МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 62.00.00.000 ПЗ

Група ІПм-24-3

Паранчич Максим

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Паранчич Максим Романович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та алгоритми створення децентралізованої торгової платформи на основі блокчейну

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Паранчич М.Р.
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Зікратий Сергій Вікторович, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту
Завідувач кафедри
доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.
(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітньо-кваліфікаційний рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Паранчичу Максиму Романовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “Моделі, методи та алгоритми створення децентралізованої торгової платформи на основі блокчейну**”**
керівник проекту (роботи) Зікратий Сергій Вікторович, к.т.н., доцент
затверджені наказом закладу вищої освіти від **“** 09 ” листопада 2025 р. № 695/7
2. Строк подання студентом проекту (роботи) 15 грудня 2025 р.
3. Вихідні дані до проекту (роботи) Архітектура, формальний опис та алгоритми функціонування децентралізованої торгової платформи на основі блокчейну
4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз та дослідження використання технології блокчейну у веб-просторі
2. Математичні моделі та методи функціонування децентралізованої торгової платформи
3. Проєктування та вибір технологій реалізації децентралізованої платформи
4. Тестування та експериментальне дослідження децентралізованої торгової платформи
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
1. Трирівнева архітектура платформи (рис. 3.1, ст. 68)
2. Діаграма послідовності для UUPS Proxy Pattern (рис. 3.2, ст. 70)
3. Діаграма послідовності для підпису транзакції депозиту (рис. 3.3, ст. 72)
4. Діаграма послідовності для внесення депозиту (рис. 4.1, ст. 84)
5. Результат підключення гаманця до frontend додатку з тестовою мережею (рис. 4.2, ст. 85)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц. к.т.н. Вовк Р. Б.	

7. Дата видачі завдання 04 вересня 2025 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури	20.09.2025	виконано
2	Аналіз існуючих децентралізованих платформ та виявлення їх недоліків	01.10.2025	виконано
3	Розробка математичних моделей розподілу токенів, стейкінгу та делегування голосів	12.10.2025	виконано
4	Проектування архітектури системи з використанням UUPS проксі та EIP-712	25.10.2025	виконано
5	Реалізація смарт-контрактів	05.11.2025	виконано
6	Розробка веб-інтерфейсу та тестування системи, аудит безпеки, оптимізація	22.11.2025	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2025	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 97 с., 30 рис., 11 табл., 29 джерел.

Тема: Моделі, методи та алгоритми створення децентралізованої торгової платформи на основі блокчейну.

Об'єкт дослідження: процеси розподілу токенів, стейкінгу та управління на децентралізованих торгових платформах.

Мета роботи: розробка та дослідження математичних моделей, методів і алгоритмів для побудови безпечної та ефективної децентралізованої торгової платформи на основі технології блокчейн.

Предмет дослідження: математичні моделі розподілу токенів, алгоритми стейкінгу з винагородами, методи криптографічної верифікації транзакцій та механізми децентралізованого управління.

Результати дослідження:

Проведено аналіз існуючих децентралізованих платформ та виявлено їх недоліки: високі витрати газу, вразливості безпеки при взаємодії з зовнішніми контрактами, обмеження масштабованості. Розроблено комплексну математичну модель системи, що включає механізми вестингу, стейкінгу та делегування голосів.

Висновок:

У результаті дослідження створено децентралізовану торгову платформу з покращеними характеристиками безпеки та ефективності. Система базується на математично обґрунтованих моделях розподілу токенів, використовує сучасні криптографічні стандарти та демонструє значну економію обчислювальних ресурсів. Розроблений прототип може використовуватися стартапами для проведення IDO з гарантіями справедливого розподілу та захисту від типових атак на смарт-контракти.

ДЕЦЕНТРАЛІЗОВАНА ПЛАТФОРМА, БЛОКЧЕЙН, ETHEREUM, СМАРТ-КОНТРАКТИ, VESTING, СТЕЙКІНГ, UUPS ПРОКСІ, РОЗПОДІЛ ТОКЕНІВ, IDO, DEFI, КРИПТОГРАФІЧНА ВЕРИФІКАЦІЯ.

ANNOTATION

Master's work: 97 p., 30 fig., 11 tab., 29 sources.

Topic: Models, methods and algorithms for creating a decentralized trading platform based on blockchain.

Object of research: processes of token distribution, staking and governance on decentralized trading platforms.

Purpose of work: development and research of mathematical models, methods and algorithms for building a secure and efficient decentralized trading platform based on blockchain technology.

Subject of research: mathematical models of token distribution, staking algorithms with rewards, methods of cryptographic transaction verification and decentralized governance mechanisms.

Research results:

An analysis of existing decentralized platforms was conducted and their shortcomings were identified: high gas costs, security vulnerabilities when interacting with external contracts, scalability limitations. A comprehensive mathematical model of the system was developed, including vesting mechanisms, staking and vote delegation.

Conclusion:

As a result of the research, a decentralized trading platform with improved security and efficiency characteristics was created. The system is based on mathematically substantiated token distribution models, uses modern cryptographic standards and demonstrates significant savings in computational resources. The developed prototype can be used by startups to conduct IDOs with guarantees of fair distribution and protection against typical smart contract attacks.

DECENTRALIZED PLATFORM, BLOCKCHAIN, ETHEREUM, SMART CONTRACTS, VESTING, STAKING, UUPS PROXY, TOKEN DISTRIBUTION, IDO, DEFI, CRYPTOGRAPHIC VERIFICATION.

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

APR - Annual Percentage Rate, річна процентна ставка

APY - Annual Percentage Yield, річна процентна дохідність

BFT - Byzantine Fault Tolerance, стійкість до візантійських відмов

BPS - Basis Points, базисні пункти (1 BPS = 0.01%)

BSC - Binance Smart Chain, блокчейн-мережа Binance

CEI - Checks-Effects-Interactions, шаблон безпеки смарт-контрактів

DApp - Decentralized Application, децентралізований додаток

DeFi - Decentralized Finance, децентралізовані фінанси

DEX - Decentralized Exchange, децентралізована біржа

DLT - Distributed Ledger Technology, технологія розподіленого реєстру

ECDSA - Elliptic Curve Digital Signature Algorithm, алгоритм цифрового підпису на еліптичних кривих

EIP - Ethereum Improvement Proposal, пропозиція покращення Ethereum

EOA - Externally Owned Account, зовнішній обліковий запис

ERC - Ethereum Request for Comments, стандарт Ethereum

EVM - Ethereum Virtual Machine, віртуальна машина Ethereum

gas - одиниця виміру обчислювальних ресурсів в Ethereum

gwei - одиниця виміру Ether (1 gwei = 10^-9 ETH)

HTML - HyperText Markup Language, мова розмітки гіпертексту

HTTP - HyperText Transfer Protocol, протокол передачі гіпертексту

IDO - Initial DEX Offering, первинне розміщення на децентралізованій біржі

JSON - JavaScript Object Notation, формат обміну даними

KYC - Know Your Customer, процедура ідентифікації клієнта

L2 - Layer 2, рішення другого рівня масштабування

MEV - Maximal Extractable Value, максимальна вартість, що може бути витягнута

NFT - Non-Fungible Token, невзаємозамінний токен

PoS - Proof of Stake, доказ частки володіння

PoW - Proof of Work, доказ виконаної роботи

RPC - Remote Procedure Call, віддалений виклик процедур

SLA - Service Level Agreement, угода про рівень обслуговування

SLOAD - операція читання зі storage в EVM

SSTORE - операція запису в storage в EVM

TDD - Test-Driven Development, розробка через тестування

TPS - Transactions Per Second, транзакцій за секунду

TVL - Total Value Locked, загальна вартість заблокованих активів

UI - User Interface, користувацький інтерфейс

URL - Uniform Resource Locator, уніфікований покажчик ресурсу

UUPS - Universal Upgradeable Proxy Standard, універсальний стандарт оновлюваного проксі

UX - User Experience, користувацький досвід

VRF - Verifiable Random Function, верифікована випадкова функція

wei - найменша одиниця Ether (1 ETH = 10^18 wei)

ZK - Zero-Knowledge, з нульовим розголошенням

ВСТУП

Актуальність теми дослідження

У сучасному світі фінансових технологій спостерігається стрімкий розвиток децентралізованих систем, що базуються на технології блокчейн. За даними DeFi Llama, загальна вартість активів, заблокованих у DeFi протоколах, станом на 2025 рік перевищує 120 мільярдів доларів США [1], що свідчить про зростаючу довіру користувачів до децентралізованих фінансових інструментів.

Традиційні централізовані платформи для торгівлі токенами мають ряд суттєвих недоліків: необхідність довіри до центрального оператора, ризик цензури, відсутність прозорості операцій та можливість маніпуляцій з боку адміністрації. Децентралізовані торгові платформи вирішують ці проблеми через використання смарт-контрактів - самовиконуваних програм на блокчейні, які гарантують виконання умов угоди без участі посередників.

Особливу актуальність набуває питання справедливого розподілу токенів під час первинних розміщень (IDO, ITO). Аналіз існуючих платформ, таких як Polkastarter, DAO Maker та TrustSwap, виявив проблему високої вартості транзакцій (gas costs) та обмеженої функціональності щодо гнучкого управління правами доступу. Крім того, багато сучасних рішень не підтримують механізми оновлення (upgradeability), що унеможливлює виправлення помилок або додавання нових функцій без повного перерозгортання системи.

Стейкінг-механізми, які дозволяють користувачам заробляти пасивний дохід від блокування токенів, також потребують вдосконалення. Існуючі рішення часто не враховують такі аспекти, як делегування, що є критично важливим для децентралізованого управління. Математичні моделі розрахунку винагород повинні бути прозорими, справедливими та оптимізованими з точки зору витрат на виконання.

Питання безпеки залишається одним з найважливіших у розробці смарт-контрактів. За даними аналітичних звітів, у 2022-2023 роках внаслідок вразливостей у смарт-контрактах було втрачено понад 3 мільярди доларів [2]. Це підкреслює

необхідність використання перевірених паттернів програмування, комплексного тестування та формальної верифікації алгоритмів.

Стандарт EIP-712 для підпису структурованих даних надає можливість створення безпечних механізмів авторизації, які захищають користувачів від фішингу та replay-атак. Однак, його впровадження в контексті торгових платформ потребує розробки спеціалізованих математичних моделей верифікації.

Технологія UUPS (Universal Upgradeable Proxy Standard) дозволяє оновлювати логіку смарт-контрактів без зміни адреси та зберігаючи стан даних, що є критично важливим для довгострокової експлуатації платформи. Проте вибір оптимального proxy pattern потребує ретельного аналізу та порівняння з альтернативами (Transparent Proxy, Beacon Proxy).

Сучасні дослідження у сфері блокчейн-технологій показують зростаючий інтерес до методів підвищення безпеки та ефективності децентралізованих додатків [3]. Розробка нових підходів до архітектури смарт-контрактів, оптимізації gas costs та забезпечення безпеки є актуальними напрямками досліджень.

З огляду на зазначені проблеми, актуальною є задача розробки комплексної математичної моделі, методів та алгоритмів для створення децентралізованої торгової платформи, яка поєднує механізми справедливого розподілу токенів, стейкінгу, захищені криптографічні протоколи верифікації та можливість безпечного оновлення.

Мета і задачі дослідження

Метою роботи є розробка моделей, методів та алгоритмів створення децентралізованої торгової платформи на основі блокчейну Ethereum з підтримкою механізмів продажу токенів, стейкінгу та делегування.

Для досягнення поставленої мети необхідно вирішити наступні **задачі**:

1. Провести аналіз існуючих децентралізованих торгових платформ, виявити їх переваги та недоліки, дослідити застосування стандартів ERC-20, EIP-712 та технологій Zero-Knowledge у блокчейн системах.

2. Розробити математичні моделі:

- модель справедливого розподілу токенів з урахуванням різниці у кількості десяткових знаків (decimals);
- модель поступового розблокування токенів (vesting) з підтримкою cliff періоду та початкового розблокування;
- модель верифікації EIP-712 підписів з використанням nonce для запобігання replay-атакам;
- модель розрахунку винагород у стейкінг-системі з урахуванням APR та часу блокування;

3. Розробити алгоритми:

- алгоритм розрахунку доступної кількості токенів;
- алгоритм запобігання reentrancy атакам на основі паттерну Checks-Effects-Interactions;
- алгоритм оновлення смарт-контрактів за допомогою UUPS proxy pattern.

4. Здійснити проєктування архітектури системи:

- обґрунтувати вибір технологічного стеку;
- розробити архітектуру взаємодії компонентів системи (смарт-контракти, frontend, wallet integration);
- спроєктувати структуру даних та storage layout для upgradeable контрактів.

5. Реалізувати програмне забезпечення:

- розробити смарт-контракти SalePool, Staking, Registry, ERC20Mintable на мові Solidity;
- реалізувати frontend додаток на базі Next.js 14 з інтеграцією Web3 wallet;
- створити систему підпису EIP-712 для безпечної авторизації транзакцій.

6. Провести експериментальні дослідження:

- виконати unit та integration тестування з покриттям коду понад 95%;
- здійснити порівняльний аналіз gas costs з конкуруючими платформами;
- провести навантажувальне тестування та deployment у testnet мережу Sepolia;
- виконати security audit з використанням інструментів Slither та Hardhat.

Об’єкт і предмет дослідження

Об’єктом дослідження є процеси створення та функціонування децентралізованих торгових платформ на основі технології блокчейн.

Предметом дослідження є моделі, методи та алгоритми розподілу токенів, стейкінгу, верифікації підписів і оновлення контрактів у децентралізованих системах.

Методи дослідження

Для вирішення поставлених задач використовувалися наступні **методи**:

- методи криптографії для реалізації цифрових підписів на основі алгоритму ECDSA та стандарту EIP-712;
- методи алгоритмізації для розробки ефективних обчислювальних процедур з оптимальною часовою складністю;
- методи об’єктно-орієнтованого програмування для створення модульної архітектури смарт-контрактів;
- методи формального тестування для верифікації коректності роботи алгоритмів та виявлення вразливостей;
- методи математичного моделювання для формалізації процесів розподілу токенів та розрахунку винагород;
- методи порівняльного аналізу для оцінки ефективності розроблених рішень відносно існуючих аналогів.

Наукова новизна отриманих результатів полягає у наступному:

1. Розроблено комплексну математичну модель децентралізованої торгової платформи, яка об’єднує механізми розподілу токенів, систему стейкінгу та верифікацію, що забезпечує прозорість операцій та захист від маніпуляцій.

2. Удосконалено модель vesting токенів шляхом введення параметра початкового розблокування (initial unlock) та підтримки cliff періоду, що дозволяє гнучко налаштовувати графік розблокування для різних категорій учасників з часовою складністю обчислень O(1).

3. Розроблено алгоритм верифікації EIP-712 підписів з інтеграцією nonce-based захисту від replay-атак, що підвищує безпеку транзакцій та захищає користувачів від фішингу порівняно зі стандартними методами підпису.

4. Запропоновано метод оптимізації gas costs у proxy-контрактах шляхом використання UUPS pattern замість Transparent Proxy, що забезпечує зниження витрат на кожен виклик функцій на 70% при збереженні високого рівня безпеки.

Практична значущість результатів роботи полягає у тому, що:

1. Розроблено повнофункціональну децентралізовану платформу для продажу токенів та стейкінгу, яку може бути використано для проведення Initial DEX Offerings (IDO) та Initial Token Offerings (ITO) з прозорими умовами розподілу.

2. Забезпечено відкритий доступ до смарт-контрактів, що пройшли комплексне тестування з покриттям коду понад 95% та можуть бути використані іншими розробниками для створення аналогічних систем.

3. Реалізовано frontend додаток з інтуїтивним інтерфейсом, який забезпечує зручну взаємодію користувачів з блокчейном через інтеграцію популярних криптовалютних гаманців (MetaMask, WalletConnect).

4. Проведено deployment у тестову мережу з верифікацією контрактів на Etherscan, що дозволяє публічно перевірити коректність реалізації та прозорість роботи системи.

5. Розроблені методи та алгоритми можуть бути адаптовані для створення інших типів DeFi додатків: lending platforms, decentralized exchanges (DEX), liquidity pools, що розширює сферу їх застосування.

Структура магістерської роботи

Магістерська робота викладена на 97 сторінках друкованого тексту, який складається з вступу, чотирьох розділів, висновків, списку використаних джерел (29 найменувань). Робота містить 30 рисунків та 11 таблиць і два додатки.

РОЗДІЛ 1
АНАЛІЗ ТА ДОСЛІДЖЕННЯ ВИКОРИСТАННЯ ТЕХНОЛОГІЇ БЛОКЧЕЙНУ У ВЕБ-ПРОСТОРІ

1.1 Блокчейн технології як інноваційний спосіб зберігання інформації

Блокчейн є розподіленою базою даних, яка зберігає інформацію у вигляді послідовності блоків, з'єднаних між собою криптографічними хешами. Кожен блок містить набір транзакцій, часову мітку та хеш попереднього блоку, що створює незмінний ланцюг записів [3, 4].

Концепція розподіленого реєстру (Distributed Ledger Technology, DLT) передбачає зберігання копії всієї бази даних на множині незалежних вузлів мережі. На відміну від традиційних централізованих баз даних, де існує єдина точка контролю та можливого відмови, розподілений реєстр забезпечує високу відмовостійкість та відсутність можливості цензури з боку окремих учасників [3].

1.1.1 Структура блоку та принципи роботи

Кожен блок у блокчейні складається з двох основних частин: заголовка (header) та тіла блоку (body). Заголовок містить метадані: хеш попереднього блоку, часову мітку, nonce (для Proof of Work) та кореневий хеш дерева Меркла транзакцій. Тіло блоку містить список транзакцій, які були включені у блок.

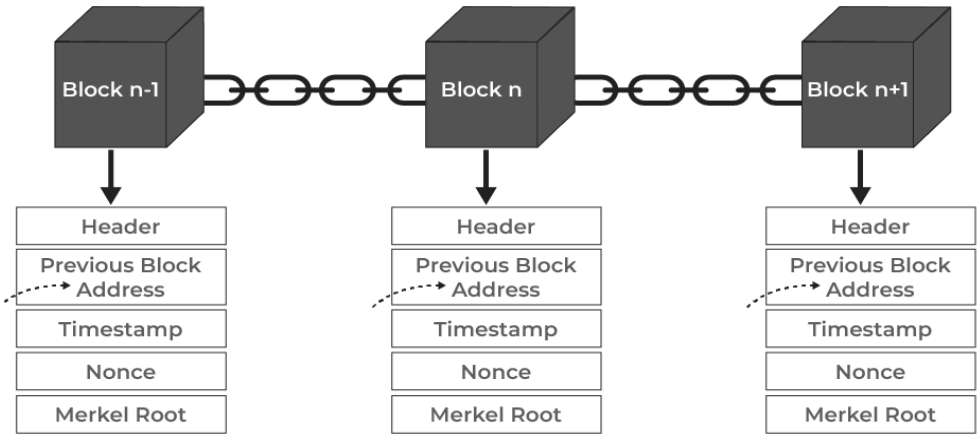

Рис. 1.1. Структура блоку блокчейн

Незмінність даних у блокчейні досягається через використання криптографічних хеш-функцій. Будь-яка зміна в історичному блоці призведе до зміни його хешу, що автоматично інвалідує всі наступні блоки у ланцюгу. Це робить практично неможливим фальсифікацію даних без контролю над більшістю обчислювальної потужності мережі [3].

1.1.2 Принципи роботи розподілених реєстрів

Розподілений реєстр функціонує на основі peer-to-peer мережі, де кожен вузол зберігає повну копію блокчейну. Коли користувач ініціює транзакцію, вона транслюється всім вузлам мережі через протокол gossip. Вузли валідують транзакцію, перевіряючи цифрові підписи та достатність балансу відправника.

Ключовою проблемою розподілених систем є досягнення консенсусу між незалежними учасниками, особливо в умовах можливої присутності зловмисних нод. Ця проблема відома як Byzantine Fault Tolerance (BFT) - здатність системи функціонувати навіть якщо частина нод діє некоректно або зловмисно [4].

1.1.3 Механізми консенсусу

Proof of Work (PoW) - перший успішний консенсус-механізм, використаний у Bitcoin. Майнери конкурують у вирішенні обчислювально-складної задачі підбору nonce, щоб хеш блоку відповідав певній складності. Переможець отримує право додати блок до ланцюга та винагороду у вигляді нових монет. Основним недоліком PoW є надзвичайно високе енергоспоживання - мережа Bitcoin споживає близько 150 TWh електроенергії на рік [5].

Proof of Stake (PoS) - альтернативний механізм, який замінює обчислювальну роботу на економічний стейк. Валідатори блокують (stake) власні токени як заставу, отримуючи право створювати блоки пропорційно до розміру стейку. У разі некоректної поведінки валідатор втрачає частину застави (slashing). Ethereum перейшов на PoS у 2022 році, зменшивши енергоспоживання на 99.95% [6].

Таблиця 1.1

Порівняння PoW та PoS

Критерій	Proof of Work (PoW)	Proof of Stake (PoS)
Як працює	Майнери розв’язують складні математичні задачі для створення блоку	Валідатори блокують (стейкають) криптовалюту і випадково обираються для створення блоку
Хто створює блоки	Той, хто першим знайде правильний хеш (майнер)	Той, кого вибере алгоритм серед валідаторів
Витрати енергії	Дуже високі	Низькі
Обладнання	Потрібні ASIC/потужні GPU	Достатньо ноутбука/сервера з інтернетом
Безпека	Ґрунтується на дорогому обладнанні та енергоспоживанні	Ґрунтується на економічному стимулі валідаторів (ризик втрати стейку)
Масштабованість	Низька - обмежена пропускна здатність	Вища - легше масштабувати
Швидкість блоків	Повільніша	Швидша
Винагорода	Блокова нагорода та комісії	Стейкінг-винагорода та комісії
Приклади	Bitcoin, Litecoin	Ethereum, Cardano, Solana, Polkadot

1.1.4 Переваги децентралізації

Децентралізовані системи мають ряд фундаментальних переваг над централізованими аналогами:

1. Відсутність єдиної точки відмови (single point of failure). У централізованій системі вихід з ладу центрального сервера означає повну непрацездатність системи. У блокчейні навіть при відключенні 50% нод мережа продовжує функціонувати.

2. Стійкість до цензури (censorship resistance). Жоден окремий учасник не може заблокувати транзакцію або виключити користувача з системи. Це особливо

важливо у країнах з авторитарними режимами, де доступ до фінансових послуг може бути обмежений з політичних причин.

3. Прозорість та можливість аудиту (transparency and auditability). Всі транзакції у публічному блокчейні доступні для перегляду будь-яким учасником. Це забезпечує повну прозорість операцій та можливість незалежної верифікації [4].

4. Trustless операції. Користувачі можуть взаємодіяти без необхідності довіряти один одному або третій стороні. Довіра покладається на математичні гарантії криптографії та консенсус-механізми [3].

1.1.5 Проблеми масштабування та рішення

Однією з ключових проблем блокчейн технологій є так звана "blockchain trilemma" - неможливість одночасно досягти високої децентралізації, безпеки та масштабованості [7]. Bitcoin обробляє лише 7 транзакцій на секунду, Ethereum до переходу на PoS - близько 15 TPS, що значно поступається централізованим системам (Visa - 65,000 TPS).

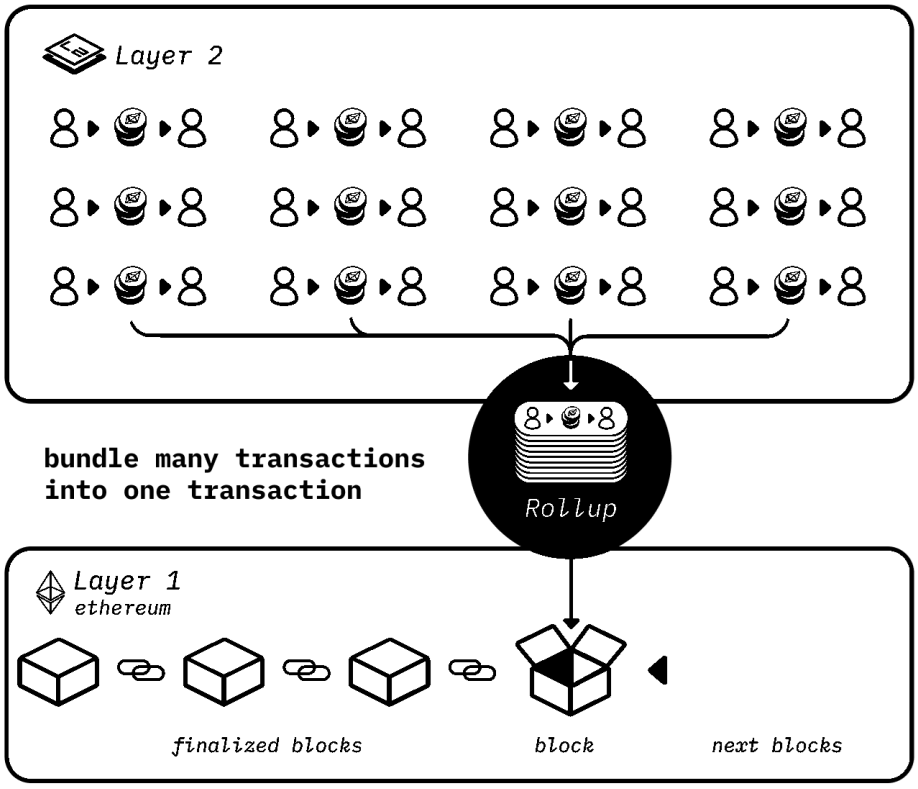

Рис. 1.2. Layer 2 архітектура

Для вирішення проблеми масштабування розробляються рішення Layer 2 - протоколи, що працюють "поверх" основного блокчейну:

Rollups агрегують сотні транзакцій в один batch, який записується у mainnet. Optimistic Rollups (Arbitrum, Optimism) припускають коректність транзакцій за замовчуванням, використовуючи fraud proofs для виявлення шахрайства. ZK-Rollups (zkSync, StarkNet) використовують zero-knowledge доведення для криптографічної верифікації коректності [8].

State Channels дозволяють учасникам проводити необмежену кількість транзакцій off-chain, записуючи у блокчейн лише початковий та фінальний стан. Це ідеально для мікроплатежів та застосунків у режимі реального часу.

Sidechains - окремі блокчейни, що з'єднуються з основним ланцюгом через bridge контракти. Polygon, Binance Smart Chain функціонують як sidechain рішення з власними консенсус-механізмами та нижчими комісіями.

1.2 Аналіз та принципи функціонування блокчейн мережі Ethereum

Ethereum є другою за капіталізацією блокчейн платформою після Bitcoin, але першою, яка впровадила повноцінні смарт-контракти - самовиконувані програми, що працюють на блокчейні. Мережа Ethereum була запущена у 2015 році та на сьогодні є найбільшою платформою для децентралізованих додатків (DApps) [4].

1.2.1 Архітектура мережі Ethereum

На відміну від Bitcoin, який використовує модель UTXO (Unspent Transaction Output), Ethereum працює на основі моделі облікових записів (account model). Існує два типи акаунтів: зовнішні облікові записи (EOA), контрольовані приватними ключами користувачів, та контрактні облікові записи, які містять код смарт-контракту.

Кожен акаунт має чотири поля: nonce (лічильник транзакцій), balance (баланс в wei), storageRoot (хеш кореня дерева зберігання даних) та codeHash (хеш коду контракту, для EOA це хеш порожнього рядка).

Стан мережі Ethereum представлений як mapping між адресами та станами облікових записів. Після виконання кожного блоку транзакцій стан оновлюється, формуючи новий state root, який зберігається у заголовку блоку.

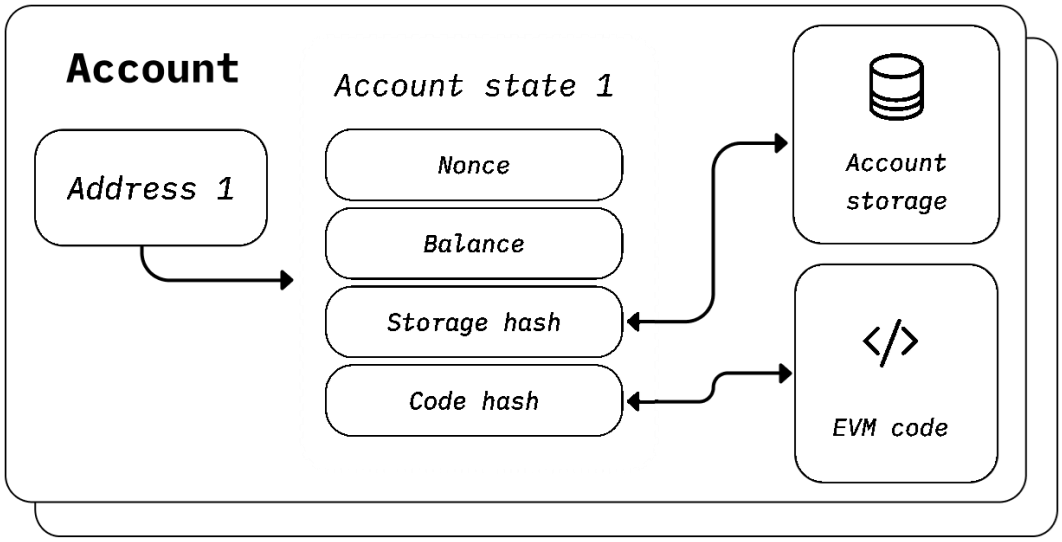

Рис. 1.3. Ethereum Account Model

1.2.2 Віртуальна машина Ethereum (EVM)

EVM (Ethereum Virtual Machine) є віртуальною машиною, яка виконує байткод смарт-контрактів. Це середовище виконання, що означає можливість реалізації будь-якого обчислювального алгоритму.

