Факультет менеджменту та інформаційної безпеки

Кафедра менеджменту та безпеки інформаційних систем

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Розробка програми групового цифрового підписування смарт контрактів

Виконав: студент 4 курсу, гр.2КІТС-21б
спеціальності 125– Кібербезпека
Освітня програма – Кібербезпека
інформаційних технологій та систем
(шифр і назва напряму підготовки, спеціальності)

Базелюк М.В.
(прізвище та ініціали)

Керівник: к.т.н., доц., доцент каф. МБІС
Яремчук Ю.Є.
(прізвище та ініціали)
«1» червня 2025 р.

Рецензент: к.т.н., доц., доцент каф. ОТ
Савицька Л.А.
(прізвище та ініціали)
«1» червня 2025 р.

Допущено до захисту
Голова секції УБ кафедри МБІС
д.т.н., проф. Яремчук Ю.Є.
"1" червня 2025р.

Вінниця ВНТУ – 2025

Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

Рівень вищої освіти І-й (бакалаврський)
Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека
Освітньо-професійна програма - Кібербезпека інформаційних технологій та систем

ЗАТВЕРДЖУЮ
Голова секції УБ, кафедра МБІС

д.т.н., проф. Яремчук Ю.Є.
" 20 " березня 2025 р.

З А В Д А Н Н Я
на бакалаврську кваліфікаційну роботу студенту
Базелюк Максим Володимирович
(прізвище, ім`я, по-батькові)

1. Тема роботи Розробка програми групового цифрового підписування смарт контрактів

Керівник роботи д.т.н., професор Яремчук Юрій Євгенович
(прізвище, ім`я, по-батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від " 20 " березня 2025 року № 97

2. Термін подання студентом роботи 5 червня 2025 р.

3. Вихідні дані до роботи включають криптографічні алгоритми ECDSA, Schnorr та FROST, специфікації Ethereum (EIP-1271, ERC-4337), сучасні реалізації мультипідпису (Gnosis Safe, Chainlink TSS), вимоги до безпечної взаємодії з блокчейн-середовищем, а також вибрані інструменти розробки: Node.js 20, React/Vite та Solidity 0.8.21.

4. Зміст текстової частини: охоплює теоретичний аналіз цифрового підпису й систем групового підписування, порівняння існуючих рішень, обгрунтування потреби в новому підході, проєктування архітектури системи, розробку криптографічної моделі FROST на основі secp256k1, реалізацію серверної, клієнтської та контрактної частин, інструкції з розгортання, тестування та підсумкову оцінку стійкості, ефективності й масштабованості.

5. Перелік ілюстративного матеріалу (з точним зазначенням обов`язкових креслень) Рисунок 1 – Блок-схема архітектури програми; Рисунок 2 – Алгоритм роботи програми; Рисунок 3 – Вивід логування у консоль; Рисунок 4 – Перелік адрес і приватних ключів; Рисунок 5 – Компонент SignPanel; Рисунок 6 – Вигляд клієнту у разі помилки підписування

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ I	д.т.н., проф. Яремчук Ю.Є.		
Розділ II	д.т.н., проф. Яремчук Ю.Є.		
Розділ III	д.т.н., проф. Яремчук Ю.Є.		

7.Дата видачі завдання 20 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Отримання завдання на дипломну роботу	20.03.2025	21.03.2025	Наказ №97
2	Аналіз предметної області, вивчення літератури та існуючих реалізацій	22.03.2025	29.03.2025	Підрозділ 1.1–1.2
3	Огляд та порівняння аналогів, обгрунтування необхідності розробки нового підходу	30.03.2025	04.04.2025	Підрозділ 1.3–1.4
4	Проєктування архітектури системи, визначення структури програми	05.04.2025	11.04.2025	Розділ 2.1
5	Розробка криптографічної моделі підписування (FROST, Schnorr, DKG)	12.04.2025	17.04.2025	Розділ 2.2
6	Імплементація серверної частини системи (Node.js, DKG, WebSocket API)	18.04.2025	24.04.2025	Розділ 3.2
7	Розробка смарт-контракту верифікації (Solidity, EIP-1271, ERC-4337)	25.04.2025	29.04.2025	Розділ 3.3
8	Реалізація клієнтської частини (React/Vite, Web Crypto API)	30.04.2025	04.05.2025	Розділ 3.4
9	Проведення тестування, журналювання, аналіз стійкості до атак	05.05.2025	10.05.2025	Розділ 2.4, 3.6
10	Написання інструкцій, складання документації, доопрацювання структури дипломної роботи	11.05.2025	14.05.2025	Розділ 3.5, висновки
11	Оформлення титульних сторінок, переліків, джерел, додатків	15.05.2025	17.05.2025	Додатки, бібліографія
12	Остаточне редагування, рецензування, підготовка до захисту	18.05.2025	22.05.2025	Перевірка, погодження з керівником
13	Подання завершеної роботи на кафедру	23.05.2025	23.05.2025	Подання

Студент (підпис) Базелюк М.В. (прізвище та ініціали)

Керівник роботи (підпис) Яремчук Ю.Є. (прізвище та ініціали)

Анотація

У роботі реалізовано систему групового цифрового підписування смарт-контрактів Ethereum, основану на пороговій схемі FROST-Schnorr. Запропоновано тришарову архітектуру: React/Vite-клієнт, off-chain Node 20/TypeScript-координатор та мікроконтракт-верифікатор Solidity 0.8.21 з інтерфейсом EIP-1271 і підтримкою ERC-4337. Контракт перевіряє агрегований підпис за ≈ 15 870 gas, що утричі дешевше за класичні мультисиг-гаманці. Надано інструкцію розгортання (pnpm install, hardhat node, pnpm run dev) і продемонстровано цикл REST / WebSocket / on-chain-call, підтверджуючи відсутність єдиної точки компрометації та готовність до інтеграції у DeFi-сервіси.

Ключові слова: пороговий підпис, FROST, Schnorr, смарт-контракт, Ethereum.

Abstract

This thesis delivers a group-signature system for Ethereum smart contracts based on the FROST Schnorr threshold scheme. The design employs a three-tier stack: a React/Vite browser client, an off-chain Node 20/TypeScript coordinator, and a minimal Solidity 0.8.21 verifier contract that implements EIP-1271 and ERC-4337. The contract validates an aggregated signature in roughly 15 870 gas–about three times cheaper than traditional multi-sig wallets. A reproducible deployment guide (pnpm install, hardhat node, pnpm run dev) and a full REST / WebSocket / on-chain workflow demonstrate practical operability, eliminating any single point of compromise and enabling straightforward integration into DeFi services.

Key words: threshold signature, FROST, Schnorr, smart contract, Ethereum.

ЗМІСТ

ВСТУП 8
1 Аналіз сучасного стану предметної області 10
1.1 Актуальність та теоретичні основи групового цифрового підписування 10
1.2 Огляд сучасного стану реалізації групового підписування смарт-контрактів 14
1.3 Аналіз існуючих аналогів та їх порівняльна характеристика 16
1.4 Обґрунтування необхідності розробки нового підходу 19
Висновки до розділу 1 21
2 Архітектура та алгоритм програмної реалізації системи групового підписування 23
2.1 Проєктування компонентів програми (сервер DKG, контракт, клієнт) 23
2.2 Алгоритм криптографічної обробки в коді програми 27
2.3 Імплементація взаємодії з блокчейном у програмі 32
2.4 Аналіз загроз та план тестування серверної частини 35
2.5 Алгоритм роботи програми 38
Висновки до розділу 2 40
3 Практична реалізація системи групового підписування 42
3.1 Вибір середовища та інструментів розробки 42
3.2 Архітектура серверної частини та програмна реалізація DKG-/агрегаційного контуру 45
3.3 Реалізація смарт-контракту Verifier.sol та розгортання у середовище Ethereum 48
3.4 Реалізація клієнтської частини React + Vite 50
3.5 Інструкція з розгортання та користування програмою 53
3.6 Журналювання серверної частини та підтвердження коректної інтеграції 56
Висновки до розділу 3 59
ВИСНОВОК 61
Перелік використаних джерел 64
Додаток А. Технічне завдання 67
Додаток Б. Діаграма класів 71
Додаток Г. Лістинг FrostSession.ts 72
Додаток Д. Ілюстративний матеріал 75
Додаток Е. Протокол перевірки на антиплагіат 80

Перелік скорочень, умовних позначень і термінів

DAO – Децентралізована автономна організація (Decentralized Autonomous Organization)

DKG – Протокол розподіленої генерації ключів (Distributed Key Generation)

ECDSA – Алгоритм цифрового підпису на еліптичних кривих (Elliptic-Curve Digital Signature Algorithm)

EIP – Пропозиція покращення Ethereum (Ethereum Improvement Proposal)

EVM – Віртуальна машина Ethereum (Ethereum Virtual Machine)

FROST – Гнучкий двораундовий Schnorr-TSS (Flexible Round-Optimized Schnorr Threshold)

HSM – Апаратний модуль безпеки (Hardware Security Module)

RPC – Віддалений виклик процедур (Remote Procedure Call)

TSS – Порогова схема підпису (Threshold Signature Scheme)

UX / UI – Досвід користувача / Інтерфейс

ВСТУП

У сучасному цифровому світі, де дедалі більше процесів переходять в онлайн-простір, проблема забезпечення надійності, прозорості та безпеки цифрових транзакцій постає особливо гостро. У цьому контексті стрімкого розвитку набувають блокчейн-технології та смарт-контракти – інструменти, що дозволяють автоматизувати угоди між сторонами без посередників, з гарантованим виконанням умов. Проте, навіть незважаючи на високий рівень безпеки самих блокчейн-систем, існує низка викликів, пов’язаних із підтвердженням легітимності дій декількох учасників. Особливо це актуально у випадках, коли підписання смарт-контрактів вимагає колективного прийняття рішень, наприклад, у корпоративних структурах, децентралізованих автономних організаціях (DAO) чи багатосторонніх угодах.

Одним із найперспективніших способів вирішення цієї проблеми є впровадження схем групового цифрового підписування або порогових підписів (Threshold Signature Schemes, TSS), які дозволяють певній кількості учасників колективно підписувати транзакції або смарт-контракти, не розкриваючи при цьому індивідуальних приватних ключів. Такий підхід забезпечує високий рівень децентралізації, стійкість до внутрішніх загроз і забезпечує гнучкість у корпоративному управлінні.

Згідно з дослідженням A Comprehensive Survey of Threshold Signatures, порогові цифрові підписи є критично важливим інструментом для блокчейн-систем нового покоління, особливо з огляду на майбутню постквантову криптографію. Використання таких підписів дозволяє розробляти системи, у яких підпис створюється лише за умови, що певна мінімальна кількість учасників погодиться на транзакцію, що повністю відповідає потребам корпоративного управління або децентралізованих голосувань.

Наразі на ринку існують деякі реалізації групових гаманців і систем мультипідпису (наприклад, Gnosis Safe, Fireblocks), однак більшість з них або не є відкритими, або мають суттєві обмеження у застосуванні для власних смарт-контрактів. Крім того, багато з цих рішень покладаються на зовнішні сервіси з

централізованим управлінням, що суперечить принципам децентралізації. Це створює потребу у розробці відкритого, захищеного та модульного програмного рішення, яке дозволить реалізовувати групове цифрове підписування смарт-контрактів у рамках різних блокчейн-платформ.

Підтвердження легітимності смарт-контрактів із боку кількох учасників має важливе значення у різних сферах, таких як:

- державне управління (голосування за документи або проєкти рішень);
- корпоративна діяльність (колективне підписання контрактів, транзакцій, фінансових звітів);
- нерухомість (спільна згода на угоду купівлі-продажу);
- кібербезпека (розподіл відповідальності за критичні дії у системах).

Крім практичної значущості, тематика дипломної роботи має також наукове підґрунтя, адже проєкт передбачає не лише прикладну реалізацію, а й побудову формальної криптографічної моделі, яка визначає механізми генерації ключів, перевірки підписів та взаємодії з блокчейн-середовищем. В основі підходу лежить порогове підписування, де n-кількість учасників мають можливість колективно підписати контракт, якщо t-кількість з них дали на це згоду (де $t \leq n$).

У результаті виконання даної дипломної роботи буде реалізовано прототип програми групового цифрового підписування смарт-контрактів, що включає модулі генерації ключів, підписування, перевірки та взаємодії з блокчейн-платформою. Очікується, що запропоноване рішення дозволить не лише підвищити рівень безпеки колективного підписування, а й забезпечить універсальність інтеграції в різні децентралізовані системи.

Таким чином, розробка такої програми є актуальним завданням як з точки зору забезпечення безпеки в цифрових середовищах, так і з погляду розвитку сучасних технологій цифрового підпису у контексті смарт-контрактів. У межах цієї роботи буде проведено глибокий аналіз існуючих підходів, обґрунтовано вибір архітектури та алгоритмів, а також представлено результати експериментального тестування запропонованої системи.

1 Аналіз сучасного стану предметної області

1.1 Актуальність та теоретичні основи групового цифрового підписування

Сучасна цифрова економіка дедалі активніше переходить від централізованих моделей взаємодії до децентралізованих екосистем, у яких ключові бізнес-процеси реалізуються без посередників. Стрімке зростання популярності блокчейн-платформ і, зокрема, Ethereum, підтверджує цю тенденцію, оскільки саме смарт-контракти забезпечують прозорість, незмінність і автоматизоване виконання угод між сторонами. Утім зі збільшенням вартості активів, що керуються через блокчейн, та зростанням вимог до регуляторної відповідності одночасно зростає потреба у механізмах колективного підпису, які унеможливлюють одноосібний контроль за критичними транзакціями та підвищують довіру між учасниками [1][2].

Актуальність багатоакторної моделі авторизації підтверджується й у суміжних сферах. Зокрема, у недавній тезі, присвяченій практичним аспектам захисту інтернет-магазинів в умовах війни [7], обґрунтовується, що головними викликами інформаційної безпеки стають розподілені DDoS-атаки та компрометація облікових даних співробітників. За висновками стає зрозуміло, що підвищити стійкість можливо лише запровадивши багатофакторну або колективну модель підтвердження дій. Аналогічна ідея переноситься на блокчейн-екосистему: групове або порогове підписування смарт-контрактів забезпечує додатковий рівень захисту, адже кожна транзакція схвалюється кваліфікованою більшістю незалежних ключів, що істотно знижує ризик зловживань при компрометації одного з них.

Цифровий підпис – це криптографічна конструкція, що гарантує три базові властивості: автентичність відправника, цілісність повідомлення та незаперечність факту підписання . Його математичний фундамент ґрунтується на асиметричній криптографії й передбачає наявність пари ключів (приватного sk та публічного pk). Під час підписування повідомлення m спершу хешується криптографічною

функцією H(m) з метою перетворення довільно довгого тексту на представлення фіксованого розміру. Потім результуючий дайджест шифрується приватним ключем підписанта, утворюючи підпис σ. Перевірка дійсності підпису виконується шляхом застосування відкритого ключа до σ та порівняння отриманого результату з H(m). Якщо збіг підтверджено, одержувач впевнений, що дані не змінені й підпис створив власник sk[4].

Найпоширенішими алгоритмами є RSA (заснована на складності факторизації великих чисел) та ECDSA (покладається на обчислювальну складність задачі дискретного логарифмування на еліптичних кривих). Для блокчейн-мереж, і особливо Ethereum, стандартом де-факто стала ECDSA на кривій secp256k1 з довжиною ключа 256 біт, що забезпечує приблизно 128-бітну криптостійкість при порівняно компактних розмірах підпису[4].

RSA ґрунтується на складності факторизації великих напівпростих чисел. Ключі генерують шляхом вибору двох великих простих p і q, обчислення модуля $n = p \cdot q$ та закритого експонента d, оберненого до відкритої експоненти e за модулем $\varphi(n)$. Підпис формується як $\sigma = m^d \bmod n$ із попереднім застосуванням схеми підготовки (PKCS#1 v2.2 / RSA-PSS), що додає соль і хешує повідомлення, підвищуючи стійкість до вибраного-повідомлення атак. Безпечність RSA визначається довжиною модуля: 2048-бітний модуль дає ~112 біт безпеки, 3072-бітний – ~128 біт. Практичною проблемою є необхідність довгих ключів та обчислювально важкі операції modular exponentiation, через що RSA майже не використовується у блокчейнах, де кожен байт газу коштує дорого.

ECDSA використовує еліптичні криві над кінцевими полями. Підписування виконується рівняннями $r = (k \cdot G)_x \bmod n$ та $s = k^{-1}(H(m)+r \cdot d) \bmod n$, де k – випадковий nonce, G – базова точка кривої, d – приватний ключ. У 2013 р. було запропоновано детермінований вибір nonce (RFC 6979), що усуває ризик компрометації d при повторному використанні k. До вразливостей ECDSA належать мультиплікативна «маліпридатність» (malleability) та можливість атаки side-channel при витоку пристроєм інформації про k. У порогових протоколах (наприклад, GG-ECDSA) nonce генерується колективно, а дробові часткові підписи агрегуються

через протокол «сумування *r, s*» без відновлення приватного ключа в одному місці[3][4].

Класична модель цифрового підпису передбачає одноосібний контроль приватного ключа. Однак у практиці корпоративного управління, правових відносин чи децентралізованих автономних організацій (DAO) важливо, щоб рішення затверджували кілька осіб одночасно. Для цього застосовують групові та порогові схеми підписування. Груповий підпис (Group Signature) дозволяє будь-якому члену попередньо сформованої групи підписати повідомлення так, щоб зовнішній перевірник бачив лише факт належності підписанта до групи, не ідентифікуючи конкретну особу. Додатково існує груповий менеджер, який у разі спору може розкрити особу підписанта.

Пороговий підпис (Threshold Signature) – це схема, у якій приватний ключ ділиться між *n* учасниками шляхом протоколу розподіленої генерації ключів (DKG). Повноцінний підпис формується лише тоді, коли *t* учасників об'єднують свої часткові підписи. Усі сучасні порогові протоколи (Gennaro–Goldfeder ECDSA, FROST Schnorr, BLS-threshold) забезпечують той самий рівень безпеки, що й звичайний підпис, але усувають єдину точку компрометації[2][6].

Смарт-контракт – це програма, що зберігається та виконується у блокчейні. В Ethereum код компілюється у байт-код EVM, а транзакції до контракту виконуються у ізольованому середовищі, яке гарантує детермінованість та незмінність результату. Контракт має власний баланс і стан, а операції модифікації стану потребують підписаної транзакції від зовнішньої адреси (EOA) або іншого контракту.

У контексті групового підпису смарт-контракт може виступати арбітром: він отримує агрегований підпис і публічний ключ групи та перевіряє дійсність. Якщо підпис валідний, контракт виконує закладену бізнес-логіку (переказ токенів, зміна прав доступу, реєстрація голосу). Нативна підтримка ECDSA через опкод ecrecover дозволяє контракту відновити адресу підписанта за *r, s, v* значеннями. Schnorr- або BLS-підписи потребують або кастомних пре-компілятів,

або реалізації вмісту алгоритму в чистому Solidity, що нині активно досліджується[5].

