Плюс, обов’язковим

елементом є постійний моніторинг і логування – система має фіксувати всі дії, щоб вчасно виявляти підозрілу активність.

Також варто згадати концепцію Defense in Depth. Суть її проста – система не покладається на один захисний бар'єр, а створює кілька рівнів захисту. Наприклад, навіть якщо сервер зламаний або злили окремі ключі, інші шари системи не дозволять зловмисникам повністю захопити контроль. Це може включати в себе й апаратну ізоляцію, й управління ключами, й спеціалізовані протоколи шифрування.

1.2.3 Ізольовані середовища

Важливим елементом побудови безпечної архітектури є ізоляція середовищ, де зберігаються ключі або виконуються криптографічні операції. Якщо такої ізоляції немає, критичні дані можуть опинитися під загрозою, адже у разі компрометації будь-якого компоненту системи (операційної системи, додатка чи навіть адміністратора) зловмисники можуть отримати до них доступ.

Для захисту від таких ризиків використовують спеціальні апаратні або програмні рішення, які створюють окремі середовища з ізольованою пам'яттю та обчислювальними ресурсами. Наприклад HSM забезпечують проведення всіх криптографічних операцій всередині пристрою, не виводячи ключі назовні й у випадку спроб фізичного втручання такі модулі здатні автоматично знищувати ключі.

Подібні принципи реалізують і хмарні рішення, де використовуються механізми, що розділяють ресурси таким чином, що навіть за компрометації основного інстансу чи операційної системи дані та ключі залишаються захищеними. Це доволі розповсюджений підхід і він дозволяє значно зменшити ризики як зовнішніх атак, так і внутрішніх помилок в системі.

Додати можна, що сучасні ізольовані середовища мають цікаву функцію, яка активно впроваджується, і це технологія віддаленої атестації, вона дозволяє

клієнту переконуватись в правдивості коду, що виконується віддалено на сервері.

Висновки до першого розділу

У першому розділі я розглянув основні підходи до захисту конфіденційних даних, зокрема проблеми централізованих рішень, які створюють ризик єдиної точки зламу. Було проаналізовано переваги блокчейн-технологій, які завдяки децентралізованій природі й використанню смарт-контрактів дозволяють підвищити надійність систем.

Також я окреслив огляд сучасних хмарних рішень для ізольованої генерації й обробки ключів, а саме апаратних та програмних методів, що дозволяють мінімізувати ризики компрометації даних. Було проведено аналіз конкурентних продуктів, де стало очевидно, що комбінований підхід, що поєднує локальне шифрування, блокчейн і апаратну ізоляцію, на сьогодні є одним із найефективніших.

Окремий фокус розділу присвячений вимогам до безпечної архітектури: вибору перевірених алгоритмів шифрування, правильному управлінню ключами, застосуванню принципів Zero Trust та Defense in Depth. Разом ці елементи формують міцну основу для створення захищених систем, які здатні протистояти сучасним загрозам.

РОЗДІЛ 2 ЗАГАЛЬНА МЕТОДИКА, ОСНОВНІ МЕТОДИ ДОСЛІДЖЕНЬ

2.1 Використані програмні інструменти

Для побудови системи, при виборі мов програмування та інструментів, я в першу чергу покладався на широко використовувані, перевірені та зручні технології саме для поставленої задачі.

Основна мова – це Python. Обрав я її, тому що вона є дуже популярною, має багато готових бібліотек для різних задач, наприклад для того ж шифрування PyCryptodome, для побудови серверу FastAPI та для зв'язку з блокчейном Web3.py (хоча Web3.js вважається більш масштабною бібліотекою для веб3 розробки). Крім того, на Python швидко можна створювати прототипи й перевіряти свої ідеї.

Для серверної частини я використовую FastAPI – це фреймворк, який дозволяє швидко створювати API, що працюють відповідно до назви фрейморку, дуже швидко. Щоб розгорнути сервер всередині EC2 і налаштувати самі EC2 разом з інфраструктурою, я застосував Terraform, тому що з ним можна прописати інфраструктуру як код і потім легко її масштабувати в AWS.

Одною з найважливіших деталей роботи є використання AWS Nitro Enclaves як ядра системи. Анклави доволі зручно інтегруються у AWS архітектуру і не потребують окремих HMS, спрощуючи розгортання системи нативно до стеку.

Для роботи з блокчейном я використовував Solidity для написання смарт-контрактів, які розгортаються у мережі Polygon. Смарт-контракти потрібні, щоб зберігати розділені частини ключів на блокчейні. Зв'язок серверу зі смарт-контрактами відбувається через бібліотеку Web3.py.

Клієнтська частина зроблена з використанням Tkinter – це проста бібліотека для створення графічного інтерфейсу як на Windows так і на MacOS/Linux. Завдяки клієнту користувач може шифрувати й дешифрувати свої файли локально. Для збереження метадати зашифрованих файлів я обрав

мінімалістичну SQLite.

Для безпечної аутентифікації я використав протокол SRP, який дозволяє підтверджувати знання пароля без його передачі. Це не класичний ZKP, але працює за схожим принципом: одна сторона доводить іншій, що вона знає пароль, не розкриваючи його. Щоб перевірити, що все працює правильно, я додав автоматичні тести з використанням pytest – популярного фреймворку для тестування на Python.

Загалом, я вибрав ці інструменти, бо вони добре підходять для захищеного шифрування, роботи з блокчейном і хмарою, а також дозволяють зручно працювати з програмою як користувачу, так і розробнику.

2.2 Теоретичні основи криптографії

Коли ми говоримо про захист даних, то перше, що приходить на думку, – це шифрування. Без шифрування захистити файли чи особисту інформацію практично неможливо. У цьому розділі я розповім про два ключові елементи, які використовуються в моєму проєкті – це AES-256 і SSSS.

2.2.1 AES-256

AES – це один із найпопулярніших алгоритмів симетричного шифрування у світі. «Симетричне» означає, що один і той самий ключ використовується як для шифрування, так і для дешифрування даних. Це дуже зручно, бо не треба вигадувати дві різні частини ключа, але є й при цьому цей ключ треба розумно захищати, адже у випадку його крадіжки зловмисник зможе розшифрувати всі зашифровані дані.

Число 256 у AES-256 означає довжину ключа 256 біт. Це доволі надійний рівень безпеки, який зараз вважається стандартом навіть для державних структур. Наприклад, 128-бітний ключ також досить сильний, але 256-бітний вважається ще більш стійким до зламу. AES працює блоками даних (по 128 біт) і обробляє їх за допомогою різних раундів шифрування, які включають заміни,

перестановки й підмішування ключів. Це робить шифрування майже неможливим для злому навіть за допомогою суперкомп'ютерів.

Однією з переваг AES є його швидкість. Він добре працює як на звичайних комп'ютерах, так і на мобільних пристроях, а також підтримується апаратно, тобто процесор сам вміє його обробляти без навантаження на програму. В моїй роботі AES-256 використовується як на клієнтській так і на серверній частині системи [2].

2.2.2 Shamir's Secret Sharing Scheme

А от як захистити сам ключ шифрування? Для цього використовується не менш цікаве рішення – SSSS. Ця ідея належить Адді Шаміру, одному з творців криптографії. Суть у тому, що секрет (наприклад, ключ шифрування) можна поділити на кілька частин, і для того, щоб відновити цей секрет, потрібно зібрати лише певну мінімальну кількість частин. Це називається порогова схема.

Уявіть собі ситуацію: є 5 частин ключа, а для того, щоб його відновити, потрібно будь-які 3 з них. Це зручно, бо навіть якщо одна або дві частини загубляться або хтось не захоче співпрацювати, ми все одно зможемо зібрати секрет. Але якщо зібрати менше ніж 3 частини – нічого не вийде, бо кожна частина окремо нічого не означає.

SSSS працює на основі математики багаточленів і інтерполяції. Секрет задається як значення у нулі деякої випадкової функції – тобто, це просто f(0). Потім генерується кілька частин, які є точками на цій функції, наприклад (1, f(1)), (2, f(2)) і т.д. Якщо зібрати достатньо таких точок, то можна відновити сам багаточлен і дізнатися f(0), тобто секрет. Без достатньої кількості точок відновити нічого не вийде. У моєму проєкті ця схема потрібна для того, щоб ключ користувача, який генерується всередині ізольованого середовища, був поділений на частини й записаний у децентралізоване сховище блокчейн. Це дозволяє уникнути ситуації, коли один єдиний злам або витік дасть доступ до всіх зашифрованих даних [3].

Таке поєднання AES-256 і SSSS – це класичний приклад того, як сучасна криптографія може захистити як дані, так і ключі від злому. Навіть якщо зловмисники отримають доступ до однієї частини системи, без повного набору ключових компонентів вони нічого не зможуть зробити. Такий підхід допомагає будувати багаторівневий захист, який сьогодні стає стандартом у багатьох серйозних IT-рішеннях.

2.3 Використання блокчейн-мережі Polygon для зберігання ключів

У даному проєкті я використовую блокчейн Polygon Amoy – це тестова мережа, яка спеціально створена для того, щоб розробники могли перевіряти, як працюють їхні смарт-контракти, не витрачаючи справжні гроші. Вона поводиться дуже схоже до реального блокчейну, але замість реальних монет використовуються тестові токени. Якби система запускалась у продакшн, я б використовував Polygon PoS – це основна мережа, яка вже обробляє справжні транзакції з реальними користувачами.

Щоб розуміти, чому саме Polygon, слід коротко розглянути принцип роботи блокчейну. Блокчейн – це послідовний ланцюг блоків, кожен з яких містить дані (наприклад, транзакції), захищені криптографічно та пов’язані з попереднім блоком. Головна ідея в тому, що цей ланцюжок розподілений між багатьма комп’ютерами (вузлами), і всі ці вузли разом підтверджують, що дані в ланцюжку справжні. Це робить блокчейн дуже захищеним від підробки.

Polygon працює на основі алгоритму Proof of Stake (PoS). У такій системі замість того, щоб витрачати величезну кількість обчислювальних ресурсів, як це робить Bitcoin, валідатори (учасники мережі) «заморожують» певну кількість своїх монет (це називається stake) і використовують їх як гарантію чесної роботи. Якщо вони будуть намагатися шахраювати, їхній stake можуть забрати. Завдяки цьому PoS є більш швидким і дешевим способом захисту мережі. Polygon – це ще й Layer 2-рішення, тобто воно працює поверх основного блокчейну Ethereum. Це дозволяє йому залишатися сумісним з Ethereum, але бути набагато дешевшим і швидшим у роботі [4].