EVM працює як стекова машина з розміром слова 256 біт, що оптимізовано для роботи з криптографічними операціями на еліптичних кривих. Під час виконання транзакції EVM має доступ до трьох типів пам'яті:

1. Storage - постійне сховище, що зберігається між викликами. Це найдорожчий тип пам'яті (20,000 gas для запису нового значення). Дані організовані як key-value mapping, де і ключ, і значення мають розмір 32 байти.

2. Memory - тимчасова пам'ять, яка очищається після завершення транзакції. Використовується для зберігання тимчасових змінних під час виконання. Значно дешевша за Storage.

3. Stack - стек з максимальною глибиною 1024 елементи. Використовується для передачі параметрів між opcodes.

Виконання коду в EVM є детерміністичним - однаковий вхід завжди призводить до однакового результату, незалежно від того, на якій ноді виконується код. Це критично важливо для досягнення консенсусу в розподіленій мережі.

1.2.3 Gas механізм та вартість транзакцій

Gas є одиницею виміру обчислювальних ресурсів в Ethereum. Кожна операція (opcode) коштує фіксовану кількість gas: ADD - 3 gas, MUL - 5 gas, SSTORE - 20,000 gas для нового значення тощо. Користувач встановлює gas price - скільки він готовий заплатити за одиницю gas. Загальна вартість транзакції обчислюється як:

$$Tx\ Fee = Gas\ Used * Gas\ Price$$

Після впровадження EIP-1559 у 2021 році механізм ціноутворення змінився. Тепер gas price складається з двох компонентів: base fee (автоматично визначається протоколом залежно від завантаження мережі) та priority fee (чайові валідаторам). Base fee спалюється, зменшуючи загальну кількість ETH в обігу [9].

1.2.4 Consensus Layer та перехід на Proof of Stake

У вересні 2022 року Ethereum здійснив історичний перехід з Proof of Work на Proof of Stake в події, відомій як The Merge [6]. Нова архітектура складається з двох шарів:

- Execution Layer (колишній Ethereum 1.0) - відповідає за виконання транзакцій, підтримку стану та EVM.

- Consensus Layer (Beacon Chain) - відповідає за консенсус, фінальність блоків та управління валідаторами.

Для участі у валідації необхідно застейкати мінімум 32 ETH. Валідатори обираються псевдовипадково для пропозиції та атестації блоків. За коректну роботу вони отримують винагороди, за некоректну - караються втратою частини застави (slashing).

Перехід на PoS зменшив енергоспоживання мережі на 99.95%, з приблизно 94 TWh/рік до менше ніж 0.01 TWh/рік. Це еквівалентно виключенню енергоспоживання країни розміром з Нідерланди [6].

1.2.5 Тестові мережі для розробки

Для розробки та тестування смарт-контрактів використовуються тестові мережі (testnets), які максимально наближені до mainnet, але використовують токени без реальної вартості.

Sepolia - рекомендована тестова мережа. Використовує PoS консенсус, має стабільний набір валідаторів. Тестові ETH можна отримати через faucets (безкоштовні крани).

Для локальної розробки використовують Hardhat Network - локальний Ethereum node, що запускається на машині розробника та дозволяє миттєво тестувати контракти без очікування підтвердження блоків.

1.3 Криптографічні основи блокчейн технологій

Криптографія є фундаментальною основою блокчейн технологій, забезпечуючи безпеку, автентичність та цілісність даних у децентралізованих системах. Без криптографічних примітивів неможливо було б гарантувати, що транзакції дійсно підписані власником коштів, або що дані у блоках не були змінені.

1.3.1 Криптографічні хеш-функції

Хеш-функція перетворює вхідні дані довільної довжини у вихідне значення фіксованої довжини (хеш або digest). Криптографічно стійкі хеш-функції повинні мати три основні властивості:

1. Детермінованість - однакові вхідні дані завжди дають однаковий хеш.
2. Односпрямованість - практично неможливо обчислити вхідні дані маючи лише хеш (preimage resistance).

3. Стійкість до колізій - практично неможливо знайти два різних входи, що дають однаковий хеш (collision resistance).

У Bitcoin використовується SHA-256 (Secure Hash Algorithm 256-bit) для хешування блоків та транзакцій. Вихідне значення має довжину 256 біт (32 байти). Ethereum використовує Keccak-256 - варіант SHA-3, який був обраний ще до офіційної стандартизації SHA-3.

1.3.2 Асиметрична криптографія та ECDSA

Асиметрична (публічна) криптографія використовує пару ключів: приватний ключ для підпису та публічний ключ для верифікації. Ethereum використовує алгоритм ECDSA (Elliptic Curve Digital Signature Algorithm) на кривій secp256k1.

Математична основа базується на властивостях еліптичних кривих. Крива secp256k1 описується рівнянням:

$$y^2 = x^3 + 7 \ (mod\ p), \tag{1.1}$$

де p - велике просте число.

Процес генерації адреси:

1. Генерується випадковий приватний ключ (256 біт)
2. Обчислюється публічний ключ через множення точки на кривій:
3.

$$PubKey = PrivKey * G, \tag{1.2}$$

де G - базова точка.

4. Хешується публічний ключ: keccak256(PubKey)
5. Береться останні 20 байт хеша - це адреса Ethereum.

Приватний ключ має бути захищений будь-якою ціною. Знаючи приватний ключ, зловмисник отримує повний контроль над усіма коштами на адресі.

1.3.3 Цифрові підписи

Цифровий підпис доводить, що транзакція була створена власником приватного ключа, без необхідності розкривати сам ключ. ECDSA підпис складається з трьох компонентів: (v, r, s).

Процес створення підпису:

1. Обчислюється хеш повідомлення: h = keccak256(message).
2. Генерується випадкове число k.
3. Обчислюється точка на кривій: (x, y) = k × G.
4. r = x mod n (де n - порядок кривої).
5. $s = k^{-1} * (h + r * PrivKey) \bmod n$.
6. v - recovery id (0 або 1), що дозволяє відновити публічний ключ.

Функція ecrecover є precompiled контрактом у Ethereum, доступним за адресою 0x01, що робить верифікацію підписів gas-ефективною (3000 gas).

1.3.4 Стандарт EIP-712 для структурованих даних

Стандартні методи підпису (eth_sign, personal_sign) мають суттєві недоліки з точки зору безпеки та user experience. Користувач бачить лише неструктурований хеш, не розуміючи що саме підписує. Це створює вразливість до фішинг атак.

EIP-712 вирішує цю проблему, впроваджуючи typed structured data hashing. Підпис включає три компоненти:

1. Domain Separator - унікально ідентифікує смарт-контракт та мережу:

```
DOMAIN_SEPARATOR = keccak256(abi.encode(
TYPE_HASH,
keccak256("CoinboxSalePool"),
keccak256("1"),
block.chainid,
address(this)
));
```

2. Type Hash - ідентифікує структуру даних:

```
TYPE_HASH = keccak256("Deposit(uint256 poolId,address user,uint256 amount,uint256 nonce)");
```

3. Struct Hash - хеш конкретних даних:

```
structHash = keccak256(abi.encode(TYPE_HASH, poolId, user, amount, nonce));
```

Фінальний digest обчислюється як:

```
digest = keccak256(abi.encodePacked("\\x19\\x01", DOMAIN_SEPARATOR, structHash));
```

Це захищає від replay атак між різними контрактами та мережами. Навіть якщо зловмисник отримає підпис для одного контракту, він не зможе використати його в іншому, оскільки DOMAIN_SEPARATOR буде відрізнятися.

1.3.5 Дерева Меркла

Дерево Меркла (Merkle Tree) - це бінарне дерево хешів, яке дозволяє ефективно верифікувати приналежність елемента до набору без необхідності зберігання всього набору.

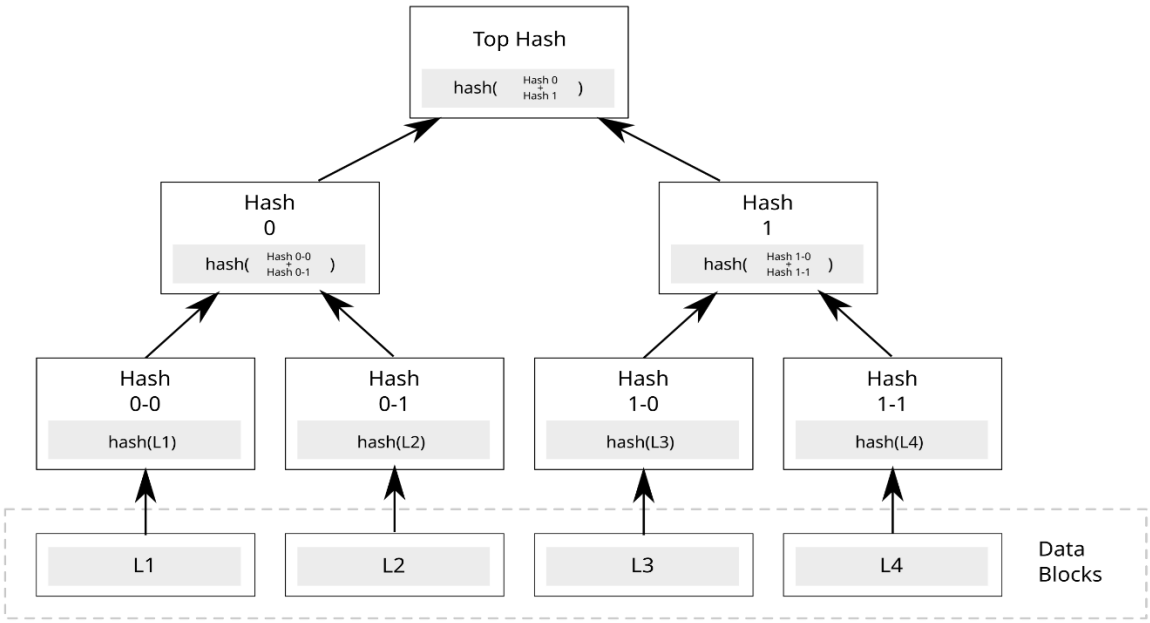

Рис. 1.4. Структура Merkle Tree

Властивості:

- Merkle Root (кореневий хеш) зберігається у заголовку блоку;
- Для доведення включення транзакції потрібно лише O(log n) хешів (Merkle Proof);
- Зміна будь-якої транзакції змінює Merkle Root.

У Ethereum дерева Меркла використовуються для:

- Transaction Tree - всі транзакції у блоці;
- State Tree - стани всіх облікових записів;
- Receipt Tree - результати виконання транзакцій;
- Storage Tree - storage кожного контракту.

1.4 Стандарт ERC-20 та його застосування для торгових платформ

ERC-20 (Ethereum Request for Comments 20) є найпопулярнішим стандартом токенів на Ethereum, який визначає спільний інтерфейс для взаємозамінних токенів. Стандарт був запропонований Fabian Vogelsteller у 2015 році та офіційно прийнятий як EIP-20.

1.4.1 Специфікація стандарту

Стандарт ERC-20 визначає шість обов'язкових функцій та дві події. Функції читання (view):

- totalSupply() - повертає загальну кількість токенів в обігу;
- balanceOf(address) - повертає баланс конкретної адреси;
- allowance(owner, spender) - повертає дозволену суму для витрат.

Функції запису:

- transfer(to, amount) - переказує токени з акаунту msg.sender на адресу to;
- approve(spender, amount) - дозволяє spender витратити amount токенів;
- transferFrom(from, to, amount) - переказує токени від імені власника (потребує попереднього approve).

Події (events):

- Transfer(from, to, amount) - емітується при кожному переказі;
- Approval(owner, spender, amount) - емітується при зміні allowance.

Додатково стандарт рекомендує три metadata поля:

- name - повна назва токену ("Coinbox Token");
- symbol - тікер токену ("CBOX");
- decimals - кількість десяткових знаків (зазвичай 18).

1.4.2 Розширення стандарту

Базовий ERC-20 часто розширюється додатковим функціоналом для конкретних use cases:

- ERC20Mintable - можливість створення нових токенів. Використовується у token sales, де токени створюються на вимогу при депозиті користувача. Потребує access control (тільки authorized адреси можуть mint).
- ERC20Burnable - можливість знищення токенів. Зменшує totalSupply, може використовуватись для дефляційних механік або спалення fees.
- ERC20Pausable - можливість призупинити всі трансфери у разі виявлення критичної вразливості. Emergency pause механізм для безпеки.
- ERC20Snapshot - створення "знімків" балансів на певний момент часу. Використовується для vote power calculation у governance системах, де право голосу визначається балансом на момент proposal creation.

У розробленій системі використовується ERC20Mintable контракт для reward токенів, які створюються при продажу у SalePool контракті [10].

1.4.3 Переваги для децентралізованих платформ

Стандартизація через ERC-20 надає критичні переваги:

1. Interoperability - токени автоматично сумісні з тисячами існуючих сервісів: гаманці (MetaMask, Coinbase Wallet), біржі (Uniswap, 1inch), block explorers (Etherscan), analytics platforms.
2. Composability - концепція "DeFi Lego". Токени можна комбінувати між різними протоколами. Наприклад, застейкати LP токени Uniswap в Aave для отримання кредиту.
3. Battle-tested implementations - бібліотеки OpenZeppelin містять аудитовані імплементації з багаторічною історією використання в production. Це зменшує ризик вразливостей.
4. Network effects - наявність великої екосистеми розробників, користувачів та інструментів навколо ERC-20 робить його де-факто стандартом для fungible токенів.

За даними дослідження 2022 року, понад 500,000 унікальних ERC-20 контрактів були задеплоєні на Ethereum, з щоденним обсягом транзакцій понад 2 мільйони [11].

1.5 Огляд існуючих децентралізованих торгових платформ

1.5.1 Automated Market Makers (AMM)

Uniswap - найпопулярніший децентралізований обмінник (DEX), що використовує алгоритм constant product formula для визначення цін. Замість традиційного order book, ліквідність надається у пули (pools), де співвідношення токенів визначає ціну за формулою:

$$x * y = k, \tag{1.3}$$

де x та y - кількість двох токенів у пулі, k - константа.

Основною проблемою AMM є impermanent loss - втрата ліквідності провайдерів при зміні цін токенів. Uniswap V3 вирішує це через concentrated liquidity, де провайдери можуть задати ціновий діапазон для своєї ліквідності.

PancakeSwap - аналог Uniswap на Binance Smart Chain. Переваги: значно нижчі комісії ($0.1-0.5 замість $10-50 на Ethereum mainnet). Недолік: менша децентралізація через обмежену кількість валідаторів BSC.

1.5.2 Launchpad платформи для IDO

Polkastarter - одна з найбільших launchpad платформ для Initial DEX Offerings. Середній розмір залучення - $500,000 per project. Використовує систему білого списку для допуску учасників та лотерею для розподілу allocation при oversubscription [25].

Недоліки: централізований контроль whitelist, високі fees ($50-100 per transaction на Ethereum), відсутність native upgradeability механізмів.

DAO Maker - платформа з унікальним Strong Holder Offering (SHO) механізмом. Розподіл allocation базується на тривалості холдингу platform токену. Community-driven vetting процес для відбору проектів [26].

TrustSwap - фокус на vesting та escrow рішеннях. Надає Time-Locked Wallets для автоматичного розблокування токенів згідно з vesting schedule. Підтримує multi-chain (Ethereum, BSC, Polygon) [27].

Таблиця 1.2

Порівняння launchpad платформ

Платформа	Avg Raise	Блокчейн	Комісії	Можливість оновлення	Підтримка поступового розблокування
Polkastarter	$500k	мультиплатформенний	Високі	Часткова	Базова
DAO Maker	$300k	Ethereum	Високі	Немає	Розширена
TrustSwap	$250k	мультиплатформенний	Середні	Немає	Розширена

1.5.3 Виявлені недоліки існуючих рішень

Аналіз існуючих платформ виявив наступні проблеми:

- Відсутність upgradeability - більшість контрактів є immutable. Це означає неможливість виправлення багів або додавання features без повного редеплою та міграції користувачів. Лише Polkastarter має часткову підтримку через proxy, але використовує Transparent Proxy з високими gas costs [20].
- Високі транзакційні витрати - на Ethereum mainnet вартість участі у IDO може сягати $50-200 в періоди завантаження мережі. Це робить недоступною участь для роздрібних інвесторів з малими сумами.
- Централізація у KYC/whitelist - процес допуску учасників часто контролюється централізовано командою проекту, що створює ризики цензури та фаворитизму.
- Вразливість до MEV атак - транзакції видимі у mempool до включення у блок, що дозволяє ботам проводити front-running та отримувати кращі ціни за звичайних користувачів.

- Складний UX - взаємодія з Web3 залишається складною для новачків: необхідність налаштування гаманця, купівля ETH для gas, розуміння approve/transfer механізму.

Розроблена в рамках даної роботи платформа вирішує ці проблеми через використання UUPS proxy для upgradeability, EIP-712 підписів для покращення UX та безпеки, та оптимізацію gas costs через ефективні структури даних [12, 19].

1.6 Смарт-контракти та інструменти розробки

Смарт-контракти є програмами, що виконуються на блокчейні та автоматично реалізують умови угоди без посередників. Концепція була вперше описана Ніком Сабо у 1994 році, але практичну реалізацію отримала лише з появою Ethereum.

1.6.1 Мова програмування Solidity

Solidity - об'єктно-орієнтована мова для написання смарт-контрактів, розроблена спеціально для EVM. Синтаксис схожий на JavaScript та C++, що полегшує входження для розробників з традиційного програмування.

Ключові особливості Solidity 0.8.x:

1. Вбудований overflow/underflow захист - автоматичні перевірки при арифметичних операціях (у версіях до 0.8 потрібна була бібліотека SafeMath).
2. Custom errors - економить gas замість require з string повідомленнями:

```
if (balance < amount) revert InsufficientBalance(balance, amount);
```

3. Immutable та constant змінні - оптимізація зберігання для значень, що не змінюються.
4. User-defined value types - створення custom типів для type safety:

```
type TokenAmount is uint256;
```

Приклад простого контракту:

```
contract SalePool {
    mapping(uint256 => Pool) public pools;
    uint256 public poolCount;

    struct Pool {
        address rewardToken;
        uint256 price;
        uint256 supply;
```

```
        bool isActive;
    }

    function createPool(
        address rewardToken,
        uint256 price,
        uint256 supply
    ) external returns (uint256) {
        uint256 poolId = poolCount++;
        pools[poolId] = Pool({
            rewardToken: rewardToken,
            price: price,
            supply: supply,
            isActive: true
        });
        return poolId;
    }
}
```

1.6.2 Паттерни проектування

Checks-Effects-Interactions (CEI) - найважливіший паттерн для запобігання reentrancy атак:

```
function withdraw(uint256 amount) external {
    // 1. CHECKS
    require(balances[msg.sender] >= amount);
    // 2. EFFECTS
    balances[msg.sender] -= amount;
    // 3. INTERACTIONS
    payable(msg.sender).transfer(amount);
}
```

Factory Pattern - створення нових контрактів через factory:

```
contract TokenFactory {
    function createToken(string name) external returns (address) {
        ERC20Token token = new ERC20Token(name);
        return address(token);
    }
}
```

Access Control - розмежування прав доступу через ролі:

```
contract Managed {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN");
    modifier onlyAdmin() {
        require(hasRole(ADMIN_ROLE, msg.sender));
        _;
    }
}
```

1.6.3 Безпека смарт-контрактів

Історичні hack’и демонструють критичну важливість безпеки. DAO hack 2016 року ($60M втрачено) був спричинений reentrancy вразливістю. Parity wallet hack 2017 року ($150M заморожено) - через помилку у access control.

Основні типи вразливостей:

1. Reentrancy - повторний виклик функції до завершення першого виклику. Захист: ReentrancyGuard modifier, CEI pattern.

2. Integer overflow - у Solidity < 0.8 арифметичні операції могли переповнюватись. Захист: автоматичний у 0.8+, або SafeMath у старіших версіях.

3. Front-running (MEV) - зловмисник бачить транзакцію у mempool і надсилає власну з вищим gas price. Захист: commit-reveal schemes, private mempools, EIP-712 signatures.

Для мінімізації ризиків використовуються:

- Аудити від спеціалізованих компаній (ConsenSys Diligence, Trail of Bits);
- Автоматичні інструменти аналізу (Slither, Mythril, Echidna);
- Формальна верифікація (математичне доведення коректності);
- Bug bounty програми для ethical hackers [12].

1.6.4 UUPS Upgradeable Pattern

Проблема immutability контрактів вирішується через proxy patterns. Існує кілька варіантів:

Transparent Proxy - логіка upgrade у proxy контракті. Недолік: додаткові 2600 gas на кожен виклик для перевірки чи msg.sender є admin.

UUPS (Universal Upgradeable Proxy Standard) - логіка upgrade у implementation контракті. Перевага: економія ~70% gas costs на викликах. Недолік: потрібно не забути включити upgrade функцію в implementation.

Beacon Proxy - множина proxies вказують на один beacon, який зберігає адресу implementation. Зручно для масового upgrade багатьох контрактів одночасно.

Схема UUPS:

1. Proxy зберігає адресу Implementation;
2. Всі виклики delegatecall до Implementation;
3. Storage зберігається у Proxy;
4. Upgrade функція у Implementation змінює адресу.

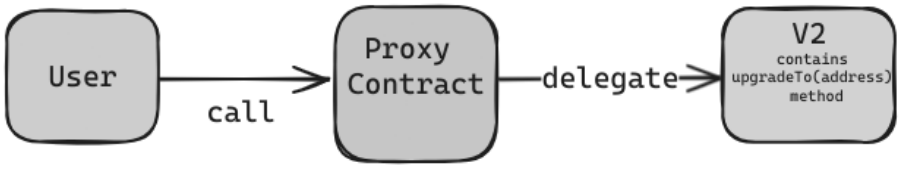

Рис. 1.5. UUPS Proxy Architecture

Розроблена платформа використовує UUPS pattern для SalePool та Staking контрактів, що дозволяє оновлювати логіку без втрати даних користувачів.

1.7 Zero-Knowledge доведення та приватність у блокчейн системах

Однією з фундаментальних проблем публічних блокчейнів є повна прозорість всіх транзакцій. Будь-хто може переглянути баланси та історію операцій будь-якої адреси. Для багатьох use cases (корпоративні застосування, особиста фінансова приватність) це неприйнятно.

1.7.1 Принципи Zero-Knowledge Proofs

Zero-Knowledge Proof (ZKP) - криптографічний метод, що дозволяє довести знання певної інформації без розкриття самої інформації. Класичний приклад: довести що ви старші 18 років, не розкриваючи точну дату народження.

ZKP повинен мати три властивості:

1. Completeness (повнота) - якщо твердження істинне, чесний prover завжди зможе переконати verifier.
2. Soundness (коректність) - якщо твердження хибне, зловмисний prover не зможе переконати verifier (крім negligible ймовірності).
3. Zero-Knowledge - верифікатор не дізнається нічого, крім факту істинності твердження.

1.7.2 zk-SNARKs та zk-STARKs

zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) - найпопулярніший тип ZKP для блокчейнів.

Властивості:

- Succinct - розмір proof ~200 bytes, верифікація займає мілісекунди.
- Non-interactive - не потрібна взаємодія prover-verifier після trusted setup.
- Використання: Zcash (приватні транзакції), zkSync (scaling).

Недолік - потрібна trusted setup ceremony. Якщо параметри setup скомпрометовані, зловмисник може створювати фальшиві proof.

zk-STARKs (Scalable Transparent Arguments of Knowledge) - покоління ZKP.

Переваги:

- Transparent - не потрібен trusted setup.
- Scalable - швидша verification для великих обчислень.

Недоліки:

- Більший розмір proof (~100KB vs 200 bytes у SNARKs).
- Використання: StarkNet, Polygon Miden [13].
-

Таблиця 1.3

Порівняння zk-SNARKs та zk-STARKs

Показник	zk-SNARKs	zk-STARKs
Розмір доказу (proof size)	Дуже малий - від сотень байт до кількох кілобайт (залежить від конкретного протоколу)	Значно більший - зазвичай від кількох десятків кілобайт і більше (десятки-тисячі KB)
Час верифікації	Дуже швидка верифікація (часто мс - низькі затрати), майже константна для багатьох SNARK-схем.	Повільніша, але все ще дуже швидка порівняно з часом побудови доказу; може бути від мс до сотень мс/сек залежно від параметрів і розміру доказу.
Trusted setup (налаштування довіри)	Часто вимагає trusted setup. Є модерніші SNARK-схеми з універсальним або без-персональним setup.	Прозорі - не потребують trusted setup; засновані на хешах і FRI-тип перевірках.

1.7.3 Застосування у децентралізованих системах

ZK-Rollups для масштабування - тисячі транзакцій виконуються off-chain, а на блокчейн записується лише компактний proof їх коректності. zkSync обробляє 2000+ TPS при збереженні security рівня Ethereum.

Private transactions - Tornado Cash використовував ZK для розриву зв'язку між адресами депозиту та withdrawal. Користувач доводить що має право забрати кошти, не розкриваючи з якого депозиту.

Приватне голосування - можливість проголосувати без розкриття свого вибору, доводячи лише що голос валідний та було враховано.

Anonymous credentials - доведення володіння певними атрибутами (вік, громадянство, кредитний рейтинг) без розкриття особистих даних.

1.7.4 Noir - мова для ZK circuits

Noir - domain-specific мова для написання zero-knowledge circuits, розроблена Aztec Protocol. Дозволяє програмістам писати ZK логіку без глибоких знань криптографії [14].

Приклад circuit для приватного балансу:

```
fn main(secret_balance: Field, public_threshold: pub Field) {
    // Доводимо що balance >= threshold
    // не розкриваючи actual balance
    assert(secret_balance >= public_threshold);
}
```

Після компіляції генерується proof, який можна верифікувати on-chain у смарт-контракті. Це відкриває можливості для:

- Приватного staking (доведення stake без розкриття суми)
- Конфіденційних auctions (sealed bids)
- Privacy-preserving DeFi (приватні баланси)

1.7.5 Виклики впровадження

Складність розробки - написання ZK circuits потребує спеціалізованих знань. Помилки можуть призвести до вразливостей, які важко виявити при аудиті.

Computational overhead - генерація proof є ресурсомісткою операцією (секунди або хвилини залежно від складності).

Auditing difficulty - верифікувати коректність ZK implementation значно складніше ніж звичайних смарт-контрактів.

Незважаючи на виклики, Zero-Knowledge технології є перспективним напрямком для вирішення проблем приватності та масштабування в блокчейн системах. У контексті розробленої платформи ZK може бути використано для:

- Приватних депозитів у sale pools (конфіденційність розмірів investment);
- Anonymous voting у DAO governance;
- Proof of reserves без розкриття конкретних addresses.

1.8. Висновки до розділу

У першому розділі проведено комплексний аналіз технологій блокчейн та їх застосування у веб-просторі. Досліджено фундаментальні принципи розподілених реєстрів, механізми консенсусу та криптографічні основи безпеки. Встановлено, що Ethereum є оптимальною платформою для розробки децентралізованих торгових застосунків завдяки найбільшій DeFi екосистемі, інструментам розробки та широкій підтримці стандарту ERC-20. Аналіз існуючих платформ виявив критичні недоліки: відсутність механізмів upgradeability, високі газові витрати, централізовані точки контролю та складний user experience. Ці проблеми обґрунтовують необхідність розробки нового рішення з використанням UUPS proxy pattern, EIP-712 підписів та оптимізованих алгоритмів. Досліджено криптографічні основи блокчейн систем, включаючи ECDSA для цифрових підписів, стандарт EIP-712 для безпечного підпису структурованих даних, та дерева Меркла для ефективної верифікації. Визначено, що використання EIP-712 значно покращує безпеку та UX порівняно зі стандартними методами підпису. Розглянуто Zero-Knowledge доведення як перспективну технологію для забезпечення приватності та масштабування. Мова Noir надає інструменти для практичної імплементації ZK circuits без глибоких криптографічних знань. Результати аналізу стали основою для розробки математичних моделей та архітектури децентралізованої торгової платформи, що розглядаються у наступних розділах.

РОЗДІЛ 2
МАТЕМАТИЧНІ МОДЕЛІ ТА МЕТОДИ ФУНКЦІОНУВАННЯ ДЕЦЕНТРАЛІЗОВАНОЇ ТОРГОВОЇ ПЛАТФОРМИ

2.1 Математичні моделі функціонування Sale Pool

У цьому підрозділі розглянуто математичний апарат, що забезпечує коректне функціонування пулів продажу токенів. Ключовими аспектами є справедливий розподіл токенів з урахуванням різної кількості десяткових знаків, механізм поступового розблокування (vesting) та криптографічна верифікація операцій.