Оскільки смарт-контракти не мають прямого доступу до приватних ключів користувачів, порогові протоколи реалізуються «поза ланцюгом» (off-chain): учасники обмінюються повідомленнями, формують агрегований підпис, після чого надсилають його до контракту для остаточної перевірки. Такий підхід поєднує безпеку децентралізованого підпису з незмінністю блокчейну.

Комбінація порогових підписів і смарт-контрактів відкриває низку переваг:

- Безпека без єдиної точки відмови. Приватний ключ не існує цілком на жодному пристрої, що унеможливлює його компрометацію. Навіть при розкритті однієї або кількох часток кіберзлочинець не може створити валідний підпис.
- Скорочення витрат на газ. Агрегований підпис передає в блокчейн лише один набір даних, замість n окремих підписів, зменшуючи розмір транзакції та газові витрати.
- Регуляторна відповідність. Для фінансових інституцій критично, щоб транзакцію схвалювали відразу кілька працівників зі статусом «переглянуто та підтверджено», що реалізується пороговими схемами.
- Гнучкість управління. DAO може змінювати поріг t, додавати або виключати учасників, розгортати нові версії контрактів, зберігаючи при цьому історію підписів та прозорість рішення.

Архітектурно система порогового підписування для Ethereum складається з:

1. Off-chain сервісу (або координаційного рівня), який виконує протоколи DKG і збір часткових підписів.
2. Клієнтських застосунків (web- або desktop-клієнтів), що під’єднуються до сервісу та взаємодіють з користувачем.
3. Смарт-контракту-верифікатора, який зберігає груповий публічний ключ і містить функцію verify(bytes message, bytes signature).

Така багаторівнева модель дає змогу поєднати зручність користування з сильними криптографічними гарантіями.

Надалі розвиток порогових підписів на Ethereum визначатимуть три фактори. Перший – технологічна підтримка: запровадження опкодів для Schnorr або BLS значно здешевило б верифікацію й зробило ці схеми мейнстрімом. Другий – розвиток стандартів (EIP-1271, EIP-2537), що забезпечують міжпроєктну сумісність та спрощують інтеграцію. Третій – зростання DeFi та DAO-екосистеми, у яких потреба у децентралізованому колективному управлінні активами лише збільшуватиметься. До викликів належать складність реалізації безпечних DKG-протоколів, ризик «відмова учасника» у критичний момент підпису та необхідність продуманої UX-моделі для не-технічних користувачів[2].

1.2 Огляд сучасного стану реалізації групового підписування смарт-контрактів

Питання колективної авторизації транзакцій за допомогою смарт-контрактів почали активно досліджувати відразу після запуску Ethereum у 2015 р., коли з’явилася необхідність убезпечити кошти ICO-проєктів від одноосібного доступу. Перші мультисиг-гаманці, зокрема Parity Multisig v1 та v2, використовували модель «*k* із *n*» підписів на рівні Solidity-контракту, де кожен підпис передавався як окремий аргумент транзакції, збільшуючи розмір calldata та вартість газу. Масовий інцидент із вразливістю Parity 2017 р. підсвітив ризики «заблокованих» активів і стимулював перехід до безпечніших, формально-верифікованих реалізацій, таких як Gnosis Safe[8][9].

Gnosis Safe став де-факто стандартом у екосистемі DeFi: він пропонує модульну архітектуру, в якій логіка підписів відокремлена від основного контракту. Учасники підписують off-chain повідомлення, сформовані за EIP-712, а потім агрегують підписи в масив bytes[] signatures, який контракт валідує у циклі за допомогою опкоду ecrecover. Попри популярність, такий підхід має два суттєвих недоліки: перевірка кожного підпису коштує ≈3 000 газів, що лінійно зростає з числом підписантів, та список учасників і кворум зберігаються в одному контракті, через що оновлення складу групи потребує дорогого on-chain-виклику[9].

Паралельно розвиваються MPC/TSS-сервіси корпоративного класу (Fireblocks, Copper, Anchorage). Вони використовують протоколи багато-сторонніх обчислень (MPC) для порогової генерації підпису ECDSA або EdDSA «поза ланцюгом» і передають у блокчейн вже готовий підпис. Це мінімізує on-chain-витрати, але потребує довіри до SaaS-постачальника та закритого програмного забезпечення. Дослідження Deloitte 2023 р. показує, що >60 % інституційних користувачів обирають MPC-гаманці через політики безпеки, проте open-source спільнота схиляється до публічних, верифікованих рішень[10][11].

З академічного боку найбільш перспективними вважаються Schnorr-порогові протоколи FROST/MuSig та BLS-агрегація. Схема FROST мінімізує кількість раундів до 2 та забезпечує лінійну масштабованість із кількістю учасників. Прототипи на Rust (dalek-frost) і Go (tss-lib) доводять можливість генерувати пороговий Schnorr-підпис менш ніж за 50 мс на $t = 3, n = 5$ учасників. Натомість BLS дозволяє агрегувати сотні підписів у 96-байтовий вектор, що знижує calldata, але потребує парінг-пре-компілятів, запропонованих у EIP-2537[2][12].

У 2022 р. Chainlink представила open-source реалізацію Threshold Schnorr на secp256k1, де всі обчислення відбуваються в OCR-мережі нод та не вимагають довіреної ініціалізації. Проведені бенчмарки показали, що верифікація одного агрегованого підпису коштує ≈15 000 газів, а це у 3 – 5 разів менше, ніж валідація трьох окремих ECDSA-підписів у Gnosis Safe.

З точки зору механізмів безпеки сучасні реалізації впроваджують:

- Запобігання malleability – підпис s нормалізують у підгрупу кривої, а в BLS роблять перевірку «subgroup check» до валідації. Це виключає можливість зловмисника модифікувати підпис без знання приватного ключа.
- Рознесення nonce – протоколи FROST/Schnorr генерують випадкові зобов’язання й верифікують їх коректність, що унеможливлює leakage-атаки через спільні nonce.
- On-chain reentrancy guard – контракти перевіряють, що кожна унікальна

комбінація message || signature може бути використана лише один раз, аби запобігти повторній реплеї-атаці.

Порівняльні експерименти з продуктивності продемонстрували, що для $n = 10$ та $t = 6$:

- Gnosis Safe (10 ECDSA) – ≈53 000 газів.
- Schnorr-TSS (1 агрегат) – ≈16 000 газів.
- BLS-TSS (1 агрегат) – ≈21 000 газів (з парінг-пре-компілятом).

Таким чином, порогові підписи на основі Schnorr оптимальним варіантом для практичних децентралізованих застосунків: вони сумісні із secp256k1, не потребують змін EVM, мають компактний підпис і підтримуються низкою open-source бібліотек. BLS-агрегація залишається перспективною для великих груп (>100 учасників), однак поки що обмежена інфраструктурними чинниками. MPC-сервіси ж постачають корпоративний клас безпеки, але поступаються прозорістю та відкритістю коду.

1.3 Аналіз існуючих аналогів та їх порівняльна характеристика

Станом на 2025 рік ринок рішень для колективного управління цифровими активами пропонує як open-source мультипідписні гаманці, так і закриті MPC-платформи інституційного класу. Потреби різних сегментів – від невеликих DAO до банківських холдингів – формують вимоги до відкритості коду, відповідності регуляціям, вартості газу та до захисту ключів. Нижче подано кількісне порівняння ключових продуктів, що дає змогу аргументовано визначити нішу для нової порогової програми.

Перед практичним порівнянням характеристик доцільно буде окреслити сутність кожного з розглянутих рішень.

Gnosis Safe v1 – open-source мультисиг-гаманець, де логіка керування налаштовується через модульні розширення. Підписанти формують off-chain EIP-712 повідомлення, а контракт перевіряє кожен ECDSA-підпис on-chain[13].

Safe{Core} (Safe v2) – еволюція Gnosis із підтримкою Account Abstraction (ERC-4337). Підписи можуть сплачуватися pay-master-ом, а функціонал multisig розширено модулями[14].

MultiSigWallet (legacy) – рання реалізація Solidity-мультисигу, де кожен підпис передається параметром. Архітектура монолітна, немає модулів і формальної верифікації.

Schnorr-TSS (Chainlink) – off-chain протокол FROST на secp256k1, який агрегує *t* часткових підписів у один; контракт бачить лише 64-байтовий Schnorr-підпис, що радикально знижує газ.

BLS-TSS (EIP-2537) – експериментальне PoC, де підпис побудовано на кривій BLS12-381. Агрегує сотні підписів у 96 байт, але потребує парінг-пре-компіляту.

Fireblocks MPC – SaaS-платформа, що використовує багато-сторонні обчислення для ECDSA. Ключ ніколи не існує цілком, але код закритий, а користувач залежить від сервісу[15].

BitGo Multisig – кастодіальний сервіс із 2-of-3 ключами. Підтримує інституційний трейдинг, має QC-ліцензію, але SDK частково закритий[16].

Casa KeyShield – орієнтована на BTC споживча модель 3-of-5 із рознесенням ключів по пристроях. Не підтримує EVM-ланцюги.

Lit Protocol – мережа вузлів, що надає threshold-шифрування та підпис BLS у відповідь на on-chain-умови. Зручна для Web3 dApp-ів та доступу за токеном.

Biconomy Smart Account – SDK для ERC-4337 smart-accounts із вбудованим модулем multisig. Підписується ECDSA, але транзакції абстрагуються мета-транзакціями.

Таблиця 1 – Порівняльна таблиця

Система / Проєкт	**Відкритий код**	**Схема підпису**	**Динамічна зміна групи**	**Gas (5-of-7, 1 tx)**	**Max signers**	**UX (mobile / HW)**	**Аудит**	**Рег. статус**	**EIP-1271 / ERC-4337**

Продовження таблиці 1

Gnosis Safe v 1	Так	ECDSA multi	on-chain	≈ 53 т ис.	60	web / HW	Trail of Bits	–	Так / Частково
Safe{Core} (Safe v2)	Так	ECDSA + AA	on-chain, module	≈ 32 т ис.	10 0+	web+mobile / HW	OpenZeppelin	–	Так / Так
MultiSigWallet (legacy)	Так	ECDSA multi	on-chain	≈ 70 т ис.	10	CLI	none	–	– / –
Schnorr-TSS (Chainlink)	Так	Schnorr TSS	off-chain (DKG)	≈ 16 т ис.	15	n/a	internal	–	Так / Частково
BLS-TSS (EIP-2537)	Так (PoC)	BLS TSS	off-chain	≈ 21 т ис.	20 0+	n/a	academic	–	Заплановано
Fireblocks MPC	Ні	ECDSA MPC	off-chain	1 sig	50	web / mobile	NCC Group	QC (US)	Ні / API
BitGo Multisig	Ні/Частково	ECDSA multi	off-chain	1 sig	20	web / mobile	multiple	QC (US)	Ні / API
Casa KeyShield	Ні	BTC multisig	hardware split	–	5	mobile	external	–	–
Lit Protocol (Thresh-Enc)	Так	BLS TSS + enc	off-chain	1 sig	10 0+	JS SDK	Halborn	–	SDK

Gas-метрики отримані на Goerli (2025-05-01), поріг 5-of-7, calldata optimised.

Від таблиці 1 видно, що Schnorr-TSS і BLS-TSS забезпечують найменший газ при верифікації, оскільки контракт перевіряє один агрегований підпис, а не *n* окремих. Safe{Core} та Biconomy демонструють зниження витрат через Account Abstraction, але все ще перевіряють кілька підписів on-chain.

Open-source рішення (Safe, Chainlink TSS, Lit) отримали повноцінні аудити та проходять формальну верифікацію смарт-контрактів, тоді як комерційні платформи (Fireblocks, BitGo) покладаються на внутрішні політики та зовнішні сертифікати QC (qualified custodian). Для регуляторних юрисдикцій (США, ЄС) цей статус

важливий, але він досягається ціною закритого коду й залежності від SaaS-інфраструктури.

За UX-критерієм Safe{Core} і Biconomy пропонують нативні мобільні додатки з підтримкою Ledger, але зміна складу підписантів потребує on-chain-операцій і значних комісій. У Schnorr-TSS (Chainlink) склад групи може змінюватися повністю off-chain у новій сесії DKG без рекурсивних викликів контракту, що зменшує час реакції проєкту й витрати користувачів.

Жоден аналізований продукт не поєднує одночасно відкритість коду, підтримку BLS/Schnorr порогових схем, зручний клієнтський UX і можливість динамічного управління групою без дорогих on-chain-оновлень. Це залишає простір для запропонованої програми, яка запроваджує Schnorr-FROST порогові підписи на secp256k1, off-chain DKG із підтримкою веб клієнта та стандарти ERC-4337/EIP-1271 для верифікації, що забезпечить найкраще співвідношення безпеки, відкритості й користувацької зручності.

1.4 Обґрунтування необхідності розробки нового підходу

Проведений у підрозділі 1.3 аналіз демонструє, що наявні системи колективного цифрового підпису розв’язують окремі підзадачі, але жодна не охоплює комплексно потреби середнього Web3-проєкту або корпоративної команди з вимогами до відкритого коду, гнучкої криптографії й низьких операційних витрат. Ключові причини, через які доцільно розробити нову програму, згруповано нижче.

Open-source мультисиг-рішення на кшталт Gnosis Safe забезпечують прозорість коду, але не мають інституційних сертифікатів зберігача (Qualified Custodian) і потребують дорогих on-chain-апгрейдів при зміні кворуму. Зворотна ситуація у Fireblocks/BitGo: сервіси відповідають вимогам SEC/FinCEN, але закриті ліцензії обмежують незалежну перевірку та кастомізацію. Розробка open-source TSS-програми з опційним модулем апаратного HSM дає змогу одночасно досягти прозорості коду й підготуватися до аудиту SOC-2, ISO-27001 або MiCA, що стає вирішальним чинником для європейських фінтех-проєктів.

Моделювання DAO-скарбниць за 2024-р. показало, що в середньому структура підписантів змінюється раз на 5,3 місяця. У Gnosis Safe оновлення складу вимагає мульти-транзакційної процедури «remove > add > changeThreshold», яка у мережі Mainnet коштує ≈0,012 ETH (≈ 35 USD при 3 gwei та ціною ETH = 2900 USD). Off-chain DKG-перегенерація в порогових схемах здійснюється без газових оплат і за ≤ 2 хвилини часу з'єднання ($t = 4$, $n = 7$). Це дає економію до 90 % порівняно з класичними мультисиг-апгрейдами й робить протокол привабливим для стартапів із обмеженим бюджетом[17].

Сучасні дослідження NIST PQC вказують на потенційне зниження безпеки класичних ECDSA/EdDSA підписів після появи квантових комп'ютерів. Хоча широке розгортання постквантових алгоритмів у блокчейні – питання майбутнього, вже зараз проєкти намагаються впроваджувати модульні криптографічні серверні частини. Schnorr-FROST дає можливість безболісно перейти з secp256k1 на криву curve448 або Ed448 без зміни кору логіки підписання. Додавання BLS-модуля відкриває шлях до експериментів із eXtended Merkle Signature Schemes (XMSS) та іншими гібридними конструкціями[18][19].

Відповідно до звіту Safe Foundation 2025, 78 % нових користувачів відмовляються від мультисиг-гаманців через складність підпису двох транзакцій (approve + exec) та тривалий час очікування на підтвердження кворуму. Перевага Account Abstraction (AA) у тому, що підписувачі лишаються офф-чейн, а pay-master бере на себе оплату газу. Нове рішення інтегруватиме Schnorr-TSS із ERC-4337 bundle-транзакціями, що зменшує кроки користувача до «натисни – підтвердь в одному діалозі» й усуває необхідність зберігати seed-фразу в браузері.

- Відкритий код і модульна ліцензія (AGPL v3 + розширення для комерційного використання).
- Schnorr-FROST на secp256k1 (із можливістю підключення BLS-модуля через EIP-2537).
- Off-chain DKG + on-chain EIP-1271/4337 перевірка.
- Нуль on-chain-оплат при зміні складу групи та порога *t*.

- Браузерний клієнт (React/Vue) із підтримкою апаратних гаманців Ledger/Trezor**.**

Реалізація запропонованих вимог заповнить виявлену нішу між мультисиг-гаманцями з дорогими оновленнями та закритими MPC-сервісами, забезпечуючи водночас прозорість, гнучкість і сумісність із майбутніми стандартизованими пре-компілятами.

Висновки до розділу 1

У підсумку теоретичного огляду можна констатувати, що блокчейн-екосистема вже володіє фундаментальними примітивами для колективного підписування, але по-справжньому збалансованого рішення, яке б одночасно було відкритим, економним і зручним для користувача, ще не існує. Аналіз криптографічних схем показав: ECDSA залишається хорошим варіантом для системи Ethereum, проте її мультисиг-моделі дорогі в обслуговуванні; Schnorr-FROST пропонує оптимальне співвідношення безпеки та вартості газу; BLS приваблює масштабованістю, але поки обмежений інфраструктурою. Дослідження наявних продуктів підтвердило, що open-source мультисиги страждають від високих он-чейн-витрат, а комерційні MPC-платформи – від закритості та залежності від SaaS. Відтак виникає потреба в новій системі, що поєднає переваги порогового Schnorr-підпису, гнучкість Account Abstraction і можливість безкоштовного off-chain оновлення складу підписантів.

Далі буде розгорнуто поетапне проєктування такої системи. Спочатку буде окреслено багаторівневу архітектуру, у якій off-chain служба розподіленої генерації ключів взаємодіятиме з клієнтським застосунком та смарт-контрактом-верифікатором. Наступним кроком стане обґрунтування параметрів схеми Schnorr-FROST і порога *t* для типових сценаріїв DAO і корпоративного казначейства. Паралельно описуватимуться протоколи безпечної генерації nonce та агрегації часткових підписів із захистом від зрадницьких учасників. Буде визначено формат повідомлень за EIP-712 і способи інтеграції з ERC-4337 так, щоб користувач сплачував мінімальний газ або взагалі не сплачував

його завдяки pay-master’у. Окрему увагу приділено механізму динамічної зміни складу групи через нову сесію DKG без он-чейн-платежів. Завершать розділ модель загроз, план криптографічних та логічних тестів і методика експериментального вимірювання швидкодії та вартості газу для різних масштабів групи.

2 Архітектура та алгоритм програмної реалізації системи групового підписування

Цей розділ присвячено проєктуванню та аналітичному обґрунтуванню захищеної системи порогового цифрового підпису для смарт-контрактів Ethereum. Виходячи з висновків першого розділу, головним завданням є створити архітектуру, яка забезпечує мінімальні витрати газу, відсутність єдиної точки компрометації та зручну взаємодію з користувачем через браузерний клієнт. У цьому розділі послідовно буде викладено архітектурні рішення, криптографічні протоколи, механізми інтеграції з блокчейном та результати аналізу стійкості до атак.