У моєму проєкті Polygon використовується для зберігання частин шифрувальних ключів. Коли користувач реєструється, система генерує для нього спеціальний майстер-ключ, щоб не зберігати цей ключ повністю в одному місці, я розбиваю його на частини за допомогою алгоритму SSSS як було зазначено раніше.

Після розділення ключа його частини (шарди) записуються у смарт-контракт на Polygon.

Коли користувач потім хоче отримати свій ключ назад, наприклад, щоб розшифрувати дані, система надсилає запити до смарт-контракту, збирає потрібну кількість шардів і відновлює оригінальний ключ. Дуже важливо, що весь цей процес відбувається децентралізовано – дані не залежать від одного сервера чи однієї компанії, а розподілені між багатьма учасниками мережі.

Щоб інтегрувати роботу з Polygon у свій Python-код, я використовую бібліотеку Web3.py. Вона дозволяє моїй програмі спілкуватися з блокчейном, а саме надсилати транзакції, викликати функції смарт-контракту, перевіряти вартість транзакцій для вибору найбільш оптимальної комісії, тощо.

Завдяки використанню Polygon система отримує не тільки захищене сховище, а й швидкість та низькі комісії. Тут важливо ще відзначити, що мій сервер реалізує так звану абстракцію акаунту (account abstraction). Це означає, що користувачам не потрібно мати власний криптогаманець або самостійно проводити транзакції в блокчейні, єдиний гаманець системи автоматично виконує всі операції запису шардів у блокчейн.

Завдяки цьому користувачі взагалі не стикаються з технічними деталями, такими як підписання транзакцій або оплата газу. Крім того, за рахунок оптимізації логіки смарт-контракту та особливостей мережі Polygon комісії за запис (зберігання частин ключа) дуже низькі – часто менше одного цента. А от зчитування даних із блокчейну є повністю безкоштовним, бо воно не змінює стан мережі й не створює платних транзакцій. Це робить рішення зручним, доступним і дешевим для кінцевих користувачів, яким важлива простота використання без глибокого занурення у технічні нюанси.

2.4 Застосування AWS Nitro Enclaves для захищених обчислень

AWS Nitro Enclaves – це не просто ізольоване середовище, а технологія, що побудована на основі AWS Nitro Hypervisor, спеціально розробленого для досягнення високого рівня безпеки та продуктивності. Що відрізняє Nitro Enclaves від звичайних HSM, так це поєднання апаратної ізоляції, гнучкості хмарних сервісів і масштабованості.

На технічному рівні Nitro Enclaves створюються шляхом виділення частини ресурсів (CPU, RAM) EC2-інстансу в окремий ізольований блок, що працює без постійного сховища, мережевих інтерфейсів чи навіть доступу до Amazon EC2 Instance Metadata Service. Це означає, що всередині анклаву немає традиційних каналів зв’язку із зовнішнім світом, окрім спеціального протоколу vsock, який дозволяє контролювати, які дані передаються всередину анклаву та назад.

Однією з ключових особливостей Nitro Enclaves є інтеграція з AWS KMS через механізм підписів, що дозволяє виконувати криптографічні операції без необхідності отримувати доступ до самих ключів. Наприклад, система може підписати запит або зашифрувати дані, але не має змоги прочитати чи експортувати приватний ключ – це забезпечує додатковий захист навіть у випадку логічного доступу до анклаву.

Ще один важливий момент – Nitro Enclaves SDK, який надає інструменти для розробки власних додатків усередині анклавів. Це дозволяє налаштовувати гнучкіші сценарії: наприклад, виконувати перевірку аутентичності даних, обробку конфіденційних обчислень, чи інтегрувати систему з зовнішніми сервісами без порушення принципів Zero Trust.

У моєму проєкті Nitro Enclaves використовується для генерації криптографічних ключів, шифрування та підготовки за SSSS до безпечного розподілу через блокчейн. Що важливо, анклав не зберігає стан між сесіями – тобто після завершення обробки він може бути повністю скинутий, що мінімізує

ризики довготривалого зберігання чутливих даних, але й змушує придумувати додаткову логіку для коректного розшифровування даних. Крім того, Nitro Enclaves підтримує використання attestation – процесу, що дозволяє підтвердити справжність анклаву перед зовнішніми сторонами.

Таким чином, Nitro Enclaves у моєму проєкті не просто ізолює операції, а стає ядром архітектури довіри, що дозволяє забезпечити максимальний захист критичних обчислень і зберігання ключових елементів безпеки.

Висновки до другого розділу

Отже, у цьому розділі було розглянуто використані технології з обґрунтування їх використання. Були покриті теоретичні основи криптографії, такі як AES-256 та SSSS, оскільки це основні алгоритми, які використовуються як в моєму проєкті так і в перевірених системах захисту даних навіть державних масштабів.

Таким самим чином було розібрано використання блокчейну Polygon, принципи його роботи та інтеграцію з усіма компонентами системи для надання нового стійкого рішення захисту інформації.

Розглянута інтеграція AWS Nitro Enclaves для захищених обчислень ключів користувачів, як основної технології для надання ізоляції та відповідності до архітектурних вимог.

РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Розроблення клієнтської частини

Клієнтський додаток виступає в ролі точки входу для взаємодії зі створеною системою. Програма має бути завантажена локально і використовуватись на комп'ютері користувача.

Не дивлячись на комплексний механізм із розподілом ключів через хмару в блокчейні, саме шифрування файлів відбувається локально в Python-програмі для надання найбільшої швидкості використання, зменшення навантаження на сервери та для зниження ризику хакерських атак.

Python-програма також використовується для комунікації з серверною частиною для створення головного ключа шифрування користувача при реєстрації та для його отримання при аутентифікації. Таким чином зашифровані файли зберігаються та оброблюються локально, але ключ до файлів розподілений та збережений децентралізовано.

3.1.1 Впровадження безпечної аутентифікації

Аутентифікація клієнтської частини з сервером виконана з використанням SRP протоколу, що неформально є ZKP. Тобто ця комунікація забезпечує аутентифікацію без розголошення паролю користувача, що надає значні переваги у безпеці порівняно з традиційними методами [5].

В коді SRP реалізовано за допомогою srp Python-бібліотеки для основних криптографічних операцій і слідує SRP-6a варіанту протоколу.

Основні характеристики SRP-6a:

1. пароль ніколи не передається по мережі;
2. сервер не зберігає сам пароль, а лише верифікатор (значення, похідне від пароля);
3. після обміну обидві сторони обчислюють спільний ключ сесії;

4. обчислення значення параметра u як хешу від конкатенації відкритих значень клієнта й сервера.

Фрагмент коду для генерації випадкового значення (salt) та верифікатора (verifier) наведено нижче (рисунок 3.1).

```
# Create verifier and salt using the password
salt, verifier = create_salted_verification_key(username, password)

# Convert to hex strings to match server expectations
salt_hex = salt.hex()
verifier_hex = verifier.hex()
```

Рисунок 3.1 – Генерація salt та verifier під час реєстрації

Процес аутентифікації відбувається за стандартним сценарієм SRP:

1. клієнт генерує пару ключів;
2. клієнт надсилає свій відкритий ключ (A) на сервер;
3. сервер відповідає своїм відкритим ключем (B) і користувацьким salt;
4. клієнт обчислює ключ сесії і доказ;
5. сервер перевіряє доказ клієнта і повертає свій власний;
6. клієнт перевіряє доказ сервера.

3.1.2 Створення графічного інтерфейсу

Графічний інтерфейс повністю побудований із використанням стандартної Python-бібліотеки Tkinter, вона надає зручний та візуально приємний інтерфейс на будь-якій операційній системі.

Інтерфейс поділений на два основних компоненти:

1. LoginWindow – керує реєстрацією та аутентифікацією користувача;
2. MainWindow – надає функціонал шифрування, дешифрування файлів а також управління самими файлами.

Оскільки при написанні коду я чітку слідував принципам SOLID, при створенні клієнтського застосунку я зробив великий акцент на принцип єдиної

відповідальності (Single Responsibility Principle), гарантуючи, що кожен клас вікна інкапсулює чітко визначену, незалежну задачу.

Також при створенні графічного інтерфейсу я використовував такі техніки:

1. адаптивний зворотній зв'язок за допомогою спеціальних статусних рядків;
2. валідація форм з чіткими повідомленнями про помилки;
3. обробка асинхронних операцій із заморожуванням та розморожуванням інтерфейсу під час тривалих операцій;
4. ефективне управління ресурсами та обробка виключень.

Ці техніки є рекомендованими для дотримання, тому що вони гарантують чіткий та зручний користувацький досвід [6].

3.1.3 Криптографічні операції для локального шифрування

Локальне шифрування відбувається з використанням AES-256 в CTR (Counter) режимі. Його перевага в тому, що він є високо продуктивним для великих даних, не має паддінгу і може використовуватись для паралельного оперування.

Сам по собі CTR режим може бути потенційно вразливим до зламу, тому треба обов’язково змінювати nonce, що є одноразовим унікальним числом для внесення унікальності в кожну операцію шифрування. Якщо той самий nonce використовується з тим самим ключем більше одного разу, це відкриває шлях до plaintext recovery (текст може бути вирахуваний XOR-ом) [7].