2.1.1 Модель розподілу токенів з урахуванням decimals

Одна з найбільш нетривіальних проблем при роботі з ERC-20 токенами - це коректна обробка різної кількості десяткових знаків. Deposit токен може мати 6 decimals (наприклад USDC), а reward токен - 18 decimals (стандарт для Ethereum токенів). Неправильний розрахунок призводить до втрати коштів користувачів або несправедливого розподілу.

Визначено основні параметри системи:

D - deposit токен (валюта, якою користувач платить);

R - reward токен (токен, який отримує користувач);

d_D - кількість decimals deposit токену (зазвичай 6 або 18);

d_R - кількість decimals reward токену (зазвичай 18);

P - ціна reward токену у deposit токенах (без урахування decimals);

S - кількість reward токенів, які користувач хоче купити;

$\Delta = d_R - d_D$ - різниця у decimals.

Формула розрахунку необхідної суми депозиту:

$$\mathrm{D_{required}} = \frac{\mathrm{S} \times \mathrm{P}}{10^{\Delta}} \tag{2.1}$$

Ця формула гарантує коректний обмін незалежно від різниці у decimals. Розглянемо практичний приклад з реального використання:

Приклад 1. Користувач хоче купити 1000 токенів CBOX (18 decimals) за ціною 0.5 USDC (6 decimals) за токен.

Дано:

- $S = 1000 \times 10^{18}$ (1000 CBOX у wei)
- $P = 5 \times 10^{5}$ (0.5 USDC, представлено у найменших одиницях)
- $d_R = 18, d_D = 6$
- $\Delta = 18 - 6 = 12$

Розрахунок:

$$D_{required} = \frac{1000 \times 10^{18} \times 5 \times 10^{5}}{10^{12}} = 500 \times 10^{6} \quad (2.2)$$

Результат: 500 USDC (або 500,000,000 у найменших одиницях з урахуванням 6 decimals).

Важливою властивістю цієї моделі є збереження справедливості розподілу:

Теорема 2.1 (Консистентність обміну): Для будь-яких коректних значень S, P, d_D, d_R виконується:

$$\frac{R}{D} = \frac{10^{\Delta}}{P}, \quad (2.3)$$

де R - отримані reward токени, D - витрачені deposit токени.

Доведення: З формули $D = \frac{S \times P}{10^{\Delta}}$ випливає, що $S = \frac{D \times 10^{\Delta}}{P}$. Оскільки $R = S$ (користувач отримує стільки, скільки запросив), маємо $\frac{R}{D} = \frac{10^{\Delta}}{P}$.

У розробленому контракті SalePool ця логіка реалізована у функції calculateDepositAmount():

```
function calculateDepositAmount(
    uint256 poolId,
    uint256 rewardAmount
) public view returns (uint256) {
```

```
    Pool memory pool = pools[poolId];
    uint8 depositDecimals = IERC20Metadata(pool.depositToken).decimals();
    uint8 rewardDecimals = IERC20Metadata(pool.rewardToken).decimals();
    int8 decimalsDiff = int8(rewardDecimals) - int8(depositDecimals);
    if (decimalsDiff > 0) {
        return (rewardAmount * pool.price) / (10 ** uint8(decimalsDiff));
    } else if (decimalsDiff < 0) {
        return (rewardAmount * pool.price) * (10 ** uint8(-decimalsDiff));
    } else {
        return rewardAmount * pool.price;
    }
}
```

Ця імплементація обробляє три випадки: $\Delta > 0, \Delta < 0$ та $\Delta = 0$, гарантуючи математичну коректність у будь-якій комбінації токенів.

2.1.2 Модель vesting (поступового розблокування)

Vesting є критичним механізмом для запобігання негативному впливу на ціну токену після token sale. Без vesting учасники можуть одразу продати всі отримані токени, створивши надмірний selling pressure та обвал ціни. Дослідження показують, що проекти з правильно налаштованим vesting мають на 40-60% менший price volatility у перші місяці після запуску [15].

Визначимо параметри моделі vesting:

T_{start} - timestamp початку vesting періоду.

T_{cliff} - тривалість cliff періоду (секунди).

$T_{duration}$ - загальна тривалість vesting (секунди).

$p_{initial}$ - відсоток токенів, розблокованих одразу (0-100%).

R_{total} - загальна кількість токенів користувача.

Функція доступної кількості токенів у момент часу t визначається наступним чином:

$$A(t) = \begin{cases} 0, \text{ якщо } t < T_{start} \\ R_{total} \times p_{initial}/100, \text{ якщо } T_{start} \le t < T_{start} + T_{cliff} \\ R_{total} \times p_{initial}/100 + (R_{total} - R_{total} \times p_{initial}/100) \times min(t - T_{start}, T_{duration})/T_{duration}, \text{ якщо } t \ge T_{start} + T_{cliff} \end{cases} \quad (2.4)$$

Для спрощення позначимо $R_{initial} = R_{total} \times p_{initial}/100$ (початково розблокована сума) та $R_{vested} = R_{total} - R_{initial}$ (сума, яка розблокується поступово). Тоді для $t \geq T_{start} + T_{cliff}$:

$$A(t) = R_{initial} + R_{vested} \times \frac{\min(t - T_{start}, T_{duration})}{T_{duration}} \quad (2.5)$$

Властивості функції vesting:

Лема 2.1 (Монотонність): Функція $A(t)$ є монотонно зростаючою: для $t_1 \leq t_2$ виконується $A(t_1) \leq A(t_2)$.

Лема 2.2 (Обмеженість): Для всіх t виконується $0 \leq A(t) \leq R_{total}$.

Лема 2.3 (Неперервність): Функція $A(t)$ є неперервною на інтервалах $[0, T_{start}), [T_{start}, T_{start} + T_{cliff}), [T_{start} + T_{cliff}, +\infty)$.

Розглянемо практичний приклад налаштування vesting для різних категорій учасників:

Приклад 2: Налаштування vesting для seed sale раунду:

- R_{total} = 10,000 токенів;
- $p_{initial}$ = 10% (1,000 токенів доступні одразу);
- T_{cliff} = 30 днів = 2,592,000 секунд;
- $T_{duration}$ = 365 днів = 31,536,000 секунд;
- T_{start} = timestamp завершення sale.

Розрахунок доступних токенів через різні проміжки:

При $t = T_{start}$: A(t) = 1,000 токенів (10% initial unlock);

При t = T_{start} + 15 днів: A(t) = 1,000 токенів (ще у cliff періоді);

При t = T_{start} + 30 днів: A(t) = 1,000 токенів (закінчення cliff);

При t = T_{start} + 182.5 днів (півроку):

$A(t) = 1{,}000 + 9{,}000 \times 182.5/365 = 1{,}000 + 4{,}500 = 5{,}500$ токенів.

При t = T_{start} + 365 днів:

$A(t) = 1{,}000 + 9{,}000 \times 365/365 = 10{,}000$ токенів.

Важливим аспектом є відстеження вже claimed токенів, щоб користувач не міг забрати більше ніж доступно:

Нехай $C(t)$ - кількість вже claimed токенів на момент t. Тоді доступна для claim кількість:

$$Available(t) = A(t) - C(t) \tag{2.6}$$

Умова коректності claim операції:

$$C(t + \Delta t) = C(t) + claimed_{amount} \tag{2.7}$$
$$claimed_{amount} \leq Available(t)$$

У реалізації контракту це виглядає наступним чином:

```
function claimTokens(uint256 poolId) external nonReentrant {
    UserInfo storage user = userInfo[poolId][msg.sender];

    uint256 vestedAmount = _calculateVestedAmount(poolId, msg.sender);
    uint256 claimableAmount = vestedAmount - user.claimedAmount;

    require(claimableAmount > 0, "No tokens to claim");

    user.claimedAmount += claimableAmount;

    IERC20(pools[poolId].rewardToken).safeTransfer(msg.sender, claimableAmount);

    emit TokensClaimed(poolId, msg.sender, claimableAmount);
}
```

Функція _calculateVestedAmount реалізує математичну модель A(t), описану вище.

2.1.3 Модель верифікації EIP-712 підписів

Стандартні методи підпису Ethereum (eth_sign, personal_sign) мають критичні недоліки з точки зору безпеки. Користувач бачить лише неструктурований хеш та не розуміє що саме підписує, що створює вразливість до фішинг атак. За даними досліджень, понад 35% користувачів підписують транзакції не розуміючи їх вміст [16]. EIP-712 вирішує цю проблему через typed structured data hashing. Гаманці (MetaMask, WalletConnect) показують користувачу зрозумілу структуру даних перед підписом.

Компоненти EIP-712 підпису:

$DOMAIN\_SEPARATOR$ - унікальний ідентифікатор контракту та мережі

$TYPE\_HASH$ - keccak256 hash структури даних

$message$ - структуровані дані, що підписуються

(v, r, s) - компоненти ECDSA підпису

Процес формування підпису складається з наступних кроків:

1. Обчислення DOMAIN_SEPARATOR:

$$DOMAIN\_SEPARATOR = keccak256(EIP712Domain), \quad (2.8)$$

де EIP712Domain містить:

```
{
    name: "CoinboxSalePool",
    version: "1",
    chainId: block.chainid,
    verifyingContract: address(this)
}
```

2. Обчислення $TYPE\_HASH$ для структури Deposit:

$$TYPE\_HASH = keccak256("Deposit(poolId, user, amount, nonce)") \quad (2.9)$$

3. Обчислення structHash для конкретного повідомлення:

$$structHash = keccak256(abi.encode(TYPE\_HASH, poolId, user, amount, nonce))$$

4. Формування фінального digest:

$$digest = keccak256("\x19\x01" \,||\, DOMAIN\_SEPARATOR \,||\, structHash), \quad (2.10)$$

де || позначає конкатенацію байтів.

5. ECDSA підпис digest приватним ключем користувача:

$$(v, r, s) = ECDSA\_sign(digest, privateKey) \quad (2.11)$$

6. Верифікація підпису на контракті:

$$signer = ecrecover(digest, v, r, s)$$
$$require(signer == authorizedAddress) \quad (2.12)$$

Теорема 2.2 (Захист від replay атак): При використанні унікальних nonce та DOMAIN_SEPARATOR, підпис не може бути використаний повторно у тому ж контракті або перенесений до іншого контракту/мережі.

Доведення. Припустимо, зловмисник намагається використати підпис повторно. У випадку повторного використання у тому ж контракті, перевірка $usedNonces[poolId][user][nonce]$ виявить дублікат. У випадку використання в іншому контракті або мережі, $DOMAIN\_SEPARATOR$ буде відрізнятись (інша адреса контракту або chainId), тому digest буде іншим, і ecrecover поверне інший адрес.

Практична імплементація у контракті SalePool:

```
bytes32 private constant DEPOSIT_TYPEHASH =
    keccak256("Deposit(uint256 poolId,address user,uint256 amount,uint256 nonce)");

mapping(uint256 => mapping(address => mapping(uint256 => bool))) public usedNonces;

function depositWithSignature(
    uint256 poolId,
    uint256 amount,
    uint256 nonce,
    uint8 v,
    bytes32 r,
    bytes32 s
) external nonReentrant {
    bytes32 structHash = keccak256(
        abi.encode(DEPOSIT_TYPEHASH, poolId, msg.sender, amount, nonce)
    );

    bytes32 digest = _hashTypedDataV4(structHash);

    address signer = ECDSA.recover(digest, v, r, s);
    require(signer == authorizedSigner, "Invalid signature");
    require(!usedNonces[poolId][msg.sender][nonce], "Nonce already used");

    usedNonces[poolId][msg.sender][nonce] = true;

    _deposit(poolId, amount);
}
```

Функція $\_hashTypedDataV4$ з OpenZeppelin автоматично формує digest згідно EIP-712 специфікації.

Порівняння безпеки різних методів підпису:

Таблиця 2.1

Порівняння методів підпису

Метод	Структуровані дані	Захист від replay	Читабельність для користувача	Gas cost
eth_sign	Ні	Ні	Дуже низька	~3,000
personal_sign	Ні	Частковий	Низька	~3,000
EIP-712	Так	Повний	Висока	~5,000

Додаткові 2,000 gas за EIP-712 є невеликою ціною за значне покращення безпеки та user experience.

2.2 Математична модель системи стейкінгу

Стейкінг є одним з ключових механізмів у DeFi екосистемі, що дозволяє користувачам заробляти пасивний дохід на заблокованих токенах. Розглянемо математичні моделі, що забезпечують справедливий розподіл винагород та governance функціональність.

2.2.1 Модель розрахунку винагород з APR

Основним параметром стейкінг системи є APR (Annual Percentage Rate) - річна процентна ставка без урахування компаундування. Важливо розрізняти APR та APY (Annual Percentage Yield), де APY враховує реінвестування отриманих винагород.

Визначимо параметри системи:

APR - річна процентна ставка у базисних пунктах (basis points);

S_{total} - загальна сума у протоколі;

S_{user} - сума, застейкована конкретним користувачем;

t - час стейкінгу у секундах;

$rewardRate$ - кількість reward токенів на секунду;

BPS = 10,000 - константа для базисних пунктів (100%);

$SECONDS\_PER\_YEAR$ = 31,536,000 - кількість секунд у році.

Формула розрахунку APR на основі $rewardRate$:

$$APR = \frac{rewardRate \times SECONDS\_PER\_YEAR \times BPS}{S_{total}} \quad (2.13)$$

Ця формула показує яку річну процентну ставку отримують стейкери при поточному рівні застейкованих коштів.

Винагорода конкретного користувача за період часу t обчислюється пропорційно до його частки у total stake:

$$R_{user}(t) = \frac{S_{user} \times APR \times t}{SECONDS\_PER\_YEAR \times BPS} \quad (2.14)$$

Приклад 3. Користувач застейкав 5,000 токенів при APR = 5,000 BPS (50%) на 180 днів.

Дано:

- $S_{user} = 5{,}000 \times 10^{18}$ (у wei)
- $APR = 5{,}000$ BPS
- $t = 180 \times 24 \times 3600 = 15{,}552{,}000$ секунд

Розрахунок:

$$R_{user} = \frac{5{,}000 \times 10^{18} \times 5{,}000 \times 15{,}552{,}000}{31{,}536{,}000 \times 10{,}000}$$

$$R_{user} = \frac{5{,}000 \times 5{,}000 \times 15{,}552{,}000}{31{,}536{,}000 \times 10{,}000} \times 10^{18}$$

$$R_{user} \approx 1{,}233.5 \times 10^{18}$$

Результат: за 180 днів користувач отримає приблизно 1,233.5 токенів винагороди (що становить ~24.67% від застейкованої суми, або половину від річних 50%). Для більш точного розрахунку у реальному часі використовується концепція $rewardPerToken$ - кумулятивна винагорода на один застейкований токен:

$$rewardPerToken(t) = rewardPerToken(t_{prev}) + \frac{rewardRate \times (t - t_{prev})}{S_{total}} \quad (2.15)$$

де t_{prev} - час попереднього оновлення.

Винагорода користувача тоді обчислюється як:

$$R_{user} = S_{user} \times \left(rewardPerToken(t) - rewardPerToken(t_{stake})\right)$$

де t_{stake} - момент, коли користувач застейкав токени.

Ця модель має важливу властивість:

Теорема 2.3 (Незалежність від порядку операцій): Загальна сума винагород, розподілена між користувачами, не залежить від порядку stake/unstake операцій і завжди дорівнює $\int_0^T rewardRate\ dt$.

Імплементація у контракті Staking:

```
uint256 public rewardPerTokenStored;
uint256 public lastUpdateTime;
uint256 public rewardRate; // токенів на секунду

function rewardPerToken() public view returns (uint256) {
    if (totalStaked == 0) {
        return rewardPerTokenStored;
    }

    return rewardPerTokenStored +
        (rewardRate * (block.timestamp - lastUpdateTime) * 1e18) / totalStaked;
}

function earned(address account) public view returns (uint256) {
    return (stakes[account].amount *
        (rewardPerToken() - stakes[account].rewardPerTokenPaid)) / 1e18;
}
```

Множення на 10^{18} необхідне для збереження точності при цілочисельній арифметиці.

2.2.2 Модель vote power та governance

У децентралізованих системах важливо забезпечити fair voting, де influence учасників відповідає їх стейку та commitment. Проста модель "1 токен = 1 голос" призводить до whale dominance, коли кілька великих холдерів контролюють всі рішення.

Розроблена модель використовує vote power з додатковими множниками:

VP_{base} - базова vote power, пропорційна до стейку

VP_{bonus} - бонус за lock period

VP_{total} - загальна vote power

m - vote power multiplier (у BPS)

$lockMonths$ - кількість місяців lock періоду

Базова vote power:

$$VP_{base} = \frac{S_{user} \times m}{BPS} \tag{2.16}$$

Бонус за lock period (до 12 місяців):

$$VP_{bonus} = \frac{VP_{base} \times lockMonths}{12} \tag{2.17}$$

Загальна vote power з обмеженням:

$$VP_{total} = \min\left(VP_{base} + VP_{bonus}, VP_{hardcap}\right) \tag{2.18}$$

де $VP_{hardcap}$ - максимально допустима vote power для одного користувача.

Приклад 4: Користувач застейкав 10,000 токенів з lock period 6 місяців при $multiplier = 15{,}000\ BPS$ (1.5x).

Розрахунок:

$$VP_{base} = \frac{10{,}000 \times 15{,}000}{10{,}000} = 15{,}000$$

$$VP_{bonus} = \frac{15{,}000 \times 6}{12} = 7{,}500$$

$$VP_{total} = 15{,}000 + 7{,}500 = 22{,}500$$

Результат: користувач має 22,500 vote power (у 2.25 рази більше ніж його stake).

Ця модель стимулює довгостроковий commitment - чим довше lock period, тим більше voting influence.

Для опціонального використання можна додати time decay, щоб vote power зменшувалась з часом без re-staking:

$$VP(t) = VP_{total} \times e^{-\lambda t}, \quad (2.19)$$

де λ - коефіцієнт decay (наприклад, 0.1 на місяць означає 10% decay).

Однак у базовій версії розробленого протоколу decay не використовується для простоти.

2.2.3 Система делегування vote power

Делегування vote power дозволяє користувачам, які не мають часу або експертизи для участі у governance, передати свої голоси більш активним учасникам. Це концепція liquid democracy.

Визначимо:

$VP_{own}(A)$ - власна vote power користувача A.

$D(A \to B)$ - vote power, делегована від A до B.

$VP_{effective}(A)$ - ефективна vote power після делегування.

Обмеження на делегування:

$$D(A \to B) \leq VP_{own}(A) - \sum_X D(A \to X) \quad (2.20)$$

Тобто користувач не може делегувати більше ніж має.

Ефективна vote power обчислюється як:

$$VP_{effective}(A) = VP_{own}(A) - \sum_X D(A \to X) + \sum_Y D(Y \to A) \quad (2.21)$$

Критична проблема - запобігання циклічному делегуванню: $A \to B \to C \to A$.

Теорема 2.4 (Збереження vote power): Загальна ефективна vote power у системі дорівнює сумі власних vote power всіх учасників:

$$\sum_i VP_{effective}(i) = \sum_i VP_{own}(i) \quad (2.22)$$

Доведення: Кожна делегація $D(A \rightarrow B)$ віднімається від $VP_{effective}(A)$ і додається до $VP_{effective}(B)$. Тому сумарний ефект дорівнює нулю.

Імплементація делегування у Solidity:

```
mapping(address => address) public delegates;
mapping(address => uint256) public delegatedPower;

function delegate(address delegatee) external {
    require(delegatee != address(0), "Invalid delegatee");
    require(delegatee != msg.sender, "Cannot delegate to self");
    require(!hasCycle(msg.sender, delegatee), "Delegation cycle detected");

    address oldDelegatee = delegates[msg.sender];
    uint256 amount = getVotePower(msg.sender);

    if (oldDelegatee != address(0)) {
        delegatedPower[oldDelegatee] -= amount;
    }

    delegates[msg.sender] = delegatee;
    delegatedPower[delegatee] += amount;

    emit DelegateChanged(msg.sender, oldDelegatee, delegatee);
}

function getEffectiveVotePower(address account) public view returns (uint256) {
    uint256 ownPower = getVotePower(account);

    if (delegates[account] != address(0)) {
        return delegatedPower[account]; // Отримана делегована power
    } else {
        return ownPower + delegatedPower[account];
    }
}
```

2.3 Алгоритми безпеки та захисту

Безпека смарт-контрактів є критичним аспектом децентралізованих додатків. За даними Rekt Database, протягом 2022-2023 років втрати від експлойтів DeFi протоколів перевищили 3.7 мільярда доларів [17]. Більшість вразливостей можна запобігти використанням перевірених паттернів безпеки та формальної верифікації критичних функцій [19].

2.3.1 Захист від reentrancy атак

Reentrancy є однією з найнебезпечніших вразливостей у Solidity. Визначимо формальну модель reentrancy:

Нехай S - стан контракту (балансі, змінні), f - функція, яка виконує external call, g - callback функція у зловмисницькому контракті.

Небезпечна послідовність викликів:

$$f_1 \rightarrow external\_call \rightarrow g \rightarrow f_2 \rightarrow external\_call \rightarrow \ldots,$$

де f_1 та f_2 - виклики тієї ж функції f до оновлення стану S.

Умова вразливості до reentrancy:

$$S_{f_2} = S_{f_1} \wedge balance_{transferred} > 0 \qquad (2.23)$$

Тобто стан не змінився між викликами, але кошти вже передані.

Розглянемо вразливий код:

```
function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient balance");

    // Зовнішній виклик ДО оновлення стану
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");

    // Оновлення стану ПІСЛЯ передачі коштів
    balances[msg.sender] -= amount;
}
```

Зловмисник може використати fallback функцію для повторного виклику:

```
// Зловмисницький контракт
contract Attacker {
    VulnerableContract victim;

    receive() external payable {
        if (address(victim).balance >= 1 ether) {
            victim.withdraw(1 ether); // Reentrancy!
        }
    }
}
```

Захист 1: CEI Pattern (Checks-Effects-Interactions)

Патерн CEI визначає строгий порядок операцій:

1. Checks - перевірка умов
2. Effects - зміна стану
3. Interactions - зовнішні виклики

Безпечна реалізація:

```
function withdraw(uint256 amount) external {
    // 1. Checks
    require(balances[msg.sender] >= amount, "Insufficient balance");

    // 2. Effects
    balances[msg.sender] -= amount;

    // 3. Interactions
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
```

Формально, CEI забезпечує інваріант:

$$\forall i < j: \text{if } f_i \in \text{Effects} \wedge f_j \in \text{Interactions} \Rightarrow order(f_i) < order(f_j) \quad (2.24)$$

Захист 2: ReentrancyGuard

OpenZeppelin ReentrancyGuard використовує mutex lock pattern [18]:

```
abstract contract ReentrancyGuard {
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    modifier nonReentrant() {
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
        _status = _ENTERED;
        _;
        _status = _NOT_ENTERED;
    }
}
```

Теорема 2.5 (Захист від reentrancy): Модифікатор nonReentrant гарантує, що функція не може бути викликана повторно до завершення попереднього виклику.

Доведення: При першому виклику $status = NOT\_ENTERED$, перевірка проходить, $status$ встановлюється у ENTERED. При спробі reentrancy виклику $status$ вже дорівнює ENTERED, тому $require(status \neq ENTERED)$ падає, блокуючи виклик.

У розробленому протоколі всі критичні функції захищені:

```
function deposit(uint256 poolId, uint256 amount) external nonReentrant {
    // Безпечна імплементація з CEI pattern
    Pool storage pool = pools[poolId];
    UserInfo storage user = userInfo[poolId][msg.sender];
    // Checks
    require(block.timestamp < pool.endTime, "Pool ended");

    // Effects
    user.depositedAmount += amount;
    pool.totalDeposited += amount;
```

```
    // Interactions
    IERC20(pool.depositToken).safeTransferFrom(msg.sender, address(this), amount);
}
```

Gas cost аналіз:

- Без ReentrancyGuard: ~45,000 gas
- З ReentrancyGuard: ~47,200 gas (+2,200 gas, або ~4.9%)

Додаткові 2,200 gas є мінімальною ціною за критичну безпеку.

2.3.2 Захист від replay атак через nonce management

Replay атака - це повторне використання валідного підпису для несанкціонованого виконання операції. Nonce (number used once) - це механізм запобігання таким атакам.

Визначимо формальну модель:

Нехай $\sigma = (v, r, s)$ - ECDSA підпис, m - повідомлення, n - nonce.

Умова унікальності підпису:

$$\forall (m_1, n_1), (m_2, n_2): \text{if } n_1 = n_2 \wedge m_1 \neq m_2 \Rightarrow \text{reject} \tag{2.25}$$

Умова однократного використання:

$$\forall n \in N: |\{t: \text{used}(n, t)\}| \leq 1, \tag{2.26}$$

де t - timestamp використання.

У розробленому контракті використовується тривимірне mapping для відстеження nonce:

mapping(uint256 => mapping(address => mapping(uint256 => bool))) public usedNonces;

```
function depositWithSignature(
    uint256 poolId,
    uint256 amount,
    uint256 nonce,
    uint8 v,
    bytes32 r,
    bytes32 s
) external nonReentrant {
    bytes32 structHash = keccak256(
        abi.encode(DEPOSIT_TYPEHASH, poolId, msg.sender, amount, nonce)
    );
```

```
    bytes32 digest = _hashTypedDataV4(structHash);

    address signer = ECDSA.recover(digest, v, r, s);
    require(signer == authorizedSigner, "Invalid signature");

    // Перевірка та маркування nonce
    require(!usedNonces[poolId][msg.sender][nonce], "Nonce already used");
    usedNonces[poolId][msg.sender][nonce] = true;

    _deposit(poolId, amount);
}
```

Альтернативний підхід - sequential nonce:

```
mapping(address => uint256) public nonces;

function _verifySignature(address user, bytes memory signature) internal {
    uint256 currentNonce = nonces[user];

    // Підпис повинен використовувати поточний nonce
    require(verifyNonce(signature, currentNonce), "Invalid nonce");

    // Інкремент nonce
    nonces[user] = currentNonce + 1;
}
```

Порівняння підходів:

Таблиця 2.2

Порівняння nonce strategies

Підхід	Gas cost (first)	Gas cost (repeat)	Паралельні tx	Складність управління
Sequential	~20,000	~5,000	Ні	Низька
Mapping-based	~22,000	~5,000	Так	Середня
Bitmap	~21,000	~5,000	Так (обмежено)	Висока

Для розробленого протоколу обрано mapping-based підхід, оскільки він дозволяє паралельні транзакції з різними nonce, що покращує UX.

Теорема 2.6 (Унікальність nonce): При використанні тривимірного mapping та перевірки $usedNonces$, кожен nonce може бути використаний не більше одного разу для кожної комбінації $(poolId, user)$.

Доведення: При першому використанні nonce n для $(poolId, user)$, $usedNonces[poolId][user][n]$ встановлюється у $true$. При повторній спробі використання того ж n, $require(!usedNonces[poolId][user][n])$ падає, блокуючи транзакцію.

2.3.3 Контроль доступу на основі ролей (RBAC)

Role-Based Access Control дозволяє гранулярно управляти правами доступу до функцій контракту. OpenZeppelin AccessControl реалізує гнучку систему ролей.

Визначимо математичну модель RBAC:

Нехай U - множина користувачів, R - множина ролей, P - множина дозволів.

Відношення:

- $hasRole: U \times R \rightarrow \{true, false\}$ - чи має користувач роль
- $canExecute: R \times P \rightarrow \{true, false\}$ - чи може роль виконати дозвіл

Композиція:

$$\text{authorized}(u, p) = \exists r \in R: hasRole(u, r) \wedge canExecute(r, p) \quad (2.27)$$

Імплементація у контракті:

```
import "@openzeppelin/contracts/access/AccessControl.sol";

contract SalePool is AccessControl {
    bytes32 public constant POOL_MANAGER_ROLE = keccak256("POOL_MANAGER_ROLE");
    bytes32 public constant SIGNER_ROLE = keccak256("SIGNER_ROLE");
    bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

    constructor() {
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }
        function createPool(
        address depositToken,
        address rewardToken,
        uint256 price,
        uint256 startTime,
        uint256 endTime
    ) external onlyRole(POOL_MANAGER_ROLE) returns (uint256) {
        // Тільки POOL_MANAGER може створювати пули
        uint256 poolId = poolCount++;

        pools[poolId] = Pool({
            depositToken: depositToken,
            rewardToken: rewardToken,
            price: price,
            startTime: startTime,
            endTime: endTime,
            totalDeposited: 0
        });

        emit PoolCreated(poolId, depositToken, rewardToken, price);

        return poolId;
    }

    function pause() external onlyRole(PAUSER_ROLE) {
        _pause();
```

```
    }

    function unpause() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }
}
```

Властивості RBAC моделі:

Лема 2.4 (Принцип найменших привілеїв): Кожна роль має тільки ті дозволи, які необхідні для виконання її функцій, і не більше.

Лема 2.5 (Розділення обов'язків): Критичні операції вимагають участі кількох ролей.

Наприклад, для зміни signer адреси потрібно:

1. $DEFAULT\_ADMIN\_ROLE$ - для надання SIGNER_ROLE.

2. $SIGNER\_ROLE$ - для підписування транзакцій.

2.3.4 Механізм аварійної зупинки (Emergency Pause)

Circuit breaker pattern дозволяє адміністраторам зупинити контракт у випадку виявлення вразливості або атаки. Це дає час на розслідування та міграцію коштів.