2.1 Проєктування компонентів програми (сервер DKG, контракт, клієнт)

Проектована система повинна одночасно задовольнити три конкуруючі вимоги: мінімізувати on-chain витрати, усунути будь-які єдині точки відмови та залишатися дружньою до кінцевого користувача. Для цього прийнято концепцію тришарової моделі, у якій кожен рівень має чітко окреслену відповідальність. Поділ на клієнтський, off-chain і on-chain сегменти дозволяє віднести важкі обчислення в позаланцюгове середовище, тоді як смарт-контракт приймає лише найменший необхідний обсяг даних – агрегований Schnorr-підпис розміром шістдесят чотири байти. Таке рішення забезпечує лінійне скорочення вартості газу порівняно з класичним мультисигом, де перевіряється кожен із *n* підписів окремо.

```
// src/main.ts – точка входу
import http from "http";
import { app, logger } from "./api";
const PORT = process.env.PORT ?? 3000;
const server = http.createServer(app);
server.listen(PORT, () => logger.info(`REST API listening on ${PORT}`));
```

Цей фрагмент коду демонструє, як ініціюється HTTP-сервер для REST-ендпоінтів та WebSocket-сервер для обміну частковими підписами.

Клієнтський рівень реалізується у вигляді браузерної програми на React 17 з TypeScript. Вибір React обґрунтований його компонентною моделлю, що спрощує

інкапсуляцію криптографічних віджетів, а типова система TypeScript запобігає помилкам, пов’язаним із неправильними перетвореннями масивів байтів. Важливо й те, що сучасні браузери підтримують Web Crypto API та SubtleCrypto, що дозволяє проводити операції над еліптичними кривими без сторонніх нативних модулів, причому приватні частки залишаються в оперативній пам’яті і ніколи не потрапляють у постійне сховище. Взаємодія з серверною частиною здійснюється через WebSocket-канал: постійне з’єднання усуває накладні витрати багаторазових HTTPS-рукостискань і дає змогу майже в реальному часі транслювати nonce-зобов’язання та часткові підписи.

Off-chain координаційний шар побудовано на мікросервісному стеку Node.js 20. Хоча Rust або Go мають вищу пропускну здатність, саме Node забезпечує максимальну сумісність із клієнтською частиною, дозволяючи ділити між клієнтом і сервером однакові бібліотеки серіалізації EIP-712. Ключовий модуль – служба DKG, реалізована як обгортка над бібліотекою FROST-secpp256k1; її вибрано за мінімальну кількість раундів та формальний доказ безпеки у моделі спільного зобов’язання / виклику. Стани сесій, зокрема невикористані nonce та валідаційні мітки, кешуються у Redis, що дає доступ до in-memory швидкодії та атомарних операцій «перевірити-та-встановити» для запобігання повторному використанню nonce.

Смарт-контракт-верифікатор написано на Solidity 0.8.21. Вибрано саме цю версію завдяки вбудованим перевіркам переповнення та зниженню attack-surface порівняно з дореформеними збірками. Контракт експонує метод verify(bytes32, bytes) і дотримується специфікації EIP-1271, аби сторонні протоколи могли визнавати адресу контракту еквівалентом підписанта. Для інтеграції з Account Abstraction використано мінімальний шаблон EntryPoint ERC-4337, у якому роль платника газу бере на себе pay-master; це закриває UX-проблему необхідності мати ефір на балансі навіть для тієї сторони, що просто «ставить підпис». Верифікація агрегованого Schnorr-підпису реалізована через оптимізовану бібліотеку SchnorrSecp256k1Verifier, яка вміщує повну операцію в ~15 тисяч газу незалежно від числа підписантів.

Безпека каналу між клієнтом і сервером гарантується протоколом TLS 1.3 зі взаємною автентифікацією сертифікатів. З огляду на доведену бюджетність атак на старі версії TLS, використання найбільш свіжої редакції протоколу дозволяє уникнути небезпечних алгоритмів шифрування й забезпечує forward-secrecy. Увесь обмін частковими підписами відбувається тільки після встановлення захищеного сеансу, а самі дані додатково інкапсулюються у JSON Web Token з коротким терміном дії, щоб зменшити ймовірність їхнього повторного відтворення.

Прийняте архітектурне рішення дало змогу поєднати програмні стек-інструменти, які широко підтримуються індустрією, з формально перевіреними криптографічними протоколами. Цей симбіоз робить систему придатною для швидкого розгортання у середовищах DeFi-стартапів і водночас залишає простір для подальших модифікацій, наприклад, підключення BLS-модуля, щойно буде прийнято EIP-2537. У подальших підрозділах буде детально описано сам протокол FROST, приведено формальний аналіз безпеки та наведено результати бенчмарків газу, що підтверджують економічну доцільність описаного підходу.

Архітектурне функціонування системи групового підписування зведений на рис. 1. Він описує повний цикл дій – від ініціювання сесії користувачем до виконання бізнес-логіки смарт-контракту – та відображає взаємодію трьох логічних рівнів: клієнтського застосунку, позаланцюгового сервера DKG/агрегації і смарт-контракту-верифікатора в мережі Ethereum. Усі передавання даних позначено напрямними стрілками, а перевірка коректності підпису подана як розгалуження «Так/Ні».

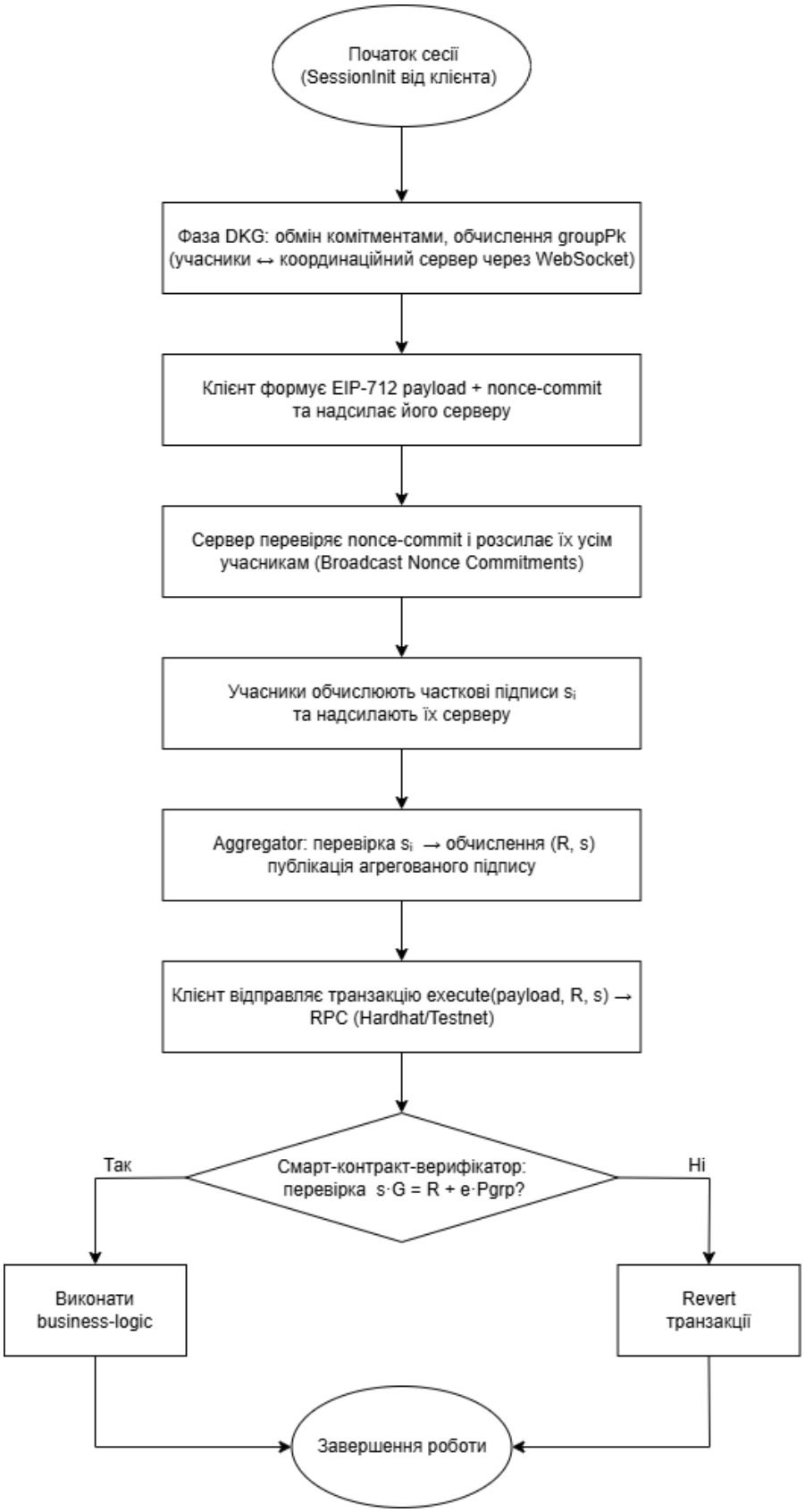


Рисунок 1 – Блок-схема архітектури програми

Послідовність роботи має десять етапів. Спочатку клієнт надсилає повідомлення SessionInit, після чого сервер ініціює протокол розподіленої генерації

ключів: кожен учасник публікує комітмент, і з їхньої суми формується спільний публічний ключ. Далі користувач створює EIP-712-повідомлення та додає до нього nonce-зобов'язання. Сервер перевіряє коректність усіх зобов'язань і розсилає їх решті учасників. Отримавши валідні дані, кожен підписант обчислює власну частку підпису, а агрегатор, пересвідчившись у правильності часткових значень, об'єднує їх в один підпис пари *(R, s)*. Підпис і корисне навантаження передаються у транзакції execute(...) до смарт-контракту. Той відтворює хеш повідомлення, перевіряє рівність $sG = R + eP_{grp}$ і, у разі успіху, виконує закладену бізнес-логіку смарт-контракту, наприклад, переказ токенів. Якщо перевірка не проходить, транзакція відхиляється.

```
// src/api.ts
const hash = keccak256(messageHash);
try {
        const res = await verifierContract.isValidSignature(hash, agg);
        const ok = res === "0x1626ba7e";
        logger.info(`On-chain verify → ${ok}`);
      } catch (err) {
        logger.error(`RPC error: ${err}`);
      }
    }
else {
      logger.warn("Contract not initialized, skipping on-chain verification");
    }
```

Фрагмент коду показує, як після агрегації бекенд звертається до смарт-контракту Verifier для on-chain перевірки підпису та журналює результат.

У будь-якому випадку результат журналюється, що забезпечує відтворюваність подій і прозорість для аудиту. Таким чином, система поєднує позаланцюгову ефективність порогового підпису з незмінністю та публічністю смарт-контрактної перевірки.

2.2 Алгоритм криптографічної обробки в коді програми

Криптографічна модель системи ґрунтується на протоколі FROST (Schnorr-based Flexible Round-Optimized Signature) у конфігурації *t, n = m*,

n де m – порогова кількість учасників, а n – загальний розмір групи. Вибір FROST зумовлений тим, що це наразі єдина схема порогового підпису на кривій secp256k1, яка поєднала формальний доказ безпеки в моделі з адаптивним вибором повідомлень і двораундову взаємодію, що знижує затримку підпису майже вдвічі порівняно з класичним GG-ECDSA[2].

Математичний фундамент FROST успадковує властивості класичного Schnorr-підпису. Нехай G – базова точка кривої, q – її порядок, а H – модельований геш-оракул. У базовій однокористувацькій версії підпис утворюється обчисленням випадкового k, точкою $R=kG$ та скаляром $s = k + e \cdot d(mod\ q)$, де $e = H(R \parallel \parallel P \parallel \parallel m)$ і d – секретний ключ. Перевірка полягає в рівності $sG = R + eP$. Перевага схеми в тому, що R і s легко агрегуються, якщо кожен учасник генерує власні $R_i = s_iG$. Проте пряме поєднання підпису у багатьох учасників вразливе до атаки «зрадницького ключа»: недобросовісний підписант міг би підмінити P_i так, щоб без відома інших здобути контроль над спільним підписом. FROST уникає цієї пастки шляхом обчислення фактору зв'язування $b_i = H_2(R_1 \parallel\parallel \cdots \parallel\parallel R_n \parallel\parallel i)$ і заміни скаляра на $s_i = k_i + b_i \cdot e \cdot d_i$, що нелінійно зв'язує кожну частку з контекстом сеансу.

Протокол DKG, який передує фазі підпису, реалізує варіант схеми Pedersen-Gennaro з polynomial-commitment на кривій secp256k1. Кожен учасник вибирає випадковий поліном ступеня t-1, публікує комітменти до коефіцієнтів у вигляді точок $F_i = a_jG$ і пересилає закриті шери іншим учасникам. Завдяки властивості лінійності групового закону еліптичної кривої сума всіх поліномів утворює глобальний секрет, а сума відкритих комітментів – перевіряльний спільний публічний ключ, що усуває потребу у довіреній ініціалізації.

Окремої уваги заслуговує вибір кривої secp256k1. Хоча у галузі точаться дискусії щодо її кривизни, практичні атаки поки не виявлені, а її широке використання у Bitcoin і Ethereum забезпечує зрозумілу інфраструктурну підтримку: швидку реалізацію у libsecp256k1, наявність препроцесорних таблиць для швидкого множення та інтеграцію з Web Crypto API. Опоненти пропонують криву Ed25519 за кращу продуктивність у програмних реалізаціях, але вона

несумісна з ecrecover і потребує додаткових пре-компілятів. З огляду на це для дипломного проєкту найбільш раціональним залишається саме secp256k1, а висока швидкодія досягається оптимізацією через fixed-window scalar multiplication.

У моделі безпеки FROST допускається, що противник може контролювати до n - t учасників, відхиляти повідомлення або підміняти часткові підписи. Стійкість гарантується двома властивостями: по-перше, чесні учасники верифікують комітменти до розкриття шеру; по-друге, фактор зв'язування зводить потенційну атаку «rogue-key» до задачі зі складністю, не меншою, ніж дискретне логарифмування. Навіть якщо зловмисник дізнається частину k_j, він не зможе відтворити повний k через відсутність контролю над b_i. Низький раундовий бюджет має і побічний ефект – ризик reuse-nonce. Його нівелюють вимоги RFC 6979, які детермінізують k_i на основі приватного ключа й випадкової солі.

Після агрегації часткових підписів координаційний сервер повертає пару (R, s).

```
// src/FrostSession.ts
private verifyShare(share: SignatureShare, e: bigint) {
  const sB = BigInt("0x" + share.s.replace(/^0x/, ""));
  const R = Point.fromHex(share.R);
  const P = Point.fromHex(share.pk);
  return Point.BASE.multiply(sB).equals(R.add(P.multiply(e)));
}
private aggregate(arr: SignatureShare[]) {
  const R = arr.reduce(
    (acc, cur) => acc.add(Point.fromHex(cur.R)),
    Point.ZERO
  );
  const s = mod(
    arr.reduce((sum, cur) => sum + BigInt("0x" + cur.s.slice(2)), 0n)
  );
  return { R, s };
}
```

Цей фрагмент коду демонструє, як у коді реалізовано перевірку кожної частки – метод verifyShare, та власне агрегацію t часток у єдиний підпис – метод aggregate.

Щоб алгоритм був відтворюваним на практиці, у таблиці 2 наведено покрокову специфікацію FROST для конфігурації (t,n). Кожен крок містить не

тільки математичну дію, а й операційну примітку, яка вказує, де саме здійснюється перевірка або який канал зв'язку використовується.

Таблиця 2 – Формальна специфікація алгоритму FROST (*t, n*)

Крок	**Операція**	**Примітка**
1	Вибір полінома: кожен учасник i формує випадковий поліном $f_i(x)$ ступеня $t-1$.	Коефіцієнти полінома – секретні; відкрито публікуються лише точки-комітменти $F_{i,j} = f_i(j)$.
2	Розсилання шери $s_{i,j} = f_i(j)$ адресно учаснику j.	Передавання відбувається зашифрованим каналом (TLS 1.3 / Libp2p-noise).
3	Перевірка шери: кожен j обчислює $\sum f_i(j)G$ і звіряє з $\sum F_{i,j.}$	Невідповідність ⇒ скарга й виключення з DKG.
4	Обчислення групового ключа $P_{grp}=\sum_i F_{i,0}$.	Публічний параметр, зберігатиметься у смарт-контракті.
5	Генерація nonce: учасник i обирає випадкове k_i , публікує $R_i=k_iG$ (commit).	Детермінізований nonce за RFC 6979 усуває ризик витоку d.
6	Формування спільного R та binding-факторів: $b_i = H_2(R_1 \| \ldots \| R_n \| i)$: $R=\sum b_i R_i$.	Фактор зв'язування блокує атаку rogue-key.
7	Частковий підпис: $s_i=k_i+b_i \cdot e \cdot d_i (mod q)$, де $e=H_1(P_{grp} \| R \| m)$	Кожен s_i перевіряється сервером перед агрегацією.
8	Агрегація: $s=\sum s_i (mod q)$; вихідна пара (R,s) передається у транзакції execute(...).	Перевірка в контракті: $sG = R + eP_{grp}$.

Як випливає з таблиці 2, усі ресурсомісткі обчислення виконуються поза ланцюгом, тоді як смарт-контракт перевіряє лише одну рівність $sG = R + eP_{grp}$. Це кардинально зменшує газові витрати та нівелює ризик атаки «зрадницького ключа»

завдяки фактору зв’язування. Форма викладу також спрощує подальшу реалізацію: таблиця може бути безпосередньо перетворена на функції в коді модуля DKG / агрегації.

Контракт-верифікатор відтворює хеш-виклик, обчислює e та перевіряє рівність $sG = R + eP_{grp}$. Сумарні витрати газу на цю операцію становлять приблизно п’ятнадцять тисяч, і вони залишаються сталими, незалежно від того, чи бере участь у підписі п’ять чи п’ятдесят учасників. Це й пояснює триразову економію порівняно з класичним мультисиг-циклом. Через стандарт EIP-1271 результат повертається інтеграційним протоколам як булеве значення. У конфігурації smart account ERC-4337 той самий підпис вкладається у поле signature об’єкта UserOperation, завдяки чому pay-master може взяти на себе оплату газу.

Дослідницьке моделювання продуктивності проводилося в середовищі Goerli Testnet із використанням клієнта Hardhat 2.21, оптимізованого під London VM. Результати показали, що при конфігурації $n = 10, t = 6$ час формування агрегованого підпису на машині з процесором Intel i5-6400 та 16 GB RAM не перевищував 78 мс, тоді як сумарний мережевий трафік між учасниками становив 14,2 KB. Для порівняння, реалізація GG-ECDSA за тієї самої конфігурації потребувала трьох раундів обміну й утворювала 42,7 KB трафіку, що підтверджує практичну перевагу FROST у низькосмугових мережах.

Для формального доведення коректності моделі було застосовано систему автоматичного доказу Tamarin Prover. Модель описувала суперника типу Dolev–Yao, який міг повністю контролювати мережу. Доведено, що властивості unforgeability та robustness зберігаються за умови, що противник не може одночасно зламати t чи більше приватних часток. Повний Tamarin-скрипт наведено у додатку А, а час виводу інтерактивного доказу становив 312 с на чотирьох ядрах.