Процес шифрування починається з генерації криптографічно захищеного вісімбітного nonce, код наведено нижче (рисунок 3.2).

```
# Generate a random nonce for AES-CTR mode
nonce = os.urandom(8)  # 8 bytes nonce for AES-CTR
```

Рисунок 3.2 – Генерація nonce

Далі використовується AES імплементація з бібліотеки PyCryptodome для створення об'єкта шифру зі згенерованим nonce, код наведений нижче (рисунок 3.3).

```
cipher_object = AES.new(
    self.hashed_key_salt["key"],
    AES.MODE_CTR,
    nonce=nonce
)
```

Рисунок 3.3 – Створення об'єкта шифру

Для додаткової безпеки я використовую такий формат шифрування, що перші вісім байтів файлу – це nonce, за яким слідує сам зашифрований вміст файлу. Код реалізації шифрування наведено нижче (рисунок 3.4).

```
with open(self.user_file, "rb") as input_file:
    file_data = input_file.read()

encrypted_content = cipher_object.encrypt(file_data)

with open(self.encrypt_output_file, "wb") as output_file:
    output_file.write(nonce)
    output_file.write(encrypted_content)
```

Рисунок 3.4 – Шифрування даних

Тобто таким чином nonce є і додатковою змінною для розшифрування. І під час розшифрування файлу система спочатку аналізує nonce із заголовка файлу, а потім реконструює об’єкт шифру. Код реалізації наведено нижче (рисунок 3.4).

```
with open(self.user_file, "rb") as input_file:
    nonce = input_file.read(8)
    encrypted_data = input_file.read()

    cipher_object = AES.new(
        self.hashed_key_salt["key"],
        AES.MODE_CTR,
        nonce=nonce
    )

    decrypted_content = cipher_object.decrypt(encrypted_data)
```

Рисунок 3.4 – Розшифрування даних

Аспекти безпеки під час шифрування:

1. Безпечна генерація ключів за допомогою хешування SHA-256
2. Випадкова генерація nonce за допомогою os.urandom() для криптографічного захисту (для повноцінного масштабного рішення краще використовувати інкрементальні лічильники або аутентифікаційне шифрування)
3. Належне очищення чутливих об'єктів після використання
4. Спеціальний формат файлів .encryp для збереження інформації про розширення файлів

Імплементація слідує ключовим рекомендаціям по безпечному використанню режимів роботи блокових шифрів [7].

Робота алгоритму побудована на повному завантажені файлів в систему та шифрування даних в одному процесі та в одному потоці і завантажує весь файл у пам’ять, замість того, щоб оброблювати його по частинах, але цьому рішенню є обґрунтування і воно буде розкрите у розділі з тестуванням.

3.1.4 Локальна база даних

Локальна база даних SQLite використовується для керування метаданими зашифрованих файлів на комп’ютері користувача. Це зроблено для надання стійкого зберігання інформації про файли, навіть якщо вони розкидані по різним папкам.

Хочу додати, що за кожним користувачем окремо закріплена інформація про їх файли, тому один користувач на одному комп’ютері не зможе бачити чи оперувати (розшифровувати) файлами іншого.

SQLite пропонує ідеальне рішення для клієнтських додатків завдяки своїй безсерверній архітектурі та розгортанню з нульовою конфігурацією [8].

Схема бази даних була розроблена для відстеження:

1. шляхів до зашифрованих файлів;
2. штампів часу шифрування;

3. розмірів файлів;
4. ключових хеш-ідентифікаторів, щоб гарантувати, що файли можуть бути розшифровані лише відповідним користувачем.

Фрагмент коду ініціалізації бази даних наведено нижче (рисунок 3.3).

```
def _initialize_database(self):
    """Initialize the SQLite database."""
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS encrypted_files (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            file_path TEXT NOT NULL UNIQUE,
            encrypted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            file_size INTEGER,
            encryption_key_hash TEXT
        )
    """)
    conn.commit()
    conn.close()
```

Рисунок 3.3 – Ініціалізація бази даних SQLite

3.1.5 Інтеграція з віддаленими сервісами

Клієнтський модуль інтегрується з віддаленим сервером аутентифікації через RESTful API. Такий дизайн відповідає принципам архітектури розподілених систем, де сервіси аутентифікації відокремлені від загальної логіки [9]. Такий принцип використовується, основному, при побудові сайтів, але в моєму рішенні це дуже зручно використовувати навіть з клієнтською частині локально на комп’ютері користувача.

Інтеграція передбачає:

1. реєстрацію користувачів з безпечною передачею ключів верифікації;
2. аутентифікацію з нульовим рівнем знань;
3. отримання ключів шифрування з сервера;
4. обробку помилок при виникненні мережевих проблем.

Також, реалізація включає механізми забезпечення безвідмовності:

1. таймаути на запити для запобігання зависання з'єднань;
2. комплексна обробка помилок при збоях в мережі

3. зручні для користувача повідомлення про помилки для поширених проблем.

3.2 Побудова серверної логіки

Для забезпечення роботи усієї моєї системи сервер виступає основним компонентом забезпечення безпеки у тому вигляді в якому це уявлялось. Сервер всередині EC2 відповідає за обчислення, збереження та обмін даними між компонентами.

Серверна логіка керує аутентифікацією та реєстрацією користувачів, генерацією ключів, їх шифруванням, розшифруванням та розподілом завдяки комунікації з блкочейном.

3.2.1 Архітектура сервісу аутентифікації FastAPI

Першими воротами для обробки запросів користувача стає сервер на FastAPI, в основному для користувача він використовується як спосіб аутентифікація та провайдер ключа шифрування.

Отже сервіс аутентифікації структурований як RESTful API з трьома основними ендпоінтами:

1. кореневий ендпоінт (/) – використовується для простої перевірки статусу сервіса – чи працює він;
2. реєстраційний ендпоінт (/register) – містить логіку реєстрації користувачів
3. ініціація аутентифікації (/auth_init) – стартує SRP аутентифікацію
4. верифікація аутентифікації (/auth_verify) – перевірка та завершення аутентифікації з надсиланням ключа користувача.

Отже, першим етапом є реєстрація, він починається ще на стороні клієнта з вводу логіна та пароля, після чого пароль конвертується в verifier із випадковим salt, після чого сервер зберігає ці дані, навіть близько не знаючи оригінальний пароль.

Другим етапом є ініціація аутентифікації, навіть коли користувач вперше реєструється, він все одно миттєво перенаправляється на проходження аутентифікації на рівні з тими, хто намагається просто увійти в давно створений акаунт. Процес ініціації відбувається у вигляді таких етапів:

1. клієнт надсилає ім'я користувача та публічне тимчасове значення A;
2. сервер отримує salt та verifier користувача;
3. сервер генерує челендж B і повертає його разом з salt.

Спрощений приклад реалізації наведено нижче (рисунок 3.4).

```
def auth_init(request: AuthInitRequest):
    """Initialize authentication by providing salt and challenge B"""
    username = request.username
    A_hex = request.A
    A_bytes = bytes.fromhex(A_hex)

    verifier = Verifier(username, salt_bytes, verifier_bytes, A_bytes)

    _, B = verifier.get_challenge()
    B_hex = B.hex()

    session_id = str(uuid.uuid4())
    active_sessions[session_id] = {
        'username': username,
        'verifier': verifier
    }

    return {
        "status": "success",
        "salt": salt_hex,
        "B": B_hex,
        "session_id": session_id
    }
```

Рисунок 3.4 – Спрощений приклад ініціації аутентифікації

Третім етапом є верифікація:

1. клієнт вираховує ключ сесії та підтвердження ключа;
2. сервер перевіряє підтвердження клієнта та повертає власне підтвердження разом із ключем.

Спрощений приклад реалізації наведено нижче (рисунок 3.5).

```python
@app.post("/auth_verify")
def auth_verify(request: AuthVerifyRequest):
    """Verify the client's proof and return server proof"""
    client_proof = request.client_proof
    session_id = request.session_id

    session = active_sessions[session_id]
    verifier = session['verifier']
    username = session['username']

    client_proof_bytes = bytes.fromhex(client_proof)

    server_proof = verifier.verify_session(client_proof_bytes)
    server_proof_hex = server_proof.hex()

    del active_sessions[session_id]

    # ... отримання ключів з блокчейну та анклаву ...

    return {
        "status": "success",
        "server_proof": server_proof_hex,
        "user_key": user_key
    }
```

Рисунок 3.5 – Спрощений приклад верифікації аутентифікації

Важливим аспектом цієї реалізації є використання бінарних даних в контексті JSON, тому що вони не можуть передаватись в цьому форматі. Для вирішення цієї проблеми я конвертую всі бінарні значення в шістандцяткові строки і передаю значення в такому вигляді, потім при їх отримання так само конвертую назад в бінарні значення для оброблення.

3.2.2 Інтеграція з блокчейном

У межах мого проєкту використовується тестова мережа блокчейну Polygon – Polygon Amoy, завдяки ній я успішно зберігаю розподілені ключі користувачів децентралізовано.

Для забезпечення успішної інтеграції сервер спілкується з моїм смарт-контрактом за адресою 0x6b017cbf9ffbdd70e2b8a78ef06c163ab722784c (рисунок 3.6).

Не менш важливим для комунікації є контрактовий ABI. Він виступає в ролі перекладача між кодом та смарт-контрактом, щоб програма розуміла як їй спілкуватись з ним, викликати його функції або читати результати їх виконань. І саме такий ABI я також використовую (рисунок 3.6), отримуючи його з метаданих смарт-контракту після його створення.

```
w3 = Web3(Web3.HTTPProvider('https://rpc-amoy.polygon.technology'))
with open('/home/ec2-user/app/contract_abi.json', 'r') as f:
    contract_abi = json.load(f)
contract_address = Web3.to_checksum_address("0x6b017cbf9ffbdd70e2b8a78ef06c163ab722784c")
contract = w3.eth.contract(address=contract_address, abi=contract_abi)
```

Рисунок 3.6 – Підключення до смарт-контракту

Для розуміння роботи наступних компонентів я вважаю необхідним ознайомитись із таким реалізованим в мене підходом як абстракція акаунту. Він дозволяє користувачам взаємодіяти з блокчейном і підписувати транзакції без необхідності мати власний криптогаманець. Оскільки усі операції з розподілом, зберіганням та діставанням частин ключей виконує сервер, для цього використовується централізований акаунт системи. Це дуже спрощує роботу з додатком на стороні клієнта, оскільки знищує потребу у наявності криптогаманця та коштів на необхідній мережі [10].

Знаючи цю інформацію, стає зрозумілим що за ключ використовує сервер для підписання транзакцій (рисунок 3.7), який, до речі, зберігається захищено в змінних оточення (.env). В масштабному проєкті, такого роду змінні мають бути захищені краще та зберігатись ізольовано.

```
server_private_key = os.getenv('SERVER_PRIVATE_KEY')
server_account = w3.eth.account.from_key(server_private_key)
```

Рисунок 3.7 – Ініціалізація облікового запису Web3.py

Отже коли користувач реєструється, його ключ шифрування генерується в Nitro Enclaves, шифрується головним ключем анклаву і ділиться на частки за допомогою SSSS. FastAPI внутрішня логіка анклаву невідома, він робить запит на отримання частин ключа і зберігає їх в блокчейні (рисунок 3.8).

```python
def store_key_shares(username, shares_data):
    try:
        user_exists = call_contract_view(contract.functions.userExists, username)
        if user_exists:
            print(f"User {username} already exists in contract")
            return True

        chunks = shares_data.get('chunks', 0)
        total_length = shares_data.get('total_length', 0)
        shares_raw = shares_data.get('shares', [])
        serialized_shares = []

        serialized_shares = [
            share if isinstance(share, str) else json.dumps(share)
            for share in shares_raw
        ]

        receipt = execute_contract_tx(
            contract.functions.storeKeyShares,
            username,
            chunks,
            total_length,
            serialized_shares
        )
        success = receipt.status == 1
        return success
    except Exception as e:
        print(f"Error storing key shares: {str(e)}")
        return False
```

Рисунок 3.8 – Процес зберігання частин ключа в смарт-контракті

За допомогою функції execute_contract_tx() викликається функція смарт-контракту storeKeyShares, яка записує усі необхідні дані у блокчейн.