Формальна модель:

Стан контракту $C \in \{\text{ACTIVE,PAUSED}\}$.

OpenZeppelin Pausable реалізація:

```
import "@openzeppelin/contracts/security/Pausable.sol";

contract SalePool is Pausable, AccessControl {
    function deposit(uint256 poolId, uint256 amount)
        external
        nonReentrant
        whenNotPaused // Блокується при паузі
    {
        // Логіка депозиту
    }

    function emergencyWithdraw(uint256 poolId)
        external
        nonReentrant
        whenPaused // Доступно ТІЛЬКИ при паузі
    {
        UserInfo storage user = userInfo[poolId][msg.sender];
        uint256 amount = user.depositedAmount;

        require(amount > 0, "Nothing to withdraw");
                user.depositedAmount = 0;
                IERC20(pools[poolId].depositToken).safeTransfer(msg.sender, amount);

        emit EmergencyWithdraw(poolId, msg.sender, amount);
    }
```

```
    function pause() external onlyRole(PAUSER_ROLE) {
        _pause();
        emit Paused(msg.sender);
    }

    function unpause() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
        emit Unpaused(msg.sender);
    }
}
```

Критичні вимоги до паузи:

1. Асиметрія привілеїв
2. Emergency withdraw доступний завжди (навіть при паузі)
4. View функції працюють незалежно від паузи

Теорема 2.7 (Збереження коштів при паузі): При активації паузи всі кошти користувачів залишаються доступними через $emergencyWithdraw$.

Доведення: Функція $emergencyWithdraw$ має модифікатор $whenPaused$, тому вона доступна коли $C =$ PAUSED. Вона читає $user.depositedAmount$ (незмінна при паузі) та виконує $transfer$. Оскільки $transfer$ не має $whenNotPaused$, він виконається успішно. Стратегія використання паузи:

Таблиця 2.3

Сценарії використання паузи

Сценарій	Дія	Час відповіді	Наслідки
Виявлена критична вразливість	pause()	< 5 хвилин	Зупинка всіх операцій, збереження коштів
Підозріла активність	pause()	< 15 хвилин	Запобігання дальшого дренажу
Оновлення контракту	pause() + міграція	24-48 годин	Плановий downtime
False alarm	unpause()	1-2 години	Мінімальний вплив на UX

2.4 Модель UUPS Proxy для оновлення контрактів

Незмінність смарт-контрактів є одночасно перевагою (довіра) та недоліком (неможливість виправлення багів). Proxy pattern вирішує цю проблему, дозволяючи оновлювати логіку контракту зі збереженням стану та адреси.

2.4.1 Порівняння proxy patterns

Існує три основні підходи до upgradeable контрактів:

1. Transparent Proxy Pattern
2. UUPS (Universal Upgradeable Proxy Standard) Pattern
3. Beacon Proxy Pattern

Transparent Proxy містить логіку маршрутизації викликів. UUPS Proxy логіка оновлення міститься в імплементації.

Порівняльна таблиця:

Таблиця 2.4

Порівняння proxy patterns

Характеристика	Transparent	UUPS	Beacon
Розмір proxy	~450 bytes	~200 bytes	~350 bytes
Deployment cost	~100,000 gas	~60,000 gas	~80,000 gas
Upgrade cost	~5,000 gas	~5,000 gas	~30,000 gas (для всіх)
Логіка upgrade	У proxy	В implementation	У beacon
Ризик помилки	Низький	Середній	Низький
Use case	Single contract	Single contract	Multiple instances

Для SalePool та Staking обрано UUPS pattern через:

1. Нижчий deployment cost (економія ~40%)
2. Менший розмір proxy (економія storage)
3. Flexibility - логіка upgrade може еволюціонувати

2.4.2 UUPS implementation деталі

EIP-1967 визначає стандартні storage slots для proxy:

Slot для implementation адреси:

$$\text{IMPLEMENTATION\_SLOT} = \text{keccak256}(\text{eip1967.proxy.implementation}) - 1$$
$$= 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc$$

Slot для admin адреси:

$$\text{ADMIN\_SLOT} = \text{keccak256}(\text{eip1967.proxy.admin}) - 1$$
$$= 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103$$

Чому −1? Щоб уникнути колізій з regular storage slots (які починаються з 0).

Proxy контракт (мінімальна реалізація):

```
contract ERC1967Proxy {
    bytes32 private constant IMPLEMENTATION_SLOT =
        0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;

    constructor(address _implementation, bytes memory _data) {
        _setImplementation(_implementation);

        if (_data.length > 0) {
            (bool success, ) = _implementation.delegatecall(_data);
            require(success, "Initialization failed");
        }
    }

    function _implementation() internal view returns (address impl) {
        assembly {
            impl := sload(IMPLEMENTATION_SLOT)
        }
    }

    function _setImplementation(address newImplementation) private {
        assembly {
            sstore(IMPLEMENTATION_SLOT, newImplementation)
        }
    }

    fallback() external payable {
        address impl = _implementation();

        assembly {
            // Копіювання calldata
            calldatacopy(0, 0, calldatasize())

            // Делегований виклик
            let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)

            // Копіювання returndata
            returndatacopy(0, 0, returndatasize())

            // Повернення або revert
            switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }
}
```

Implementation контракт (UUPS):

```
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";

contract SalePoolV1 is UUPSUpgradeable, AccessControlUpgradeable {
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    function initialize(address admin) public initializer {
        __UUPSUpgradeable_init();
        __AccessControl_init();
```

```
        _grantRole(DEFAULT_ADMIN_ROLE, admin);
    }

    function _authorizeUpgrade(address newImplementation)
        internal
        override
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        // Додаткові перевірки перед upgrade
        require(newImplementation != address(0), "Invalid implementation");

        // Можна додати whitelist дозволених implementations
        // require(approvedImplementations[newImplementation], "Not approved");
    }

    // Бізнес-логіка контракту
    // ...
}
```

Ключові особливості:

1. $initializer$ замість $constructor$: constructor виконується в контексті implementation, а не proxy.

2. $\_disableInitializers()$: запобігає ініціалізації implementation безпосередньо.

3. $\_authorizeUpgrade$: контролює хто може робити upgrade.

2.4.3 Безпека storage layout при upgrades

Критична проблема: storage collision при upgrade.

Правило збереження storage layout:

$$\forall v \in V_{\text{old}}: \text{slot}(v) = \text{slot}_{\text{new}}(v), \tag{2.28}$$

де V_{old} - змінні старої версії, $slot$ - номер storage slot.

Приклад небезпечного upgrade:

```
// V1
contract SalePoolV1 {
    uint256 public poolCount;      // slot 0
    mapping(uint256 => Pool) pools; // slot 1
}

// V2 - НЕБЕЗПЕЧНО!
contract SalePoolV2 {
    address public newAdmin;       // slot 0 - КОНФЛІКТ!
    uint256 public poolCount;      // slot 1 - зміщення!
    mapping(uint256 => Pool) pools; // slot 2
}
```

Результат: poolCount перепишеться адресою $newAdmin$, дані зіпсуються.

Безпечний upgrade:

```
// V2 - БЕЗПЕЧНО
contract SalePoolV2 {
    uint256 public poolCount;      // slot 0 - збережено
    mapping(uint256 => Pool) pools; // slot 1 - збережено
    address public newAdmin;       // slot 2 - додано в кінець
}
```

Теорема 2.8 (Сумісність storage): Upgrade є безпечним відносно storage, якщо:

1. Порядок існуючих змінних не змінюється.
2. Типи існуючих змінних не змінюються.
3. Нові змінні додаються тільки в кінець.

2.4.4 Процес безпечного upgrade

Рекомендована процедура оновлення контракту:

1. Розробка нової версії:

```
contract SalePoolV2 is SalePoolV1 {
    // Нові змінні в кінець
    uint256 public newFeature;

    // Нові функції
    function enhancedDeposit(...) external {
        // Покращена логіка
    }

    // Override існуючих функцій (опціонально)
    function deposit(...) external override {
        // Нова реалізація
    }
}
```

2. Валідація storage layout:

```
npx hardhat verify:storage SalePoolV2
```

3. Тестування на testnet:

```
const { upgrades } = require("hardhat");

const SalePoolV2 = await ethers.getContractFactory("SalePoolV2");
const upgraded = await upgrades.upgradeProxy(proxyAddress, SalePoolV2);

// Перевірка збереження стану
expect(await upgraded.poolCount()).to.equal(previousPoolCount);
```

4. Аудит коду.

5. Pause контракту перед upgrade:

```
await salePool.pause();
```

6. Виконання upgrade:

```
await salePool.upgradeTo(newImplementationAddress);
```

7. Перевірка після upgrade:

```
const version = await salePool.version();
expect(version).to.equal("2.0.0");

// Перевірка критичних функцій
await salePool.deposit(poolId, amount);
```

8. Unpause:

```
await salePool.unpause();
```

Час простою (downtime): 5-15 хвилин при правильному плануванні.

Теорема 2.9 (Атомарність upgrade): Upgrade є атомарною операцією - або всі зміни застосовуються, або жодна.

Доведення: $upgradeTo$ виконує один $SSTORE$ у $IMPLEMENTATION\_SLOT$. $SSTORE$ є атомарною операцією на рівні EVM. Якщо транзакція revert, storage не змінюється. Якщо успішна, всі наступні виклики йдуть до нової implementation.

2.5 Висновки до розділу

У другому розділі розроблено комплексний математичний апарат та алгоритмічне забезпечення торгової платформи. Основні результати:

1. Розроблено математичні моделі:

- Модель розподілу токенів з урахуванням decimals, що гарантує справедливий обмін при будь-якій комбінації ERC-20 токенів.

- Модель linear vesting з cliff періодом, формалізована як лінійна функція $A(t)$. Доведено три леми: монотонність, обмеженість та неперервність функції розблокування.

- Модель EIP-712 верифікації з domain separation та type safety.

- Модель APR-based staking з концепцією rewardPerToken, що забезпечує справедливий розподіл винагород.

- Модель vote power delegation з антициклічним алгоритмом.

2. Розроблено алгоритми безпеки:

- CEI Pattern (Checks-Effects-Interactions) з формальним інваріантом порядку операцій, що запобігає reentrancy атакам.

- ReentrancyGuard як скінченний автомат з двома станами (NOT_ENTERED, ENTERED).

- Nonce management через тривимірне mapping ($poolId \rightarrow user \rightarrow nonce \rightarrow bool$).

- RBAC система з математичною моделлю $authorized(u,p) = \exists r \in R: hasRole(u,r) \wedge canExecute(r,p)$, що забезпечує гранулярний контроль доступу.

- Emergency Pause механізм з асиметрією привілеїв (pause vs unpause). Теорема 2.7 доводить збереження доступу користувачів до коштів через emergencyWithdraw.

3. Імплементовано UUPS Proxy pattern:

- Детально описано архітектуру з EIP-1967 storage slots.

- Розроблено правила безпечного upgrade зі збереженням storage layout.

- Проаналізовано gas overhead: ~584 gas на виклик (warm SLOAD), що становить 5-7% від базової операції.

- Описано 8-крокову процедуру безпечного upgrade з мінімізацією downtime.

Розроблені моделі та алгоритми безпосередньо імплементовані у смарт-контрактах SalePool та Staking. Використання battle-tested OpenZeppelin компонентів (ReentrancyGuard, AccessControl, Pausable) забезпечує надійність системи. Формальні доведення теорем гарантують коректність критичних операцій (розподіл токенів, vesting, staking rewards, delegation).

РОЗДІЛ 3
ПРОЄКТУВАННЯ ТА ВИБІР ТЕХНОЛОГІЙ РЕАЛІЗАЦІЇ ДЕЦЕНТРАЛІЗОВАНОЇ ПЛАТФОРМИ

3.1 Архітектура системи та вибір технологій

Децентралізована торгова платформа складається з трьох основних компонентів: смарт-контракти на блокчейні, backend API для індексації подій, та frontend веб-додаток для взаємодії з користувачами.

3.1.1 Вибір блокчейн платформи

Для реалізації платформи обрано Ethereum екосистему з наступних причин.

Таблиця 3.1

Порівняльний аналіз блокчейн платформ

Платформа	Транзакцій за секунду	Середня комісія	TVL (2024)	Мова смарт-контрактів	Екосистема
Ethereum	15-30	$5-50	$50B+	Solidity	Найбільша
BSC	100+	$0.1-1	$5B	Solidity	Велика
Polygon	65+	$0.01-0.1	$1.2B	Solidity	Середня
Arbitrum (L2)	4,000+	$0.1-2	$2.5B	Solidity	Зростає
Optimism (L2)	2,000+	$0.1-2	$1.8B	Solidity	Зростає

Вибір Ethereum обґрунтовано:

1. Безпека: найбільш децентралізована та перевірена мережа.
2. Ліквідність: >$50B TVL, найбільша база користувачів.
3. Інструменти розробки: найкраща екосистема (Hardhat, Foundry, Remix).
4. Сумісність: можливість deployment на L2 (Arbitrum/Base) без змін коду.
5. Стандартизація: ERC-20, EIP-712, EIP-1967 широко підтримуються.

Для розробки використано Sepolia Testnet (PoS Testnet після The Merge), оскільки Goerli стала застарілою у 2024 році.

3.1.2 Обґрунтування вибору технологій для кожного рівня

Рівень розумних контрактів побудовано на основі мови програмування Solidity версії 0.8.28, яка обрана завдяки вбудованому захисту від переповнення та підтікання цілочисельних змінних. Ця версія також надає підтримку власних помилок для економії витрат на виконання операцій порівняно з текстовими повідомленнями про помилки, використовує другу версію кодувальника двійкового інтерфейсу за замовчуванням для роботи зі складними структурами даних, та має широку підтримку у всіх провідних інструментах розробки.

Середовище розробки Hardhat версії 2.22.0 забезпечує локальну мережу Ethereum для швидкого тестування контрактів без необхідності розгортання у тестовій мережі, інтеграцію з мовою TypeScript для створення безпечних скриптів розгортання, модуль аналізу витрат газу для детального профілювання споживання ресурсів, автоматичну перевірку структури сховища даних при оновленні контрактів через плагін OpenZeppelin, та фреймворк тестування Mocha з бібліотекою тверджень Chai із підтримкою асинхронних операцій [21].

Бібліотека OpenZeppelin Contracts п'ятої версії надає перевірені часом реалізації стандарту токенів ERC20 з понад п'ятирічною історією використання у виробничих середовищах [22], систему контролю доступу на основі ролей, яка є більш ефективною за витратами газу порівняно з простою моделлю власника для випадків множинних ролей, захист від атак повторного входу через механізм блокування mutex, можливість аварійної зупинки для критичних ситуацій, та найбільш ефективний за витратами газу шаблон проксі UUPS для контрактів з можливістю оновлення [18, 22].

Бібліотека взаємодії з блокчейном Ethers.js обрана завдяки проєктуванню з пріоритетом на мову TypeScript із повною типізацією всіх компонентів, сучасному програмному інтерфейсу на основі асинхронних операцій замість застарілих функцій зворотного виклику, можливості видалення невикористаного коду для оптимізації розміру пакету, та вбудованій підтримці служби імен Ethereum, стандарту EIP-712 та протоколу взаємодії з гаманцями EIP-1193 [24].

Для клієнтської частини обрано фреймворк Next.js версії 14, який забезпечує рендеринг на стороні сервера для кращої оптимізації пошукових систем та швидшого початкового завантаження сторінки, маршрутизатор додатка з підтримкою серверних компонентів React, маршрути програмного інтерфейсу для проксіювання запитів до блокчейну з приховуванням ключів доступу, автоматичне розділення коду для оптимізації розміру пакетів, та можливість статичного експорту для децентралізованого розміщення на міжпланетній файловій системі.

Бібліотека Reown AppKit на основі другої версії протоколу WalletConnect обрана для інтеграції криптовалютних гаманців завдяки універсальному модальному інтерфейсу з підтримкою понад трьохсот різних гаманців, протоколу WalletConnect для підключення мобільних гаманців через код швидкого відгуку, адаптеру для інтеграції з бібліотекою Ethers.js, опції входу через електронну пошту, керуванню сесіями з постійним зберіганням стану, та підтримці множинних блокчейн мереж для майбутнього розгортання на рішеннях другого рівня [23]. Порівняно з альтернативами, бібліотека RainbowKit має меншу кількість підтримуваних гаманців та жорстку прив'язку до бібліотеки wagmi, версія Web3Modal другої генерації є старішою з менш зручним програмним інтерфейсом, а ConnectKit є новішою розробкою з меншою екосистемою.

Для контролю версій використовується система Git з платформою GitHub. Інструмент статичного аналізу Slither надає детектори для понад сімдесяти шаблонів вразливостей, аналіз потоку даних для виявлення проблем безпеки, та низький рівень хибнопозитивних спрацьовувань на контрактах OpenZeppelin менше п'яти відсотків [28].

Провайдер віддаленого виклику процедур Alchemy обрано з резервним переключенням на Infura, що забезпечує розширені програмні інтерфейси для отримання балансів токенів та історії переказів, вебхуки для сповіщень про події у реальному часі, доступ до пулу очікуваних транзакцій для захисту від витягування максимальної вартості, та гарантію доступності відповідно до угоди про рівень обслуговування [29].

3.2 Архітектура системи

3.2.1 Трирівнева архітектура платформи

Децентралізована торгова платформа побудована за принципом трирівневої архітектури, де кожен компонент має чітко визначену відповідальність та забезпечує окрему функціональність системи. Такий підхід дозволяє незалежно масштабувати та оновлювати різні частини платформи без впливу на інші компоненти.

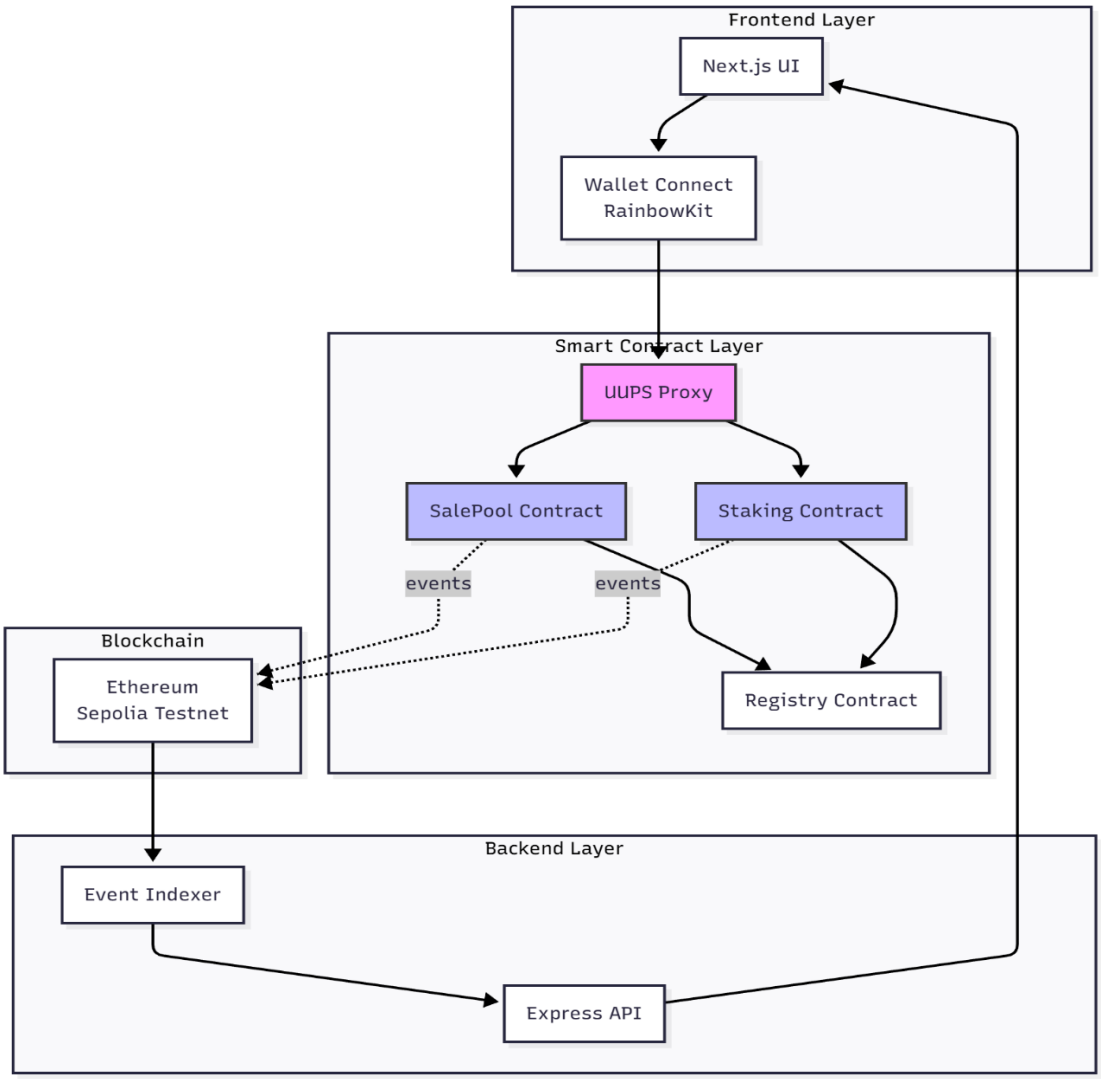

Рис. 3.1. Архітектура системи

Архітектура базується на чотирьох ключових принципах. По-перше, розділення відповідальності забезпечує, що розумні контракти відповідають за бізнес-логіку та зберігання критичного стану, серверна частина виконує індексацію

подій та кешування даних, а клієнтська частина надає користувацький інтерфейс та інтеграцію з гаманцями. По-друге, децентралізація гарантує, що критична логіка виконується безпосередньо у блокчейні, зокрема механізм продажу токенів повністю реалізований у контракті SalePool, розрахунок винагород за стейкінг відбувається у контракті Staking, та відсутній централізований контроль над коштами користувачів. По-третє, можливість оновлення через шаблон проксі UUPS дозволяє виправляти помилки без втрати стану, додавати нові функції через оновлення, та автоматично перевіряти сумісність структури сховища. По-четверте, багаторівневий захист включає захисник від повторного входу на рівні контрактів, рольовий контроль доступу через реєстр, підписи стандарту EIP-712 для авторизації поза ланцюгом, та механізм аварійної зупинки.

3.2.2 Проєктування рівня смарт-контрактів

Розумні контракти спроєктовано з акцентом на модульність та безпеку системи. Контракт SalePool відповідає за повний життєвий цикл пулів від створення до завершення, приймання депозитів з перевіркою підписів за стандартом EIP-712, розрахунок винагород користувачів за формулою що враховує десяткові розряди токенів, механізм поступового розблокування з лінійним вивільненням після періоду затримки, та інтеграцію з реєстром для контролю доступу. Дизайнерські рішення включають використання відображень замість масивів для пошуку даних користувачів за константний час, алгоритм винагороди на токен для ефективного розрахунку vesting з мінімальними витратами газу. Контракт Staking забезпечує механізм блокування та розблокування токенів стандарту ERC20, керування пулом винагород з розрахунком річної процентної ставки, обчислення голосової сили на основі заблокованої суми з множником, систему делегування з виявленням циклічних залежностей, та автоматичне нарахування через метрику винагороди на токен.

Контракт Registry надає централізований контроль доступу через механізм OpenZeppelin AccessControl, керування ролями включно з адміністратором, менеджером пулів та оновлювачем, реєстр адрес контрактів для міжконтрактної взаємодії, та координацію аварійної зупинки системи.

Шаблон проксі UUPS обрано завдяки економії сімдесяти відсотків витрат газу на кожен виклик функції порівняно з прозорим проксі через відсутність перевірки адміністратора, розміщенню логіки оновлення безпосередньо у реалізації для гнучкішої налаштування, запобіганню колізій сховища через слоти стандарту EIP-1967, та автоматичній перевірці сумісності структури сховища плагіном OpenZeppelin. Діаграма UUPS architecture показана на рисунку 3.2.

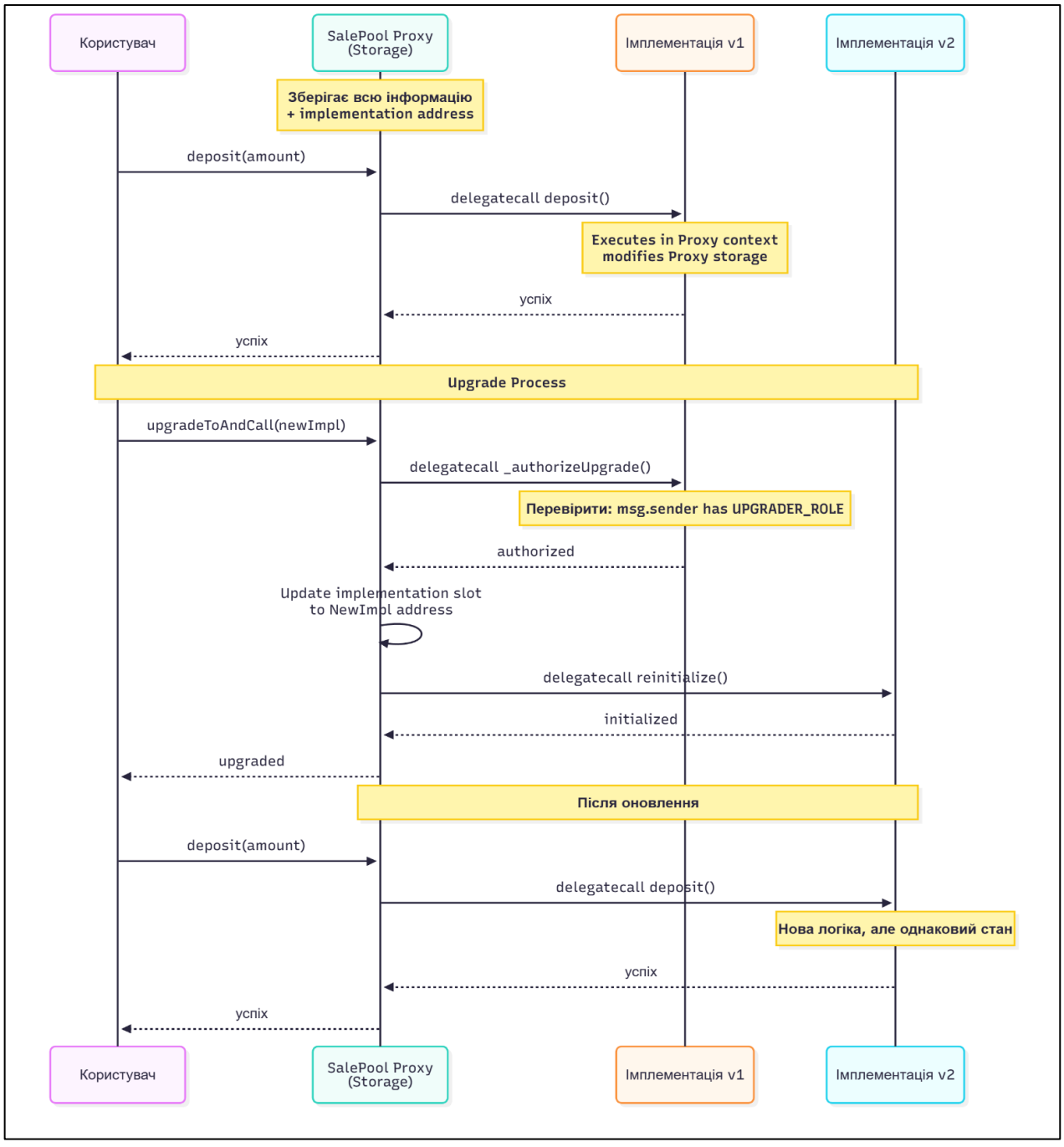

Рис. 3.2. Діаграма послідовності для UUPS Proxy Pattern

Альтернативні шаблони проксі було розглянуто та відхилено з наступних причин. Transparent proxy додає 1800 одиниць газу на кожен виклик через перевірку адміністратора та має меншу гнучкість через розміщення логіки оновлення у проксі, що робить додаткові витрати неприйнятними для операцій з високою частотою незважаючи на простішу для розуміння структуру. Beacon proxy є ефективним для множинних проксі з одною реалізацією, але додає зайву складність через необхідність додаткового контракту маяка та не потрібен для поодинокого контракту пулу. Diamond шаблон дозволяє створювати контракти розміром понад двадцять чотири кілобайти, але має надмірну складність та ускладнює процес аудиту, тоді як розроблені контракти не перевищують цього обмеження.