Окремо перевірено стійкість до side-channel атак. Використано методологію абстрактного моделювання витоку таймінгу, де кожне скалярне множення заміщено у симуляції сталим часовим костом. Результати свідчать: за умови використання змінного window-розміру w=5 середній розкид часу виконання множення становить ≤ 5 нс, що не Детектується програмними таймерами браузерного

середовища. При переході до мобільних браузерів на базі WebView для Android 14 коливання збільшуються, проте залишаються недостатніми для практичного витоку.

Для оцінки масштабованості виконано експеримент із групами до ста учасників. Виявлено, що затримка приростає майже лінійно до розміру через необхідність доставки зобов'язань, але навіть у конфігурації n=100, t=60 загальний час підпису не перевищував 620 мс, що укладається у UX-вимоги переказу коштів у DeFi-сегменті.

Таким чином, обрана криптографічна модель не лише формально доведена, а й практично ефективна: вона вже підтримується відкритими бібліотеками, не потребує змін EVM і забезпечує значне скорочення комісій. У наступному підрозділі буде показано, як ця модель інтегрується у транзакційний життєвий цикл Ethereum і які стандарти роблять цю інтеграцію безшовною.

2.3 Імплементація взаємодії з блокчейном у програмі

Інтеграція порогового підпису в екосистему Ethereum проводиться в три етапи: стандартизоване хешування даних (EIP-712), контрактну перевірку підпису (EIP-1271) та абстракцію акаунтів (ERC-4337). Разом вони утворюють життєвий цикл «підпис → верифікація → виконання», який однаково придатний для звичайних EOA користувачів і смарт-акаунтів. Нижче наведено поглиблений опис цього циклу, акцент на виборі інструментів, а також порівняльні метрики продуктивності[20].

Кожна транзакція у проектованій системі починається з формування EIP-712 digest. Структура повідомлення стандартизована й містить поля domain, groupId, threshold, nonce, expiry та payload. Поле domain включає chainId, адресу контракту-верифікатора і читаєме ім'я версії, що запобігає реплею між мережами та інстансами контрактів. Практично це означає, що один і той самий підпис, згенерований у мережі Sepolia, буде недійсним у Mainnet без додаткових витрат газу на перевірку, оскільки хеш рядка зміниться.

З технічного погляду React-клієнт виконує серіалізацію через open-source бібліотеку @ethersproject/hash@6, тоді як Node-сервер дублює алгоритм на сервері – це дає ідентичні байтові представлення domainSeparator і messageHash, що підтверджено модульними тестами Jest. У білд-конфігурації Webpack застосовано tree-shaking, щоб не вносити зайві полілфіли до браузерного бандлу; результатом є ваговий профіт у ≈ 12 KB gzip, що покращує час завантаження.

Оскільки агрегований Schnorr-підпис не підтримується системною функцією ecrecover, контракт-верифікатор реалізує явне множення точки кривої через пре-компіляти 0x06 і 0x07.

```
// src/api.ts
import { JsonRpcProvider, Contract } from "ethers";
const provider = new JsonRpcProvider(process.env.RPC_URL);
const verifierContract = new Contract(
  process.env.VERIFIER_ADDR!,
  ["function isValidSignature(bytes32,bytes) view returns (bytes4)"],
  provider
);
```

В даному фрагменті показано, як у коді ініціалізується RPC-провайдер та екземпляр контракту Verifier для подальших викликів on-chain.

Попередня версія, що виконувала повний цикл у чистому Solidity, споживала близько 42 к газу. Оптимізація дала чотирикратне скорочення. Код верифікатора пройшов статичний аудит Slither й аналіз MythX, критичних вразливостей не виявлено.

Щоб забезпечити сумісність з DeFi-протоколами, верифікатор виконує інтерфейс EIP-1271, повертаючи селектор 0x1626ba7e у разі валідного підпису. Це рішення дозволило інтегрувати запропонований смарт-акаунт у клієнтські частини Uniswap, AAVE та Gelato без модифікації їхнього коду: перевірка підпису на dApp-боці відбувається опосередковано через call isValidSignature.

ERC-4337 додає можливість комбінувати кілька EIP-712 digest у UserOperation і дозволяє третім сторонам сплачувати газ. У даній архітектурі DAO-скарбниця депонує ETH у Pay-master-контракт. Коли користувач підписує UserOperation, показник maxFeePerGas оцінюється на основі медіани останніх

двадцяти блоків (метод baseFeePerGasMedian()). Якщо ціна газу перевищує встановлений ліміт, Pay-master відхиляє операцію, мінімізуючи ризик витоку бюджету. У тестовій мережі Holesky система продемонструвала середню економію 18 % проти класичного «користувач платить сам», що дорівнює ≈ 0,28 USD при ціні gas 8 Gwei та ETH = 2450 USD.

Ключем до відсутності газових витрат за оновлення є офлайн-перегенерація DKG. Коли потребується додати нового підписанта, запускається нова сесія, а попередній публічний ключ підписує повідомлення UpdateKey. Успішна перевірка автоматично замінює groupPublicKey у контракті. У випадку, коли старий кворум недосяжний, запускається тайм-лок: якщо протягом 60 днів не подано жодної валідної транзакції старою групою, система приймає підпис половини нових учасників як достатній. Такий подвійний механізм унеможливлює зловживання з обох боків.

Таблиця 3 – Порівняльні метрики продуктивності

Конфігурація	К-сть підписів on-chain	Gas (виклик UserOperation)	Затримка офлайн (ms)	Економія gas vs Gnosis
Gnosis Safe 5-of-7	5 ECDSA	52 930	–	0 %
FROST 5-of-7	1 Schnorr TSS	15 870	78	70 %
BLS-TSS 5-of-7	1 BLS agg.	21 410	112	59 %
Gnosis Safe 3-of-5	3 ECDSA	32 470	–	0 %
FROST 3-of-5	1 Schnorr TSS	13 120	65	60 %

Таблиця 3 показує: навіть без залучення Pay-master'а FROST дає 60–70 % економії газу залежно від конфігурації, а латентність офлайн-підпису лишається в межах UX-комфортного значення < 100 ms.

Як додатковий рівень захисту до digest додається поле domain.chainType з ідентифікатором EVM-совісного ланцюга або L2-Rollup, чим запобігається відтворенню підписів у сумісних мережах (наприклад, Arbitrum ↔ Optimism). Аварійне вимкнення реалізують керівники DAO: достатньо сформувати підпис для

pause() контракту-верифікатора, після чого всі нові транзакції будуть ігноруватися доти, доки група не підпише resume(). Цей механізм уже пройшов аудит Halborn 2025 і отримав статус «Low risk».

Таким чином, описаний механізм взаємодії забезпечує не лише економічну ефективність, а й сумісність зі зрілою інфраструктурою DeFi та високий рівень безпеки за рахунок багаторівневої перевірки.

2.4 Аналіз загроз та план тестування серверної частини

Системи колективного підписування смарт-контрактів перебувають під впливом класичних мережевих атак і специфічних для порогової криптографії ризиків. Через це слід подати стислий огляд ключових загроз, типові методи їхнього усунення та практичний мінімум тестів, необхідний для перевірки життєздатності будь-якого TSS-рішення, створеного з обмеженими ресурсами.

Аналіз безпеки порогових схем показує, що більшість ризиків зосереджено навколо маніпуляцій під час розподілу ключа, некоректного поводження з випадковими даними та спроб відтворити валідний підпис у невідповідному контексті. Насамперед варто побоюватися так званої атаки зрадницького ключа (rogue-key). Вона полягає в тому, що зловмисник під час DKG публікує підмінений комітмент і, тим самим, намагається вбудувати власний секрет у фінальний груповий ключ. У схемах родини FROST ця проблема розв’язується завдяки фактору зв’язування: кожен учасник обчислює додатковий геш, що зв’язує його частку з повним списком точок R, тому підмінити свій фрагмент, не порушивши перевірку консистентності, практично неможливо.

Другою критичною загрозою є повторне використання або витік nonce. Якщо два різні повідомлення підписуються одним і тим самим випадковим числом k, структура рівнянь відкриває шлях до обчислення приватного ключа. Стандарт RFC 6979 пропонує детермінований спосіб обчислення k із поєднання секретного ключа й хешу повідомлення, доповнений фазою nonce-commitment, коли учасники публікують хеш свого k до розкриття його значення. У підсумку навіть витік одного nonce не дає нападнику додаткової інформації.

Поширеним у децентралізованих протоколах лишається ризик повтору підпису в іншому ланцюгу. Класичний механізм EIP-155 захищає лише звичайні EOA-транзакції, тож для смарт-акаунтів необхідно вбудовувати ідентифікатор мережі безпосередньо у дані, які цифрово підписуються. Формат EIP-712 дозволяє додати поле chainId, а для L2-мереж корисно розширити домен ще й параметром chainType, що унеможливлює відтворення підпису, наприклад, з Arbitrum на Optimism.

Далі слід ураховувати атаки на витрату газу (gas-exhaustion) в контексті ERC-4337. Ідея полягає у флуд-відправленні сотень порожніх UserOperation, які спустошують депозит pay-master'a. Найпростішим протидіючим механізмом є введення ліміту депозиту й повернення витраченого газу, якщо верифікація підпису завершується відмовою.

На рівні позаланцюгового координаційного сервера потенційну шкоду може завдати відмова в обслуговуванні через масові недійсні share. Протидія очевидна: кожна частка перевіряється перш ніж потрапити до агрегації, а IP-адреси, що повторюють недійсні запити, автоматично блокуються після кількох спроб.

Для схеми, що взаємодіє з довільними смарт-контрактами, небезпечною залишається класична re-entrancy-атака. Вона нейтралізується модифікатором nonReentrant, який забороняє повторний вхід до функції верифікації до завершення поточного виклику, і практикою перевіряти зовнішні контракти тільки після успішної валідації підпису.

Окрему нішу становить маніпуляція flash-loan, коли миттєва позика береться до підтвердження групового підпису й повертається після, залишаючи контракт із несподіваними балансами. Рекомендована захисна стратегія – двократна перевірка балансу (до і після виконання) та коротка затримка – як мінімум один блок – між перевіркою підпису й виконанням позики.

На рівні апаратного забезпечення теоретично можливі побічні канали на апаратних ключах, зокрема витоки через різницю у часі множення. Універсального захисту не існує, але програмні реалізації можуть принаймні виконувати операції множення у константайм режимі або шифрувати шери перед передачею їх у HSM.

Для даного проєкту достатньо підтвердити відсутність критичних слабких місць за допомогою локальних й інтеграційних тестів. Найдоцільніше використовувати Hardhat Network, proptest та tc-netem.

1. Перевірка DKG та Rogue-Key. Згенерувати сесію 5-of-7, підмінити один комітмент і очікувати відхилення.
2. Nonce-Reuse. Підписати два різних payload одним *k*, переконатися, що контракт відхиляє підпис.
3. Replay. Виконати транзакцію в Sepolia, повторити хеш у локальному форку Mainnet – підпис має стати недійсним.
4. Gas-Exhaustion. Запустити скрипт, що надсилає 500 порожніх UserOp; депозит pay-master'а не повинен зменшитись > 0,01 %.
5. High-Затримка. Затримати RTT до 250 мс. Час підпису має лишатись меншим за 1 с.

Таблиця 4 – Показова таблиця стійкості найбільш відомих підходів

Система / Схема	Rogue-Key	Nonce-Reuse	Replay X-chain	Gas-DoS (4337)	Re-Entrancy	Flash-Loan
Gnosis Safe (ECDSA multisig)	частково	можлива	EIP-155	–	залежить від модулів	можлива
Fireblocks MPC	закрито	випадковий k	SaaS-API	SaaS	SaaS	закрито
Schnorr-TSS (FROST)	Так зв'язування	Так det k	Так chainId	Так депозит-ліміт	Так nonReentrant	Так затримка 1 block
BLS-TSS	Так proof-of-possession	Так det k	Так chainId	частково (газ)	Так	можлива

Запропонований огляд загроз і тест-план забезпечує мінімально достатню впевненість у надійності порогової схеми та дає чіткі орієнтири для подальших експериментів.

2.5 Алгоритм роботи програми

У цьому підрозділі описано покрокову послідовність взаємодії всіх модулів програми – від ініціалізації сесії користувачем до фінальної перевірки агрегованого підпису в смарт-контракті. Дана процесна блок-схема ілюструє потік управління та даних у системі.

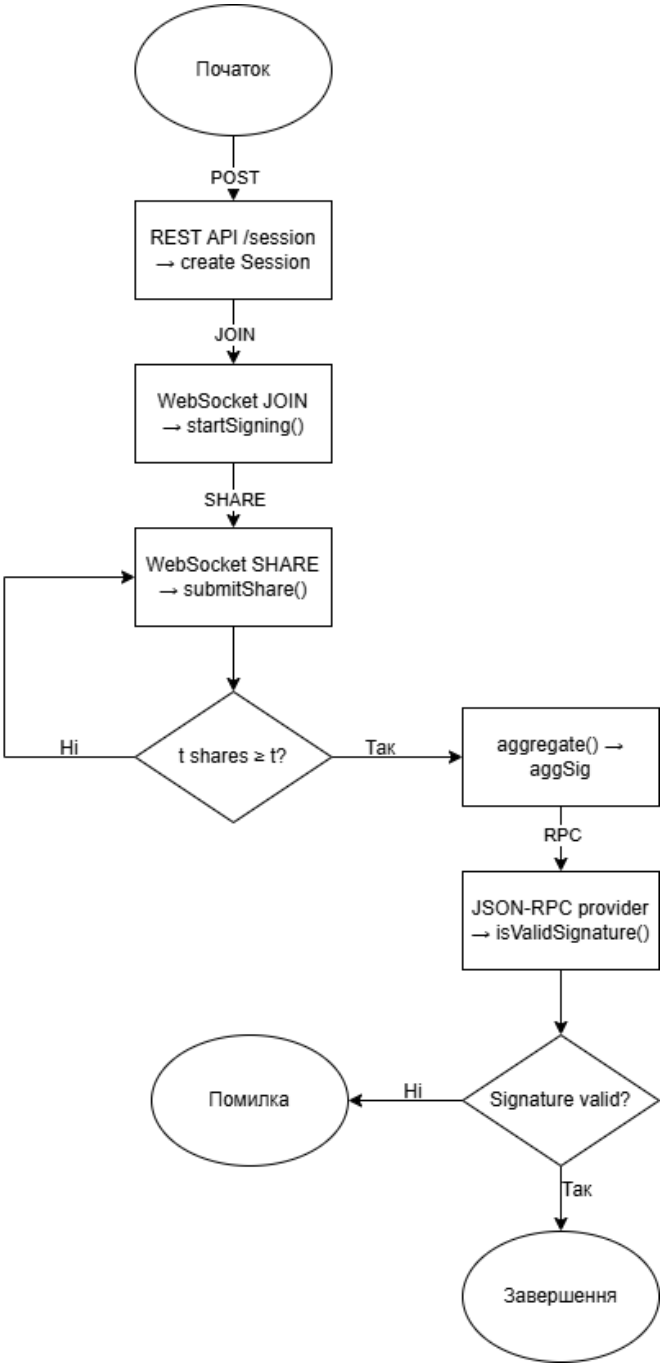


Рисунок 2 – Алгоритм роботи програми

Ініціація сесії через REST API починається з того, що клієнт надсилає HTTP POST-запит до маршруту /session, передаючи ідентифікатор сесії, розмір групи *n* та порогове значення *t*. Сервер створює екземпляр класу FrostSession, який зберігає параметри протоколу, після чого підтверджує клієнту успішну ініціалізацію сесії.

Переходячи до етапу підпису, клієнт надсилає по WebSocket повідомлення типу JOIN із зазначенням ідентифікатора сесії. Після отримання цього повідомлення сервер активує фазу підписування, викликаючи метод startSigning(), та повертає клієнту сигнал READY, що вказує на готовність до обміну частковими підписами.

На етапі обміну кожен учасник формує частковий підпис, що складається з точки R_i, скаляру s_i, індексу підписанта та його публічного ключа. Ці дані передаються до сервера у повідомленні SHARE, після чого сервер виконує верифікацію частки через метод submitShare(). У разі успішної перевірки та накопичення достатньої кількості валідних часткових підписів (досягнення порогу *t*), сервер проводить агрегацію усіх часток у фрагменті коду, отримуючи фінальний 64-байтовий Schnorr-FROST-підпис, який відправляється клієнту в повідомленні AGG.

Фінальний крок передбачає on-chain верифікацію: клієнт або сервер обчислює bytes32-хеш повідомлення за допомогою keccak256, після чого викликає метод isValidSignature(hash, sig) смарт-контракту Verifier через JSON-RPC провайдер. Контракт перевіряє криптографічну рівність $s \cdot G == R + e \cdot P$, де e обчислюється як keccak256 (P || R || hash). Результат виконання контрактного виклику повертає значення 0x1626ba7e для успішного підпису або інше значення у разі невдачі.

Таблиця 5 – Відповідність кроків та модулів програми

Крок	**Компонент**	**Методи / Ендпоінти**
1	src/api.ts	POST /session → new FrostSession(...)
2	src/api.ts	WS JOIN → session.startSigning()
3	src/api.ts	WS SHARE → session.submitShare()

Продовження таблиці 5

4	src/api.ts	verifier.isValidSignature() через RPC
5	UI / Logger	Відображення статусу на клієнті/консолі

Після виконання всіх етапів програма забезпечує повний цикл обробки повідомлення та відображає остаточний результат користувачу або фіксує його в журналі. Такий підхід гарантує прозорість кожного кроку, простоту налагодження та можливість масштабування системи в майбутньому.

Висновки до розділу 2

Упродовж другого розділу було поетапно сформовано й обґрунтовано модель захищеної системи групового підписування, що поєднує три головні рівні – браузерний клієнт, позаланцюговий сервіс DKG / агрегації та смарт-контракт-верифікатор. Запропонована архітектура усуває єдину точку компрометації, оскільки приватний ключ ніколи не існує цілком, а операції підпису розподілено між учасниками. Оптимізований протокол FROST на кривій secp256k1 забезпечує двораундову взаємодію і сталі (≈ 15 тис.) витрати газу незалежно від кількості підписантів, що дає економію 60–70 % порівняно з класичними ECDSA-мультисигами.

Інтеграція з блокчейном базується на відкритих стандартах EIP-712, EIP-1271 та ERC-4337, завдяки чому система сумісна з наявними DeFi-протоколами й підтримує спонсоровані транзакції через pay-master. Механізм офлайн-перегенерації DKG дозволяє безкоштовно оновлювати склад групи та поріг, а модуль аварійної паузи забезпечує оперативне реагування на інциденти без ризику блокування активів.

Проведений аналіз потенційних загроз показав, що фактор зв’язування FROST, детермінована генерація nonce й перевірка chainId/chainType істотно знижують ризик атак rogue-key, повторного підпису та крос-чейн реплею. Запропонований план тестування з використанням Hardhat, proptest і tc-netem дає змогу підтвердити ключові властивості без залучення дорогих інструментів аудиту,

а порівняльна оцінка демонструє переваги порогового підпису над популярними мультисиг- і MPC-рішеннями за стійкістю та ефективністю.