Критичним аспектом будь-якого проєкту пов’язаного з блокчейном є ефективне управління транзакціями та їх вартістю. Для цього я зробив додаткову функцію, яка динамічно підраховує вартість газу для виконання певної функціх смарт-контракту, а також автоматично підписує транзакцію і чекає на її виконання (Рисунок 3.9). Функція додає 10% буфер до приблизної ціни газу для врахування можливих коливань.

```python
def execute_contract_tx(func, *args):
    estimated_gas = func(*args).estimate_gas({'from': server_account.address})
    gas_limit = int(estimated_gas * 1.1)
    tx = func(*args).build_transaction({
        'from': server_account.address,
        'nonce': w3.eth.get_transaction_count(server_account.address),
        'gas': gas_limit,
        'gasPrice': w3.eth.gas_price,
        'chainId': w3.eth.chain_id
    })
    signed_tx = w3.eth.account.sign_transaction(tx, server_private_key)
    tx_hash = w3.eth.send_raw_transaction(signed_tx.rawTransaction)
    return w3.eth.wait_for_transaction_receipt(tx_hash)
```

Рисунок 3.9 – Виконання та підпис транзакції смарт-контракту

Під час аутентифікації система дістає частини ключа з блокчейну, це відбувається за допомогою функції retrieve_key_shares() (рисунок 3.10). Спочатку функція перевіряє чи існують частини ключа для такого користувача, а потім викликає функцію смарт-контракту для отримання даних. Зазначу, що оскільки операція лише зчитує дані і не змінює стан блокчейну, вона є безкоштовною.

```
def retrieve_key_shares(username):
    """Retrieve key shares from the blockchain"""
    try:
        user_exists = call_contract_view(contract.functions.userExists, username)
        if not user_exists:
            print(f"User {username} not found in contract")
            return None

        chunks, total_length, serialized_shares = call_contract_view(
            contract.functions.retrieveKeyShares,
            username
        )

        return {
            'chunks': chunks,
            'total_length': total_length,
            'shares': serialized_shares
        }
    except Exception as e:
        print(f"Error retrieving key shares: {str(e)}")
        return None
```

Рисунок 3.10 – Відновлення частин ключа зі смарт-контракту

Допоміжна функція call_contract_view() спрощує взаємодію з функціями перегляду.

3.2.3 Протокол зв’язку Nitro Enclaves

FastAPI сервер може взаємодіяти з Nitro Enclaves лише через vsock. Реалізація цього комунікаційного протоколу виконана за допомогою стандартного Python програмування сокетів зі спеціальним типом сокетів AF_VSOCK, а сам процес спілкування відбувається за схемою запит-відповідь, всі дані серіалізуються у форматі JSON. Код функції наведено нижче (рисунок 3.11).

```python
def send_to_enclave(request):
    global ENCLAVE_CID

    if ENCLAVE_CID is None:
        ENCLAVE_CID = get_enclave_cid()
        if ENCLAVE_CID is None:
            raise HTTPException(status_code=500, detail="Enclave not running")

    s = socket.socket(socket.AF_VSOCK, socket.SOCK_STREAM)
    try:
        s.connect((int(ENCLAVE_CID), 5005))
        s.send(json.dumps(request).encode())
        response = s.recv(4096).decode()
        return json.loads(response)
    except Exception as e:
        print(f"Error communicating with enclave: {e}")
        raise HTTPException(status_code=500, detail=f"Enclave communication error: {str(e)}")
    finally:
        s.close()
```

Рисунок 3.11 – Надсилання запиту до захищеного середовища через vsock

Для того щоб під’єднатись до анклаву треба також знати його ідентифікатор – CID. Це унікальне число, яке присвоюється кожному анклаву і є необхідним у vsock-з’єднанні. Я його доволі легко дістаю за допомогою запуску окремого процесу, який виконує команду AWS Nitro CLI describe-enclaves. Ця команда повертає список усіх запущених анклавів (в моєму випадку один), і дістає CID анклаву з необхідною назвою.

3.2.4 Система управління ключами в ізольованому середовищі

Анклав зберігає в собі майстер-ключ, який використовується для шифрування усіх ключів користувачів перед тим як вони будуть розподілені та збережені в блокчейні. Цей ключ генерується лише під час запуску та існує лише в пам’яті анклаву.

Ключ шифрування самого користувача шифрується майстер-ключем за допомогою AES-GCM. GCM режим шифрує дані і додатково створює тег аутентифікації, який гарантує, що дані не були змінені.

Процес шифрування:

1. генерує 12-байтовий випадковий nonce;
2. створює об’єкт шифру AES-GCM з головним ключем і nonce;
3. шифрує дані та обчислює тег аутентифікації;

4. повертає nonce, тег і зашифрований текст об’єднані разом.

Код шифрування ключа користувача наведено нижче (рисунок 3.12).

```
def encrypt_with_master_key(data):
    """Encrypt data with the master key using AES-GCM"""
    nonce = os.urandom(12)

    cipher = AES.new(MASTER_KEY, AES.MODE_GCM, nonce=nonce)
    ciphertext, tag = cipher.encrypt_and_digest(data)
    return nonce + tag + ciphertext
```

Рисунок 3.12 – Шифрування ключа користувача майстер-ключем анклава

Розшифрування відбувається у зворотному до шифрування порядку (рисунок 3.13).

```
def decrypt_with_master_key(encrypted_data):
    """Decrypt data with the master key using AES-GCM"""
    nonce = encrypted_data[:12]
    tag = encrypted_data[12:28]  # 16-byte authentication tag
    ciphertext = encrypted_data[28:]

    cipher = AES.new(MASTER_KEY, AES.MODE_GCM, nonce=nonce)

    try:
        plaintext = cipher.decrypt_and_verify(ciphertext, tag)
        return plaintext
    except ValueError:
        raise SecurityError("Decryption failed: authentication tag verification failed")
```

Рисунок 3.13 – Розшифрування ключа користувача майстер-ключем анклава

Наступним кроком захисту ключів користувачів при їх створенні є розподіл за SSSS, що дозволяє відновлення тільки за певної порогової кількості частин. Реалізація передбачає обробку зашифрованих даних довільної довжини, розбиваючи їх на фрагменти, які помістяться в алгоритм SSSS. Кожен фрагмент оброблюється окремо:

1. ключ розбивається на 30-байтові фрагменти;
2. кожен фрагмент перетворюється в ціле число;
3. для кожного фрагмента генеруються частки за допомогою поліноміальної оцінки;
4. частки форматуються за допомогою метаданих (індекс та довжина фрагмента).

Реалізація алгоритму наведена нижче (рисунок 3.14).

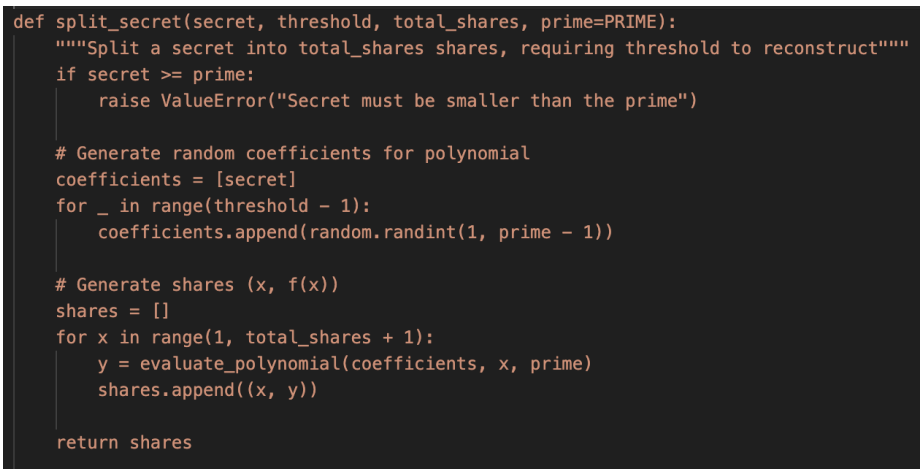
```
def split_encrypted_key(encrypted_data, total_shares=5, threshold=3):
    """Split encrypted key data into shares using SSSS"""
    chunk_size = 30
    chunks = [encrypted_data[i:i+chunk_size] for i in range(0, len(encrypted_data), chunk_size)]
    print(f"Split encrypted data into {len(chunks)} chunks")

    # Create shares for each chunk
    all_shares = []
    for i, chunk in enumerate(chunks):
        # Convert chunk to integer
        chunk_int = bytes_to_int(chunk)

        # Generate shares for this chunk
        chunk_shares = split_secret(chunk_int, threshold, total_shares)

        formatted_shares = []
        for x, y in chunk_shares:
            formatted_shares.append((str(x), str(y), str(i), str(len(chunk))))

        all_shares.append(formatted_shares)

    return {
        "chunks": len(chunks),
        "shares": all_shares,
        "total_length": len(encrypted_data)
    }
```

Рисунок 3.14 – Попередня обробка ключа та форматування його частин

Ядром оброблення ключа є функція split_secret(), яка і впроваджує SSSS для поділу секрета. Алгоритм базується на використанні полінома довільного ступеня над кінцевим полем, визначеним простим числом (рисунок 3.15).

1. Секрет задається як значення в нулі випадково згенерованого полінома.
2. Коефіцієнти полінома (окрім першого, що дорівнює секрету) вибираються випадково.
3. Для кожного значення x = 1…n обчислюється y = f(x), і так утворюється частка (x, y).