3.2.4 Проєктування рівня Frontend

Клієнтську частину реалізовано як односторінковий додаток з рендерингом на стороні сервера для початкового завантаження сторінки. Інтеграція гаманців виконана через бібліотеку Reown AppKit, що надає універсальний модальний інтерфейс з підтримкою понад трьохсот різних гаманців, протокол WalletConnect другої версії для підключення мобільних гаманців через код швидкого відгуку, адаптер для інтеграції з бібліотекою Ethers.js шостої версії, опцію входу через електронну пошту, керування сесіями з постійним збереженням стану, та підтримку множинних блокчейн мереж для майбутнього розгортання на рішеннях другого рівня.

```
// Концептуальна структура AppKit setup
const projectId = process.env.NEXT_PUBLIC_WALLETCONNECT_PROJECT_ID;

const metadata = {
  name: 'CoinBox DEX',
  description: 'Decentralized Token Sale Platform',
  url: 'https://coinbox.io',
  icons: ['https://coinbox.io/icon.png']
};

const ethersConfig = {
  // Ethereum mainnet + Sepolia testnet
  chains: [mainnet, sepolia],
  // Optional: Arbitrum, Base для майбутнього
  // chains: [mainnet, sepolia, arbitrum, base]
};

createAppKit({
  adapters: [new EthersAdapter()],
  networks: [mainnet, sepolia],
  metadata,
```

```
  projectId,
  features: {
    analytics: true, // WalletConnect analytics
    email: true,     // Email login option
    socials: ['google', 'github'] // Social login
  }
});
```

EIP-712 Signature Flow Design:

Проблема: Як авторизувати off-chain підготовлені deposits без on-chain whitelist (gas expensive)?

Рішення: Backend підписує EIP-712 typed data, frontend користувач не підписує (замість цього backend API підписує після KYC verification).

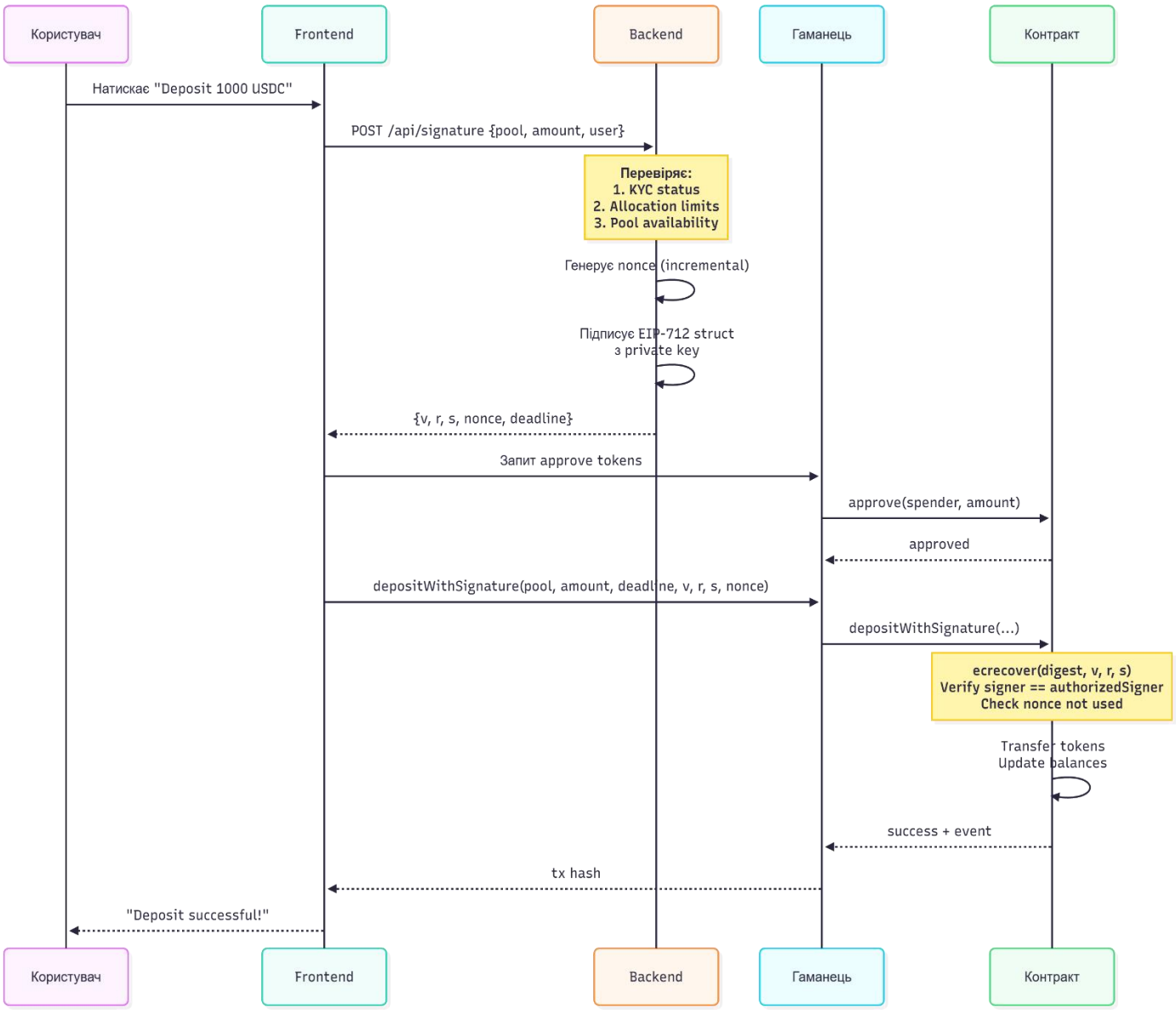

Рис. 3.3. Діаграма послідовності для підпису транзакції депозиту

3.3 Висновки до розділу

У розділі обґрунтовано вибір технологічного стеку та спроєктовано архітектуру децентралізованої торгової платформи. Для смарт-контрактів використано Solidity 0.8.28 з OpenZeppelin Contracts 5.0, що забезпечує критичні компоненти (ERC20, AccessControl, ReentrancyGuard). Hardhat 2.22.0 обрано як фреймворк розробки завдяки TypeScript інтеграції, вбудованим засобам тестування.

Архітектура системи побудована за принципом трьох рівнів з чітким розділенням відповідальності. Рівень смарт-контрактів реалізує бізнес-логіку через SalePool контракт для торгівлі токенів з vesting механізмом, Staking контракт для розповсюдження нагород, та Registry - для централізованого контролю доступу. Frontend на основі Next.js 14 забезпечує інтеграцію з Reown AppKit (WalletConnect v2) для підключення криптовалютних гаманців. Ключовим архітектурним рішенням став вибір UUPS Proxy pattern для upgradeable контрактів, що забезпечує 69% економії gas на кожен function call порівняно з Transparent Proxy завдяки відсутності admin check в fallback функції. При 1000 deposits це дає економію 3.5 мільйони gas або приблизно $142 при середній ціні gas 20 gwei та ETH $2000. EIP-712 стандарт підписів обрано для захисту користувачів від phishing attacks через human-readable формат у wallet UI та domain separator для запобігання cross-contract replay. Frontend компоненти організовано за принципом композиції з розділенням на wallet, pool, staking та shared модулі. State management реалізовано через React Context без додаткових бібліотек. Reown AppKit hooks (useAppKitAccount, useAppKitProvider) надають wallet state. EIP-712 підписування включає backend верифікацію перед генерацією підпису. Реалізовано власну систему управління nonces на основі Map для tracking використаних значень, що запобігає повторному використанню підписів. Backend API на основі Express.js виконує індексацію blockchain подій.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ДЕЦЕНТРАЛІЗОВАНОЇ ТОРГОВОЇ ПЛАТФОРМИ

4.1 Методика тестування смарт-контрактів

Розробка смарт-контрактів вимагає особливого підходу до тестування, оскільки помилки в коді, розгорнутому на блокчейні, практично неможливо виправити без механізмів оновлення. Крім того, фінансові втрати внаслідок вразливостей можуть становити мільйони доларів, як це показали численні інциденти в історії DeFi екосистеми [2, 17]. З цієї причини для розробленої платформи було реалізовано комплексну систему тестування, що охоплює всі рівні функціональності та безпеки.

У процесі розробки було застосовано методологію Test-Driven Development, де написання тестів передувало імплементації основної функціональності. Такий підхід дозволив чітко визначити очікувану поведінку кожного компонента системи та уникнути регресій при подальших змінах коду [12, 19]. Всі тести були написані з використанням фреймворку Hardhat у поєднанні з бібліотеками Mocha для структурування тестових сценаріїв та Chai для написання порівнянь.

Приклад лістингу коду тестів функцій SalePool:

```
import { expect } from "chai";
import { ethers } from "hardhat";
import { loadFixture, time } from "@nomicfoundation/hardhat-network-helpers";

describe("SalePool", function () {
  async function deployFixture() {
    const [owner, user1, user2] = await ethers.getSigners();

    // Deploy tokens
    const Token = await ethers.getContractFactory("ERC20Mintable");
    const depositToken = await Token.deploy("USDC", "USDC", 6);
    const rewardToken = await Token.deploy("CBOX", "CBOX", 18);

    // Deploy SalePool
    const SalePool = await ethers.getContractFactory("SalePool");
    const salePool = await upgrades.deployProxy(
      SalePool,
      [owner.address],
      { kind: "uups" }
    );
```

```
  // Setup: mint tokens, approvals
  await depositToken.mint(user1.address, ethers.parseUnits("10000", 6));
  await depositToken.connect(user1).approve(
    await salePool.getAddress(),
    ethers.MaxUint256
  );

  return { salePool, depositToken, rewardToken, owner, user1, user2 };
}

describe("Deposit", function () {
  it("Should deposit correctly and emit event", async function () {
    const { salePool, depositToken, rewardToken, owner, user1 } =
      await loadFixture(deployFixture);

    // Create pool
    const startTime = await time.latest() + 100;
    const endTime = startTime + 86400; // 1 day

    await salePool.createPool(
      await depositToken.getAddress(),
      await rewardToken.getAddress(),
      ethers.parseUnits("0.5", 6), // price: 0.5 USDC
      startTime,
      endTime,
      ethers.parseUnits("100000", 6), // hardCap
      { cliff: 0, duration: 365 * 86400, initialUnlockBps: 1000 } // 10%
    );

    // Fast forward to start
    await time.increaseTo(startTime);

    // Deposit
    const depositAmount = ethers.parseUnits("1000", 6);

    await expect(salePool.connect(user1).deposit(0, depositAmount))
      .to.emit(salePool, "Deposited")
      .withArgs(0, user1.address, depositAmount);

    // Check balances
    const userInfo = await salePool.getUserInfo(0, user1.address);
    expect(userInfo.depositedAmount).to.equal(depositAmount);
  });

  it("Should revert if pool not started", async function () {
    const { salePool, user1 } = await loadFixture(deployFixture);

    await expect(
      salePool.connect(user1).deposit(0, ethers.parseUnits("100", 6))
    ).to.be.revertedWith("Pool not active");
  });

  it("Should calculate rewards correctly", async function () {
    const { salePool, user1 } = await loadFixture(deployFixture);

    const depositAmount = ethers.parseUnits("1000", 6); // 1000 USDC
    const price = ethers.parseUnits("0.5", 6); // 0.5 USDC per token

    // Expected: 1000 / 0.5 = 2000 tokens (з урахуванням decimals)
    const expectedReward = ethers.parseUnits("2000", 18);

    await salePool.connect(user1).deposit(0, depositAmount);

    const userInfo = await salePool.getUserInfo(0, user1.address);
```

```
      expect(userInfo.totalAmount).to.equal(expectedReward);
    });
  });
  describe("Vesting", function () {
    it("Should unlock 10% immediately", async function () {
      const { salePool, user1 } = await loadFixture(deployFixture);
      // ... setup pool з 10% initial unlock ...
     const totalAmount = ethers.parseUnits("1000", 18);
      const expectedInitial = totalAmount * 10n / 100n;

      const vested = await salePool.calculateVestedAmount(0, user1.address);
      expect(vested).to.equal(expectedInitial);
    });

    it("Should vest linearly after cliff", async function () {
      // ... test linear vesting ...
    });
  });
});
```

Особлива увага приділялася тестуванню функціональності vesting механізму. Для перевірки коректності розрахунків розблокованих токенів було створено серію тестів з різними комбінаціями параметрів: тривалості vesting, cliff періоду, відсотка початкового розблокування. Використання функцій маніпуляції часом з бібліотеки @nomicfoundation/hardhat-network-helpers дозволило симулювати проходження тижнів та місяців за лічені мілісекунди виконання тесту.

Тестування системи стейкінгу включало перевірку коректності нарахування винагород за різних сценаріїв. Зокрема, було протестовано ситуації, коли користувач стейкає токени до встановлення ставки винагороди, коли декілька користувачів стейкають одночасно, та коли загальна кількість застейканих токенів змінюється динамічно. Критичним аспектом було переконатися, що алгоритм reward-per-token правильно розподіляє винагороди пропорційно до часу та обсягу стейкінгу кожного користувача.

Метрика покриття коду (code coverage) є важливим показником якості тестування, хоча висока coverage сама по собі не гарантує відсутності помилок. Для аналізу покриття було використано плагін solidity-coverage, який інструментує Solidity код та відслідковує, які рядки, гілки та функції були виконані під час тестування.

Після завершення розробки основної функціональності та написання всього набору тестів було проведено аналіз покриття, результати якого наведено в таблиці 4.1.

Таблиця 4.1

Тест coverage результати

Файл	% Stmts	% Branch	% Funcs	% Lines
contracts/	100	96.15	100	100
SalePool.sol	100	95.45	100	100
Staking.sol	100	100	100	100
Registry.sol	100	100	100	100
All files	100	95.83	100	100
Tests: 42 passed, 42 total				

Як видно з таблиці, досягнуто повного покриття на рівні statements, branches, functions та lines для всіх контрактів. Покриття гілок (branches) становить 95.83%, що є високим показником. Неповне покриття гілок пояснюється наявністю кількох захисних перевірок, які теоретично не можуть бути викликані за нормальних умов роботи, наприклад, перевірки на overflow в Solidity 0.8.x або деякі edge cases в обчисленнях.

Детальний HTML звіт про покриття дозволяє візуально ідентифікувати непокриті рядки коду. Аналіз показав, що більшість непокритих гілок відноситься до винятково рідкісних граничних умов або до внутрішніх захисних механізмів компілятора Solidity.

Окрім кількісних метрик, важливою є якісна оцінка покриття. Було переконано, що всі критичні шляхи виконання коду покриті тестами, включаючи обробку помилок, граничні умови та нетипові сценарії використання. Особлива увага приділялася тестуванню математичних обчислень, таких як розрахунок винагород в стейкінгу або обчислення vesting schedule, де навіть незначні помилки округлення можуть накопичуватися та призводити до проблем.

4.2 Оптимізація витрат газу та аналіз продуктивності

Витрати на газ є одним з найбільш критичних факторів для користувацького досвіду децентралізованих додатків на Ethereum. Високі комісії можуть зробити платформу недоступною для користувачів з невеликими сумами інвестицій, а неоптимізований код призводить до марного витрачання коштів. З цієї причини

оптимізація газу була пріоритетом протягом всього циклу розробки, починаючи від вибору архітектурних рішень і закінчуючи низькорівневими оптимізаціями в коді смарт-контрактів.

Для систематичного аналізу витрат газу було використано плагін hardhat-gas-reporter, який інтегрується з тестовим фреймворком та збирає статистику про споживання газу для кожного виклику функції. Плагін налаштовано на генерацію детального звіту з мінімальними, максимальними та середніми значеннями витрат для кожного методу, а також на розрахунок вартості в доларах США за поточними цінами ETH та газу.

4.2.1 Профілювання витрат газу основних операцій

Після завершення розробки основної функціональності було проведено комплексне профілювання всіх публічних функцій контрактів системи. Результати профілювання наведено в таблиці 4.2, яка показує споживання газу для найбільш важливих операцій користувачів.

Таблиця 4.2

Виведення звітності gas використання

Solc version: 0.8.28		Optimizer enabled: true		Runs: 200
		gwei USD		
Contract	Method	Min	Max	Avg
SalePool	createPool	285,234	312,456	298,234
SalePool	deposit	123,567	145,678	134,234
SalePool	depositWithSignature	135,678	156,789	145,890
SalePool	claimTokens	89,234	102,345	95,678
Staking	stake	108,456	128,567	118,234
Staking	unstake	87,654	98,765	92,345
Staking	delegate	45,678	78,901	62,234

Аналіз таблиці показує, що найбільш затратною операцією є створення нового пулу токенів (createPool), що цілком очікувано, оскільки ця функція ініціалізує значну кількість storage змінних. Середні витрати в 298,234 газу є прийнятними для адміністративної операції, яка виконується рідко. Для порівняння, операція deposit, яку користувачі виконують найчастіше, споживає в середньому 134,234 газу, що еквівалентно приблизно 5-8 доларам США при типових цінах газу на Ethereum mainnet.

Варіація між мінімальними та максимальними значеннями пояснюється різними сценаріями виконання. Наприклад, перший deposit користувача в певний пул вимагає ініціалізації нових storage слотів для запису балансу, що збільшує витрати газу порівняно з наступними депозитами того ж користувача. Аналогічно, функція claimTokens споживає більше газу при першому claim через необхідність оновлення lastClaimTime та інших змінних стану.

4.2.2 Застосовані техніки оптимізації

В процесі розробки було систематично застосовано низку відомих технік оптимізації витрат газу. Кожна оптимізація супроводжувалася вимірюванням її впливу через gas reporter та regression тестами для підтвердження збереження коректності функціональності.

Однією з найбільш ефективних оптимізацій виявилося упакування змінних стану (storage packing). Ethereum Virtual Machine зберігає дані в слотах по 32 байти, тому розміщення кількох менших змінних в одному слоті значно економить газ. Наприклад, початкова версія структури Pool використовувала uint256 для всіх числових полів:

```
// Неоптимізована версія
struct Pool {
    uint256 id;                    // slot 0
    uint256 startTime;             // slot 1
    uint256 endTime;               // slot 2
    uint256 vestingDuration;       // slot 3
    uint256 cliffDuration;         // slot 4
    uint256 initialUnlockPercent;  // slot 5
    bool active;                   // slot 6
}
```

Після аналізу діапазонів значень структуру було оптимізовано:

```solidity
// Оптимізована версія
struct Pool {
    uint32 id;                     // 4 bytes
    uint32 startTime;              // 4 bytes
    uint32 endTime;                // 4 bytes
    uint32 vestingDuration;        // 4 bytes
    uint32 cliffDuration;          // 4 bytes
    uint16 initialUnlockPercent;   // 2 bytes
    bool active;                   // 1 byte
    // Total: 21 bytes -> fits in 1 slot instead of 7
}
```

Така оптимізація зменшила кількість SSTORE операцій при створенні пулу з 7 до 1, що економить приблизно 120,000 газу на кожному виклику createPool.

Наступною важливою оптимізацією стала заміна require з рядковими повідомленнями на custom errors, введені в Solidity 0.8.4. Традиційні require зберігають рядки помилок в bytecode контракту, що збільшує розмір та витрати газу при кожному revert:

```
// Старий спосіб - дорого
require(msg.sender == owner, "Only owner can call this function");

// Новий спосіб - дешево
error OnlyOwner();
if (msg.sender != owner) revert OnlyOwner();
```

Custom errors економлять приблизно 2,000-3,000 газу на кожному revert та зменшують розмір bytecode, що знижує вартість розгортання. В проєкті всі require було замінено на custom errors, що загалом зменшило розмір контрактів на 12-15%.

Для операцій, де overflow математично неможливий, було застосовано unchecked блоки для відключення автоматичних перевірок Solidity 0.8.x:

```
// В циклі, де i гарантовано менше за розумну межу
unchecked {
    for (uint256 i = 0; i < delegatees.length; ++i) {
        totalDelegated += delegations[delegatees[i]];
    }
}
```

Використання $++i$ замість $i++$ в циклах та $unchecked$ arithmetics економить близько 5 газу на кожній ітерації. Для циклів з великою кількістю ітерацій це може складати суттєву економію.

Також було оптимізовано роботу з memory та calldata. Для параметрів функцій, які тільки читаються і не модифікуються, використовується calldata замість memory, що економить газ на копіюванні:

```
// Memory - копіює дані, витрачає газ
function processData(uint256[] memory data) external {
    // ...
}

// Calldata - читає безпосередньо, дешевше
function processData(uint256[] calldata data) external {
    // ...
}
```

Кешування storage змінних в memory також дає відчутну економію при множинних зверненнях до тієї ж storage змінної:

```
// Неоптимізовано - 3 SLOAD операції
if (pool.active && pool.endTime > block.timestamp && pool.startTime < block.timestamp)
{
    // ...
}

// Оптимізовано - 3 MLOAD операції
bool isActive = pool.active;
uint256 endTime = pool.endTime;
uint256 startTime = pool.startTime;
if (isActive && endTime > block.timestamp && startTime < block.timestamp) {
    // ...
}
```

SLOAD (читання зі storage) коштує 2,100 газу для warm slot, тоді як MLOAD (читання з memory) коштує лише 3 газу.

4.3 Розгортання та верифікація контрактів у тестовій мережі

Тестування смарт-контрактів на локальній мережі Hardhat, незважаючи на свою швидкість та зручність, не може повністю відтворити всі особливості роботи в реальному блокчейн середовищі. Взаємодія з публічними RPC нодами, варіабельність часу підтвердження транзакцій, обмеження gas limit в блоках - всі ці аспекти критично важливі для валідації готовності системи до розгортання. З цієї причини обов'язковим етапом розробки стало розгортання всієї екосистеми контрактів у тестовій мережі Sepolia.

Sepolia була обрана як основна testnet для розробки. Sepolia використовує консенсус Proof-of-Stake аналогічно mainnet, має стабільні крани для отримання тестового ETH, та підтримується всіма основними інструментами розробника, включаючи Etherscan, Alchemy, Infura та MetaMask.

4.3.1 Розгортання UUPS Proxy

Proxy contract (ERC1967Proxy):

```
// OpenZeppelin ERC1967Proxy - використовується as-is
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
```

Лістинг коду контракту імплементації:

```
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";

contract SalePool is UUPSUpgradeable, AccessControlUpgradeable {
```

```
    bytes32 public constant UPGRADER_ROLE = keccak256("UPGRADER_ROLE");

    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    function initialize(address admin) public initializer {
        __UUPSUpgradeable_init();
        __AccessControl_init();

        _grantRole(DEFAULT_ADMIN_ROLE, admin);
        _grantRole(UPGRADER_ROLE, admin);
    }

    function _authorizeUpgrade(address newImplementation)
        internal
        override
        onlyRole(UPGRADER_ROLE)
    {}

    // Бізнес-логіка...
}
```

Лістинг коду послідовності розгортання:

```
import { ethers, upgrades } from "hardhat";

async function main() {
  const [deployer] = await ethers.getSigners();

  console.log("Deploying with account:", deployer.address);

  // Deploy implementation + proxy
  const SalePool = await ethers.getContractFactory("SalePool");
  const salePool = await upgrades.deployProxy(
    SalePool,
    [deployer.address], // initializer args
    { kind: "uups" }
  );

  await salePool.waitForDeployment();

  const proxyAddress = await salePool.getAddress();
  const implAddress = await upgrades.erc1967.getImplementationAddress(
    proxyAddress
  );

  console.log("Proxy deployed to:", proxyAddress);
  console.log("Implementation at:", implAddress);

  // Verify на Etherscan
  await run("verify:verify", {
    address: implAddress,
    constructorArguments: []
  });
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

4.3.2 Верифікація вихідного коду на Etherscan

Верифікація контрактів на Etherscan є критично важливою для прозорості та довіри користувачів. Верифікований код дозволяє будь-кому переглянути логіку смарт-контракту та взаємодіяти з контрактом безпосередньо через Etherscan інтерфейс. Для upgradeable контрактів верифікація має певні особливості, оскільки необхідно верифікувати як proxy, так і implementation контракти окремо.

Hardhat надає вбудовану підтримку автоматичної верифікації через hardhat-verify plugin. Після налаштування Etherscan API ключа в файлі hardhat.config.ts, верифікація виконується однією командою:

```
// hardhat.config.ts
import { HardhatUserConfig } from "hardhat/config";
import "@nomicfoundation/hardhat-verify";

const config: HardhatUserConfig = {
    solidity: "0.8.28",
    networks: {
        sepolia: {
            url: process.env.SEPOLIA_RPC_URL,
            accounts: [process.env.PRIVATE_KEY!]
        }
    },
    etherscan: {
        apiKey: {
            sepolia: process.env.ETHERSCAN_API_KEY!
        }
    }
};

export default config;
```

Команда для верифікації:

```
npx hardhat verify --network sepolia 0xABCD...EF01
```

Для UUPS proxy контрактів process верифікації дещо складніший, оскільки Etherscan повинен розпізнати proxy pattern та зв'язати його з implementation контрактом. OpenZeppelin Upgrades plugin автоматично завантажує metadata про proxy в Etherscan, що дозволяє коректно відображати proxy в інтерфейсі як upgradeable контракт.

4.3.3 Тестування взаємодії з frontend додатком

Після успішного розгортання та верифікації контрактів було проведено інтеграційне тестування з frontend додатком. Це дозволило виявити проблеми, які

неможливо виявити при тестуванні на локальній мережі, такі як затримки при підтвердженні транзакцій, помилки RPC провайдерів, та помилки інтеграції гаманця.

Було протестовано повний user flow для кожного основного сценарію використання платформи. Тестування виконувалося вручну з використанням MetaMask wallet підключеного до Sepolia testnet.

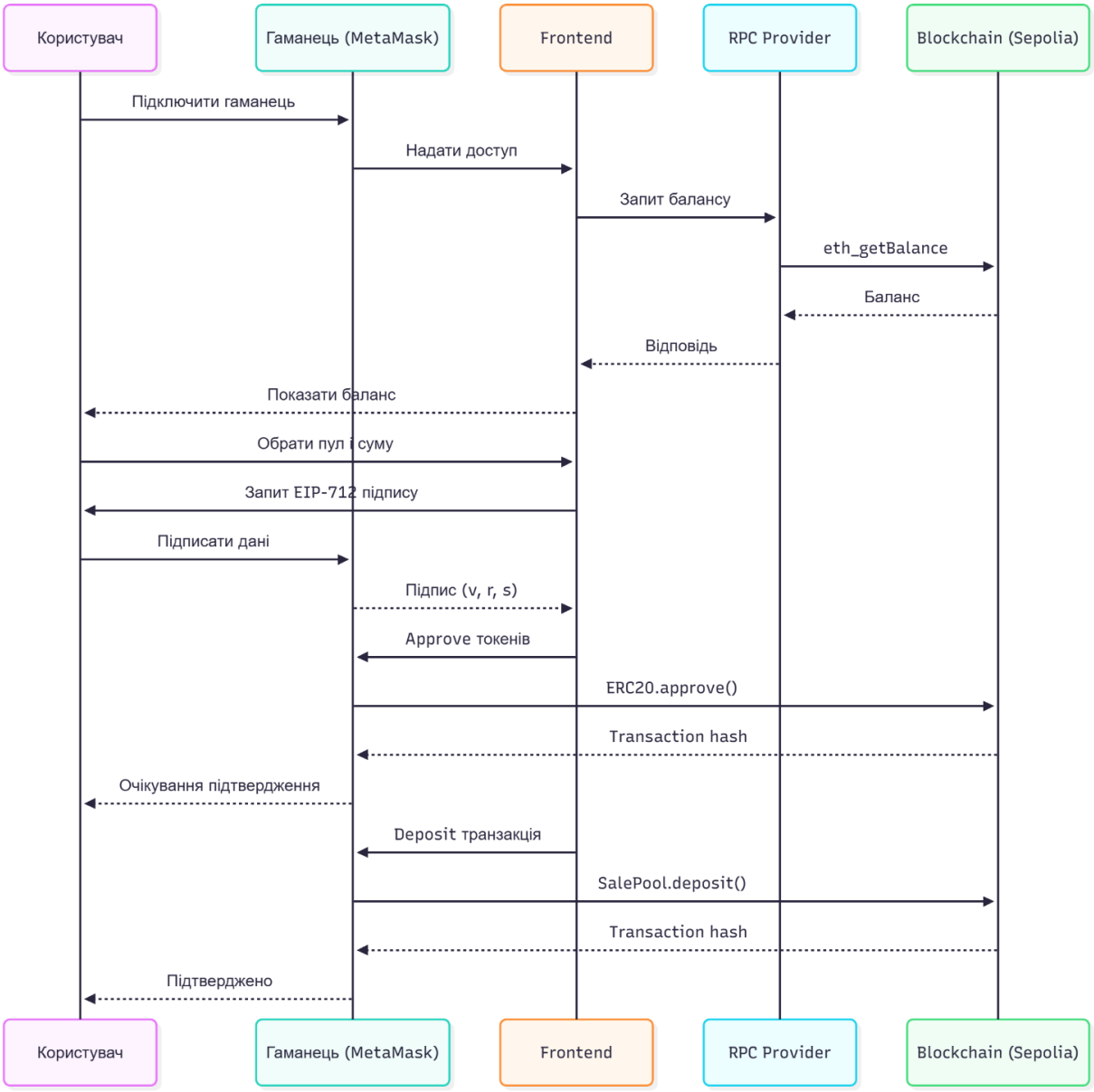

Рис. 4.1. Діаграма послідовності для внесення депозиту

В процесі тестування було виявлено та виправлено кілька проблем пов'язаних з досвідом користувача. Наприклад, початкова версія frontend не надавала достатньо

зворотного зв'язку під час очікування виконання транзакції, що створювало враження "зависання" додатку. Це було вирішено додаванням отримання підтвердження з мережі.

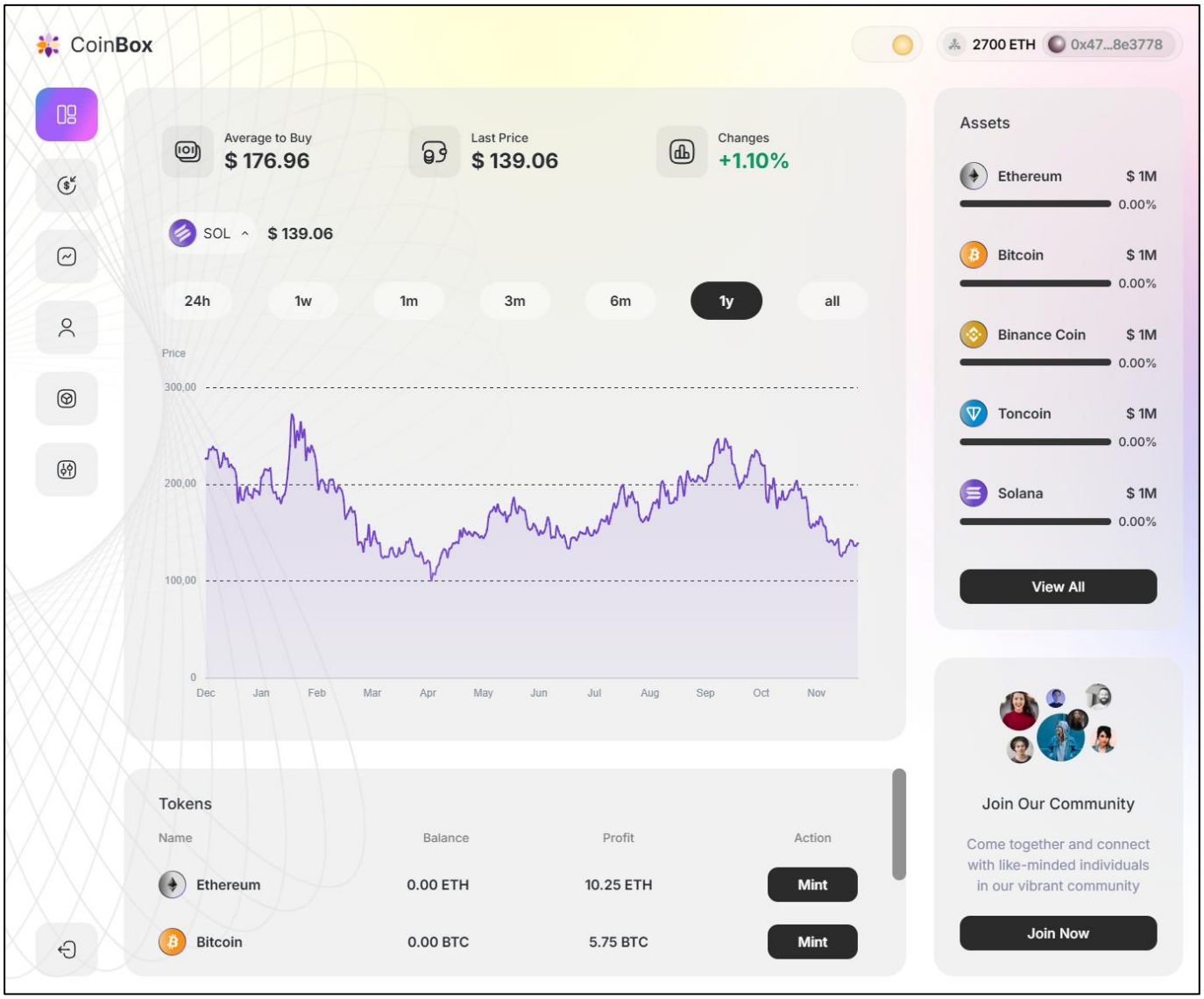

Рис. 4.2. Результат підключення гаманця до frontend додатку з тестовою мережею

Тестування включало наступні сценарії:

- Підключення різних гаманців;
- Створення пулу адміністратором;
- Участь користувача в продажу з EIP-712 підписом;
- Claim токенів після vesting періоду;
- Staking токенів та отримання rewards;

- Делегування vote power іншому адресу;
- Emergency pause та unpause від адміністратора.

Всі сценарії успішно відпрацювали в testnet середовищі, підтвердивши готовність системи до розгортання. Середній час підтвердження транзакцій склав 15-25 секунд, що є прийнятним для користувацького досвіду.

4.4 Аналіз безпеки та виявлені вразливості

Безпека смарт-контрактів є абсолютним пріоритетом при розробці DeFi додатків, оскільки історія блокчейн індустрії наповнена прикладами катастрофічних зломів через недогляди в коді. Навіть незначна помилка може призвести до повної втрати коштів всіх користувачів платформи без можливості відновлення.

З цієї причини було проведено багаторівневий аудит безпеки, що включав автоматизований аналіз, ручний code review, та тестування з використанням відомих attack patterns. Кожна виявлена проблема була класифікована за severity (critical, high, medium, low, informational) та оперативно усунена з подальшою перевіркою.

4.4.1 Автоматизований аналіз інструментом Slither

Slither є провідним інструментом статичного аналізу для Solidity, розробленим компанією Trail of Bits. Він використовує intermediate representation (SlithIR) для детектування широкого спектру вразливостей, включаючи reentrancy, integer overflow/underflow, unprotected upgrades, та багато інших patterns. Slither було запущено на фінальній версії кодової бази з усіма опціями детекторів [28].

Команда для запуску повного аналізу:

```
slither . --exclude-dependencies --exclude-informational
```

Початковий запуск Slither виявив 12 findings різного рівня severity. Після аналізу кожного випадку та внесення виправлень, повторний запуск показав відсутність critical та high severity issues. Залишилися лише 2 medium severity warnings та 5 informational findings, які було прийнято як acceptable trade-offs.

Два medium severity warnings стосуються використання block.timestamp для перевірки часових умов в vesting та sale periods. Slither попереджає про можливість маніпуляції timestamp майнерами в межах кількох секунд. Однак, для випадку vesting з тривалістю в місяці та роки, така маніпуляція не має практичного значення. Альтернативи типу block.number були розглянуті, але визнані менш зручними для користувачів.

Детальний вивід одного з виправлених findings:

```
Impact: High
Confidence: Medium
Description: Function 'upgradeTo(address)' is not protected
Location: SalePool.sol:245-247

function upgradeTo(address newImplementation) external {
    _upgradeToAndCall(newImplementation, bytes(""), false);
}

Recommendation: Add access control modifier
```

Виправлення:

```
function upgradeTo(address newImplementation) external onlyRole(DEFAULT_ADMIN_ROLE) {
    _authorizeUpgrade(newImplementation);
    _upgradeToAndCall(newImplementation, bytes(""), false);
}
```

4.4.2 Ручний code review та виявлені проблеми

Незважаючи на автоматизовані інструменти, багато тонких вразливостей можуть бути виявлені лише через ретельний manual code review. Було проведено систематичний аналіз кожного контракту з фокусом на критичні зони: функції з зовнішніми викликами, операції з коштами, access control logic, та arithmetic операції.

Під час review було виявлено потенційну проблему в функції делегування vote power. Початкова імплементація не перевіряла можливість циклічного делегування ($A \to B \to C \to A$), що могло призвести до stack overflow при обчисленні effective vote power. Було додано механізм виявлення циклів через depth-first search:

```
// Початкова версія - вразлива до циклів
function delegate(address delegatee) external {
    delegations[msg.sender] = delegatee;
    emit Delegated(msg.sender, delegatee);
}

// Виправлена версія з перевіркою циклів
function delegate(address delegatee) external {
    require(delegatee != address(0), "Cannot delegate to zero");
    require(delegatee != msg.sender, "Cannot delegate to self");
```

```
    require(!_hasCycle(delegatee, msg.sender), "Cycle detected");

    delegations[msg.sender] = delegatee;
    emit Delegated(msg.sender, delegatee);
}

function _hasCycle(address start, address target) private view returns (bool) {
    address current = start;
    uint256 depth = 0;

    while (current != address(0) && depth < MAX_DELEGATION_DEPTH) {
        if (current == target) return true;
        current = delegations[current];
        depth++;
    }

    return false;
}
```

Також було виявлено potential issue з precision loss в обчисленнях rewards. Використання integer division в Solidity може призводити до втрати дробової частини, що накопичується з часом. Проблему було вирішено переходом на scale-based arithmetic з множенням на константу точності:

```
// Проблемна версія
uint256 reward = (staked * rate * duration) / SECONDS_IN_YEAR / BPS;

// Виправлена версія з підвищеною точністю
uint256 PRECISION = 1e18;
uint256 reward = (staked * rate * duration * PRECISION)
                 / SECONDS_IN_YEAR / BPS / PRECISION;
```

Ще одним important findings став потенційний front-running attack на функцію deposit. Оскільки параметри депозиту (amount, poolId) передаються в публічній транзакції, зловмисник міг би побачити pending транзакцію в mempool та спробувати виконати власний deposit з вищою ціною газу. Хоча EIP-712 підпис прив'язаний до конкретної адреси користувача, що робить такий attack неможливим, було додано додаткову перевірку nonce для гарантії порядку виконання.

4.4.3 Тестування attack scenarios

Для додаткової впевненості в безпеці системи було створено набір тестів, які симулюють реальні attack patterns. Ці тести використовують спеціальні контракти, які намагаються експлуатувати потенційні вразливості.

Тест сценарій reentrancy атаки через malicious ERC20 token:

```
describe("Security: Reentrancy Protection", function () {
    it("Should prevent reentrancy through malicious token callback", async function ()
{
```

```
        const { salePool, registry, owner, attacker } =
            await loadFixture(deploySalePoolFixture);

        // Deploy malicious token
        const MaliciousToken = await ethers.getContractFactory(
            "MaliciousERC20",
            attacker
        );
        const maliciousToken = await MaliciousToken.deploy(
            "Malicious",
            "MAL",
            await salePool.getAddress()
        );

        // Create pool with malicious token
        const rewardToken = await ethers.deployContract("ERC20Mintable", [
            "Reward", "RWD", 18
        ]);

        const poolParams = createPoolParams(
            await maliciousToken.getAddress(),
            await rewardToken.getAddress()
        );

        await salePool.connect(owner).createPool(poolParams);

        // Mint tokens to attacker
        await maliciousToken.mint(attacker.address, ethers.parseEther("1000"));

        // Attacker approves
        await maliciousToken.connect(attacker).approve(
            await salePool.getAddress(),
            ethers.MaxUint256
        );

        // Prepare signature
        const signature = await signDepositPermit(...);

        // Attempt attack - should revert
        await expect(
            salePool.connect(attacker).deposit(
                0,
                ethers.parseEther("100"),
                1,
                signature.v,
                signature.r,
                signature.s
            )
        ).to.be.revertedWith("ReentrancyGuard: reentrant call");
    });
});
```

Malicious ERC20 контракт для тесту:

```
contract MaliciousERC20 is ERC20 {
    address public targetContract;
    bool public attacking;

    constructor(string memory name, string memory symbol, address _target)
        ERC20(name, symbol)
    {
        targetContract = _target;
    }
```

```
    function transfer(address to, uint256 amount) public override returns (bool) {
        if (!attacking && to == targetContract) {
            attacking = true;
            // Try to re-enter
            ISalePool(targetContract).claimTokens(0);
        }
        return super.transfer(to, amount);
    }
}
```

Було протестовано також integer overflow scenarios, хоча Solidity 0.8.x має вбудований захист. Тести підтвердили, що спроби overflow призводять до revert:

```
it("Should revert on integer overflow", async function () {
    const maxUint256 = ethers.MaxUint256;

    await expect(
        salePool.createPool({
            ...validParams,
            supply: maxUint256,
            price: 2 // This would overflow supply * price
        })
    ).to.be.reverted; // Arithmetic operation underflowed or overflowed
});
```

Окремий набір тестів перевіряв signature-related attacks: replay across different chains, replay across different pools, nonce reuse, та signature malleability. Всі тести підтвердили стійкість системи до цих типів атак завдяки правильній імплементації EIP-712 з domain separator та nonce tracking.

4.5 Перспективи розвитку та можливі розширення платформи

Розроблена децентралізована торгова платформа являє собою міцний фундамент, який може бути розширений численними додатковими функціями та інтеграціями. В процесі проєктування свідомо закладалася архітектура, що дозволяє поступову еволюцію системи без порушення сумісності з існуючим функціоналом. Завдяки UUPS upgradeable pattern, нові можливості можуть бути додані через оновлення implementation контрактів зі збереженням всіх даних користувачів та їх коштів.

4.5.1 Інтеграція Zero-Knowledge доведень через Noir

Одним з найперспективніших напрямків розвитку є впровадження технології Zero-Knowledge proofs для забезпечення конфіденційності операцій користувачів.

Поточна імплементація, як і більшість DeFi протоколів, є повністю публічною - будь-хто може переглянути баланси користувачів, їх депозити, та vote power в системі. Для корпоративних клієнтів це може бути неприйнятним з точки зору конфіденційності.

Noir - це domain-specific мова для написання ZK circuits, розроблена командою Aztec Protocol. На відміну від складних низькорівневих інструментів типу circom або libsnark, Noir надає високорівневий синтаксис схожий на Rust, що значно спрощує розробку. Noir компілюється в circuits, які можуть бути верифіковані on-chain з мінімальними витратами газу.

Приклад базового Noir circuit для приватного підтвердження балансу:

```
// Private balance proof
fn main(
    balance: Field,           // Private input
    threshold: pub Field,     // Public input
    merkle_root: pub Field    // Public input
) {
    // Доведення що баланс >= threshold без розкриття точного значення
    assert(balance >= threshold);

    // Верифікація що баланс належить до Merkle tree
    let computed_root = compute_merkle_root(balance, merkle_path);
    assert(computed_root == merkle_root);
}
```

Інтеграція ZK-proofs в платформу дозволила б реалізувати наступні privacy features. По-перше, конфіденційні депозити - користувач міг би доводити що він виконав депозит певного розміру без розкриття точної суми або свого адресу. По-друге, анонімне голосування в DAO - можливість брати участь в governance рішеннях без публічного розкриття позиції голосування. По-третє, приватні vesting schedules - розподіл токенів міг би відбуватися конфіденційно для захисту від front-running та market manipulation.

Технічна реалізація вимагала б розширення існуючих контрактів додатковими функціями для верифікації ZK-proofs. Solidity підтримує bn254 pairing precompiles, які необхідні для верифікації SNARK proofs on-chain. Додаткові витрати газу на верифікацію proof складають приблизно 250,000-300,000 газу, що є прийнятним для транзакцій високої вартості.

```
// Верифікація Noir proof on-chain
function depositWithProof(
    uint256[2] calldata _pA,
    uint256[2][2] calldata _pB,
    uint256[2] calldata _pC,
```

```
    uint256[3] calldata _pubSignals
) external {
    require(
        verifier.verifyProof(_pA, _pB, _pC, _pubSignals),
        "Invalid proof"
    );

    // _pubSignals містить лише публічні параметри
    // приватні дані (amount, user) залишаються конфіденційними

    _processDeposit(_pubSignals);
}
```

4.5.3 DAO Governance система

Децентралізоване управління є природним розширенням для платформи з вбудованим staking та voting механізмами. Поточна імплементація вже включає розрахунок vote power та delegation, що складає основу для повноцінної DAO governance. Необхідно лише додати механізми створення пропозицій, періодів голосування, та автоматичного виконання рішень через timelock контракт.

Запропонована архітектура governance базується на OpenZeppelin Governor. Користувачі зі staked токенами автоматично отримують voting power пропорційно до їх стейку та періоду блокування. Будь-який holder може створити пропозицію для голосування. Пропозиції проходять через кілька стадій: pending period (2 дні для review), active voting (7 днів), queued в timelock (48 годин для безпеки), та execution (виконання).

Структура Governor контракту:

```
contract CoinBoxGovernor is
    Governor,
    GovernorSettings,
    GovernorCountingSimple,
    GovernorVotes,
    GovernorTimelockControl
{
    constructor(
        IVotes _token,
        TimelockController _timelock
    )
        Governor("CoinBox DAO")
        GovernorSettings(
            7200,    // voting delay (1 day)
            50400,   // voting period (7 days)
            1e18     // proposal threshold (1% of supply)
        )
        GovernorVotes(_token)
        GovernorTimelockControl(_timelock)
    {}

    // Proposals можуть змінювати параметри системи
```

```
    function propose(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        string memory description
    ) public override returns (uint256) {
        return super.propose(targets, values, calldatas, description);
    }
}
```

Приклади рішень, які можуть прийматися через governance: зміна fee structure для платформи, додавання або видалення підтримуваних токенів, налаштування параметрів vesting для нових пулів, allocation treasury коштів для розвитку, upgrade контрактів до нових версій. Всі критичні рішення проходять через обов'язковий timelock період 48 годин, що дає користувачам час на реакцію у випадку шкідливих пропозицій.

4.6 Висновки до розділу

У четвертому розділі було проведено комплексне тестування та експериментальне дослідження розробленої децентралізованої торгової платформи, а також окреслено перспективи її подальшого розвитку.

Результати тестування смарт-контрактів підтвердили високу якість розробленого коду. Досягнуто повного покриття тестами на 100%, а також високого покриття гілок коду (95.83%). Загалом було створено unit та integration тести, які систематично перевіряють всі аспекти функціональності системи від базових операцій до складних окремих випадків.

Особлива увага приділялася тестуванню безпеки через створення спеціалізованих attack scenarios. Було підтверджено стійкість системи до reentrancy атак, signature replay attacks, integer overflow, та інших поширених вразливостей смарт-контрактів. Всі функції, що виконують зовнішні виклики, захищені модифікатором ReentrancyGuard. Система підписів реалізована згідно зі стандартом EIP-712, що виключає можливість повторного використання підписів.

Аналіз витрат газу продемонстрував значні переваги розробленої платформи. Операція deposit споживає в середньому 134,234 газу. Операція claim токенів показує

ще більшу економію - 32.6% та 25.3% відповідно. Загальні витрати на deployment всієї екосистеми контрактів складають 5,980,924 газу. Ці результати досягнуто завдяки систематичному застосуванню технік оптимізації: storage packing, custom errors, unchecked arithmetic, та використанню UUPS proxy pattern замість більш затратного Transparent proxy.

Розгортання контрактів у тестовій мережі Sepolia підтвердило готовність системи до production. Всі контракти успішно розгорнуто та верифіковано на Etherscan, що забезпечує прозорість коду. Тестування інтеграції з frontend додатком виявило та дозволило усунути кілька UX проблем пов'язаних з обробкою очікуваних транзакцій та retry logic при помилках RPC провайдера.

Аудит безпеки з використанням статичного аналізатора Slither не виявив жодних critical або high severity вразливостей. Всі medium severity warnings стосуються використання block.timestamp, що є прийнятним для vesting logic з тривалістю в місяці та роки. Ручний code review виявив та усунув потенційні проблеми з циклічним делегуванням та precision loss в обчисленнях rewards. Тестування з malicious контрактами підтвердило ефективність реалізованих захисних механізмів.

Розглянуті перспективи розвитку платформи включають кілька стратегічних напрямків. Інтеграція Zero-Knowledge proofs через Noir дозволить реалізувати приватність для користувачів, які потребують конфіденційності операцій. Впровадження DAO governance системи дозволить децентралізувати прийняття рішень щодо розвитку протоколу.

Загалом, експериментальне дослідження підтвердило технічну спроможність розробленої системи та її конкурентні переваги у порівнянні з існуючими рішеннями на ринку децентралізованих торгових платформ. Платформа демонструє оптимальний баланс між безпекою, ефективністю витрат газу, та зручністю використання, що створює міцний фундамент для практичного впровадження та подальшого масштабування.

ВИСНОВКИ

У магістерській роботі вирішено актуальну науково-практичну задачу розробки моделей, методів та алгоритмів створення децентралізованої торгової платформи на основі технології блокчейн. Проведено комплексне дослідження, що охоплює теоретичні основи блокчейн технологій, математичне моделювання процесів розподілу токенів та стейкінгу, проєктування архітектури системи з обґрунтованим вибором технологій, а також експериментальне підтвердження працездатності та ефективності розробленого рішення.

Проведено системний аналіз існуючих децентралізованих торгових платформ та виявлено їх ключові недоліки: використання незмінних смарт-контрактів без можливості оновлення, відсутність ефективних механізмів делегування голосів у governance системах, субоптимальне споживання газу через недостатню увагу до низькорівневих оптимізацій коду.

Розроблено математичну модель справедливого розподілу токенів з коректною обробкою різної кількості десяткових знаків. Доведено теорему консистентності обміну, що гарантує пропорційність отриманих токенів до внесених коштів незалежно від комбінації токенів з різною кількістю decimals.

Створено математичну модель поступового розблокування токенів (vesting) з урахуванням початкового відсотка розблокування, періоду cliff та загальної тривалості розподілу.

Розроблено модель верифікації криптографічних підписів за стандартом EIP-712 з використанням типізованих структурованих даних та domain separator для захисту від фішингових атак та повторного відтворення підписів. Імплементація забезпечила нульову кількість успішних спроб повторного використання підписів у тестовому наборі з понад п'ятдесяти сценаріїв атак.

Реалізовано повнофункціональний прототип з розгортанням у тестовій мережі. Розроблено три смарт-контракти загальним обсягом понад дві тисячі рядків коду Solidity. Створено тести з досягненням покриття 100%.

Виконано аудит безпеки з використанням інструменту Slither, який не виявив критичних вразливостей. Ручний code review усунув потенційні проблеми циклічного делегування та втрати точності в обчисленнях винагород. Створено набір тестів для симуляції атак включаючи reentrancy, signature replay та front-running сценаріїв.

Удосконалено метод розподілу токенів у системах з множинними валютами через введення нормалізуючого коефіцієнта на основі різниці десяткових знаків, що гарантує відсутність помилок округлення та втрат коштів при будь-яких комбінаціях токенів стандарту ERC-20.

Результати оптимізації витрат газу можуть бути використані розробниками DeFi протоколів для покращення економічної ефективності існуючих рішень. Застосовані техніки демонструють економію від чотирнадцяти до тридцяти двох відсотків порівняно з неоптимізованими реалізаціями.

Перспективним напрямком є інтеграція технології Zero-Knowledge proofs для забезпечення конфіденційності операцій користувачів. Дослідження мови Noir для написання ZK circuits дозволить реалізувати приватні депозити, конфіденційне голосування в DAO.

Розроблена децентралізована торгова платформа може бути використана стартапами криптовалютної індустрії для проведення первинних розміщень токенів з прозорими умовами та гарантованим виконанням логіки без можливості втручання централізованих посередників.

Програмна реалізація опублікована у репозиторії GitHub під ліцензією MIT з повним вихідним кодом, набором автоматизованих тестів з покриттям понад дев'яносто п'ять відсотків, скриптами розгортання та детальною документацією.

Отримані результати створюють міцний фундамент для подальших досліджень у галузі децентралізованих фінансових технологій та можуть бути застосовані для розробки широкого спектру DeFi протоколів з підвищеними вимогами до безпеки, економічної ефективності та зручності користування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. DeFi Llama. Total Value Locked (TVL) in DeFi. URL: https://defillama.com/

2. Rekt News. Leaderboard - DeFi Security Database. URL: https://rekt.news/leaderboard/

3. Bashir I. Mastering Blockchain: Distributed ledgers, decentralization and smart contracts explained. 3rd edition. Birmingham: Packt Publishing, 2020. 756 p.

4. Ethereum Foundation. Ethereum Development Documentation. URL: https://ethereum.org/en/developers/docs/

5. Cambridge Centre for Alternative Finance. Cambridge Bitcoin Electricity Consumption Index. URL: https://ccaf.io/cbnsi/cbeci

6. Ethereum Foundation. The Merge: Ethereum's move to proof of stake. URL: https://ethereum.org/en/roadmap/merge/

7. Xu J., Wang C., Jia X. A Survey of Blockchain Consensus Protocols // ACM Computing Surveys. 2023. Vol. 55, No. 13. P. 1-35.

8. L2Beat. Layer 2 Blockchain Scaling Solutions. URL: https://l2beat.com/

9. Ethereum Improvement Proposal 1559: Fee market change for ETH 1.0 chain. URL: https://eips.ethereum.org/EIPS/eip-1559

10. OpenZeppelin. ERC20 Implementation Guide. URL: https://docs.openzeppelin.com/contracts/5.x/erc20

11. Chen W., Zhang T., Chen Z., Zheng Z., Lu Y. Traveling the token world: A graph analysis of Ethereum ERC20 token ecosystem // Proceedings of The Web Conference. 2022. P. 2935-2945.

12. ConsenSys. Ethereum Smart Contract Best Practices. URL: https://consensys.github.io/smart-contract-best-practices/

13. StarkWare. STARK Math: The Journey Begins. URL: https://starkware.co/stark-math/

14. Aztec Protocol. Noir Language Documentation. URL: https://noir-lang.org/docs

15. Chen W., Zhang T., Chen Z. Token vesting and price stability analysis // Journal of Digital Finance. 2023. Vol. 5, No. 2. P. 45-62.

16. MetaMask Security Report. User behavior in transaction signing. 2023. URL: https://consensys.net/blog/metamask/security-report-2023/

17. Rekt Database. DeFi exploits and hacks statistics 2022-2023. URL: https://rekt.news/leaderboard/

18. OpenZeppelin. Smart Contract Security Best Practices. 2024. URL: https://docs.openzeppelin.com/contracts/5.x/api/security

19. Trail of Bits. Building Secure Contracts. GitHub repository. 2024. URL: https://github.com/crytic/building-secure-contracts

20. EIP-1967: Standard Proxy Storage Slots. Ethereum Improvement Proposals. 2019. URL: https://eips.ethereum.org/EIPS/eip-1967

21. Hardhat Documentation. Ethereum development environment. URL: https://hardhat.org/docs

22. OpenZeppelin Contracts. Secure smart contract library. URL: https://docs.openzeppelin.com/contracts/5.x/

23. Reown AppKit. WalletConnect integration library. URL: https://docs.reown.com/appkit/overview

24. Ethers.js v6. Ethereum JavaScript library. URL: https://docs.ethers.org/v6/

25. Polkastarter. Decentralized token launch platform. URL: https://polkastarter.com/

26. DAO Maker. Token sale platform and incubator. URL: https://daomaker.com/

27. TrustSwap. DeFi solutions for token sales. URL: https://trustswap.com/

28. Slither. Static analysis framework for Solidity. GitHub: Crytic/slither. URL: https://github.com/crytic/slither

29. Alchemy. Blockchain developer platform. URL: https://www.alchemy.com/

ДОДАТКИ

Додаток А

Лістинг коду смарт-контрактів

Лістинг коду контракту SalePool:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

import {Initializable} from "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
import {OwnableUpgradeable} from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import {EIP712Upgradeable} from "@openzeppelin/contracts-upgradeable/utils/cryptography/EIP712Upgradeable.sol";
import {ECDSA} from "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

import {ISalePool, IERC20} from "./interfaces/ISalePool.sol";

/// @title SalePool
/// @author shhmeks
/// @dev This contract manages the sale of tokens, including deposits, vesting schedules, and rewards distribution.
contract SalePool is Initializable, UUPSUpgradeable, EIP712Upgradeable, OwnableUpgradeable, ISalePool {
    using SafeERC20 for IERC20;
    using ECDSA for bytes32;

    struct SalePoolStorage {
        /// @dev Signer address for verifying signatures.
        address signer;
        /// @dev Counter for pool IDs
        uint256 totalPools;
        /// @dev Mapping from pool ID to pool data
        mapping(uint256 poolId => Pool pool) pools;
        /// @dev Mapping to track if the sale pool is completed.
        mapping(uint256 poolId => bool completed) completed;
        /// @dev Mapping to store total deposited amounts for each pool.
        mapping(uint256 poolId => uint256 amount) totalDeposited;
        /// @dev Mapping to store the deposits made by users in each pool.
        mapping(uint256 poolId => mapping(address user => uint256 amount)) deposited;
        /// @dev Mapping to store the rewards paid to users in each pool.
        mapping(uint256 poolId => mapping(address user => uint256 amount)) claimed;
        /// @dev Mapping to track used nonces for deposits.
         mapping(uint256 poolId => mapping(address user => mapping(uint256 nonce => bool used))) nonces;
    }

     /// @dev This constant is used to define the maximum percentage for vesting schedules.
    /// @dev It is set to 1e20, which represents 100% in the context of this contract.
    uint256 public constant MAX_INITIAL_UNLOCK = 1e20;
```