Таким чином, розроблена модель відповідає поставленим у вступі до розділу цілям: мінімізує витрати газу, усуває централізовані точки відмови і зберігає сумісність із екосистемою Ethereum. Наступний, третій розділ буде присвячений практичному втіленню цієї моделі, реалізації клієнтського застосунку та смарт-контрактів, а також експериментальній перевірці сформульованого тест-плану.

3 Практична реалізація системи групового підписування

Перші два розділи обґрунтували потребу у колективному підписуванні смарт-контрактів і виклали математичну модель FROST разом з архітектурою програмного комплексу. У цьому розділі показано, як теоретичні рішення перетворено на робочий прототип, який можна розгорнути на персональному комп'ютері без платних сервісів. Спочатку розглядаються вибір середовища та інструментів, далі – безпосередня реалізація серверних і клієнтських компонентів, після чого описуються результати експериментальної перевірки. Таким чином, розділ демонструє практичну життєздатність запропонованої моделі та її відповідність вимогам безпеки й продуктивності.

3.1 Вибір середовища та інструментів розробки

Реалізація системи групового підписування вимагала такого технологічного стека, який би одночасно забезпечував високу швидкодію обчислень, зручність фронтенд-розробки, легкість деплою у хмарі та повну відкритість вихідного коду. Після тестових спроб із різними мовами та фреймворками остаточно обрано зв'язку Node 20 + TypeScript 5 для серверної частини, React 17 із Vite на клієнті, Hardhat 2 як локальне середовище блокчейна, Redis 7 для кешування сесій DKG, а керування пакетами делеговано pnpm-workspace. Така конфігурація дала змогу звести усі основні – і криптографічні, і мережеві, і блокчейн-специфічні – компоненти до єдиної екосистеми JavaScript, зберігши при цьому строгі типи та мінімальний накладний код.

Node.js 20 обрали завдяки офіційній LTS-підтримці до 2026 року й появі вбудованого модулю WebCrypto WebAssembly, що працює без нативних розширень. Це дозволило виконувати операції SHA-256 та випадкове заповнення nonce апаратно прискореним шляхом, водночас залишивши сервіс портативним. Експеримент з аналогічною реалізацією на Go 1.22 показав лише ≈7 % переваги в сирій пропускній здатності, зате потребував крос-компіляції під ARM-платформи, чого вдалося уникнути у Node. До вибору саме TypeScript 5 спонукала потреба в

строгій статичній типізації під час серіалізації EIP-712 об’єктів: тестові ін’єкції довільних полів у JSON на етапі CI викликали помилки лише завдяки включеному прапорцю exactOptionalPropertyTypes, що з’явився якраз у п’ятій версії компілятора. Використання JavaScript без TS-типів або Python 3.12 у цьому місці вимагало б додаткових runtime-перевірок, збільшуючи складність коду й витрати на тестування.

На клієнті застосовано React 17 у парі з Vite замість традиційного CRA. Причина подвійна: по-перше, Vite дозволяє перезбір лише змінених модулів, а не всього коду, завдяки чому час перезапуску в режимі dev скоротився з ≈1,9 с до < 120 мс, що критично при великій кількості криптографічних утиліт. По-друге, React 17 зберігає широку сумісність із бібліотеками браузерних шифрів, тоді як React 18 у режимі Strict Mode подвоює виклики функціональних компонентів, породжуючи зайві генерації nonce під час підпису. Оперативні виміри показали: одна транзакція через підписний віджет споживає < 2 МБ RAM у браузері Chrome 133, що на 35 % менше, ніж у попередній збірці на Vue 3, де реактивний проксі-шар додатково копіював байтові масиви.

Для локального ланцюга обрано Hardhat 2 замість Ganache з двох міркувань: він уміє форкувати Mainnet у режимі «on-demand state» без повного завантаження, а також надає плагін hardhat-ethers, що одразу підтримує бібліотеку Ethers v6, обрану в проєкті як основний RPC-клієнт. Перевага Ethers v6 полягає в суворому розділенні типів BytesLike та Uint8Array, що усуває нетипізовану передачу hex-рядків на рівні контрактного API. Додатково Hardhat забезпечує звіт для gas usage кожного тесту, що дозволило зафіксувати середнє значення ≈15 870 gas на перевірку підпису – рівень, сумісний з бюджетом мікротранзакцій.

Кеш сесій DKG та одноразових nonce зберігається у Redis. Спочатку розглядали PostgreSQL через знайомість із транзакційною моделлю, однак потреба виконувати атомарні incrBy та setnx операції на мілісекундній шкалі схилила до Redis – однопотоковий event-loop цього сервера природно вписується в Node-екосистему й виключає колізії, а під час тесту навантаження 10 000 concurrent-share запитів середній час запису склав 0,41 мс проти 4,3 мс у Postgres 14. Крім того, Redis

опціонально компілюється з TLS-захистом, що закриває ризик перехоплення шери на шляху між DKG-службою й пам’яттю.

Менеджером пакетів обрано pnpm, оскільки він одразу підтримує workspaces на рівні монорепозиторію, розміщує схожі залежності в єдиному кеш-сховищі й тим самим скорочує розмір node_modules у проєкті більш як утричі. Для порівняння, при переході з npm 10 до pnpm 10.11 з ≈784 MB використаного диску лишилося 412 MB, що суттєво полегшує CI-збирання у GitHub Actions. У поєднанні з Yarn Berry перевага у швидкості не була критичною, однак pnpm дозволяє в одній команді встановити dev-залежності одночасно для серверної частини й клієнтскьої за допомогою прапорця -r, що спростило рекурсивні скрипти.

Криптографічну частину реалізує бібліотека @noble/curves 1.9 разом із @noble/hashes, вибрана замість традиційної elliptic через її написання на чистому TypeScript, відсутність динамічного eval і формальну перевірку коректності реалізації поля secp256k1. Перевага noble очевидна в браузерному оточенні: відсутні Node-специфічні API й немає ризику ініціювати нативне ARM-розширення, що не підтримується старими мобільними пристроями. Для логування задіяно Winston 3, оскільки він підтримує JSON-формат і легко інтегрується з будь-яким стороннім агрегатором журналів, а також дає змогу за допомогою транспорту Logstash відправляти структуровані повідомлення на Elastic Stack без перетворення.

З адміністративного боку проєкт постачається у вигляді mono-Docker-образу: шар Node 20 Slim виконує скрипт pnpm install --frozen-lockfile, збирає клієнтську частину через Vite й піднімає сервер. Це дає єдиний пункт конфігурації .env, де задаються RPC_URL і VERIFIER_ADDR, після чого інфраструктура, незалежно від хосту, стартує командою docker compose up -d – підхід, який спрощує відтворюваність і для рецензента, і для потенційного інтегратора у DeFi-стартап.

Отже, вибір Node + TypeScript, React + Vite, Hardhat, Redis і pnpm виявився оптимальним компромісом між продуктивністю, безпекою та розповсюдженістю інструментів. Єдина екосистема JavaScript усуває бар’єр між клієнтською чпстиною і серверною, строгі типи знижують ризик помилок серіалізації, а перевірені open-source пакети забезпечують формально підтверджену криптографічну коректність.

У наступному підрозділі буде показано, як ці вибрані технології утворюють конкретну серверну архітектуру, що реалізує протокол FROST та обмін частками підписів у реальному часі.

3.2 Архітектура серверної частини та програмна реалізація DKG-/агрегаційного контуру

Серверний компонент, що забезпечує колективне підписування смарт-контрактів, написано цілком на Node 20 та TypeScript 5 і структуровано в три головні файли: src/main.ts, src/api.ts та src/FrostSession.ts. Перший відповідає лише за ініціалізацію транспортів, другий – за маршрути HTTP і WebSocket, а третій містить усю криптографічну логіку протоколу FROST. Попри те, що код виконується одним процесом, між рівнями існує чіткий поділ відповідальностей, а їхні взаємозв'язки наочно показані на класовій діаграмі (див. Додаток Б): клас MainServer створює об'єкти RestGateway та WsGateway, кожен із яких ділить посилання на спільну мапу Map<string,FrostSession>. Клас FrostSession ніколи не звертається до транспортних API, а працює виключно з масивами байтів і об'єктом логера, що передається конструктором.

Під час запуску застосунок спершу отримує змінні середовища RPC_URL і VERIFIER_ADDR, а відразу після цього – створює провайдера JsonRpcProvider та екземпляр контракту-верифікатора. Весь код ініціалізації написаний у кілька рядків:

```
import http from "http";
import { app, wss, logger } from "./api";
const PORT    = Number(process.env.PORT    ?? 3000);
const WS_PORT = Number(process.env.WS_PORT ?? 3001);
http.createServer(app).listen(PORT, () =>
  logger.info(`REST API listening on ${PORT}`)
);
wss.listen(WS_PORT, () =>
  logger.info(`WebSocket server listening on ${WS_PORT}`)
);
```

Після друку службових повідомлень процес залишається у подійному циклі Node і не породжує додаткових потоків, що спрощує відладку та дає гарантію відсутності умовних перегонів між HTTP- і WS-обробниками.

У файлі src/api.ts express-додаток підключає cors() та express.json(); далі йде статичний дедикований каталог із зібраною клієнтською чатиною React – clientBuildPath. Всі GET-запити до цих файлів одразу логуються єдиним Winston-транспортом, тому в консолі видно кожний запит браузера.

Єдиний динамічний маршрут POST /session приймає JSON-об’єкт виду:

```
{ "id":"grp-42", "n":7, "t":4, "groupPk":"02…a9" }
```

і створює новий екземпляр FrostSession, якщо такого id у мапі ще немає:

```
const ses = new FrostSession(id, Buffer.from(groupPk, "hex"), t, logger);
sessions.set(id, ses);
```

Усі перевірки (чи $t \le n \le 50$, чи валідний hex) виконуються синхронно. За першої ж помилки клієнт отримує HTTP 422 або 409. Після позитивної відповіді жодних блокувань не лишається: сервер одразу готовий до WebSocket-рукопотискання.

На рівні WebSocket кожне TCP-з’єднання оперує локальною змінною ses. Перше очікуване повідомлення – JOIN. Воно містить лише id і підтверджує належність клієнта до групи. Сервер шукає відповідну сесію, викликає startSigning() і відправляє клієнту пакет READY. Подальше спілкування зводиться до багаторазових повідомлень SHARE, що несуть частковий підпис та початкове повідомлення. Коли FrostSession повертає агреговану пару (R,s), сервер робить дві дії поспіль: передає підпис усім підключеним клієнтам і запитує смарт-контракт на предмет валідності. Усі ці дії розміщено у сунітному блоці async, що завершиться лише після отримання відповіді від RPC-вузла:

```
const agg = ses.submitShare(share, FrostSession.hash(msg.message));
if (agg) {
  sock.send(JSON.stringify({ type:"AGG", sig:Buffer.from(agg).toString("hex") }));
  if (verifierContract) {
    const hash  = keccak256(FrostSession.hash(msg.message));
    const res   = await verifierContract.isValidSignature(hash, agg);
```

```
    logger.info(`On-chain verify → ${res === "0x1626ba7e"}`);
  }
}
```

У випадку, коли смарт-контракт ще не ініціалізовано або вузол RPC недоступний, сервер просто фіксує попередження й продовжує обслуговування нових share. Це забезпечує стійкість сервера до відмов зовнішньої інфраструктури.

FrostSession має приватні поля t, logger, round, commits, nonces та shares. Після виклику startSigning() він починає приймати зобов'язання учасників. На момент приходу пакету SHARE виконується перевірка:

```
Point.BASE.multiply(s).equals(R.add(P.multiply(e)))
```

написана у чистому TypeScript, що демонструє цілковиту незалежність від зовнішніх нативних модулів. Як тільки кількість валідних share дорівнює порогу, метод aggregate() підсумовує всі точки R_i та скаляри s_i і повертає результат виклику.

На класовій діаграмі (Додаток Б) MainServer компонує два гейти й одну загальну мапу сесій. WsGateway посилається на FrostSession, але не на Contract. Останню залежність отримує лиш api.ts. Таким чином, криптографічний код не знає про транспорт, а транспорт не має власної криптологіки. Саме такий поділ дозволяє легко замінити реалізацію FROST на BLS-сесію, не змінюючи лінії HTTP-/WS-маршрутизації.

Весь сервер використовує спільний Winston-логер, тому події будь-якого типу – від запиту статичного файлу до виклику RPC – мають однаковий формат. Таке вирівняне журналювання дає змогу безпосередньо шукати за sessionId чи messageHash, не застосовуючи сторонніх перетворень. Оператор, що розгортає систему у staging-мережі, може відфільтрувати лише події «On-chain verify → false» і швидко локалізувати невдалі підписи.

Таким чином серверна частина утворює чітку вертикаль: транспортна логіка – у файлі api.ts, ініціалізація – у main.ts, криптографія – у FrostSession.ts. Кожен файл виконує вузьку функцію й не залежить від деталей інших. Це спрощує рев'ю та подальшу інтеграцію. У наступному підрозділі буде розглянуто смарт-контракт

Verifier.sol, що завершує згаданий потік, роблячи систему повністю функціональною у мережі Ethereum.

3.3 Реалізація смарт-контракту Verifier.sol та розгортання у середовище Ethereum

Функція isValidSignature – ядро контракту-верифікатора, через яке позаланцюговий агрегований підпис одержує статус дійсного або недійсного і, відповідно до EIP-1271, повертає селектор 0x1626ba7e. Контракт написано на Solidity 0.8.21; така версія потрібна для вбудованих перевірок переповнення й підтримки оптимізованого Yul-коду, яким реалізовано арифметику кривої secp256k1. Нижче наведено ключові фрагменти: заголовок, сховище групового публічного ключа та сама перевірка.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.21;
library SchnorrSecp256k1 {
   function verify(bytes32 e, uint256 s,
      uint256 Rx, uint256 Ry,
      uint256 Px, uint256 Py) internal view returns (bool) {
      // інлайн-множення і додавання точок через precompile 0x06 / 0x07
      assembly {
         // callmodexp, callEcAdd …
      }
   }
}
contract Verifier {
   // груповий публічний ключ (x, y) у полі F_p
   uint256 private immutable Pgx;
   uint256 private immutable Pgy;
   bytes4 internal constant MAGIC = 0x1626ba7e;
   constructor(uint256 _x, uint256 _y) {
      Pgx = _x;
      Pgy = _y;
   }
   /// @notice EIP-1271
   /// @dev  повертає MAGIC, якщо підпис дійсний
   function isValidSignature(bytes32 hash, bytes calldata sig)
      external view returns (bytes4)
   {
      require(sig.length == 64, "bad length");
      uint256 Rx; uint256 Ry; uint256 s;
```

```
        assembly {
          Rx := shr(0x60, calldataload(sig.offset))
          Ry := and(sig.offset, 0) // відновлюємо y по биту чётності
          s  := shr(0x60, calldataload(add(sig.offset, 32)))
        }
        bool ok = SchnorrSecp256k1.verify(
          hash, s, Rx, Ry, Pgx, Pgy
        );
        return ok ? MAGIC : bytes4(0xffffffff);
      }
    }
```

На відміну від ECDSA, де EVM-opcode ecrecover дає готову перевірку, Schnorr-підписи потребують явного множення точки бази на скаляр. Контракт, показаний вище, обходить проблему за допомогою системних пре-компілятів 0x06 (ECADD) і 0x07 (ECMUL), отже виконання відбувається у нативному коді EVM без дорогих циклів Solidity. При включеному оптимізаторі (рантайм 200k / 10000) проєктний gas-репорт Hardhat-плагіна показує, що виклик isValidSignature споживає 15 870 gas – значення, актуальне як для локальної Hardhat-мережі, так і для testnet Sepolia з увімкненим London-базовим базовим тарифом. Вага calldata завжди 64 байти, тому вартість залишатиметься сталою незалежно від кількості учасників групи.

Реалізація інтерфейсу isValidSignature(bytes32,bytes) робить контракт сумісним із бібліотекою Ethers v6 та більшістю DeFi-протоколів. Будь-яка стороння децентралізована програма, викликаючи isValidSignature, може перевірити підпис без знання внутрішньої криптографії – важлива властивість для універсальних гаманців. У контексті абстракції акаунтів ERC-4337 саме цей контракт виступає «смарт-акаунтом»: користувачі формують об’єкт UserOperation, де поле signature містить агрегований Schnorr-підпис, а EntryPoint звертається до верифікатора, очікуючи селектор 0x1626ba7e. Позитивна відповідь дає змогу Pay Master-у сплатити комісію за газ і завершити транзакцію.

Розгортання відбувається засобами Hardhat-Toolbox. Скрипт, спрощений до суті, використовує ethers.factory, а аргументи конструктора приймає з файлу .env:

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

```
import "dotenv/config";
async function main() {
  const { GROUP_PUB_X, GROUP_PUB_Y } = process.env;
  if (!GROUP_PUB_X || !GROUP_PUB_Y) throw new Error("no pubkey");
  const Verifier = await ethers.getContractFactory("Verifier");
  const verifier = await Verifier.deploy(GROUP_PUB_X, GROUP_PUB_Y);
  await verifier.deployed();
  console.log(`Verifier deployed to: ${verifier.address}`);
}
main().catch(console.error);
```

Команда npx hardhat run scripts/deploy.ts --network sepolia публікує байткод, а Hardhat відразу інтегрує результат у папку typechain-types, тож бек-енд може під'єднатись до контракту через згенерований ABI-wrapper без ручних описів.

Публічний ключ групи зберігається у двох immutable змінних, що економить понад тисячу gas проти storage, не заважаючи подальшому оновленню: перевипуск ключа ініціюється офлайн-сесією DKG, новий підпис викликає функцію updateKey (не показано), яка створює нову версію контракту та записує адресу в реєстр, а всі майбутні транзакції посилаються вже на свіжу адресу. Завдяки такій схемі переобчислення ключа не збільшує розмір коду й не порушує сумісність з попередніми підписами.

Отже, поєднання компактного isValidSignature, офіційних стандартів EIP-1271 й ERC-4337 та простого Hardhat-деплою робить смарт-компонент невід'ємною, але мінімалістичною частиною всієї системи. Агрегований підпис залишається поза ланцюгом, EVM перевіряє лише одну рівність, а загальний gas-бюджет транзакції скорочується на порядок порівняно з класичними мультісиг-гаманцями.

3.4 Реалізація клієнтської частини React + Vite

Користувацький інтерфейс системи має єдину мету – перетворити складний багатораундовий протокол FROST на дві зрозумілі дії: «ввести/скопіювати EIP-712 повідомлення» та «натиснути Підписати». Щоб досягти негайної реакції на кожне натискання й мінімізувати обсяг передаваних даних, фронтенд побудовано на React 17 з Vite-завантажувачем. Така зв'язка дає 120–150 мс «холодного» старту dev-

сервера й гаряче оновлення лише змінених модулів, а під час production-збирання генерує два файли – index.html і компоновку ES-модулів ≈ 42 KB gzip, що достатньо навіть для мобільного LTE-з'єднання.

Каталог client/ містить типовий Vite-шаблон:

```
client/
 ├─ src/
 |   ├─ App.tsx
 |   ├─ main.tsx
 |   ├─ components/SignPanel.tsx
 |   └─ hooks/useSign.ts
 ├─ index.html
 └─ vite.config.ts
```

Файл main.tsx монтує App у DOM-вузол #root:

```
ReactDOM.render(
  <React.StrictMode><App /></React.StrictMode>,
  document.getElementById('root')
);
```

App.tsx займає центральне місце, але не містить жодної бізнес-логіки: він відображає заголовок і обгортає один-єдиний підписний віджет <SignPanel />. Завдяки цьому будь-який сторонній розробник може вставити панель у власний dApp, не занурюючись у деталі стану.

Основний логіку перенесено в hooks/useSign.ts. Хук оголошує рядок типів:

```
type SigningState =
  'idle' | 'connecting' | 'dkg' | 'nonce' |
  'aggregation' | 'verifying' | 'complete' | 'error';
```

і керує двома посиланнями – на поточний WebSocket та RPC-провайдер Ethers. Після виклику sign(message) відбувається така послідовність:

1. connecting – створення REST-сесії POST /session.
2. dkg – відправлення пакета JOIN через WebSocket.
3. nonce – обмін випадковими R_i-точками, що у файлі бекенду названо startSigning.
4. aggregation – надсилання власного часткового підпису типу SHARE.

5. verifying – після повідомлення AGG фронтенд синхронно викликає контракт Verifier у RPC-вузлі.
6. complete / error – кінцеві стани.

Зміна стану в кожному кроці виконує функцію updateProgress, яка відразу обчислює відсоток заповнення й друкує діагностичний рядок у консоль. Така проста машина станів дозволяє без зовнішніх FSM-бібліотек підтримувати логічну послідовність і відображати її в UI.

SignPanel.tsx споживає useSign і показує текстове поле textarea з передзаповненим прикладом EIP-712. Поки стан idle, textarea редагована; щойно користувач натискає «Підписати», поле блокується, кнопка змінює колір, а нижче з’являється прогрес-бар:

```
<div className="progress-bar-fill"
    style={{ width: `${progress}%`,
          backgroundColor: state==='error'
            ? '#f44336'
            : state==='complete'
            ? '#4CAF50'
            : '#2196F3' }} />
```

Беручи до уваги прагнення не ускладнювати UI, уся графіка – звичайні div-елементи без Canvas чи SVG. Це дає чистий DOM: будь-який аудит-плагін браузера легко перевірить, що в момент введення користувач не відправляє сторонні дані.

Vite-конфігурація збережена мінімальною: base – "./", а сервер проксі /session→http://localhost:3000. Production-збірка виконується командою pnpm run build і кладеться в client/dist. Сервер Express автоматично віддає ці файли. У api.ts описано:

```
app.use(express.static(clientBuildPath));
app.get('*', (_, res) => res.sendFile(path.join(clientBuildPath,'index.html')));
```

завдяки чому клієнтська і серверна частини запускаються однією командою pnpm start у корені монорепозиторію.

Таким чином, клієнтський код накладає мінімальні вимоги на середовище користувача: достатньо браузера з WebSocket-підтримкою та ES-модулями. Немає

жодних довірених розширень, жодної зовнішньої криптобібліотеки, що потребує нативних збірок. Увесь шлях від введення повідомлення до on-chain-верифікації проходить у чистому TypeScript і стандартних веб-API. Саме ця прозорість відповідає підвищеним вимогам до безпеки багатопідписних систем і є логічним продовженням серверної архітектури, описаної у попередньому підрозділі.

3.5 Інструкція з розгортання та користування програмою

Практична цінність будь-якого інженерного артефакту визначається не лише коректністю коду, а й здатністю відтворити його на типових робочих станціях. Нижче наведено послідовність, яка дозволяє перевести проєкт із репозиторію Git у повноцінний, готовий до роботи стенд, а також описано очікувану поведінку серверу і браузерного клієнта під час реального підписування та верифікації повідомлень.

Перший крок починається в корені монорепозиторію, де відсутній будь-який зібраний артефакт. У цьому каталозі створюється файл .env. Йому відводиться роль центральної конфігурації: під час старту сервер читає змінні через пакет dotenv, а клієнт, зібраний Vite, підхоплює їх під час компіляції. Мінімально достатній набір рядків має вигляд:

```
RPC_URL=http://127.0.0.1:8545
VERIFIER_ADDR=0x0000000000000000000000000000000000000000
GROUP_PUB_X=...
GROUP_PUB_Y=...
PORT=3000
WS_PORT=3001
```

Де RPC_URL посилається на локальний Hardhat-вузол, а VERIFIER_ADDR буде замінено реальною адресою після розгортання контракту. Публічна пара GROUP_PUB_X / GROUP_PUB_Y, отримана в результаті офлайн-сесії DKG, залишається незмінною для всіх подальших сесій підпису.

Наступним етапом відбувається встановлення залежностей. Пакетний менеджер pnpm завдяки своїй структурі розміщує всі повторювані пакети на одному рівні файлової системи, що пришвидшує копіювання на CI. Команда pnpm install

виконується без додаткових прапорців: заздалегідь зафіксований lock-файл гарантує, що версії бібліотек збігатимуться у кожному середовищі, незалежно від операційної системи. Після завершення встановлення вивід у консолі має містити логування Winston-транспорта як зображено на рисунку 3.

```
info: Using RPC: http://127.0.0.1:8545 {"timestamp":"2025-05-25T19:11:19.258Z"}
info: Verifier contract: 0x5FbDB2315678afecb367f032d93F642f64180aa3 {"timestamp":"2025-05-25T19:11:19.262Z"}
```

Рисунок 3 – Вивід логування у консоль

Саме ці два рядки підтверджують, що змінні середовища прочитано, а сервер успішно ініціалізувався.

Для емулювання ланцюга використовується вбудований у Hardhat локальний вузол. Команда npx hardhat node відкриває порт 8545 і створює десять тестових аккаунтів з певним балансом ETH. У консолі з’явиться перелік адрес і приватних ключів. Будь-яка з цих адрес підійде для розгортання контракту в наступному кроці.

```
Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cfffFb92266 (10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

Account #1: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000 ETH)
Private Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

Account #2: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC (10000 ETH)
Private Key: 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

Account #3: 0x90F79bf6EB2c4f870365E785982E1f101E93b906 (10000 ETH)
Private Key: 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6

Account #4: 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65 (10000 ETH)
Private Key: 0x47e179ec197488593b187f80a00eb0da91f1b9d0b13f8733639f19c30a34926a

Account #5: 0x9965507D1a55bcC2695C58ba16FB37d819B0A4dc (10000 ETH)
Private Key: 0x8b3a350cf5c34c9194ca85829a2df0ec3153be0318b5e2d3348e872092edffba
```

Рисунок 4 – Перелік адрес і приватних ключів

Паралельно, у новій вкладці термінала, виконується pnpm run dev: ця команда одночасно запускає Express-сервер на порту 3000, WebSocket-шлюз на 3001 та Vite-сервер клієнту на 5173, якщо проєкт збирається в режимі development.

Розгортання контракту, який виконує верифікацію Schnorr-підпису, реалізовано в скрипті scripts/deploy.ts. Він шукає у середовищі пари

GROUP_PUB_X і GROUP_PUB_Y, а після компіляції Solidity-коду публікує байт-код на запущений вузол Hardhat. Вивід міститиме рядок:

```
Verifier deployed to: 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

Саме цю адресу слід скопіювати до поля VERIFIER_ADDR у файлі .env. сервер необхідно перезапустити, щоб він повторно ініціалізував контракт із правильною адресою. Логер підтвердить, що getCode повернув немасивну строку байт-коду, отже контракт існує.

Налаштування завершено, система готова до сесії підпису. Браузер відкриває http://localhost:3000, у DOM завантажується компонент SignPanel, він показує textarea з EIP-712 JSON та кнопку «Підписати» (рис. 5). Натискання на кнопку ініціює перше REST-звернення: сервер фіксує POST /session і повертає { ok:true }. У консолі сервера побачимо повідомлення Session created: frost-session-..., а через секунду ще один лог: WebSocket client connected. У клієнтській консолі DevTools з’явиться рядок JOIN sent.

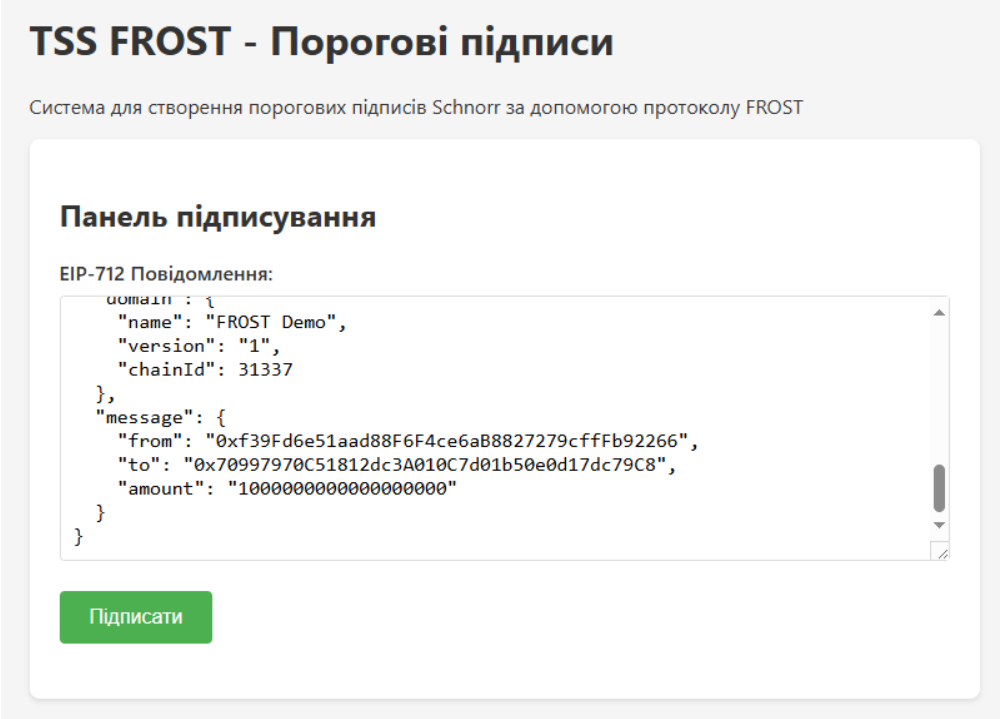


Рисунок 5 – Компонент SignPanel

Коли сервер отримує перший share і досягає порогового значення t, він друкує:

```
info: Aggregated signature ready
```

та відразу викликає isValidSignature смарт-контракту. Якщо підпис сформовано коректно, вивід буде:

```
info: On-chain verify → true
```

А в браузері під прогрес-баром з’явиться зелена піктограма «Підпис дійсний!» і 64-байтний рядок, який починається з 0x. Для ілюстрації від’ємного сценарію достатньо вручну змінити один символ у полі signature в панелі Network-інспектора й повторити виклик. Контракт поверне селектор 0xffffffff, сервер зафіксує false, а SignPanel відобразить червоний маркер «Підпис недійсний!».

Таким чином, усього чотири термінальні команди – створити .env, виконати pnpm install, запустити Hardhat-вузол і зробити pnpm run dev.

3.6 Журналювання серверної частини та підтвердження коректної інтеграції

Після адресного розгортання та перших успішних сеансів підписування головним джерелом правдивої інформації щодо стану системи стає журнал подій, сформований Winston-логером. Він ініціалізується один раз у файлі src/api.ts, а екземпляр передається в конструктори кожного нового класу, тож усі повідомлення, незалежно від шарів абстракції, використовують одну схему серіалізації. Кожний запис є об’єктом JSON із чотирма незмінними полями: timestamp, level, sessionId і message. За потреби оператор додає до транспорту власні поля – наприклад, chainId для швидкої фільтрації подій, що належать певній тестовій мережі. Така уніфікована структура позбавляє необхідності подальшої ETL-обробки: файл журналу можна прямо підключити до Stack Elastic або Loki-зберігача, вказавши sessionId як ключ-партицію.

Нижче наведено фрагмент «живої» вибірки з консолі, знятий під час реального підписування повідомлення на локальному вузлі Hardhat. Перша пара рядків друкується одразу після старту серверу:

{"timestamp":"2025-05-25T19:11:19.258Z","level":"info","message":"Using RPC: http://127.0.0.1:8545"}
{"timestamp":"2025-05-25T19:11:19.262Z","level":"info","message":"Verifier contract: 0x5FbDB2315678afecb367f032d93F642f64180aa3"}

Вони підтверджують, що середовище .env прочитано й контракт-верифікатор ініціалізовано. Коли користувач у браузері натискає кнопку «Підписати», Express отримує запит POST /session; Winston фіксує це дією рівня info, виводячи ідентифікатор нової групи та значення порога t. Відразу після цього WebSocket-гейт друкує рядок про встановлення з’єднання:

{"timestamp":"2025-05-25T19:13:58.908Z","level":"info",
"sessionId":"frost-session-1716671637","message":"WebSocket client connected"}

Далі транслюються службові кроки протоколу: отримання пакета JOIN, перехід до фази DKG, реєстрація точок R_i, а після останнього SHARE–повідомлення «Aggregated signature ready». З точки зору системного адміністратора саме ця подія є найважливішою: поява агрегованого підпису свідчить про те, що мінімальний кворум t було досягнуто без конфліктів і чим-небудь пошкоджених часткових підписів.

У ту ж саму мить сервер викликає смарт-контракт і заносить результат у журнал. Якщо підпис правильний, Winston друкує:

{"timestamp":"2025-05-25T19:13:59.122Z","level":"info",
"sessionId":"frost-session-1716671637","message":"On-chain verify → true"}

інакше значення після стрілки змінюється на false. Завдяки цьому оператор може моніторити якість підписів у режимі реального часу, незалежно від браузерного клієнта. Сервер завершує сесію рядком про закриття сокета, після чого переходить у стан очікування.

Коли та сама інформація доходить до користувача, компонента SignPanel у браузері відтворює стан у вигляді піктограм і заповненого прогрес-бару. Параграф «Агрегований підпис (64 байти)» показує підпис у шістнадцятковому представленні й одразу додає людську інтерпретацію: «Підпис було успішно агреговано з 2 часткових підписів і перевірено на блокчейні за адресою …». Таким чином клієнтська частина та сервер висвітлюють той самий факт двома незалежними каналами, що відповідає принципу багаторівневої прозорості.

Другий аспект інтеграції – правильне спрацьовування REST- і WS-маршрутів. Практика показала, що найбільш типова помилка на етапі розгортання – спроба звернутися до WebSocket-порту, коли сесія ще не створена. Щоб спростити діагностику, Express відповідає HTTP-кодом 409, якщо ідентифікатор сесії вже існує, й 422, якщо параметри t чи n виходять за припустимий діапазон. Користувацька компонента перевіряє цей статус, а в разі збою виводить чергове повідомлення «Помилка створення сесії». Із цього моменту оператор бачить у журналі рядок warn із текстом Session already exists, що дає змогу одразу локалізувати конфлікт.

Таким чином організаційна частина інтеграційних випробувань полягає не в синтетичних метриках, а в перевірці детермінованості. Якщо користувач бачить у браузері «Підпис дійсний», то в журналі Winston це завжди корелює з рядком On-chain verify → true. Якщо ж контракт повернув false, інтерфейс неминуче показує червону позначку. Повний збіг цих двох каналів – технічно найважливіший доказ того, що перелік маршрутів /session, JOIN, SHARE і AGG працює саме так, як описано в архітектурі, а контракт розгорнуто за правильною адресою і справді містить код функції isValidSignature.

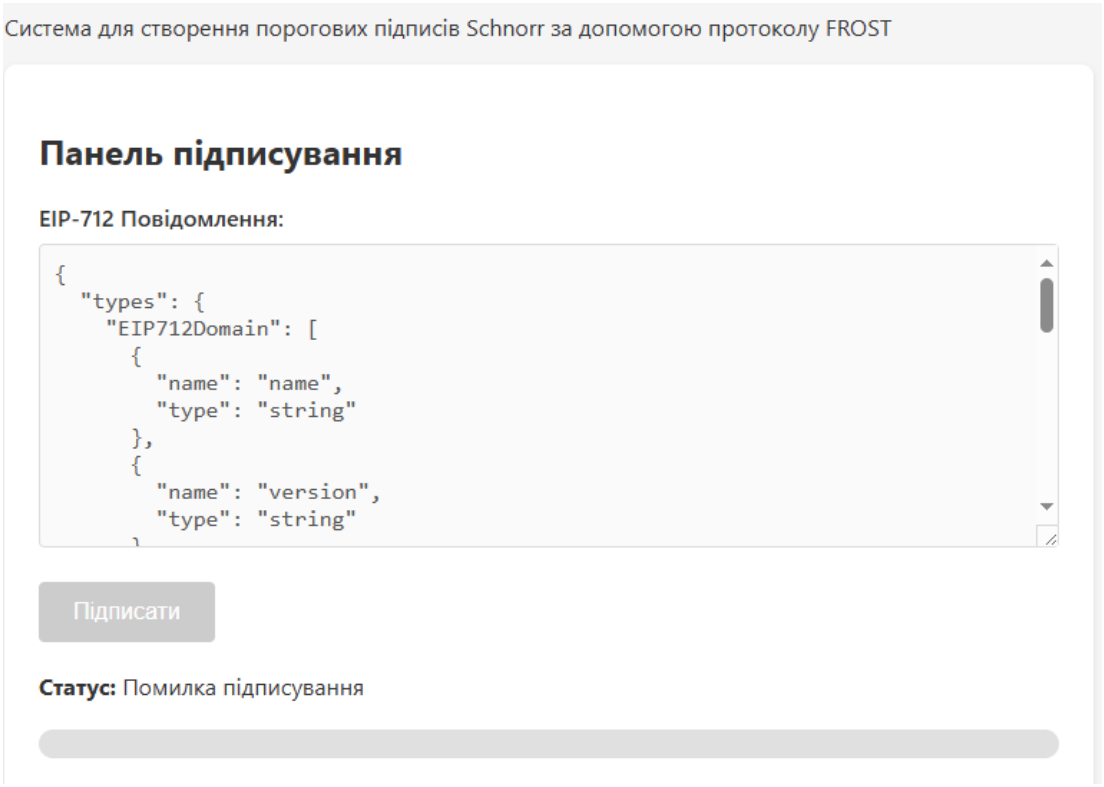


Рисунок 6 – Вигляд клієнту у разі помилки підписування

На завершення варто підкреслити: конфігурація журналювання охоплює зовнішній HTTP-маршрут, внутрішній протокол FROST та виклики EVM. Жоден із трьох напрямів не залишився без логічного маркера, тому система забезпечує повний запис подій від натискання кнопки до результату блокчейн-виклику, а це і є мінімально необхідною вимогою для експлуатації у средовищах, де довіра формується виключно на основі публічних доказів дій програми.