Що важливо, так це довжина секрету для розподілу має бути меншою за просте число prime, яке задає поле обчислень, щоб уникнути втрати даних при модульних операціях. Саме для цього я і розбиваю ключ на фрагменти в попередньому кроці [11].



```
def split_secret(secret, threshold, total_shares, prime=PRIME):
    """Split a secret into total_shares shares, requiring threshold to reconstruct"""
    if secret >= prime:
        raise ValueError("Secret must be smaller than the prime")

    # Generate random coefficients for polynomial
    coefficients = [secret]
    for _ in range(threshold - 1):
        coefficients.append(random.randint(1, prime - 1))

    # Generate shares (x, f(x))
    shares = []
    for x in range(1, total_shares + 1):
        y = evaluate_polynomial(coefficients, x, prime)
        shares.append((x, y))

    return shares
```

Рисунок 3.15 – Поділ секрету на частини за SSSS

Функція reconstruct_secret() реалізує відновлення початкового секрету на основі інтерполяції Лагранжа над кінцевим полем [12].

1. Алгоритм приймає список часток, кожна з яких є парою координат і вони являють собою точку на побудованому випадковому поліномі, що формувався при розподілі секрету.
2. За допомогою інтерполяційного полінома Лагранжа обчислюється значення в точці x=0, яке відповідає початковому секрету.
3. Усі обчислення відбуваються по модулю наперед заданого простого числа prime, що якраз гарантує збереження результатів у межах скінченного поля. Тобто, це дає гарантію, що арифметичні операції коректні та ніяка інформація не була загублена під час цих арифметичних операцій.

Важливу роль відіграє й обернений елемент по модулю, адже у скінченний полях класичне ділення неможливе. Саме тому, щоб виконати ділення в умовах обчислень за модулем, використовують обернений за модулем елемент, що дозволяє коректно обчислити частку в рамках обраного простору залишків і забезпечити точність інтерполяції.

Виконання функції в коді можна побачити на рисунку 3.16.

```python
def reconstruct_secret(shares, prime=PRIME):
    """Reconstruct the secret from shares using Lagrange interpolation"""
    if len(shares) == 0:
        raise ValueError("Need at least one share")

    # Lagrange interpolation to find f(0)
    secret = 0
    x_coords = [x for x, _ in shares]

    for i, (x_i, y_i) in enumerate(shares):
        # Calculate Lagrange basis polynomial for x_i
        numerator = 1
        denominator = 1

        for j, x_j in enumerate(x_coords):
            if i != j:
                numerator = (numerator * (0 - x_j)) % prime
                denominator = (denominator * (x_i - x_j)) % prime

        lagrange_term = (y_i * numerator * mod_inverse(denominator, prime)) % prime
        secret = (secret + lagrange_term) % prime

    return secret
```

Рисунок 3.16 – Реконструкція секрету на основі інтерполяції Лагранжа

Окрім роботи з ключами анклав також виступає в ролі сервера з яким спілкується FastAPI у ролі клієнта. Для їх комунікації сервер анклаву доводиться обробляти клієнтські запити за допомогою центральної функції, яка ті запроси розподіляє та запускає відповідні операції.

Коли сервер анклаву отримує запит на реєстрацію, система генерує новий випадковий 256-бітний ключ для користувача, ділить на частини і повертає частини (для подальшого зберігання в блокчейні) та сам ключ користувача, який йде виключно до користувача (для локального шифрування/розшифрування даних) і ніде не зберігається.

При отриманні запросу на верифікацію аутентифікації, система одночасно отримує частини ключа з блокчейну, комбінує їх для відновлення зашифрованого ключа, розшифровує його за допомогою внутрішнього майстер-ключа і повертає розшифрований ключ клієнту.

Реалізацію цього функціоналу можна знайти в Додатку Б рисунок Б.1

3.4 Розгортання середовища на AWS

Зручне розгортання та керування середовищем є доволі важливим етапом мого проєкту, а AWS надає зручність та надійність в створенні доступного

хмарного сервісу. Таким чином, моя реалізація використовує принцип інфраструктури як коду за допомогою Terraform, саме він надає контроль над процесом розгортання.

Отже, конфігурація розгортання починається з оголошення провайдерів, які встановлюють цільове середовище та обмежують версії, з них я обираю AWS як основного.

Наступним кроком є налаштування мережевої архітектури, основу становить VPC з публічною підмережею та інтернет-шлюзом. Тобто, навіть не дивлячись на те, що такий підхід відкриває сервіс для публічного інтернету, він все одно обмежує експозицію до певних портів та впроваджує кілька рівнів безпеки.

IAM відіграє ключову роль у реалізації надання безпечного доступу до хмари, під час розгортання створюється спеціальна роль IAM з дозволами, суворо обмеженими до отримання скрипту налаштування Nitro Enclaves з S3-бакету.

Основою серверної частини є сам EC2 і його тип є балансом між вартістю, продуктивність та спеціальними функціями. Обраним типом є m5.xlarge і обраний він саме через його підтримку Nitro Enclaves, що надає 4 віртуальних процесора та 16 ГБ оперативної пам’яті. Це дозволяє виділити необхідні 2 процесори та 1200 МБ оперативної пям’яті анклаву, паралельно зберігаючи достатню частину для хосту.

Процес ініціалізації застосунку реалізовано через функцію user data самого EC2, яка виконує вказаний скрипт на етапі ініціалізації. Тобто він оновлює систему, налаштовує залежності, конфігурує Nitro Enclaves, генерує сертифікати, розгортає сам застосунок та налаштовує його. Така схема усуває необхідність власноруч налаштовувати сервіси, що може бути дуже довго та повторювано.

Усі чутливі значення, як наприклад секретний ключ серверного гаманця для блокчейн-транзакцій, керується через змінні Terraform та файл середовища .env. Тобто, файли terraform.tfvars передають значення в сам Terraform при його запуску, так само і .env файли.

Анклав налаштовується початковим виділенням ресурсів як і було зазначено вище. Після цього, автоматично дістається сценарій налаштування з S3-бакету, який, в свою чергу, створює контейнер Docker з програмою керування ключами, перетворює його на EIF за допомогою Nitro CLI та запускає анклав з виділеними ресурсами. Під час запуску анклав встановлює канал зв'язку vsock, що є безпечним інтерфейсом з головною системою.

Поточна реалізація використовує локальну базу даних для зберігання даних аутентифікації користувачів (verifier та salt), що є достатнім для демонстрації концепту та цілей, але буде вдосконалено використанням Amazon DynamoDB для виробничого використання. Реалізація створить таблицю з відповідними атрибутами для зберігання даних з іменем користувача як ключем розподілу для ефективного пошуку.

Аналогічно, для високої доступності сервісу треба впровадити ALB з ASG. ALB розподілятиме трафік між кількома екземплярами, тоді як ASG підтримуватиме бажану кількість екземплярів залежно від навантаження користувачами.

3.5 Розробка та інтеграція смарт-контракту

Смарт-контракт є одним з найважливіших компонентів усієї системи, тому що саме він відповідає за зберігання найбільш критичних даних користувачів – їх частин шифрувальних ключів. Через це, необхідно було приділити найбільшу кількість уваги саме створенню надійної логіки для виконання безпечного зберігання, і я фокусувався на створенні прозорої, простої та захищеної архітектури.

3.5.1 Архітектура та дизайн смарт-контракту

Я підійшов до створення смарт-контракту з точки зору ефективності та економії газу, через що вирішив будувати основну логіку збереження даних у вигляді дворівневої моделі сховища. Тобто, я відокремлюю метадані від

фактичних даних для обходу обмежень розмірів параметрів Ethereum. Спочатку я пов’язую хеш імені користувача із такими його даними як кількість фрагментів ключа, загальна довжина даних та позначка існування, а після цього я пов’язую фактичні серіалізовані дані з фрагментами. Виконання в коді можна побачити на рисунку 3.17.

```
struct KeySharesData {
    uint256 chunks;
    uint256 totalLength;
    bool exists;
}

mapping(bytes32 => KeySharesData) private userKeyData;

mapping(bytes32 => mapping(uint256 => string)) private shareChunks;
```

Рисунок 3.17 – Формат зберігання даних в смарт-контракті

Такі вкладені структури даних надає масштабованість, тому що асоціює фрагменти ключів з конкретними користувачами та контекстам, уникаючи витрат на газ, оскільки Solidity не зберігає самі ключі у storage, а лише хешовані значення. Така структура також полегшує контроль доступу та ізоляцію структур даних.

Хешування імені користувача надає додатковий рівень безпеки конфіденційності, бо ім’я не зберігаєтсья у повноцінному відкритому вигляді. Для цього хешування смарт-контракт використовує рідну функцію хешування Ethereum – Keccak-256 (рисунок 3.18).

```
bytes32 usernameHash = keccak256(abi.encodePacked(username));
```

Рисунок 3.18 – Хешування імені користувача за допомогою Keccak-256

3.5.2 Реалізація зберігання та отримання ключів

Напевно найголовнішими функціями смарт-контракту є функції зберігання та отримання частин ключів.

Розглядаючи функцію збереження фрагментів, слід додати, що для оптимізації ціни створення транзакції, необхідно було створити просту та надійну логіку, що якраз відображається у функції storeKeyShares().

Спочатку функція вираховує хеш імені користувача, потім зберігає метадані, проходить в циклі через кожен наданий фрагмент ключа, зберігаючи під необхідним індексом. Впровадження цього функціоналу в коді можна побачити на рисунку 3.19.

```
function storeKeyShares(
    string memory username,
    uint256 chunks,
    uint256 totalLength,
    string[] memory serializedShares
) external onlyServer returns (bool) {
    bytes32 usernameHash = keccak256(abi.encodePacked(username));

    // Store metadata
    userKeyData[usernameHash] = KeySharesData({
        chunks: chunks,
        totalLength: totalLength,
        exists: true
    });

    // Store each chunk of shares
    for (uint256 i = 0; i < serializedShares.length; i++) {
        shareChunks[usernameHash][i] = serializedShares[i];
    }

    emit KeySharesStored(usernameHash);
    return true;
}
```

Рисунок 3.19 – Функція збереження частин ключа в смарт-контракті

Протилежною функцією для отримання збережених даних виступає retrieveKeyShares(). Вона спочатку перевіряє існування користувача, а потім, перебираючи кожен індекс ключа, збирає повний набір частин ключа в одній змінній для повернення (рисунок 3.20). Оскільки це функція перегляду і не змінює стан блокчейну, вона не потребує витрат на газ.