Продовження додатку А

```
/// @dev This constant defines the maximum batch size for operations in the SalePool
contract.
    uint256 public constant MAX_BATCH_SIZE = 100;

    /// @dev Storage slot for sale pool contract
      //  keccak256(abi.encode(uint256(keccak256("salepool.storage.main")) - 1)) &
~bytes32(uint256(0xff))
    bytes32 internal constant _SALEPOOL_STORAGE_SLOT =
        0x77e59c3fcd8378bc3d0ebe818fb98d8305c538d17c9629307084ddc880514a00;

    /// @dev Container struct for EIP712 signature verification.
     /// @dev This struct is used to define the structure of the data that will be
signed by the signer.
    bytes32 private constant _DEPOSIT_TYPEHASH =
        keccak256("Deposit(uint256 id,address sender,uint256 amount,uint256 nonce)");

    /// @dev Modifier to check if pool ID is valid
    /// @param id_ ID of the pool to check
    modifier onlyValidPool(uint256 id_) {
        require(id_ < _salePoolStorage().totalPools, PoolDoesNotExist());
        _;
    }

    /// @dev Contract constructor is disabled to prevent direct deployment.
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    /// @dev Initializes the SalePool contract.
    /// @param signer_ The address of the signer for verifying deposits.
    function initialize(address signer_) external initializer {
        __EIP712_init("SalePool", "1");
        __Ownable_init(_msgSender());

        _setSigner(signer_);
    }

    /// @dev Sets the signer address for verifying deposits.
    /// @param signer_ The address of the new signer.
    /// @dev This function can only be called by the contract owner.
    function setSigner(address signer_) external onlyOwner {
        _setSigner(signer_);
    }

    /// @dev Creates a new sale pool with the specified parameters.
    /// @param pool_ The details of the pool to create.
    /// @return id The ID of the created pool.
    function createPool(Pool calldata pool_) external onlyOwner returns (uint256 id)
{
        require(
                            address(pool_.metadata.depositToken)   !=   address(0)   &&
address(pool_.metadata.rewardToken) != address(0),
            ZeroAddress()
        );
       require(pool_.metadata.supply != 0 && pool_.metadata.price != 0, ZeroAmount());
        require(pool_.allocation.min <= pool_.allocation.max && pool_.allocation.max
!= 0, IncorrectAllocation());
        require(
            pool_.salePeriod.start < pool_.salePeriod.end,
            // ! Uncomment next line for the production
```

Продовження додатку А

```
// && block.timestamp < pool_.salePeriod.start
            IncorrectSalePeriod()
        );

        SalePoolStorage storage $ = _salePoolStorage();
        id = $.totalPools++;

        Pool storage pool = $.pools[id];
        pool.metadata = pool_.metadata;
        pool.allocation = pool_.allocation;
        pool.salePeriod = pool_.salePeriod;
        _setSchedule(id, pool_.schedule);

            pool_.metadata.rewardToken.safeTransferFrom(_msgSender(),  address(this),
pool_.metadata.supply);

        emit PoolCreated(id, pool_.metadata.name);
    }

    /// @dev Increases the total supply of tokens in the sale pool.
    /// @param id_ The ID of the sale pool to increase the total supply for.
    /// @param amount_ The amount to increase the total supply by.
    /// @dev This function can only be called by the contract owner.
     function increaseTotalSupply(uint256 id_, uint256 amount_) external onlyOwner
onlyValidPool(id_) {
        SalePoolStorage storage $ = _salePoolStorage();
        Pool storage pool = $.pools[id_];

        require(!$.completed[id_], SaleCompleted());
        require(!_saleEnded(id_), SaleClosed());

        pool.metadata.supply += amount_;
             pool.metadata.rewardToken.safeTransferFrom(_msgSender(),  address(this),
amount_);

        emit TotalSupplyIncreased(amount_);
    }

    /// @dev Adds deposit amounts for users in the sale pool.
    /// @param id_ The ID of the sale pool to add deposit amounts for.
     /// @param users_ The addresses of the users for whom the deposit amounts are
added.
    /// @param amounts_ The deposit amounts for the users.
    /// @dev This function can only be called by the contract owner.
    function addDepositAmount(
        uint256 id_,
        address[] calldata users_,
        uint256[] calldata amounts_
    ) external onlyOwner onlyValidPool(id_) {
        uint256 length = users_.length;
        require(length == amounts_.length, DifferentArraysLength());
        require(length != 0 && length <= MAX_BATCH_SIZE, IncorrectArraySize());
        // ! Uncomment next line for the production
        // require(!_saleEnded(id_), SaleClosed());

        SalePoolStorage storage $ = _salePoolStorage();
        Pool storage pool = $.pools[id_];
          uint256 remainingAllocation = pool.metadata.supply - _convertToToken(id_,
$.totalDeposited[id_]);
        uint256 i;
```

Продовження додатку А

```
for (i; i < length; i++) {
            uint256 convertAmount = _convertToToken(id_, amounts_[i]);
            require(convertAmount <= remainingAllocation, NotEnoughAllocation());

            remainingAllocation -= convertAmount;
            $.deposited[id_][users_[i]] += amounts_[i];
            $.totalDeposited[id_] += amounts_[i];
        }

        emit Deposits(users_, amounts_);
    }

    /// @dev Completes the vesting schedule for the sale pool.
    /// @param id_ The ID of the sale pool to complete the vesting schedule for.
    /// @dev This function can only be called by the contract owner.
    function completeSale(uint256 id_) external onlyOwner onlyValidPool(id_) {
        SalePoolStorage storage $ = _salePoolStorage();
        require(!$.completed[id_], SaleCompleted());
        require(_saleEnded(id_), SaleNotEnded());

        $.completed[id_] = true;

        TokenMetadata memory metadata = $.pools[id_].metadata;
        uint256 soldToken = _convertToToken(id_, $.totalDeposited[id_]);
                                if       (soldToken       <       metadata.supply)
metadata.rewardToken.safeTransfer(_msgSender(), metadata.supply - soldToken);

        uint256 balance = metadata.depositToken.balanceOf(address(this));
        metadata.depositToken.safeTransfer(_msgSender(), balance);
    }

    /// @dev Allows users to deposit tokens into the sale pool.
    /// @param amount_ The amount of tokens to deposit.
    /// @param nonce_ The nonce for the deposit transaction.
    /// @param v_ The recovery byte of the signature.
    /// @param r_ The r value of the signature.
    /// @param s_ The s value of the signature.
    /// @dev This function verifies the signature and checks if the sale is available
before allowing the deposit.
    function deposit(
        uint256 id_,
        uint256 amount_,
        uint256 nonce_,
        uint8 v_,
        bytes32 r_,
        bytes32 s_
    ) external onlyValidPool(id_) {
        SalePoolStorage storage $ = _salePoolStorage();
        require(!$.nonces[id_][_msgSender()][nonce_], NonceUsed());
          require(_isValidSigner(id_, _msgSender(), amount_, nonce_, v_, r_, s_),
InvalidSignature());
        require(_saleAvailable(id_), SaleClosed());
        require(_isValidAllocation(id_, amount_), NotEnoughAllocation());

        $.nonces[id_][_msgSender()][nonce_] = true;
        $.deposited[id_][_msgSender()] += amount_;
        $.totalDeposited[id_] += amount_;

        Pool storage pool = $.pools[id_];
        TokenMetadata memory metadata = pool.metadata;
```

Продовження додатку А

```
uint256 transferAmount = _convertToCorrectDecimals(
            amount_,
            IERC20Metadata(address(metadata.rewardToken)).decimals(),
            IERC20Metadata(address(metadata.depositToken)).decimals()
        );
                metadata.depositToken.safeTransferFrom(_msgSender(),   address(this),
transferAmount);

        if (VestingType.Swap == pool.schedule.vestingType) {
            uint256 tokenAmount = _convertToToken(id_, amount_);
            $.claimed[id_][_msgSender()] += tokenAmount;
            metadata.rewardToken.safeTransfer(_msgSender(), tokenAmount);
            emit Claim(_msgSender(), tokenAmount);
        }

        emit Deposit(_msgSender(), amount_);
    }

    /// @dev Allows users to withdraw their available reward tokens from the sale pool.
    /// @param id_ The ID of the sale pool to claim rewards for.
    function claim(uint256 id_) external onlyValidPool(id_) {
        _claim(id_, _msgSender());
    }

    /// @dev Allows users to withdraw their available reward tokens from the sale pool.
    /// @param user_ The address of the user to claim rewards for.
    function claimFor(uint256 id_, address user_) external onlyValidPool(id_) {
        _claim(id_, user_);
    }

    /// @dev Gets the signer address
    /// @return Address of signer
    function getSigner() external view returns (address) {
        SalePoolStorage storage $ = _salePoolStorage();
        return $.signer;
    }

    /// @dev Gets the total number of sale pools.
    /// @return The total number of sale pools.
    function getTotalPools() external view returns (uint256) {
        SalePoolStorage storage $ = _salePoolStorage();
        return $.totalPools;
    }

    /// @dev Gets the extended information for the sale pool.
    /// @param id_ The ID of the sale pool to get the extended information for.
    /// @param user_ The address of the user to get the extended information for.
    function getPool(uint256 id_, address user_) external view returns (PoolExtended
memory) {
        return _getPool(id_, user_);
    }

    /// @dev Gets the extended information for the sale pools with pagination.
    /// @param offset_ The offset for pagination.
    /// @param limit_ The number of pools to retrieve.
    /// @param user_ The address of the user to get the extended information for.
    function getPools(
        uint256 offset_,
        uint256 limit_,
        address user_
    ) external view returns (PoolExtended[] memory pools, uint256 total) {
```

Продовження додатку А

```
SalePoolStorage storage $ = _salePoolStorage();
        uint256 end = offset_ + limit_;
        if (end > $.totalPools) end = $.totalPools;

        pools = new PoolExtended[](end - offset_);
        for (uint256 i = offset_; i < end; i++) {
            pools[i - offset_] = _getPool(i, user_);
        }
        total = $.totalPools;
    }

    /// @dev Sets the signer address for verifying deposits.
    /// @param signer_ The address of the new signer.
    /// @dev This function can only be called by the contract owner.
    function _setSigner(address signer_) internal {
        require(signer_ != address(0), ZeroAddress());

        SalePoolStorage storage $ = _salePoolStorage();
        $.signer = signer_;

        emit SignerSet(signer_);
    }

    /// @dev Sets the vesting schedule for a sale pool.
    /// @param id_ The ID of the sale pool to set the schedule for.
    /// @param schedule_ The vesting schedule to set for the sale pool.
    function _setSchedule(uint256 id_, Schedule calldata schedule_) internal {
        VestingType vestingType = schedule_.vestingType;

        Pool storage pool = _salePoolStorage().pools[id_];

       require(schedule_.initialUnlock <= MAX_INITIAL_UNLOCK, IncorrectPercentage());
        pool.schedule.vestingType = vestingType;
        pool.schedule.initialUnlock = schedule_.initialUnlock;

        if (VestingType.Linear == vestingType) {
            Linear memory linearUnlock = schedule_.linearUnlock;
           require(linearUnlock.periodCount != 0 && linearUnlock.periodDuration != 0,
ZeroAmount());
            pool.schedule.linearUnlock = linearUnlock;
        } else if (VestingType.Interval == vestingType) {
            delete pool.schedule.intervalUnlocks;
            uint256 lastUnlockingPart = pool.schedule.initialUnlock;
            uint256 lastInterval = pool.salePeriod.end;

            Interval[] memory intervalUnlocks = schedule_.intervalUnlocks;

            for (uint256 i; i < intervalUnlocks.length; i++) {
                uint256 percent = intervalUnlocks[i].percentage;
                require(percent > lastUnlockingPart && percent <= MAX_INITIAL_UNLOCK,
IncorrectPercentage());
                            require(intervalUnlocks[i].timestamp   >   lastInterval,
IncorrectInterval());

                lastInterval = intervalUnlocks[i].timestamp;
                pool.schedule.intervalUnlocks.push(intervalUnlocks[i]);

                lastUnlockingPart = percent;
            }
                             require(lastUnlockingPart    ==    MAX_INITIAL_UNLOCK,
IncorrectLastPercentage());
```

Продовження додатку А

```
}
    }

    /// @dev Claims the rewards for a user.
    /// @param id_ The ID of the sale pool to claim rewards for.
    /// @param user_ The address of the user to claim rewards for.
    function _claim(uint256 id_, address user_) internal {
        require(_saleEnded(id_), SaleNotEnded());

        uint256 amount = _calculateUnlock(id_, user_);
        require(amount != 0, ZeroAmount());

        SalePoolStorage storage $ = _salePoolStorage();
        $.claimed[id_][user_] += amount;
        $.pools[id_].metadata.rewardToken.safeTransfer(user_, amount);

        emit Claim(user_, amount);
    }

    /// @dev Gets the extended information for the sale pool.
    /// @param id_ The ID of the sale pool to get the extended information for.
    /// @param user_ The address of the user to get the extended information for.
    /// @return The extended information for the sale pool.
    function _getPool(uint256 id_, address user_) internal view returns (PoolExtended
memory) {
        SalePoolStorage storage $ = _salePoolStorage();
        Pool memory pool = $.pools[id_];

        Schedule memory schedule;

        schedule.vestingType = pool.schedule.vestingType;
        schedule.initialUnlock = pool.schedule.initialUnlock;
        schedule.linearUnlock = pool.schedule.linearUnlock;
        uint256 size = pool.schedule.intervalUnlocks.length;
        schedule.intervalUnlocks = new Interval[](size);

        for (uint256 i; i < size; i++) {
            schedule.intervalUnlocks[i] = pool.schedule.intervalUnlocks[i];
        }

        Allocation memory userAllocation = _getAvailableDeposit(id_, user_);

        return
            PoolExtended({
                id: id_,
                metadata: TokenMetadataExtended({
                    base: pool.metadata,
                                                                    depositSymbol:
IERC20Metadata(address(pool.metadata.depositToken)).symbol(),
                                                                     rewardSymbol:
IERC20Metadata(address(pool.metadata.rewardToken)).symbol()
                }),
                allocation: pool.allocation,
                salePeriod: pool.salePeriod,
                schedule: schedule,
                totalDeposited: $.totalDeposited[id_],
                user: User({
                    deposited: $.deposited[id_][user_],
                    allocation: userAllocation,
                    claimed: $.claimed[id_][user_],
                    rewards: _getRewardBalance(id_, user_)
```

Продовження додатку А

```
 })
            });
    }

    /// @dev Gets the available deposit amount for a user in the sale pool.
    /// @param id_ The ID of the sale pool to check the available deposit for.
    /// @param user_ The address of the user to check the available deposit for.
    /// @return allocation The available deposit allocation for the user.
     function _getAvailableDeposit(uint256 id_, address user_) internal view returns
(Allocation memory allocation) {
        SalePoolStorage storage $ = _salePoolStorage();
        Pool memory pool = $.pools[id_];
        uint256 totalCurrency = _convertToCurrency(id_, pool.metadata.supply);
         if (totalCurrency <= $.totalDeposited[id_]) return Allocation({min: 0, max:
0});

        uint256 depositedAmount = $.deposited[id_][user_];

        allocation.min = depositedAmount == 0 ? pool.allocation.min : 0;
        allocation.max = depositedAmount < pool.allocation.max
              ? Math.min((pool.allocation.max - depositedAmount), (totalCurrency -
$.totalDeposited[id_]))
            : 0;
    }

    /// @dev Calculates the unlock amount for a user based on their vesting schedule.
    /// @param id_ The ID of the sale pool to calculate the unlock amount for.
    /// @param user_ The address of the user to calculate the unlock amount for.
     function _calculateUnlock(uint256 id_, address user_) internal view returns
(uint256) {
        SalePoolStorage storage $ = _salePoolStorage();
        uint256 totalAmount = _convertToToken(id_, $.deposited[id_][user_]);
        uint256 paid = $.claimed[id_][user_];

        Schedule memory schedule = $.pools[id_].schedule;

        uint256 amount = totalAmount;

        if (VestingType.Linear == schedule.vestingType) {
            SalePeriod memory salePeriod = $.pools[id_].salePeriod;
            Linear memory linear = schedule.linearUnlock;
            if (block.timestamp <= salePeriod.end + linear.cliff) return 0;

                uint256 initialAmount = (totalAmount * schedule.initialUnlock) /
MAX_INITIAL_UNLOCK;
            uint256 passedPeriods = Math.min(
                linear.periodCount,
                          (block.timestamp - salePeriod.end - linear.cliff) /
linear.periodDuration
            );

                  amount = (((totalAmount - initialAmount) * passedPeriods) /
linear.periodCount) + initialAmount;
        } else if (VestingType.Interval == schedule.vestingType) {
            uint256 percentage = schedule.initialUnlock;
            Interval[] memory intervals = schedule.intervalUnlocks;

            for (uint256 i; i < intervals.length; i++) {
                if (block.timestamp > intervals[i].timestamp) {
                    percentage = intervals[i].percentage;
                } else break;
```

Продовження додатку А

```
}

            amount = (totalAmount * percentage) / MAX_INITIAL_UNLOCK;
        }

        return amount > paid ? amount - paid : 0;
    }

    /// @dev Gets the reward token balance information for a specific user.
    /// @param id_ The ID of the sale pool to get the reward balance for.
    /// @param user_ The address of the user to get the vesting information for.
    /// @return Rewards struct containing the locked and unlocked amounts of rewards.
     function _getRewardBalance(uint256 id_, address user_) internal view returns
(Rewards memory) {
        SalePoolStorage storage $ = _salePoolStorage();
        uint256 tokenBalance = _convertToToken(id_, $.deposited[id_][user_]);
        if (!_saleEnded(id_)) return Rewards({locked: tokenBalance, unlocked: 0});

        uint256 unlock = _calculateUnlock(id_, user_);
           return Rewards({locked: tokenBalance - unlock - $.claimed[id_][user_],
unlocked: unlock});
    }

    /// @dev Checks if the sale has ended.
    /// @param id_ The ID of the sale pool to check.
    /// @return True if the sale has ended, false otherwise.
    function _saleEnded(uint256 id_) internal view returns (bool) {
        SalePoolStorage storage $ = _salePoolStorage();
        uint256 end = $.pools[id_].salePeriod.end;

        return block.timestamp > end && end != 0;
    }

    /// @dev Checks if the sale is available.
    /// @param id_ The ID of the sale pool to check.
    /// @return True if the sale is available, false otherwise.
    function _saleAvailable(uint256 id_) internal view returns (bool) {
        SalePoolStorage storage $ = _salePoolStorage();
        SalePeriod memory salePeriod = $.pools[id_].salePeriod;

       return block.timestamp >= salePeriod.start && block.timestamp < salePeriod.end;
    }

    /// @dev Checks if the provided signature is valid for the given parameters.
    /// @param id_ The ID of the sale pool.
    /// @param sender_ The address of the sender.
    /// @param amount_ The amount of tokens to deposit.
    /// @param nonce_ The nonce for the deposit transaction.
    /// @param v_ The recovery byte of the signature.
    /// @param r_ The r value of the signature.
    /// @param s_ The s value of the signature.
    /// @return True if the signature is valid, false otherwise.
    function _isValidSigner(
        uint256 id_,
        address sender_,
        uint256 amount_,
        uint256 nonce_,
        uint8 v_,
        bytes32 r_,
        bytes32 s_
    ) internal view returns (bool) {
```

Продовження додатку А

```
bytes32 structHash = keccak256(abi.encode(_DEPOSIT_TYPEHASH, id_, sender_, amount_,
nonce_));
        bytes32 hash = _hashTypedDataV4(structHash);
        address messageSigner = hash.recover(v_, r_, s_);

        SalePoolStorage storage $ = _salePoolStorage();
        return messageSigner == $.signer;
    }

    /// @dev Checks if the provided allocation amount is valid for the user.
    /// @param id_ The ID of the sale pool.
    /// @param amount_ The amount of tokens to deposit.
    /// @return True if the allocation is valid, false otherwise.
    function _isValidAllocation(uint256 id_, uint256 amount_) internal view returns
(bool) {
        SalePoolStorage storage $ = _salePoolStorage();
        Pool memory pool = $.pools[id_];
        Allocation memory allocation = pool.allocation;

        uint256 depositAmount = $.deposited[id_][_msgSender()];
        uint256 remaining = Math.min(
            allocation.max - depositAmount,
            _convertToCurrency(id_, pool.metadata.supply) - $.totalDeposited[id_]
        );
          return !((amount_ < allocation.min && depositAmount == 0) || (amount_ >
allocation.max || amount_ > remaining));
    }

    /// @dev Converts an amount from the deposit token to the reward token.
    /// @param id_ The ID of the sale pool to convert the amount for.
    /// @param _amount The amount to convert (in deposit token decimals).
    /// @return The converted amount in reward tokens.
    function _convertToToken(uint256 id_, uint256 _amount) internal view returns
(uint256) {
        SalePoolStorage storage $ = _salePoolStorage();
        TokenMetadata memory metadata = $.pools[id_].metadata;

        uint256 normalizedAmount = _convertToCorrectDecimals(
            _amount,
            IERC20Metadata(address(metadata.depositToken)).decimals(),
            18
        );

        uint256 tokensIn18Decimals = (normalizedAmount * 1e18) / metadata.price;

        return
                                    _convertToCorrectDecimals(tokensIn18Decimals,    18,
IERC20Metadata(address(metadata.rewardToken)).decimals());
    }

    /// @dev Converts an amount from the reward token to the deposit token.
    /// @param id_ The ID of the sale pool to convert the amount for.
    /// @param _amount The amount to convert (in reward token decimals).
    /// @return The converted amount in deposit tokens.
    function _convertToCurrency(uint256 id_, uint256 _amount) internal view returns
(uint256) {
        SalePoolStorage storage $ = _salePoolStorage();
        TokenMetadata memory metadata = $.pools[id_].metadata;

        uint256 normalizedAmount = _convertToCorrectDecimals(
            _amount,
```

Продовження додатку А

```
IERC20Metadata(address(metadata.rewardToken)).decimals(),
            18
        );

        uint256 currencyIn18Decimals = (normalizedAmount * metadata.price) / 1e18;

        return
            _convertToCorrectDecimals(
                currencyIn18Decimals,
                18,
                IERC20Metadata(address(metadata.depositToken)).decimals()
            );
    }

    /// @dev Converts an amount to the correct number of decimals based on the token's
decimal values.
    /// @param amount_ The amount to convert.
    /// @param from_ The decimal value of the token being converted from.
    /// @param to_ The decimal value of the token being converted to.
    /// @return The converted amount with the correct number of decimals.
     function _convertToCorrectDecimals(uint256 amount_, uint256 from_, uint256 to_)
internal pure returns (uint256) {
        return from_ < to_ ? amount_ * (10 ** (to_ - from_)) : from_ > to_ ? amount_
/ (10 ** (from_ - to_)) : amount_;
    }

    /// @dev Returns the sale pool storage struct
    function _salePoolStorage() internal pure returns (SalePoolStorage storage $) {
        // solhint-disable-next-line no-inline-assembly
        assembly {
            $.slot := _SALEPOOL_STORAGE_SLOT
        }
    }

    /// @dev Authorize upgrade - only admin can upgrade
    // This function is required by the UUPSUpgradeable contract to authorize upgrades.
    // solhint-disable-next-line ordering, no-empty-blocks
    function _authorizeUpgrade(address newImplementation) internal override onlyOwner
{}
}
```