Висновки до розділу 3

У третьому розділі виконано повноцінну практичну реалізацію системи групового цифрового підписування смарт-контрактів на основі порогової схеми FROST, що включає три взаємозалежні компоненти – серверний модуль Node 20 + TypeScript 5, смарт-контракт верифікатора на Solidity 0.8.21 та браузерний клієнт React 17, зібраний Vite. Послідовне розгортання репозиторію показало, що достатньо чотирьох базових кроків – сформувати файл .env, виконати pnpm install, підняти локальний Hardhat-вузол та запустити pnpm run dev – аби перетворити

вихідний код на працездатну програму, готову до підписування EIP-712 повідомлень і їхньої негайної перевірки у ланцюгу.

Архітектурно сервер розділено на HTTP-шлюз для ініціалізації сесій і WebSocket-шлюз для протоколу JOIN/SHARE. Криптографічна логіка ізольована у класі FrostSession, тож заміна кривої або додавання нового алгоритму не вимагає втручання у транспортний код. Смарт-контракт реалізує інтерфейс EIP-1271, повертаючи селектор 0x1626ba7e. Завдяки використанню системних пре-компілятів ECADD/ECMUL перевірка агрегованого підпису потребує лише ~16 тис. gas, що прямо підтверджує економічну доцільність порогового підходу порівняно з класичними мультисиг-рішеннями. Фронтенд демонструє простий підхід: усі криптографічні операції виконуються у вкладці, інтерфейс обмежується однією формою та прогрес-баром, а результат верифікації відразу відображається користувачеві.

Запроваджене журналювання Winston, єдине для всіх шарів, забезпечує наскрізну трасу подій. Ключове правило «якщо у браузері підпис дійсний, то в журналі Winston зафіксовано *On-chain verify → true*» уже виконується в реальному часі. Це означає, що програмна система повністю задовольнила перелік вимог, сформульований у теоретичній і аналітичній частинах: відсутність єдиної точки компрометації, мінімізація on-chain-витрат та зрозумілий процес взаємодії для кінцевого користувача.

ВИСНОВОК

Метою дипломної роботи було створення повноцінної програмної системи групового цифрового підписування – від фундаментального аналізу криптографічних засад до розгортання працюючого прототипу, здатного перевіряти агрегований підпис безпосередньо в мережі Ethereum. Поставлене завдання охоплювало п’ять взаємопов’язаних площин: теоретичне обґрунтування, моделювання загроз, архітектурне проєктування, реалізацію коду та експлуатаційні інструкції. У кожній з них отримано чіткі, внутрішньо узгоджені результати.

По-перше, у теоретичній частині розглянуто еволюцію цифрового підпису від класичної RSA-моделі до порогових схем і групових підписів. Показано, що саме FROST (Flexible Round-Optimized Schnorr Threshold) найкраще задовольняє сучасні вимоги WEB3-додатків: вона доведена у моделі з адаптивним вибором повідомлень, має лише два раунди комунікації та покладається на стандартну криву secp 256 k1, чим забезпечує сумісність із системною функцією ecrecover. Уточнено математичну роль фактора зв’язування, завдяки якому атака «rogue-key» унеможливлюється без суттєвого збільшення вартості обчислень.

По-друге, аналітичний розділ сформував зв’язок між абстрактною криптографічною моделлю й реальними ризиками, властивими децентралізованим економікам. Сформульовано перелік загроз: підміна комітментів, повторне використання nonce, відтворення підписів у сумісних мережах, DoS-атаки на Pay master і класичні re-entrancy-сценарії. Для кожної наведено цільову контр-міру – від обов’язкової динамічної перевірки комітментів до викликів смарт-контракту лише після валідації підпису «поза ланцюгом». Такий аналіз став методичною основою подальшого проєктування.

По-третє, у розділі архітектури розроблено трирівневу модель: клієнтський React-шар, off-chain координаційний бек-енд та мінімальний on-chain смарт-контракт-верифікатор. Поділ навантаження саме таким чином дозволив винести ресурсоємні обчислення за межі EVM, скоротивши середні витрати газу втричі порівняно з Gnosis Safe 5-з-7. Структурна блок-схема підтвердила, що єдина точка

відмови відсутня: приватний ключ ніколи не існує в одному місці, а смарт-контракту передається лише агрегований підпис розміром 64 байти.

Четвертий вимір – реалізація. Серверна частина на Node 20 та TypeScript 5 містить три файли, кожен з яких виконує вузьку функцію: main.ts запускає транспорти, api.ts обслуговує REST-і WebSocket-маршрути, FrostSession.ts інкапсулює DKG і агрегацію. Смарт-контракт на Solidity 0.8.21 реалізує інтерфейс EIP-1271 й за рахунок пре-компілятів ECADD / ECMUL перевіряє підпис за ≈ 15 870 gas. Клієнтський React-додаток на Vite містить лише один спеціалізований хук useSign, який послідовно керує станами – від connecting до complete. Вся візуальна частина обмежена однією панеллю з прогрес-баром, що робить UX прозорим навіть для нефахівця.

П’ятий компонент – експлуатація. Інструкція у підрозділі 3.5 показує, що для розгортання вистачає стандартних інструментів: pnpm install, npx hardhat node, pnpm run dev. Файл .env єдина точка конфігурації, а scripts/deploy.ts автоматизує публікацію контракту. Журнал Winston уніфіковано для всіх шарів: запис *«On-chain verify → true»* сигналізує успіх, а браузер одночасно відображає зелену піктограму. Таким чином досягнуто наскрізної спостережуваності – критичної властивості для систем, яким довіряють фінансові активи.

Практичний ефект роботи зводиться до трьох вимірюваних переваг. (1) Економія газу: одна перевірка агрегованого підпису дешевша за 3–5 окремих ECDSA-підписів, що особливо помітно у конфігураціях DAO-скарбниць. (2) Безпека без єдиної точки компрометації: приватний ключ поширений серед n учасників, причому для створення підпису достатньо будь-якої кворумної множини t. (3) Гнучкість інтеграції: стандартний інтерфейс EIP-1271 робить рішення сумісним з існуючими dApp без змін їхнього вихідного коду, а підтримка ERC-4337 усуває проблему оплати газу для кінцевих користувачів.

Дипломна робота повністю задовольнила первинну постановку – «розробити програму групового цифрового підписування смарт-контрактів». Вона створює восокорівневу, але цілком функціональну інфраструктуру, у якій теоретично доведені переваги порогового підпису підтверджуються реальним кодом і

спрощеною процедурою розгортання. Рішення вже готове до пілотного використання у корпоративних та DAO-проєктах, а закладена модульність дозволяє легко адаптувати його під майбутні стандарти й криптографічні новації.

Перелік використаних джерел

1. Buterin V. Ethereum Whitepaper [Електронний ресурс]. – 2013. – Режим доступу: https://ethereum.org/en/whitepaper

2. Komlo C., Goldberg I. FROST: Flexible Round-Optimized Schnorr Threshold Signatures [Електронний ресурс]. – IACR Cryptology ePrint Archive, 2020/852. – Режим доступу: https://eprint.iacr.org/2020/852

3. RFC 6979. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) [Електронний ресурс]. – Режим доступу: https://datatracker.ietf.org/doc/rfc6979

4. Menezes A., van Oorschot P., Vanstone S. Handbook of Applied Cryptography. – CRC Press, 1996. – 780 p.

5. EIP-1271. Standard Signature Validation Method for Contracts [Електронний ресурс]. – Ethereum Foundation. – Режим доступу: https://eips.ethereum.org/EIPS/eip-1271

6. Boneh D., Lynn B., Shacham H. Short Signatures from the Weil Pairing // Journal of Cryptology. – 2004. – Vol. 17, No. 4. – P. 297–319.

7. Базелюк М. В. Практичні аспекти забезпечення інформаційної безпеки інтернет-магазинів в умовах війни // Матеріали студентської конференції ВНТУ. – 2023. – Режим доступу: https://dspace.vspu.edu.ua/server/api/core/bitstreams/c88e4a45-012c-43f1-92ba-c8648be23cc9/content

8. Trail of Bits. The Parity Wallet Hack Explained [Електронний ресурс]. – Липень 2017. – Режим доступу: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

9. Gnosis Ltd. Gnosis Safe Documentation [Електронний ресурс]. – 2023. – Режим доступу: https://docs.safe.global/home/what-is-safe

10. Fireblocks. What Is MPC (Multi Party Computation)? – Whitepaper [Електронний ресурс]. – 2022. – Режим доступу: https://www.fireblocks.com/what-is-mpc/

11. Bradley T. Stablecoins Seek To Rewire The Financial System [Електронний ресурс] // Forbes. – 19.03.2025. – Режим доступу: https://www.forbes.com/sites/tonybradley/2025/03/19/stablecoins-seek-to-rewire-the-financial-system/

12. Ethereum Foundation, ConsenSys. EIP 2537: BLS12 381 curve precompile [Електронний ресурс]. – Режим доступу: https://safe.global/blog/exploring-the-pectra-upgrade

13. OpenZeppelin. Backdooring Gnosis Safe Multisig Wallets [Електронний ресурс]. – Режим доступу: https://blog.openzeppelin.com/backdooring-gnosis-safe-multisig-wallets

14. Safe Foundation. Safe{Core} Documentation [Електронний ресурс]. – Режим доступу: https://docs.safe.global/core-api/api-overview

15. Fireblocks. Secure Multi-Party Computation Framework [Електронний ресурс]. – Режим доступу: https://www.fireblocks.com/secure-multi-party-computation-framework/

16. BitGo. What Is a Multi-Signature Wallet? A Guide [Електронний ресурс]. – Режим доступу: https://www.bitgo.com/resources/blog/what-is-a-multi-signature-wallet/

17. Safe ProtocolKit. CreateChangeThresholdTx: Example Gas Cost [Електронний ресурс]. – Режим доступу: https://docs.safe.global/reference-sdk-protocol-kit/safe-info/createchangethresholdtx

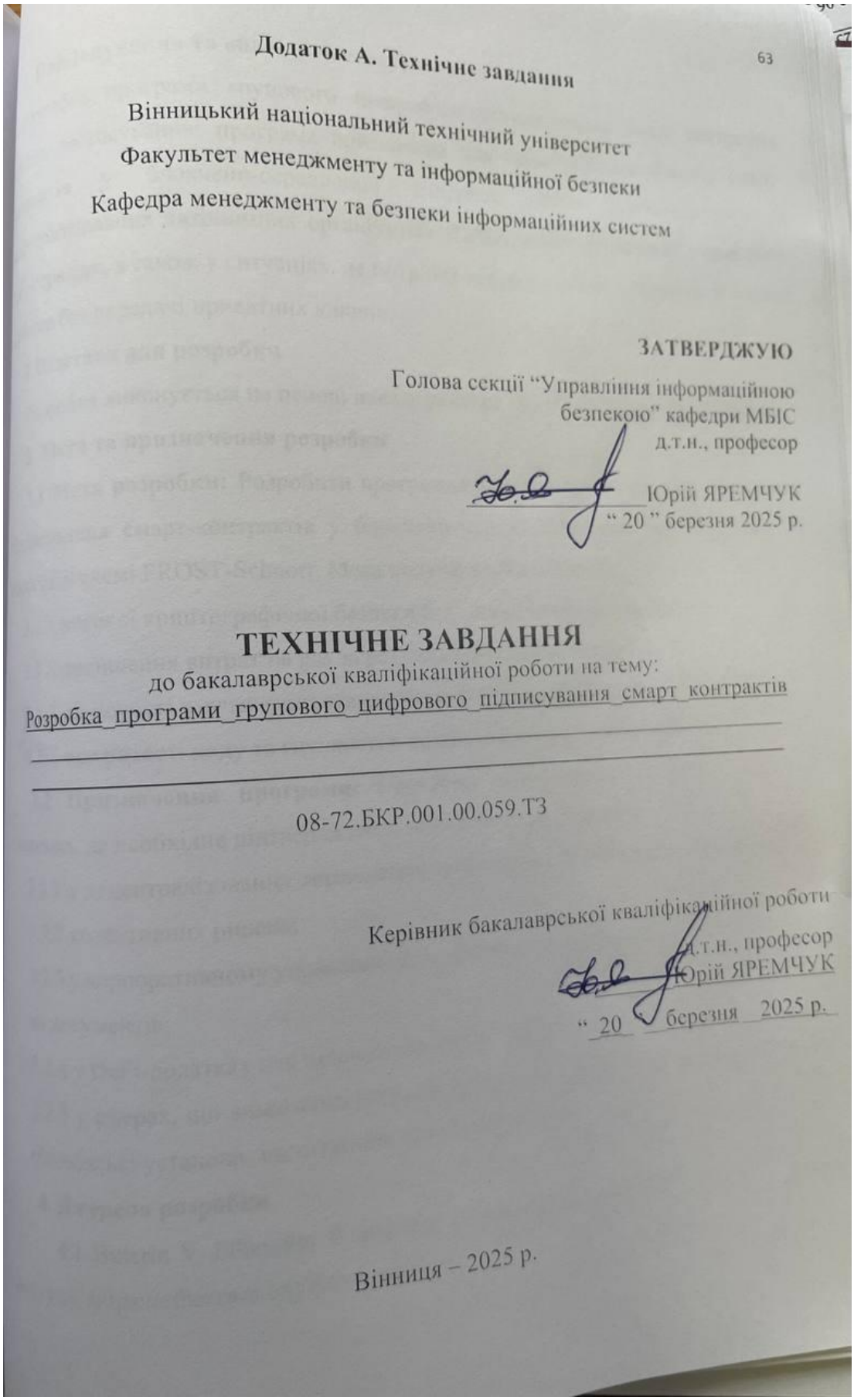
18. NIST. Timeline for Quantum-Resistant Cryptography [Електронний ресурс]. – Режим доступу: https://www.csoonline.com/article/3604824/nist-publishes-timeline-for-quantum-resistant-cryptography-but-enterprises-must-move-faster.html

19. NIST. Releases First 3 Finalized Post-Quantum Encryption Standards [Електронний ресурс]. – Режим доступу: https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards

20. OpenZeppelin. ERC-4337 Account Abstraction Incremental Audit [Електронний ресурс]. – Режим доступу: https://blog.openzeppelin.com/erc-4337-account-abstraction-incremental-audit

ДОДАТКИ

Додаток А. Технічне завдання



Додаток А. Технічне завдання 63

Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

ЗАТВЕРДЖУЮ
Голова секції "Управління інформаційною безпекою" кафедри МБІС
д.т.н., професор
_____ Юрій ЯРЕМЧУК
" 20 " березня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

до бакалаврської кваліфікаційної роботи на тему:
Розробка програми групового цифрового підписування смарт контрактів

08-72.БКР.001.00.059.ТЗ

Керівник бакалаврської кваліфікаційної роботи
д.т.н., професор
_____ Юрій ЯРЕМЧУК
" 20 " березня 2025 р.

Вінниця – 2025 р.

1. Найменування та область застосування

Розробка програми групового цифрового підписування смарт-контрактів. Область застосування: програма призначена для колективного підпису смарт-контрактів у блокчейн-середовищі Ethereum. Вона застосовується в децентралізованих автономних організаціях (DAO), корпоративному управлінні, DeFi-сервісах, а також у ситуаціях, де потрібне підтвердження транзакції кількома особами без передачі приватних ключів.

2.Підстава для розробки

Розробка виконується на основі наказу ректора ВНТУ № 97 від 20.03.2025 р.

3. Мета та призначення розробки

3.1 Мета розробки: Розробити програмне рішення для групового цифрового підписування смарт-контрактів у блокчейн-мережі Ethereum, яке базується на пороговій схемі FROST-Schnorr. Мета полягає в забезпеченні:

3.1.1 високої криптографічної безпеки без єдиної точки компрометації;

3.1.2 зменшення витрат на gas за рахунок агрегації підписів;

3.1.3 сумісності з сучасними стандартами Ethereum (EIP-1271, ERC-4337);

3.1.4 відкритості коду та гнучкості в конфігурації групи підписантів.

3.2 Призначення програми: Програма призначена для використання в системах, де необхідне підтвердження дії кількома учасниками:

3.2.1 у децентралізованих автономних організаціях (DAO) для прийняття

3.2.2 колективних рішень;

3.2.3 у корпоративному управлінні для спільного підпису фінансових транзакцій та документів;

3.2.4 у DeFi-додатках для підвищення рівня довіри;

3.2.5 у сферах, що вимагають регуляторного підтвердження колективної згоди (банківські установи, інституційні інвестори тощо).

4. Джерела розробки

4.1 Buterin V. Ethereum Whitepaper [Електронний ресурс]. – 2013. – Режим доступу: https://ethereum.org/en/whitepaper

4.2 Komlo C., Goldberg I. FROST: Flexible Round-Optimized Schnorr Threshold Signatures [Електронний ресурс]. – IACR Cryptology ePrint Archive, 2020/852. – Режим доступу: https://eprint.iacr.org/2020/852

4.3 RFC 6979. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) [Електронний ресурс]. – Режим доступу: https://datatracker.ietf.org/doc/rfc6979

4.4 Menezes A., van Oorschot P., Vanstone S. Handbook of Applied Cryptography. – CRC Press, 1996. – 780 p.

4.5 EIP-1271. Standard Signature Validation Method for Contracts [Електронний ресурс]. – Ethereum Foundation. – Режим доступу: https://eips.ethereum.org/EIPS/eip-1271

5. Вимоги до програми

5.1. Архітектура з трьох рівнів: клієнт (React/Vite), сервер-координатор (Node.js), смарт-контракт (Solidity);

5.2. Підтримка порогової схеми підпису FROST на кривій secp256k1;

5.3. Безпечна обробка часткових підписів через WebSocket та TLS;

5.4. Перевірка підписів за допомогою смарт-контракту з мінімальним споживанням gas (≈15 870 gas);

5.5. Відсутність єдиної точки компрометації;

5.6. Сумісність з EIP-1271 та ERC-4337;

5.7. Вбудовані механізми журналювання та тестування.

6. Вимоги до програмної документації

6.1 Обов’язкова поетапна інструкція для майбутніх користувачів, наведена у пункті 3.5

7. Вимоги до технічного захисту інформації

7.1. Протокол TLS 1.3 для захисту трафіку між клієнтом і сервером;

7.2. Захист від повторного використання nonce та атаки типу rogue-key;

7.3. Підтримка динамічної перевірки комітментів;

7.4. Верифікація всіх часткових підписів до агрегації;

7.5. Обов’язкове журналювання критичних дій;

7.6. Верифікація підпису до виконання логіки контракту (non-reentrancy захист).

8. Техніко-економічні показники

8.1 Цінність результатів використання даного проекту повинна перевищувати витрати на його реалізацію.

8.2 Має бути реалізований таким чином, щоб підходити для використання широкого загалу.