Це не тільки економить час і ресурси, але й сприяє створенню більш надійних та масштабованих систем.

```
function retrieveKeyShares(string memory username)
    external
    view
    onlyServer
    returns (
        uint256 chunks,
        uint256 totalLength,
        string[] memory serializedShares
    )
{
    bytes32 usernameHash = keccak256(abi.encodePacked(username));

    // Check if user exists
    require(userKeyData[usernameHash].exists, "No key shares found for this user");

    KeySharesData memory data = userKeyData[usernameHash];
    string[] memory shares = new string[](data.chunks);

    // Retrieve each chunk
    for (uint256 i = 0; i < data.chunks; i++) {
        shares[i] = shareChunks[usernameHash][i];
    }

    return (data.chunks, data.totalLength, shares);
}
```

Рисунок 3.20 – Функція отримання частин ключа зі смарт-контракту

Функція userExists() є невеликою допоміжною функцією, написаною саме під потреби цієї логіки і надає швидку перевірку чи існує користувач. На рисунку 3.21 зображений код цієї функції.

```
function userExists(string memory username) external view onlyServer returns (bool) {
    bytes32 usernameHash = keccak256(abi.encodePacked(username));
    return userKeyData[usernameHash].exists;
}
```

Рисунок 3.21 – Функція перевірки існування користувача в смарт-контракті

3.5.3 Безпека та контроль доступу

Для обмеження доступу до виклику функції Solidity має спеціальну конструкцію – модифікатор доступу. Взагалі це є доволі зручним інструментом для впровадження будь-якої повторюваної логіки для функцій, які її викликають.

У моєму випадку створюється змінна serverAddress (рисунок 3.22), яка зберігає адресу гаманця FastAPI серверу, для того щоб тільки гаманець з цією адресою міг викликати певні функції. Ключовим словом є private – це означає, що інші контракти або користувачі не зможуть прочитати значення цієї змінної напряму, воно доступне тільки всередині самого смарт-контракту. Такий дизайн привносить певного рівня централізацію, але це необхідно для забезпечення захисту від несанкціонованих модифікацій, плюс все одно гарантії незмінності та доступності блокчейну залишаються незмінними.

```
address private serverAddress;
```

Рисунок 3.22 – Змінна для збереження адреси гаманця серверу

В поєднанні з цією змінною я створюю модифікатор доступу onlyServer, який додаю до кожної функції, якщо вона надає доступ до конфіденційних даних. Виконання модифікатору доступу можна побачити на рисунку 3.23.

```
modifier onlyServer() {
    require(msg.sender == serverAddress, "Only the server can call this function");
    _;
}
```

Рисунок 3.23 – Модифікатор доступу в смарт-контракті

3.6 Тестування системи

Для забезпечення функціональної коректності та продуктивності такої комплексної системи обов'язково необхідно провести її комплексне тестування. Компоненти тестувались як ізольовано, так і у вигляді цілісного підходу, який оцінював систему за кількома вимірами: функціональна коректність, показники продуктивності, забезпечення безпеки та досвід користувача. Таким чином я зміг більш реалістично оцінити поведінку систему у виробничих умовах.

3.6.1 Функціональне тестування

Почати я хотів би саме з функціонального тестування – тестування, яке перевіряє чи правильно працює система згідно вимог, зосереджуючись на окремих функціях або поведінці системи з боку користувача. Тобто, йде перевірка що саме програма робить, а не як вона це робить. Для цього, я в першу чергу створив набір автоматизованих тестів у вигляді Python-скриптів. Набір включав сценарії успішної реєстрації, ініціалізації та завершення аутентифікації, шифрування клієнтського додатку, а також обробку помилкових ситуацій. Приклад такого тесту можна побачити на рисунку 3.24.

```
def test_login(get_auth_client, get_user_credentials):
    auth_client = get_auth_client
    username, password = get_user_credentials
    result = auth_client.login(username, password)

    assert result.get("status") == "success"
    assert "decrypted_key" in result
```

Рисунок 3.24 – Функція тестування логіну з клієнтської частини

Смарт-контракт було протестовано в середовищі Remix, оскільки там він був і розгорнутий. Ця платформа надала доволі зручне середовище для інтерактивного тестування функціональності, включно із зберіганням,

отриманням даних та механізму контролю доступу. Особливу увагу було приділено крайнім випадкам, таким як спроби доступу до неіснуючих користувачів, або несанкціоновані виклики функцій.

Для перевірки Nitro Enclaves я розробив спеціалізований тимчасовий фреймворк тестування, який імітував протокол зв'язку vsock та перевіряв криптографічні операції анклаву. Це включало тести на генерацію ключів, шифрування, дешифрування, та реалізацію алгоритму SSSS.

Інтеграційні тести надавали перевірку end-to-end робочого процесу, тобто при безшовному поєднанні, гарантуючи, що всі компоненти працюють коректно. Вони симулювали повний шлях користувача по процесам додатку з початку і до кінця, наприклад перевірка реєстрації через аутентифікацію і до отримання ключа та відкриття потрібного вікна додатку, перевіряючи як справність передачі даних між сервером, анклавом та блокчейном, так і перевіряючи коректність роботи самого клієнтського додатку на відповіді серверу.

3.6.2 Тестування продуктивності

Під час тестування ефективності системи, я зміг виявити доволі важливі показники. Криптографічні операції на стороні клієнту досягли середньої швидкості приблизно 150 МБ/с, що є непоганим для наданого рівня безпеки. Хоча це і повільніше ніж у деяких рішень, але компроміс для ізоляції на рівні користувача виправданий для безпеки даних.

Щодо витрат на транзакції для збереження даних в блокчейні, то в середньому вони коштують 0,005-0,01 долара США за реєстрацію користувача, що економічно є доволі вигідним навіть при великих масштабах. Реєстрація 100000 користувачів в середньому обійшлась би в 700 доларів США. Це значно нижче ніж аналогічні операції в основній мережі Ethereum і порівняно з іншими рішеннями другого рівня, як Arbitrum та Optimism.

Порівняння економічної ефективності зберігання секретної інформації:

- AWS CloudHSM: вартість від 1,60 доларів США на годину, або близько 1 000 доларів США на місяць, що значно перевищує показники моєї системи [18];
- HashiCorp Vault: стандартний функціонал коштує 1,58 доларів США на годину, коли моє рішення потребує одноразового платежу [19].

Вимірювання швидкості показали, що реєстрація користувача займає в середньому 4 секунди, причому більша частина цього часу (близько 3 секунд) витрачається на очікування підтвердження транзакції в блокчейні. На відміну від реєстрації, аутентифікація є набагато швидшою і займає приблизно 1 секунду, оскільки вона в основному включає процес отримання інформації блокчейну без зміни його стану. Детальніший опис операцій та їх часу виконання можна побачити в таблиці 3.1.

Таблиця 3.1 – Опис операцій системи та їх час виконання

Операція	**Середній час**	Опис
Реєстрація	4,02 с	Розрахунок SRP: 0,12 с
		Генерація ключа в анклаві: 0,34 с
		Транзакція в блокчейні: 3,56 с
Логін	1,08 с	Верифікація SRP: 0,14 с
		Зчитування з блокчейну: 0,68 с
		Реконструкція ключа в анклаві: 0,26 с

Ці характеристики порівнювались як з традиційними Web2 рішеннями так і з системами аутентифікації Web3 простору.

- Традиційні системи зазвичай аутентифікують користувачів за 250-500 мс, що може доходити до 18 разів швидше за моє рішення [13].

- Децентралізовані рішення для аутентифікації в основній мережі Ethereum можуть займати від 13 с. [14].

Отже, таким чином, моє рішення являє золоту середину, пропонуючи більш високий рівень безпеки ніж звичайні централізовані рішення, і забезпечуючи при цьому кращу продуктивність та ефективність, ніж повністю інтегровані в блокчейн альтернативи і навіть традиційні системи збереження секретів.

Стрес-тестування також показало непоганий результат – система може обробляти приблизно 200 одночасних користувачів на один екземпляр EC2 без помітного збільшення затримок. Масштабованість в більшій мірі обмежена vsock-зв'язком анклаву, тому що він може обробляти тільки один запит за раз. Теоретично, це обмеження можна усунути в наступних ітераціях за рахунок більш досконалої системи черги або кількох екземплярів анклаву.

3.6.3 Тестування безпеки

Тестування безпеки є найважливішим тестуванням, і для цього було застосовано багатосторонній підхід. Було розроблено комплексну модель загроз за методологією STRIDE – підробка, фальсифікація, відмова, розкриття інформації, відмова в обслуговуванні, підвищення привілеїв. Це доволі непогано дозволило виявити потенційні вектори атак, які потім було перевірено та усунено під час реалізації. Отже, основні результати, що було отримано:

1. система стійка до перехоплення паролів завдяки протоколу SRP з нульовим знанням;
2. Nitro Enclaves забезпечує ефективний захист від атак на хост-систему;
3. відокремлення даних аутентифікації від ключів шифрування створює кілька рівнів безпеки;

4. блокчейн-зберігання частин ключів забезпечує докази несанкціонованого втручання, стійкість до маніпуляцій з даними та безпеку їх зберігання загалом.

Висновки до третього розділу

У третьому розділі було детально розглянуто практичну реалізацію створеної системи з великим акцентом на технічні та архітектурні аспекти її побудови. Було досліджено як поєднання локального шифрування на стороні користувача та розподіленого зберігання ключів у блокчейні дозволяє досягти балансу між продуктивністю та безпекою. І аналізовані рішення демонструють, що навіть складна логіка не заважатиме в організації зручної для користувача роботи системи.

У розділі описано створення локального додатку для користувача, особливості локального шифрування даних разом із безпечною аутентифікацією SRP.

Окрему увагу я приділив серверній архітектурі, зокрема побудові API на FastAPI, створенню логіки в Nitro Enclaves для маніпуляцій з ключами користувачів та інтеграції зі смарт-контрактом.

Також розглянув етапи розгортання інфраструктури за допомогою AWS і Terraform. Усі компоненти було протестовано як окремо так і в комплексі.

РОЗДІЛ 4 ЕКОНОМІЧНА ЧАСТИНА

4.1 Характеристика бізнес-моделі

Для створення приємного досвіду використання функціоналу та одночасно для генерації прибутку було обрано freemium підхід – тобто, користувачі можуть безкоштовно користуватись базовим функціоналом, але мають можливість придбати розширену версію з додатковими перевагами.