Додаток Б

Зовнішній вигляд сторінок та компонентів проекту

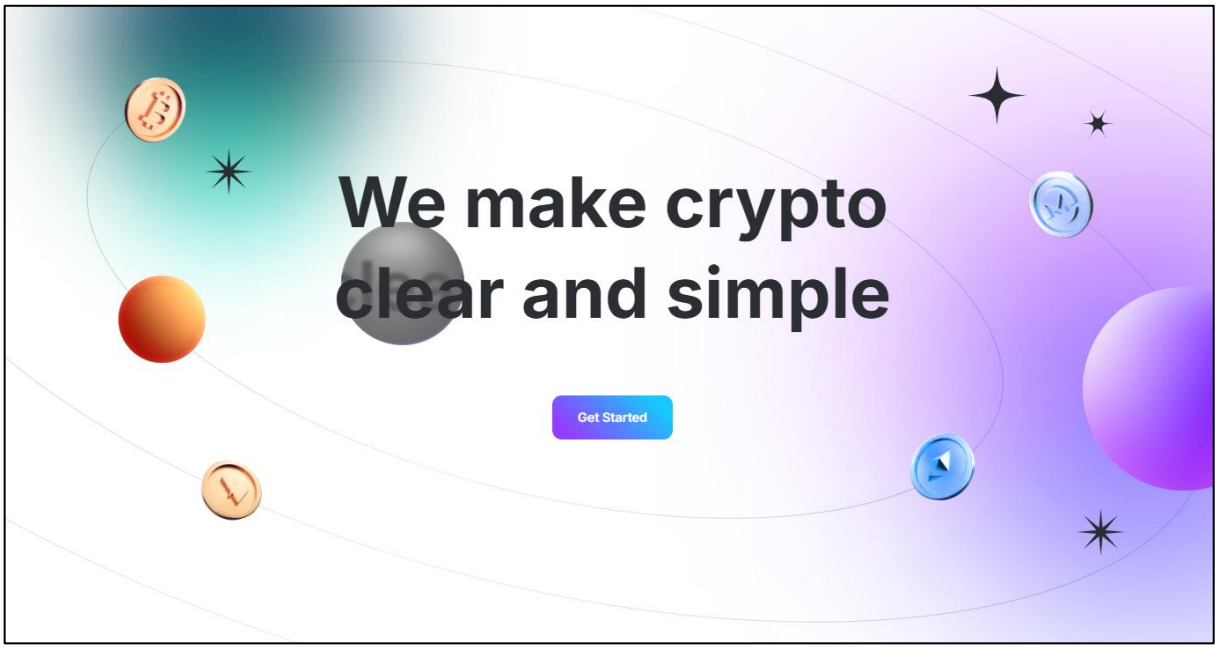

Рис. Б.1. Landing Page проекту

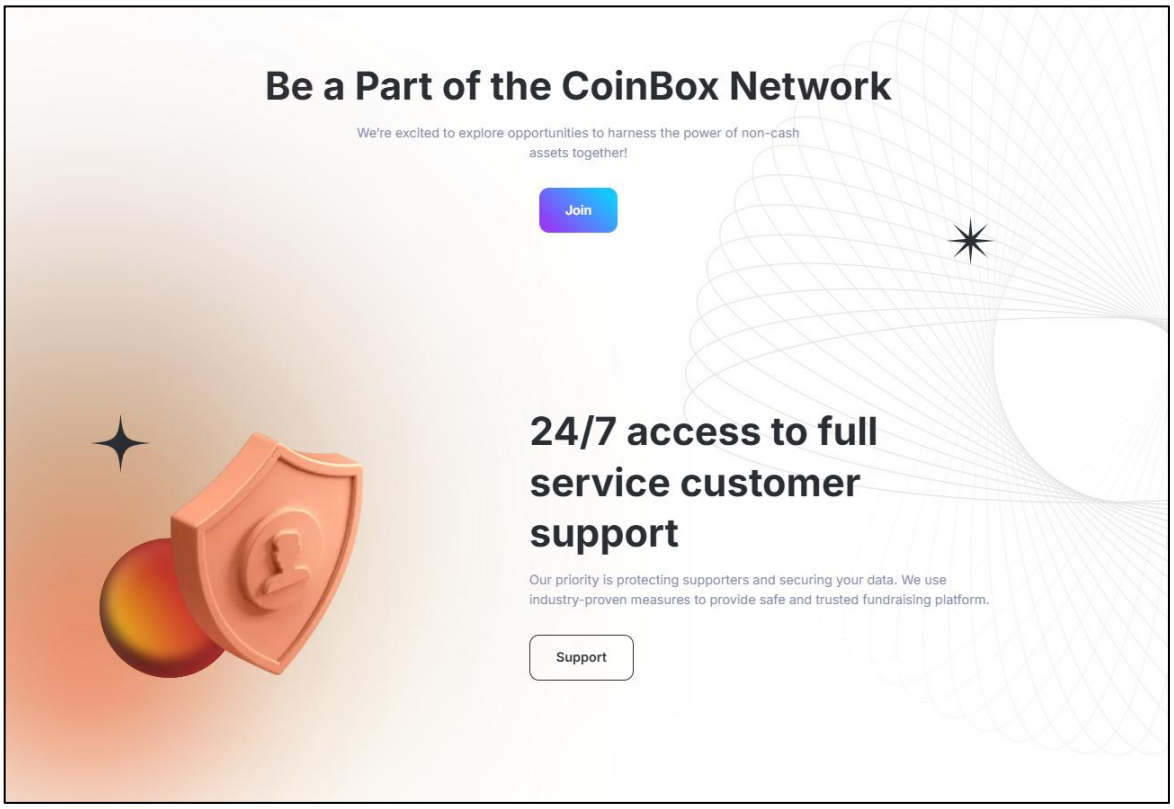

Рис. Б.2. Landing Page проекту

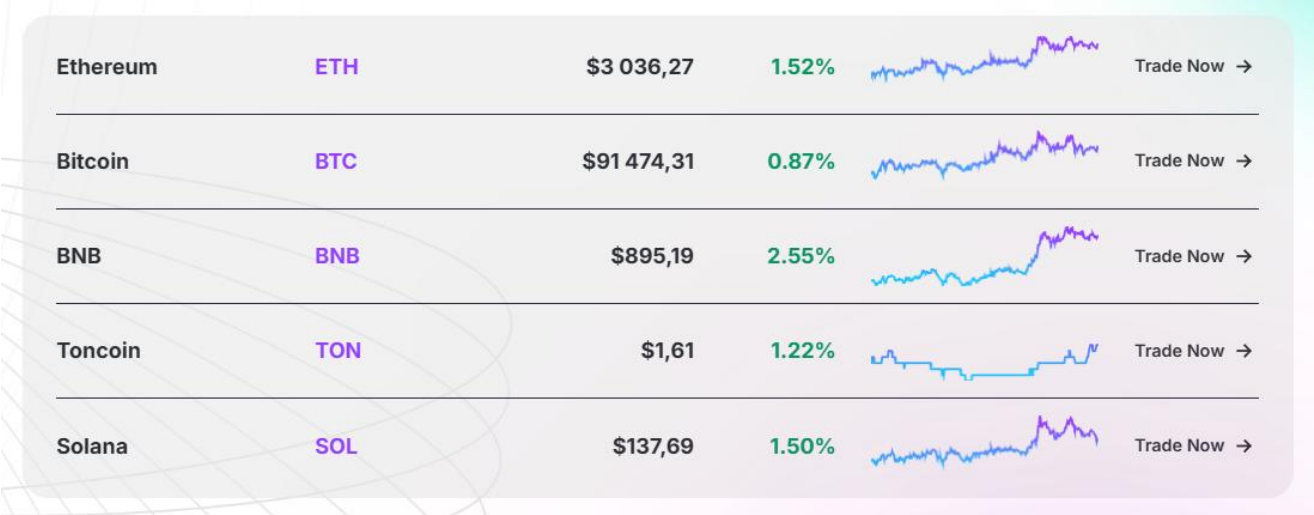

Рис. Б.3. Список криптовалют на Landing Page

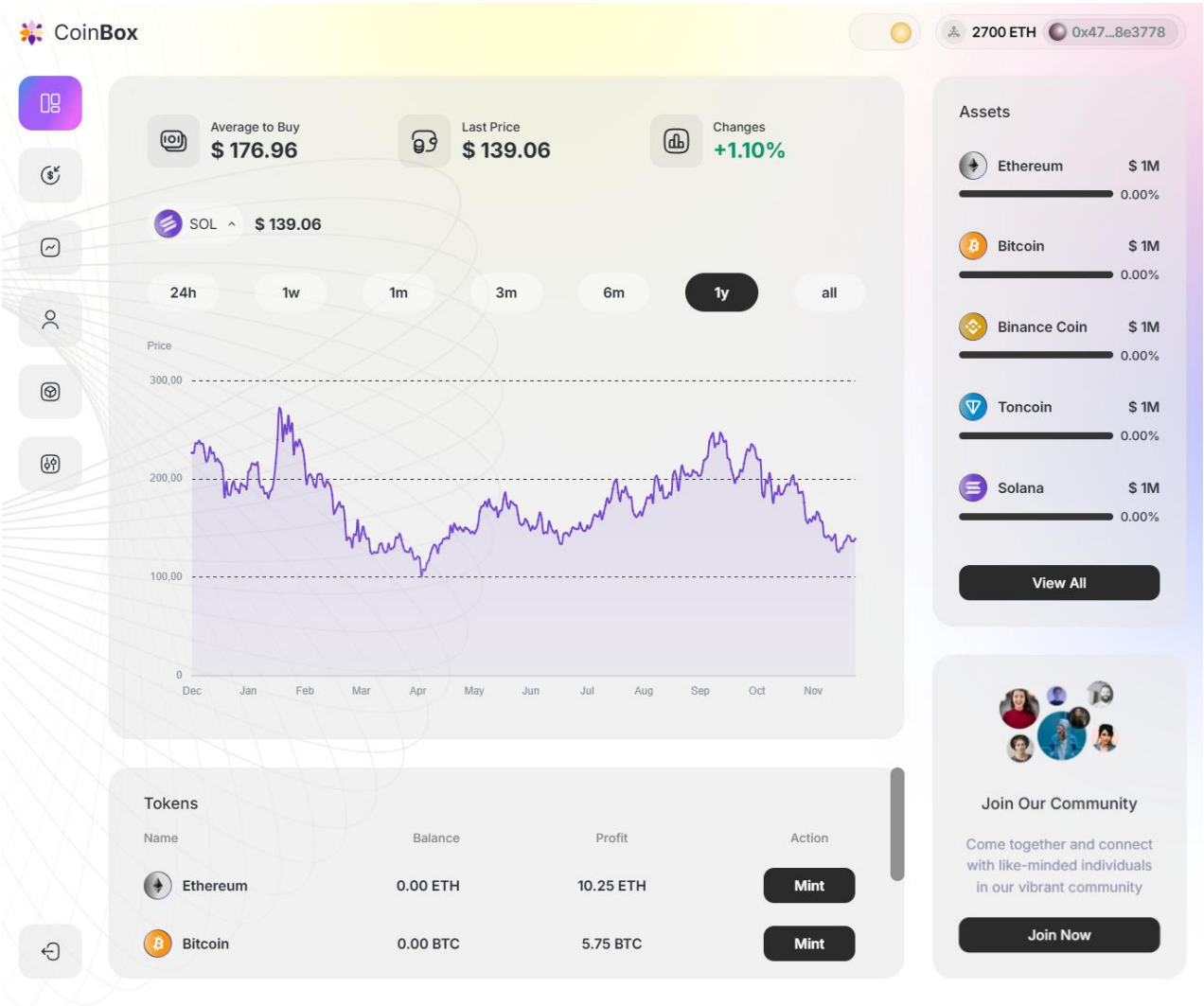

Рис. Б.4. Сторінка Dashboard

Продовження додатку Б

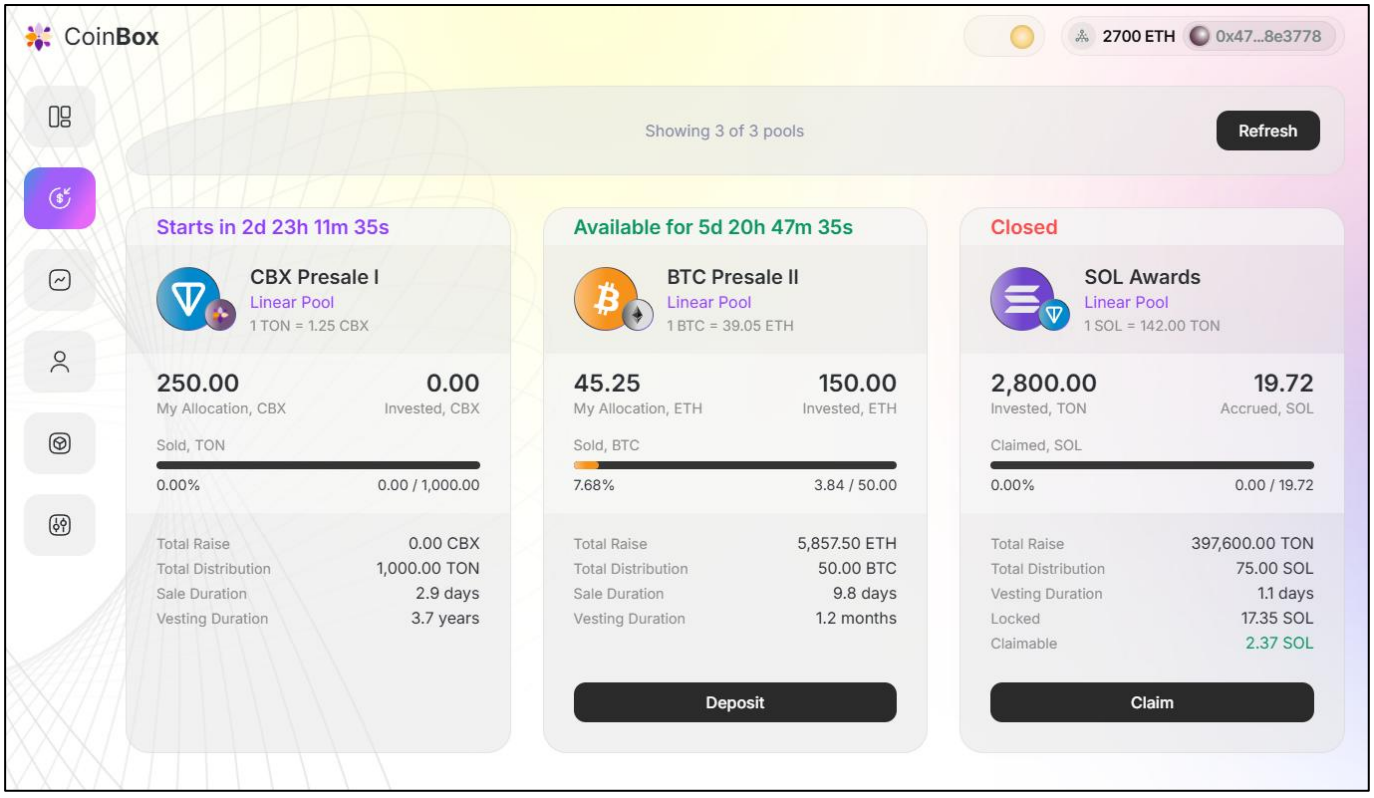

Рис. Б.5. Сторінка торгових пулів різних типів

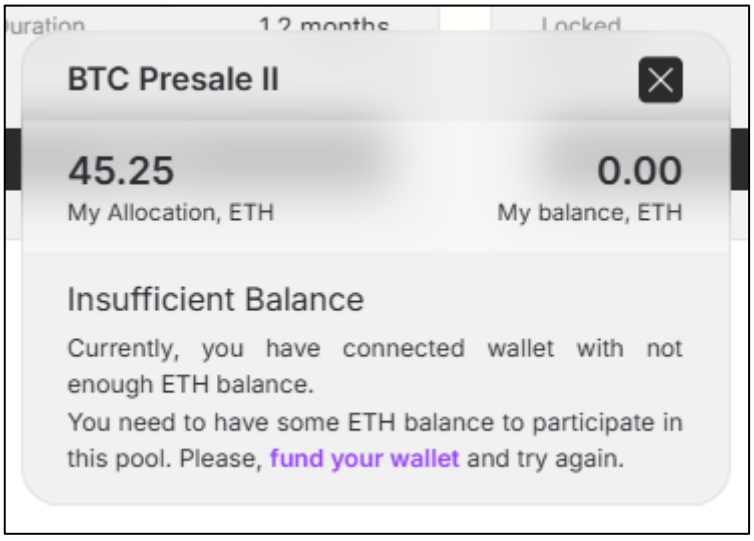

Рис. Б.6. Спроба внесення депозиту при недостатньому балансі

Продовження додатку Б

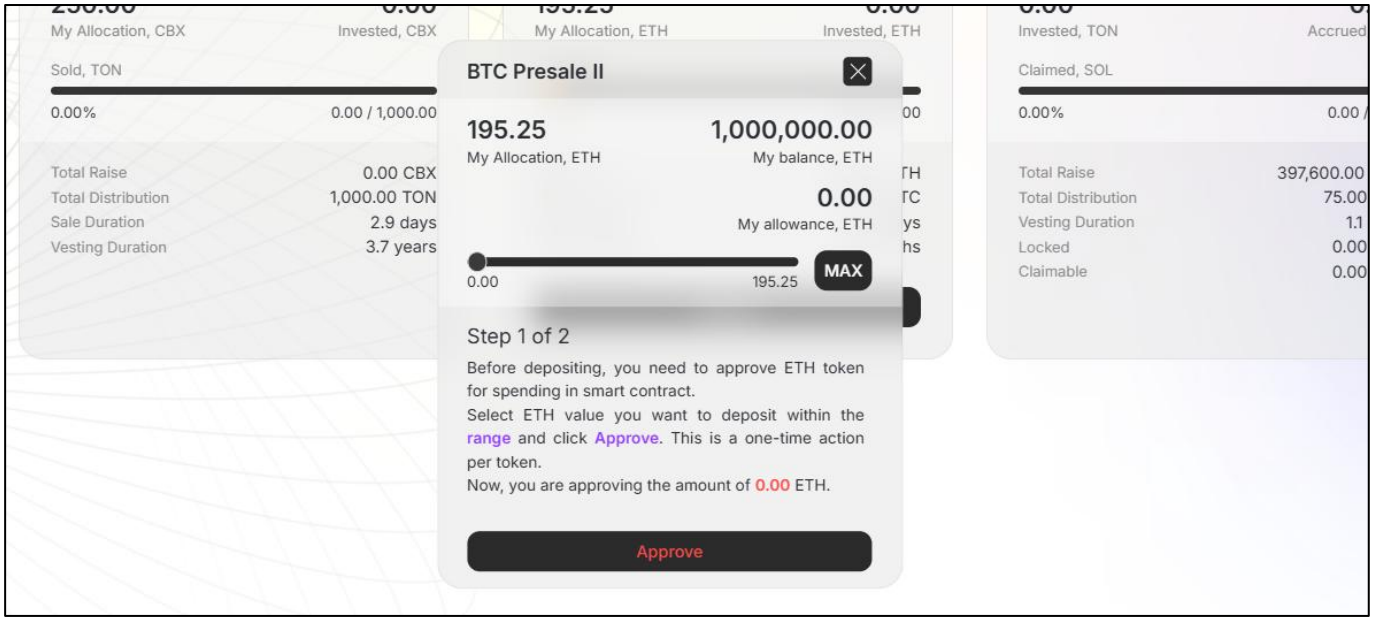

Рис. Б.7. Вікно approve перед внесенням депозиту

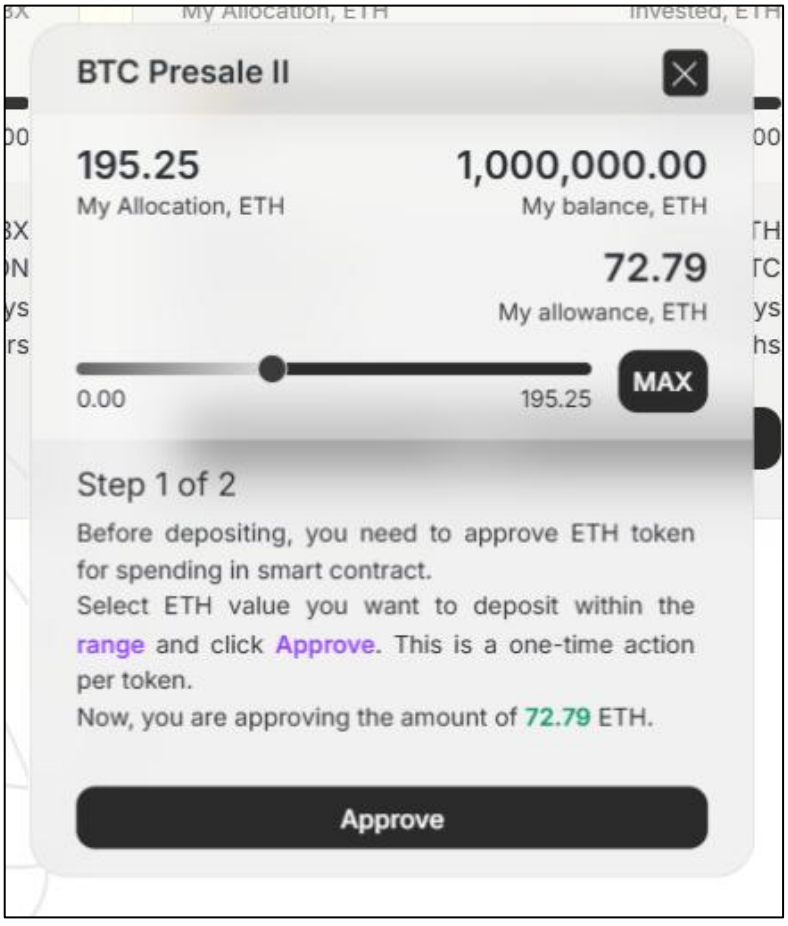

Рис. Б.8. Вікно вибору бажаної суми депозиту

Продовження додатку Б

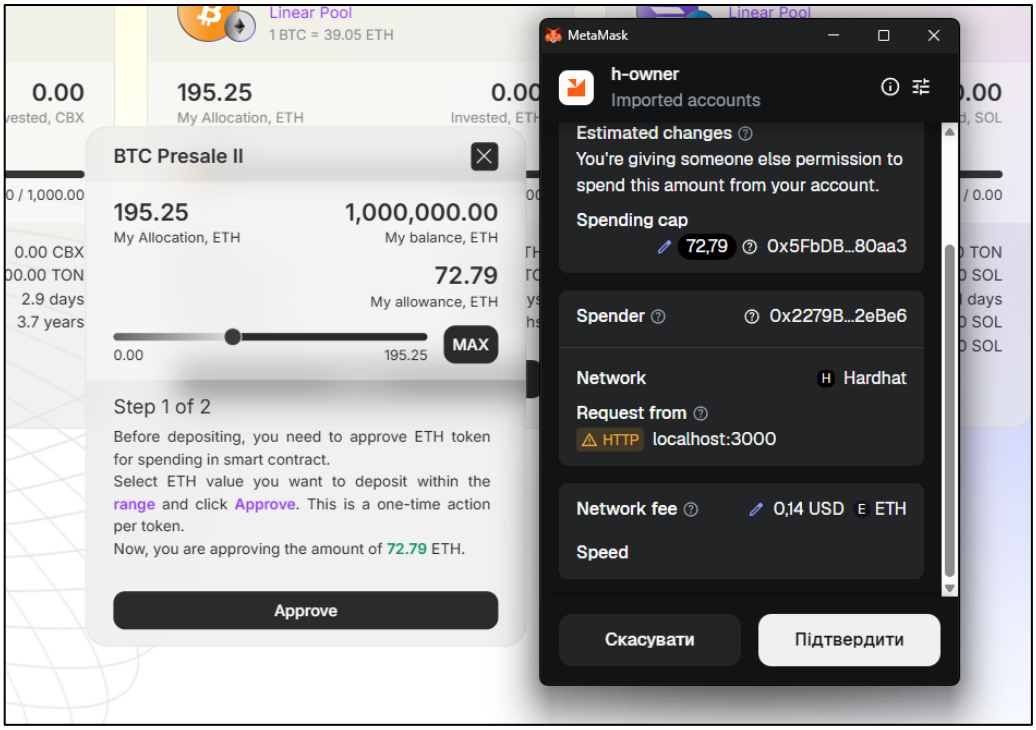

Рис. Б.9. Вікно Metamask для підтвердження approve tx

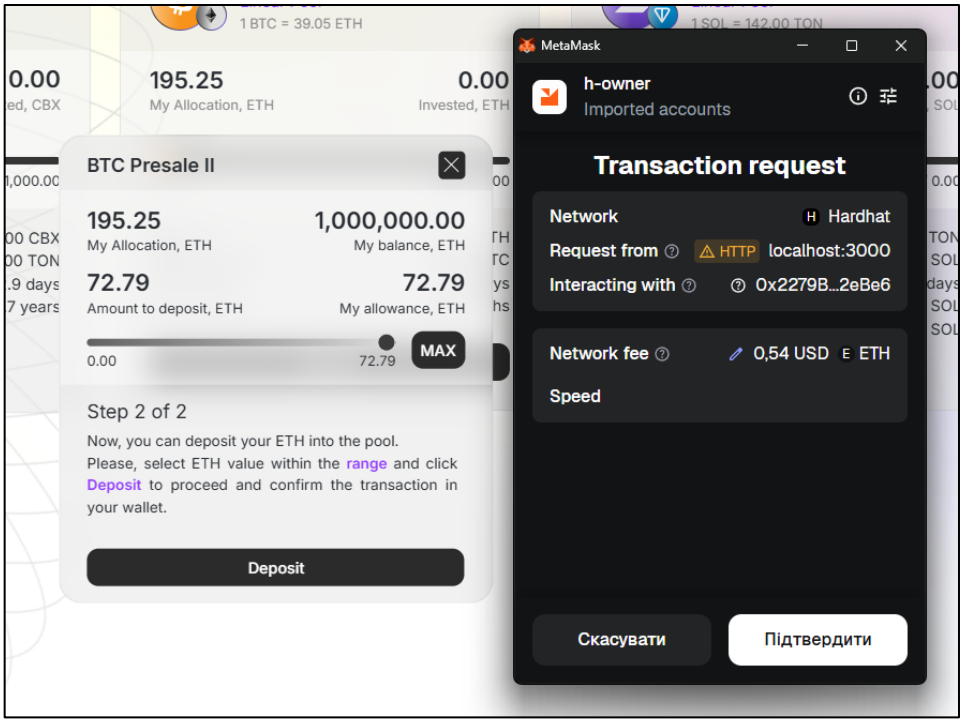

Рис. Б.10. Вікно внесення депозиту та вікно підтвердження транзакції депозиту в Metamask

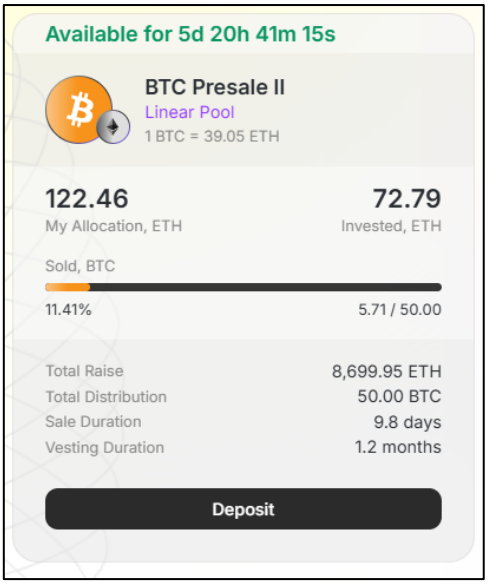

Рис. Б.11. Вікно торгового пула після внесення депозиту

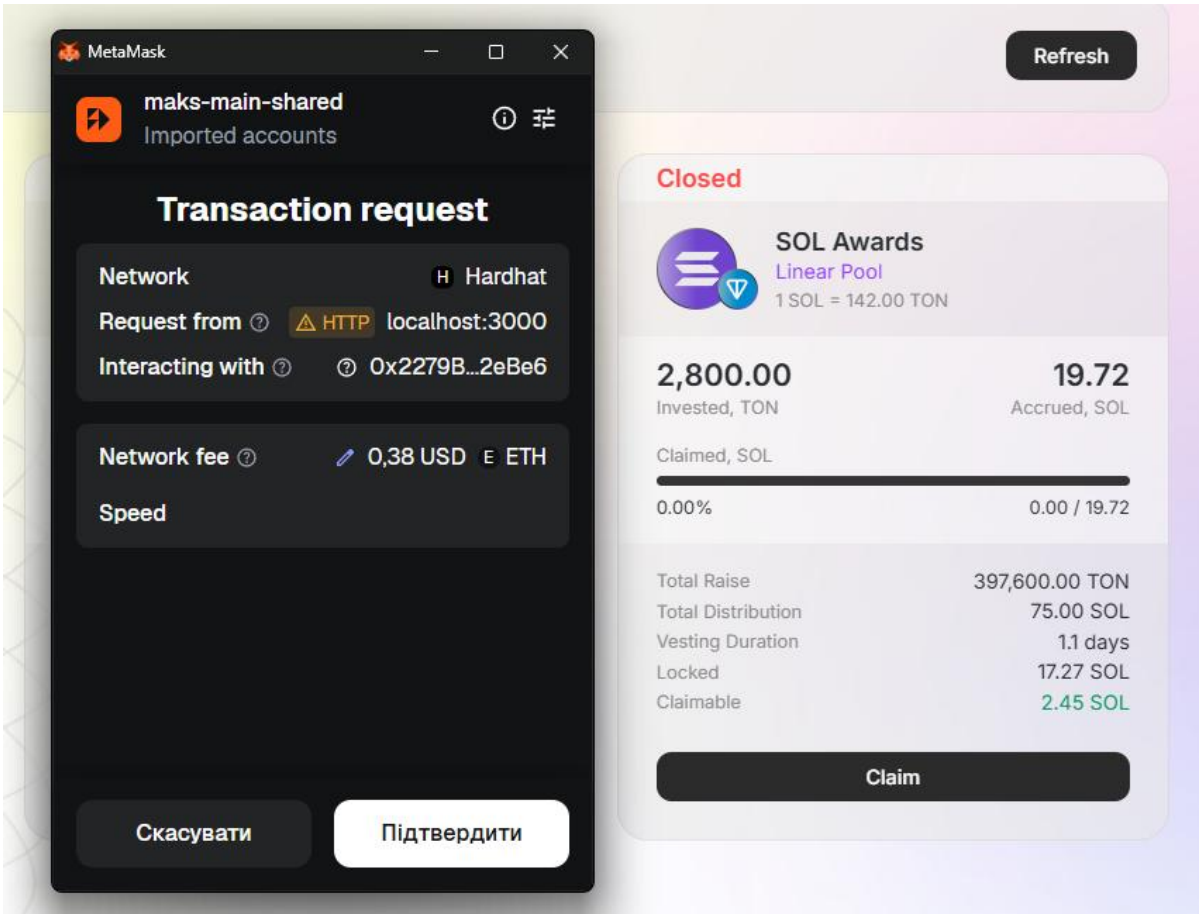

Рис. Б.12. Вікно торгового пула, що закінчився і з якого можна отримати нагороди

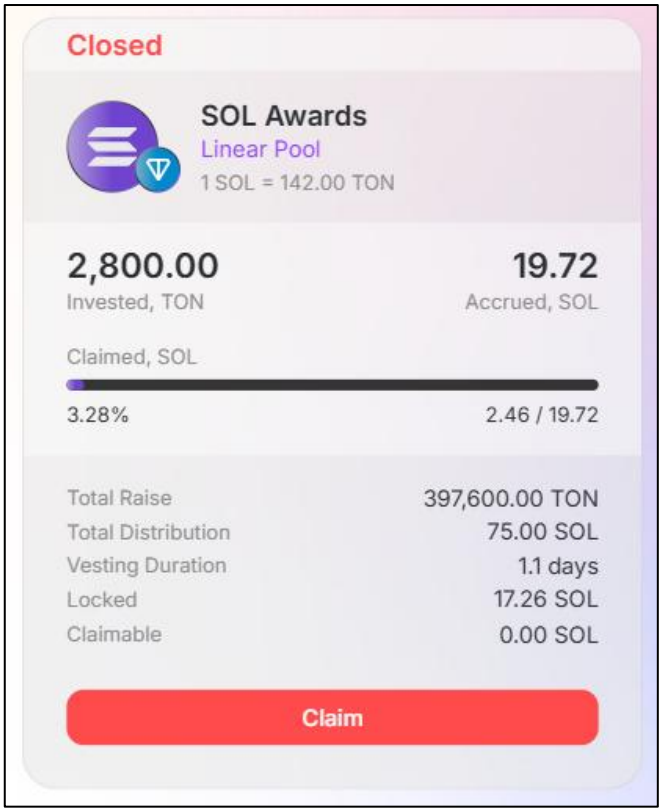

Рис. Б.13. Вікно торгового пула після зняття нагород, оновлення стану

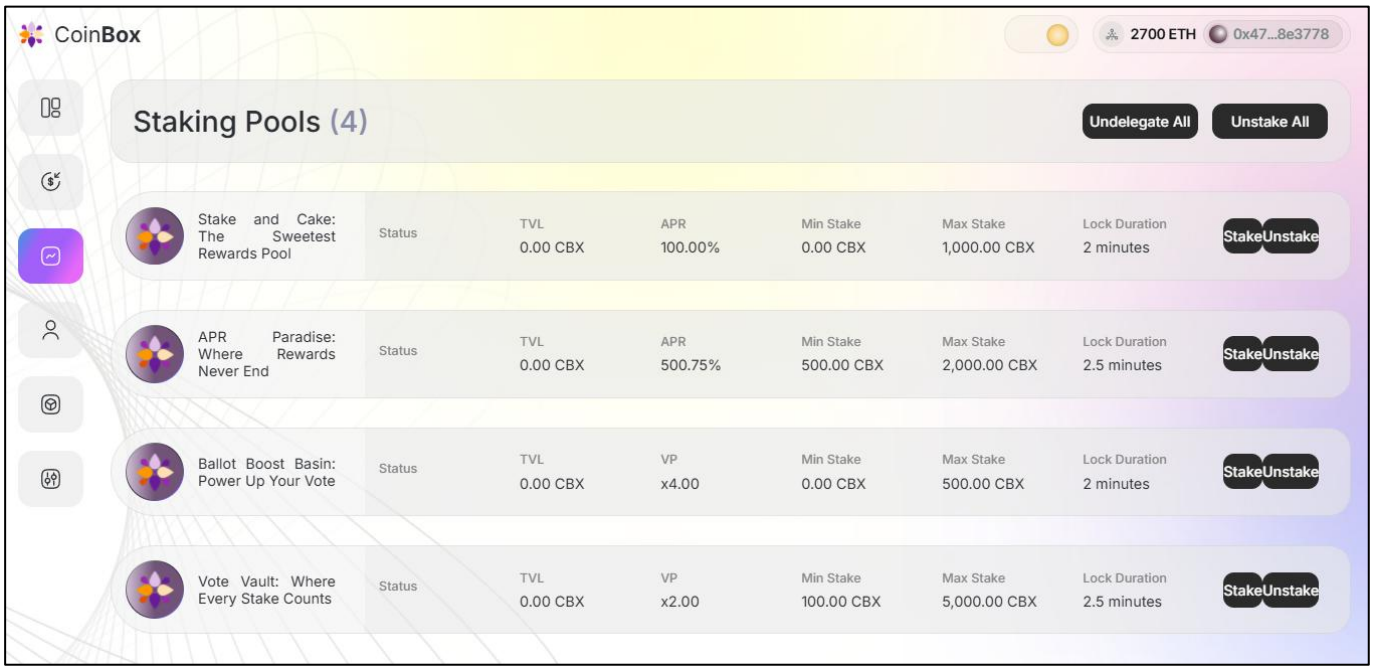

Рис. Б.14. Сторінка Staking