У базовій безкоштовній версії користувачу доступна обмежений, але все одно корисний функціонал – можливість створювати до трьох облікових записів з одного комп'ютера (тому що кожен користувач вимагає витрат на створення транзакцій в блокчейні), використання лише одного способу шифрування файлів, знижена швидкість шифрування та базовий рівень керування файлами.

На відміну від базової версії, преміум надаватиме такі переваги:

1. розширена кількість акаунтів (10 або 20);
2. кілька методів шифрування з їх додатковою оптимізацією (наприклад різні режими AES);
3. додаткові можливості керування файлами та зміни в інтерфейсі.

Розрахунки при економічному аналізі велись у гривнях, із конвертацією долара США за середнім курсом 41,2 грн/дол. США, розрахованим на основі періоду березень-травень 2025 року.

4.2 Розрахунок бюджету

Отже, при розрахунках бюджету, найбільш значним аспектом є витрати на утримання фахівців, оскільки, я вважаю, що саме від їх кваліфікацій та злагодженою роботи залежить успіх реалізації продукту. Першим кроком є аналіз витрат на утримання фахівців, для цього були враховані такі ролі: проєктний менеджер, cloud розробник, Python full-stack розробник, блокчейн розробник, security інженер, DevOps, QA інженер та технічний письменник.

Загальна сума витрат на оплату праці склала 154 633 у.о. (таблиця 4.1), якщо рахувати, що проєкт розробляється за місяць активної роботи в цілому,

враховуючи не рівномірну кількість робочих годин працівників, а лише за необхідності, виходячи з поставлених задач. Використано медіанні показники для молодших спеціалістів з досвідом роботи до двох років.

Таблиця 4.1 – Витрати на утримання фахівців

Роль	**Оплата за 1 годину роботи, у.о.**	**Кількість необхідних годин**	**Кількість потрібних фахівців**	**Сума витрат на утримання фахівців, у.о.**
Проєктний менеджер	285,00	50	1	14 250,00
Cloud розробник	304,00	85	1	25 840,00
Python full-stack розробник	274,00	160	1	43 840,00
Блокчейн розробник	323,00	45	1	14 535,00
Security інженер	342,00	70	1	23 940,00
DevOps	258,00	60	1	15 408,00
QA інженефр	209,00	50	1	10 450,00
Технічний письменник	182,00	35	1	6 370,00
Разом				154 633,00

Окрім людських ресурсів, проєкт вимагає постійні витрати на інфраструктуру хмарних сервісів, зокрема EC2 (m5.xlarge), Application Load Balancer, S3 для зберігання скриптів, DynamoDB для збереження даних авторизації та передачі даних. Сервісні витрати в сумі складають 7056 у.о. (таблиця 4.2) за місяць розроблення продукту, включаючи ресурси для обчислень, зберігання, маршрутизації трафіку та забезпечення доступності. Такий бюджет також враховує потенційні витрати на масштабування, необхідне при розгортанні продукту у відкритому доступі. Важливим зауваженням є врахування витрат на певні сервіси, які не є впроваджені на етапі MVP (Application Load Balancer, DynamoDB), але є обов’язковими в умовах

повноцінного функціонування та реалізації такої комплексної системи. Вартість сервісів оцінювалась, виходячи з офіціальних цін на сайті Амазону [15].

Таблиця 4.2 – Сервісні витрати

Назва сервісу	**Період оплати, місяців**	**Вартість сервісів, у.о. за 1 годину**	**Сума, у.о.**
EC2	1	7,91	5 695,2
Application Load Balancer	1	0,92	662,4
S3 Storage	1	0,94	676,8
DynamoDB	1	0,03	22,32
Разом			**7 056,72**

У рамках реалізації проєкту було враховано потребу в оснащенні команди розробки пристроями для повноцінної роботи у віддаленому форматі. Для спрощення обслуговування було обрано одну модель ноутбука – HP Pavilion 15 на кожного працівника, вартість яких складає 216 000 у.о. (таблиця 4.3).

Таблиця 4.3 – Вартість обладнання

Назва обладнання	**Ціна, у.о.**	**Кількість, шт.**	**Сума, у.о.**
Ноутбук HP Pavilion 15	27 000,00	8	216 000,00
Разом			**216 000,00**

Було проведено оцінку структури робіт за фазами життєвого циклу. І відповідно до разрахунків (додаток А, таблиця А.1), найбільшу частину витрат становить інвестиційна фаза – 64,68% (127 796 у.о.), що включає в себе розроблення клієнтської і серверної частини, створення смарт-контракту, інтеграцію з блокчейном та побудову логіки обробки ключів у анклаві. Передінвестиційна фаза становить 20,04% (40 307 у.о.), експлуатаційна – 14,93% (29 504 у.о.), отже, загалом витрати на розроблення проєкту оцінюються в 197 607 у.о.

Амортизаційні відрахування дають нам можливість оцінити вартість техніки з урахуванням терміну експлуатації пристроїв. Оскільки у рамках цього проєкту спеціалістам надається 8 ноутбуків HP Pavilion 15, завдяки розрахункам вдалось оцінити амортизаційні відрахування на основі вартості цього обладнання (додаток А, таблиця А.2). Таким чином загальна сума амортизації становить 5 918 у.о.

Розрахувавши усі необхідні показники, можна переходити до обчислення повного бюджету продукту. До собівартості включено оплату праці (таблиця 4.1), сервісні витрати (таблиця 4.2), амортизаційні відрахування (додаток А, табл. А.2). Бюджет становить 231 626 у.о. (таблиця 4.4).

Таблиця 4.4 – Бюджет

№ з/п	**Показник**	**Сума, у.о (виконавці наймані працівники**	**Сума, у.о (виконавці – самозайняті особи (ФОПи)**
1	Фонд оплати праці	154 633,00	154 633,00
2	ЄСВ	34 019,26	0,00
3	Сервісні витрати	7 056,72	7 056,72
4	Амортизація	5 918,00	5 918,00
5	Інші витрати (Резерв на випадок непередбачених обставин)	30 000,00	30 000,00
6	Собівартість (сума пунктів від 1 до 5)	231 626,98	197 607,72

4.3 Оцінювання економічної ефективності проєктних рішень

Почнемо з розрахунку грошових надходжень від проєкту. Ціна преміум версії складатиме 3600 у.о на рік. Наступна оцінка кількості користувачів ґрунтується на глобальній аудиторії B2C сегменту як web3, так і web2 з інтересом до власної безпеки в цифровому світі.

Прогнозована кількість підписок:

- 1 рік: 300 підписок;
- 2 рік: 600 підписок;
- 3 рік: 1000 підписок.

Виручка:

- 1 рік: 1 080 000 у.о (300 * 3600 = 1 080 000 у.о);
- 2 рік: 2 160 000 у.о (600 * 3600 = 2 160 000у.о);
- 3 рік: 3 600 000 у.о (1000 * 3600 = 3 600 000 у.о).

Загальна виручка за 3 роки – 6 840 000 у.о.

Види експлуатаційних витрат:

- підтримка серверної інфраструктури (EC2, S3, DynamoDB, Load Balancer);
- заробітна плата працівників (Python, Security, DevOps-розробники);
- амортизація обладнання.

Розрахунок експлуатаційних витрат:

- 1 рік: 84 680,64 (сервіси) + 840 000,00 (зарплата) + 27 000,00 (амортизація) = 951 680,64 у.о.;
- 2 рік: 93 148,70 (сервіси) + 912 000,00 (зарплата) + 27 000,00 (амортизація) = 1 032 148,70 у.о.;
- 3 рік: 102 463,57 (сервіси) + 960 000,00 (зарплата) + 27 000,00 (амортизація) = 1 089 463,57 у.о.

Сумарні витрати за 3 роки – 3 073 292,91 у.о.

Після розрахунку витрат та виручки, доцільно порахувати його економічну ефективність шляхом визначення чистої теперішньої вартості, індексу доходності.

Першим кроком є розрахунок грошових потоків (таблиця 4.5).

Таблиця 4.5 – Розрахунок грошових потоків, у.о.

	Показник	**1 рік**	**2 рік**	**3 рік**	**Разом**
1	Виручка від реалізації продукції	1 080 000,00	2 160 000,00	3 600 000,00	**6 840 000,00**
2	Експлуатаційні витрати	951 680,64	1 032 148,70	1 089 463,57	**3 073 292,91**
3	Амортизаційні відрахування	27 000,00	27 000,00	27 000,00	**81 000,00**
4	Загальні витрати *(n2 + n3)*	978 680,64	1 059 148,70	1 116 463,57	**3 154 292,91**
5	Валовий прибуток *(n1 – n4)*	101 319,36	1 100 851,3	2 483 536,43	**3 685 707,09**
6	Податок на прибуток *(n1 × 18%)*	194 400,00	388 800,00	648 000,00	**1 231 200,00**
7	Чистий прибуток *(n5 – n6)*	-93 080,64	712 051,3	1 835 536,43	**2 454 507,09**
8	Грошовий потік *(n3 + n7)*	-66 080,64	739 051,3	1 862 536,43	**2 535 507,09**

Для подальших розрахунків необхідним є дисконтний коефіцієнт. Норма дисконту r = 15%, оскільки фінансування за рахунок власних грошей.

- 1 рік: $1/(1 + 0{,}15)^1 = 0{,}8696$;
- 2 рік: $1/(1 + 0{,}15)^2 = 0{,}7561$;
- 3 рік: $1/(1 + 0{,}15)^3 = 0{,}6575$.

В результаті проведеного аналізу (таблиця 4.6), видно, що чиста теперішня вартість (NPV) є позитивною, тобто, грошові надходження перевищують сукупні витрати з урахуванням фактору часу. І це є свідченням, що проєкт є економічно доцільним та фінансово вигідним. Такий результат дає достатньо підстав для ухвалення рішення про реалізацію запланованого продукту з майбутнім прибутком.

Таблиця 4.6 – Розрахунок чистої теперішньої вартості за проєктом

Рік	**Грошовий потік, у.о.**	***К*д, *r* = 15%**	***NPV*, у.о.**
0	-231 626,98	1	-231 626,98
1	-66 080,64	0,8695652174	-57 461,43
2	739 051,3	0,7561436673	558 823,35
3	1 862 536,43	0,6575162324	1 224 644,86
Разом	2 303 880,11	-	1 494 379,80

Тепер, за формулою 4.2 можна вираховувати індекс дохідності інвестицій (PI):

$$PI = \frac{1}{INV} \sum_{n=1}^{T} \frac{CF_n}{(1+r)^n}$$

(4.2)

PI = 2 535 507,09/ 231 626,98 = 10,95

Отже, на кожну вкладену грошову одиницю проєкт генерує 7,45 одиниць грошових надходжень, а це свідчить про велику прибутковість.

Висновки до економічної частини

Щодо фінансової моделі, то freemium є дуже ефективною і перспективною, дозволяє залучити більшу аудиторію завдяки безкоштовному функціоналу, одночасно створюючи потенціал для монетизації [16].

Загальний бюджет складає 231 626,98 у.о., з яких найбільшу частину складають людські ресурси – 154 633 у.о. Тим не менш, проєкт демонструє доволі високу економічну ефективність – NPV становить 1 494 379,80 у.о., а PI складає 10,95

Очікується, що кількість преміум користувачів зросте з 300 до 1000 протягом трьох років.

В межах перших двох років проєкт покриє витрати та почне генерувати чистий прибуток. А виходячи з цього, можна казати, що проєкт є економічно доцільним і рекомендованим до реалізації.

ВИСНОВКИ

Під час виконання роботи було сформовано та виконано основні завдання:

1. проведено аналітичний огляд існуючих сучасних систем захисту даних, разом із проблемами централізованого зберігання інформації;
2. розроблено архітектуру клієнт-серверної системи з використанням AWS Nitro Enclaves, блокчейну Polygon, AES-256, SSSS, FastAPI, Web3.py, Terraform та інших технологій;
3. реалізовано клієнтську частину з використанням Python Tkinter та алгоритмів шифрування, а також систему SRP аутентифікації;
4. усю інфраструктуру автоматично розгорнуто за допомогою Terraform на AWS;
5. реалізовано сервер на FastAPI та смарт-контракт на Solidity з абстракцією акаунту;
6. розроблено логіку створення ключів, їх шифрування/розшифрування та розподілу за SSSS в Nitro Enclaves;
7. проведено комплексне тестування системи;
8. досліджено проєкт з економічної точки зору і доведено, що він є прибутковим та рекомендованим до реалізації.

Також, досягнуто наступних результатів:

1. швидкість шифрування файлів локально становить в середньому 150 МБ/с, що не є відчутною затримкою для користувача;
2. процес реєстрації користувача, включаючи генерацію та розподіл ключа, займає ~1.1 с, що є прийнятним показником для сучасних систем безпеки;
3. аутентифікація через SRP відбувається протягом ~0.43 с при стабільному з’єднанні;
4. вартість транзакції на запис однієї частини ключа в смарт-контракт у мережі Polygon становить < $0.01, а зчитування ключів із блокчейну є повністю безкоштовним;

5. система реалізує повну абстракцію акаунтів – користувач не взаємодіє з криптогаманцями або підписуванням, що значно спрощує користувацький досвід;
6. для підвищення продуктивності клієнтська частина виконує локальне шифрування в один потік, завантажуючи файл повністю в пам’ять, що було обґрунтовано позитивним впливом на швидкість.

У цифрових показниках система зменшує вартість зберігання секретних даних 85-95% порівняно з централізованими рішеннями на кшталт AWS CloudHSM та HashiCorp Vault. Конкретно, обробка 100,000 користувачів обходиться приблизно у 700 доларів США одноразово, при середній вартості 0,005-0,01 за одну транзакцію реєстрації. Додатково, на відміну від традиційних рішень, які вимагають постійних щомісячних витрат, дана система потребує оплати лише за фактичні транзакції, що оптимізує витрати у довгостроковій перспективі при масштабуванні.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cost of a Data Breach Report 2024. URL: https://www.ibm.com/reports/data-breach (дата звернення: 05.04.2025)
2. Advanced Encryption Standard (AES). URL: https://csrc.nist.gov/pubs/fips/197/final (дата звернення: 07.04.2025)
3. SecretCodex32. URL: https://www.secretcodex32.com/ (дата звернення: 07.04.2025)
4. Proof-of-Stake Overview. URL: https://docs.polygon.technology/pos/overview/ (дата звернення: 07.04.2025)
5. Wu T. The SRP Authentication and Key Exchange System. URL: https://datatracker.ietf.org/doc/html/rfc2945 (дата звернення: 08.04.2025)
6. Nielsen Norman Group. Visibility of System Status. URL: https://www.nngroup.com/articles/visibility-system-status/ (дата звернення: 08.04.2025)
7. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. URL: https://csrc.nist.gov/pubs/sp/800/38/a/final (дата звернення: 09.04.2025)
8. SQLite Is Serverless. URL: https://www.sqlite.org/serverless.html (дата звернення: 09.04.2025)
9. Barabanov A., Makrushin D. Authentication and Authorization in Microservice-Based Systems: Survey of Architecture Patterns. URL: https://arxiv.org/abs/2009.02114 (дата звернення: 10.04.2025)
10. Buterin V. EIP-4337: Account Abstraction via Entry Point Contract. URL: https://eips.ethereum.org/EIPS/eip-4337 (дата звернення: 11.04.2025)
11. Adi Shamir. How to share a secret. URL: https://dl.acm.org/doi/10.1145/359168.359176 (дата звернення: 11.04.2025)
12. Gold G. Shamir's Secret Sharing: A Step-by-Step Guide with Python Implementation. URL: https://medium.com/@goldengrisha/shamirs-secret-sharing-a-step-by-step-guide-with-python-implementation-da25ae241c5d

13. Website Response Time: What Is It & How to Improve It. URL: https://www.uptimia.com/learn/website-response-time (дата звернення: 12.04.2025)
14. Ethereum Transaction Speed Factors. URL: https://bithide.io/blog/ethereum-transaction-speed-factors/ (дата звернення: 12.04.2025)
15. AWS Pricing Calculator. URL: https://calculator.aws/#/?nc2=h_ql_pr_calc (дата звернення: 13.04.2025)
16. Shang Y., Jiang J., Zhang Y., Zhang R., Liu P. When does a freemium business model lead to high performance? – A qualitative comparative analysis based on fuzzy sets. URL: https://pubmed.ncbi.nlm.nih.gov/38333777/ (дата звернення: 15.04.2025)
17. Андросов Н.С., Райтер О.К. Система збереження та шифрування даних на основі хмарних та блокчейн-технологій. Зб. тез доповідей IV студентської науково-практичної конференції «Прикладні комп'ютерні науки». URL: https://itstep.edu.ua/scientific-measures#gsc.tab=0 (дата звернення: 17.05.2025)
18. AWS CloudHSM Pricing. URL: https://aws.amazon.com/cloudhsm/pricing/ (дата звернення: 17.05.2025)
19. HashiCorp Vault Pricing. URL: https://www.hashicorp.com/products/vault/pricing (дата звернення: 17.05.2025)

ДОДАТОК А

Таблиця А.1 – Структура робіт за фазами життєвого циклу проєкту

Фаза життєвого циклу	**Черговість робіт**	**Тривалість, дні**	**Вартість, у.о.**	**Частка, %**
Передінвестиційна фаза	1.1 Збір вимог та аналіз проблем	1	7 862,48	3,98%
	1.2. Формування технічного завдання	1	8 849,59	4,48%
	1.3. Планування ресурсів	1	4 916,28	2,49%
	1.4. Оцінка ризиків	1	3 932,89	1,99%
	1.5. Побудова архітектури	1	6 883,31	3,48%
	1.6. Прототипування рішень	2	7 862,48	3,98%
Інвестиційна фаза	2.1. Розроблення клієнтської частини	4	33 431,37	16,92%
	2.2. Розроблення серверної частини	4	37 366,69	18,91%
	2.3. Створення смарт-контракту	3	17 688,08	8,95%
	2.4. Побудова анклаву та логіки ключів	4	23 586,16	11,94 %
	2.5. Інтеграція з блокчейном та сервером	3	15 723,76	7,96%
Експлуатаційна фаза	3.1. Тестування системи	1	8 849,59	4,48%
	3.2. Оптимізація витрат і продуктивності	1	6 883,31	3,48%
	3.3. Написання документації	1	5 909,25	2,99%

Продовження таблиці А.1

Фази життєвого циклу	**Черговість робіт**	**Тривалість, дні**	**Вартість, у.о.**	**частка, %**
Експлуатаційна фаза	3.4. Технічна підтримка MVP	2	7 862,48	3,98%
Разом	-	30	197 607,72	100,0%

Таблиця А.2 – Розрахунок амортизаційних відрахувань

	Вартість за одиницю, у.о.	**Розрахунок вартості за одиницю, у.о.**	**Розрахунок сукупної вартості, у.о.**
1. Ноутбук HP Pavilion 15 – 8 шт.			
Початкова вартість ноутбука	27 000	27 000	216 000
Амортизація ноутбука за 1 рік (термін експлуатації-3 роки)	27 000/3 років	9 000	72 000
Амортизація ноутбука за час реалізації проєкту	(390,3/365 днів)*30 днів	739,73	5 918
Загальна сума амортизації обладнання за проєкт			**5 918**

ДОДАТОК Б

```
elif action == "register":
    username = request.get("username")
    if not username:
        return {"status": "error", "message": "Username required"}

    # Ensure master key exists
    generate_master_key()

    # Generate a new key for the user
    user_key = generate_user_key()

    # Encrypt it with the master key
    encrypted_key = encrypt_with_master_key(user_key)

    # Split the encrypted key into shares
    total_shares = 5
    threshold = 3
    key_shares = split_encrypted_key(encrypted_key, total_shares, threshold)

    return {
        "status": "success",
        "result": {
            "key_shares": key_shares,
            "decrypted_key": base64.b64encode(user_key).decode()
        }
    }

# Auth success - retrieve a key for a successfully authenticated user
elif action == "auth_verify":
    key_shares = request.get("key_shares")
    if not key_shares:
        return {"status": "error", "message": "Key shares required"}

    # Ensure master key exists
    generate_master_key()

    # Combine shares to get the encrypted key
    try:
        encrypted_key = combine_key_shares(key_shares)
        decrypted_key = decrypt_with_master_key(encrypted_key)

        return {
            "status": "success",
            "result": {
                "decrypted_key": base64.b64encode(decrypted_key).decode()
            }
        }
    except Exception as e:
        print(f"Decryption error: {e}")
        return {"status": "error", "message": f"Key decryption failed: {str(e)}"}
```

Рисунок Б.1 – Реалізація обробника запитів на сервері Nitro Enclave