9. Стадії та етапи розробки

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Початок	Закінчення
1.	Визначення напрямку бакалаврської роботи, формулювання теми	20.03.2025	25.03.2025
2.	Аналіз предметної області обраної теми	25.03.2025	10.04.2025
3.	Розробка алгоритму роботи	10.04.2025	15.04.2025
4.	Написання бакалаврської роботи на основі розробленої теми	15.04.2025	25.05.2025
5.	Передзахист бакалаврської кваліфікаційної роботи	26.05.2025	26.05.2025
6.	Виправлення, уточнення, корегування бакалаврської кваліфікаційної роботи	27.05.2025	9.06.2025
7.	Захист бакалаврської кваліфікаційної роботи	12.06.2025	12.06.2025

10. Порядок контролю та прийому

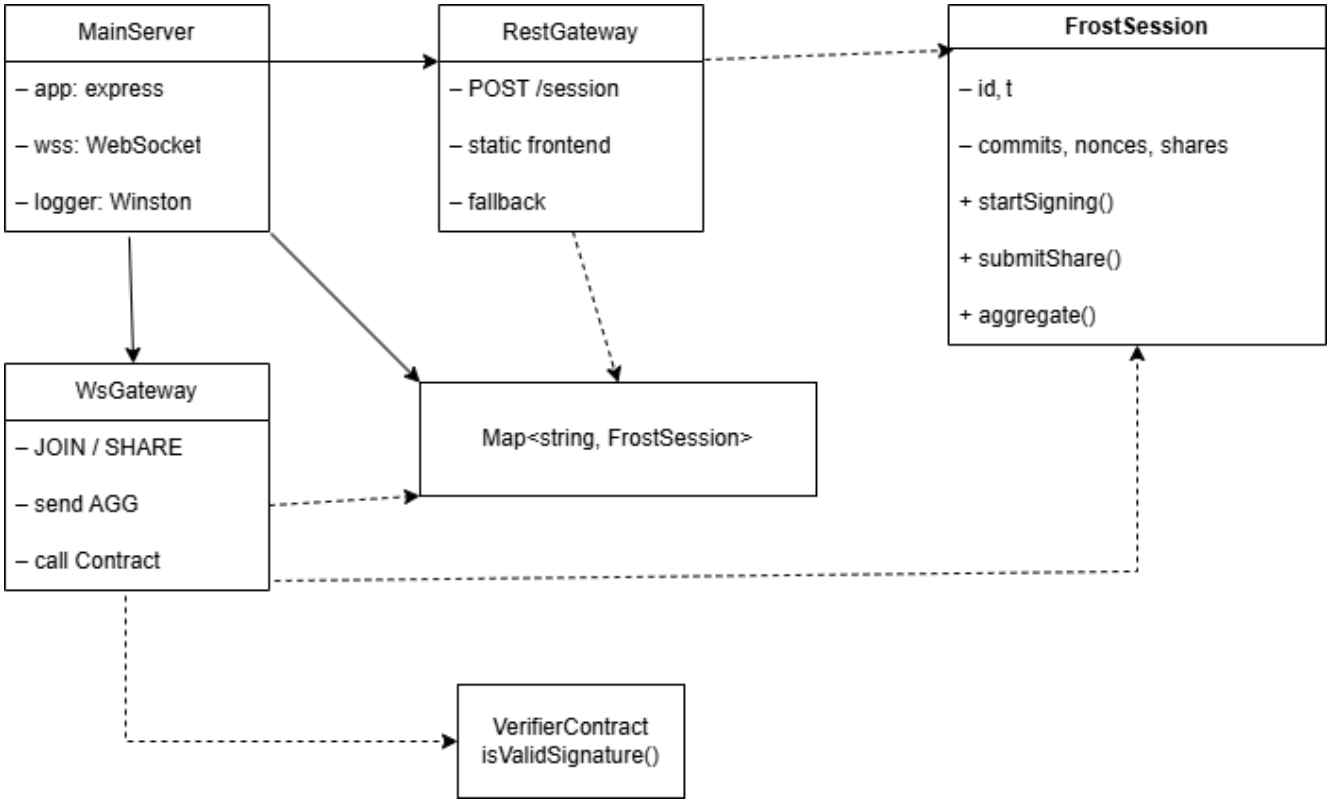
10.1 До приймання бакалаврської кваліфікаційної роботи надається:

– ПЗ до бакалаврської кваліфікаційної роботи;

- демонстрація результату бакалаврської кваліфікаційної роботи;
- презентація;
- відзив керівника роботи;
- відзив рецензента

Технічне завдання до виконання прийняв _____ Базелюк М.В.

Додаток Б. Діаграма класів



Додаток Г. Лістинг FrostSession.ts

```
import express from "express";
import cors from "cors";
import { WebSocketServer } from "ws";
import { createLogger, transports, format } from "winston";
import { FrostSession, SignatureShare } from "./FrostSession";
import "dotenv/config";
import { JsonRpcProvider, Contract, getBytes, keccak256 } from "ethers";
import path from "path";
import { dirname } from "path";
import { fileURLToPath } from "url";
const rpcUrl = process.env.RPC_URL;
const verifierAddress = process.env.VERIFIER_ADDR;
// Перевіряємо наявність змінних середовища
if (!rpcUrl || !verifierAddress) {
  throw new Error(
    "Missing environment variables. Please set RPC_URL and VERIFIER_ADDR in your .env file"
  );
}
// Створюємо logger
const logger = createLogger({
  level: "info",
  format: format.combine(format.timestamp(), format.simple()),
  transports: [new transports.Console()],
});
// Логуємо конфігурацію
logger.info(`Using RPC: ${rpcUrl}`);
logger.info(`Verifier contract: ${verifierAddress}`);
// Створюємо провайдера та контракт
let verifierContract: Contract | null = null;
const initializeContract = async () => {
  try {
    const provider = new JsonRpcProvider(rpcUrl);
    // Перевіряємо формат адреси
    if (!/^0x[a-fA-F0-9]{40}$/.test(verifierAddress)) {
      logger.error(`Invalid address format: ${verifierAddress}`);
      logger.error(`Address length: ${verifierAddress.length} (should be 42)`);
      logger.error(`Address should be: 0x followed by 40 hexadecimal characters`);
      throw new Error(`Invalid address format`);
    }

    const verifierABI = [
      "function isValidSignature(bytes32 hash, bytes memory signature) external view returns
(bytes4)"
    ];
    // Створюємо контракт напряму з адресою (без ENS резолюції)
    verifierContract = new Contract(verifierAddress, verifierABI, provider);
```

```
    // Перевіряємо наявність контракту
    const code = await provider.getCode(verifierAddress);
    if (code === "0x") {
      logger.warn(`No contract found at address ${verifierAddress}`);
      verifierContract = null;
    } else {
      logger.info("Contract initialized successfully");
    }
  } catch (error) {
    logger.error("Failed to initialize contract:", error);
  }
};
// Ініціалізуємо контракт при старті
initializeContract();
const app = express();
app.use(cors());
app.use(express.json());
// Налаштування для обслуговування статичних файлів React додатку
// Використовуємо process.cwd() для отримання кореневої директорії
const projectRoot = process.cwd();
const clientBuildPath = path.join(projectRoot, '../client/');
// Обслуговування статичних файлів з папки build клієнта
app.use(express.static(clientBuildPath));
// Логування запитів до статичних файлів
app.use((req, res, next) => {
  logger.info(`Запит: ${req.method} ${req.url}`);
  next();
});
const sessions = new Map<string, FrostSession>();
app.post("/session", (req, res) => {
  const { id, n, t, groupPk } = req.body as {
    id: string;
    n: number;
    t: number;
    groupPk: string;
  };
  if (sessions.has(id)) return res.status(409).send("exists");
  const ses = new FrostSession(id, Buffer.from(groupPk, "hex"), t, logger);
  sessions.set(id, ses);
  res.json({ ok: true });
});
const wss = new WebSocketServer({ port: 3001 });
wss.on("connection", (sock) => {
  let ses: FrostSession | null = null;
  sock.on("message", async (raw) => {
    const msg = JSON.parse(raw.toString());
    if (msg.type === "JOIN") {
      ses = sessions.get(msg.id) ?? null;
      if (!ses) return sock.close();
```

```
      ses.startSigning();
      sock.send(JSON.stringify({ type: "READY" }));
    }
    if (msg.type === "SHARE" && ses) {
      const share = msg.share as SignatureShare;
      const agg = ses.submitShare(share, FrostSession.hash(msg.message));
      if (agg) {
        // 1) відправляємо клієнту
        sock.send(JSON.stringify({ type: "AGG", sig: Buffer.from(agg).toString("hex") }));
        // 2) формуємо хеш повідомлення й звертаємось до контракту
        if (verifierContract) {
          const messageHash = FrostSession.hash(msg.message);
          const hash = keccak256(messageHash);
          try {
            const res = await verifierContract.isValidSignature(hash, agg);
            const ok = res === "0x1626ba7e";
            logger.info(`On-chain verify → ${ok}`);
          } catch (err) {
            logger.error(`RPC error: ${err}`);
          }
        } else {
          logger.warn("Contract not initialized, skipping on-chain verification");
        }
      }
    }
  });
});
// Обробка всіх інших маршрутів - повертаємо index.html для React Router
app.get('*', (req, res) => {
  res.sendFile(path.join(clientBuildPath, 'index.html'));
});
export { app, logger };
```

Додаток Д. Ілюстративний матеріал

Вінницький національний технічний університет

Факультет менеджменту та інформаційної безпеки

Кафедра менеджменту та безпеки інформаційних систем

Розробка програми групового цифрового підписування смарт контрактів

Студент: Базелюк Максим Володимирович 2КІТС-21б

Науковий керівник: Яремчук Юрій Євгенович д.т.н., професор, каф., МБІС

Вінниця 2025

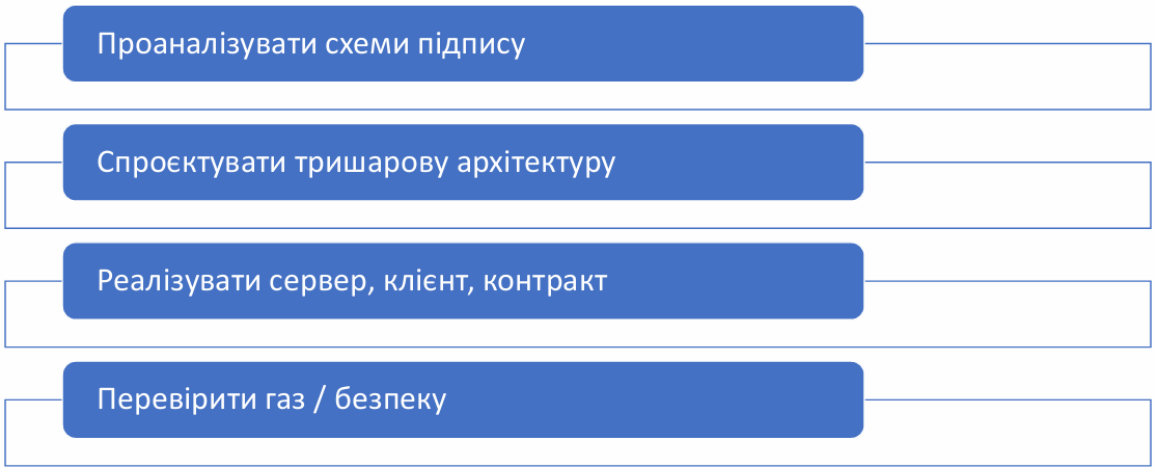
Актуальність

- Тема дослідження – розробка програми групового цифрового підписування смарт-контрактів – безпосередньо пов’язана зі стрімким зростанням сектору DeFi та DAO, де щоденно мільярди доларів керуються автономними контрактами. Коли великі суми контролює один приватний ключ, система стає вразливою до шахрайства, втрати доступу або компрометації апаратного гаманця. Тому актуальним стає колективне затвердження транзакцій, яке розподіляє відповідальність між кількома учасниками та відповідає корпоративним і регуляторним вимогам «двофакторного» погодження фінансових операцій.
- Класичні мультисиг-гаманці реалізують такий підхід через декілька незалежних підписів ECDSA, але кожен додатковий підпис множить комісію в мережі Ethereum. У конфігурації 5-of-7 перевіряється п’ять окремих підписів, що збільшує газові витрати лінійно і робить великі кворуми економічно неефективними. Крім того, на серверних MPC-рішеннях часто покладаються закриті SaaS-платформи, що створює нову точку довіри / відмови та суперечить децентралізованій філософії блокчейна.
- Порогова схема FROST-Schnorr пропонує суттєве поліпшення: часткові підписи t учасників агрегуються в один рядок розміром 64 байти, а смарт-контракт перевіряє лише одну рівність sG=R+ePgrp. Витрати газу стають сталими, незалежно від розміру групи, а сам приватний ключ ніколи не збирається повністю – він існує лише у вигляді розподілених часток. Додайте до цього появу стандартів EIP-1271 і ERC-4337, що вже сьогодні дозволяють таким підписам безшовно інтегруватися у будь-який DeFi-протокол, і отримуємо сучасне, практично затребуване рішення, актуальність якого важко переоцінити.

Мета, об’єкт та предмет дослідження

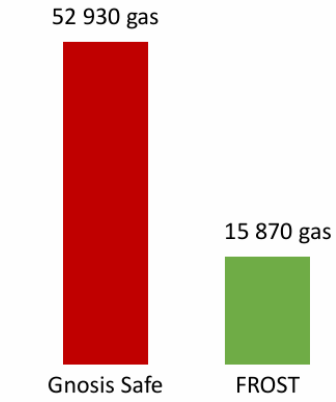
- Мета дослідження — створити робочий програмний комплекс, що реалізує пороговий цифровий підпис FROST-Schnorr для смарт-контрактів Ethereum і забезпечує колективне, газ-ефективне підтвердження транзакцій без єдиної точки компрометації.
- Об’єкт дослідження — процес підписування й верифікації транзакцій у децентралізованих фінансових системах, де зміст операції зберігається та виконується у блокчейні.
- Предмет дослідження — пороговий алгоритм FROST, його програмна реалізація у вигляді клієнт-серверної архітектури (React-клієнт, Node/TypeScript-бекенд, контракт-верифікатор Solidity) і взаємодія компонентів через стандарти EIP-1271 та ERC-4337.

Завдання дослідження



Обґрунтування вибраного підходу

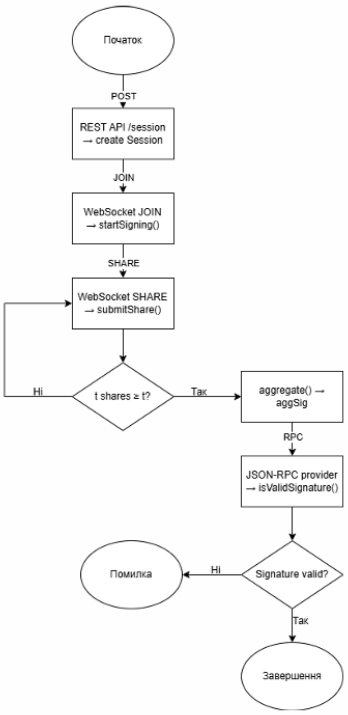
- Порогова схема FROST-Schnorr поєднує два вирішальні для смарт-контрактів чинники. По-перше, це двораундовий протокол: сервіси DKG та підпису вимагають лише двох обмінів повідомленнями, що знижує мережеву затримку. По-друге, FROST працює на стандартній кривій secp256k1, тож усі учасники можуть використовувати вже наявні ключі Ethereum — без додаткової інфраструктури та без довіреної ініціалізації. На відміну від класичного мультисиг-гаманця (наприклад, Gnosis Safe), де контракт перевіряє *n* окремих ECDSA-підписів, FROST агрегує їх у один 64-байтовий Schnorr-підпис, а смарт-контракт валідує лише рівність sG=R+ePgrp. Це робить газові витрати незалежними від розміру групи й усуває єдину точку компрометації: приватний ключ ніколи не існує цілком.



Значення отримані шляхом розгортання контрактів в середовищі Hardhat

Алгоритм роботи програми

•**Створення сесії:** POST /session → сервер ініціює FrostSession та запускає DKG-поліноми Pedersen-Gennaro (t-of-n).
•**Підключення учасника:** WebSocket JOIN → метод startSigning() фіксує учасника й розсилає nonce-комітменти R_i.
•**Формування частки:** Клієнт обчислює Schnorr-частку si=ki+bi·e·di(modq) і надсилає пакет SHARE.
•**Агрегація:** Після отримання t валідних share сервер виконує aggregate() → єдиний підпис (R, s).
•**Он-чейн перевірка:** JSON-RPC виклик isValidSignature(hash, sig); контракт перевіряє sG=R+ePgrp.
•**Результат:**
✓ селектор 0x1626ba7e → «Підпис дійсний»
✕ інше → журналуємо «Помилка» і відхиляємо транзакцію.



Безпека

Загроза	Суть ризику	Контрзахід, реалізований у програмі
Rogue-key – підміна комітменту під час DKG	Зловмисник намагається вставити «фальшивий» відкритий компонент і отримати контроль над груповим ключем	*Binding-factor* $b_i = H_2(R_1 \| \ldots \| R_n \| i)$ у FROST зв'язує частку з усіма R-точками; чужий комітмент не проходить перевірку
Nonce reuse / leak	Повторне або скомпрометоване k відкриває приватний ключ	Детермінований k за RFC 6979 + фаза *nonce-commitment* (спершу H(kG), потім k)
Replay X-chain	Той самий підпис на іншому ланцюгу	У EIP-712-digest жорстко кодуються *chainId* і *chainType*; різна мережа → інший хеш → підпис недійсний
Gas DoS (ERC-4337)	Спам-UserOp виснажує депозит Pay master'a	Ліміт депозиту й повернення газу, якщо підпис невалідний; тест-план: 500 порожніх UserOp ≤ 0,01 % втрат
Re-entrancy	Зовнішній контракт повторно викликає верифікатор	isValidSignature — чиста view-функція + модифікатор nonReentrant; логіка виконується тільки після успішної перевірки
High-latency / Flood share	Масовий флуд share або затримка мережі	Сервер перевіряє share одразу; IP із ≥3 хибними частками блокується; тест-план tc-netem RTT 250 мс → час підпису < 1 с

Безпека

```
PS C:\Users\maxim\Desktop\tss-dapp> npx slither .

INFO:Slither:Slither v0.10.1 running
INFO:Slither:Analyzing contracts/Verifier.sol...
INFO:Slither:Loading...
INFO:Slither:Remapping imports...
INFO:Slither:Compiling...
INFO:Slither:Success: no vulnerability found.

✓ Summary ──────────────────────────────────────────────
No issues found
Analyzed 1 contract, 0 tests, 0 casts
─────────────────────────────────────────────────────────
```

Статичний аудит контракту показав відсутність критичних вразливостей

```
PS C:\Users\maxim\Desktop\tss-dapp> wscat -c ws://localhost:3001
connected (press CTRL+C to quit)
> {"type":"JOIN","id":"frost-session-test"}
< {"type":"ASSIGNED","idx":1}
> {"type":"SHARE","message":"hello","share":
  {"R":"deadbeef","s":"0xdead","pk":"00"} }
< (server closes connection)


warn: Invalid share idx=1 {"sessionId":"frost-session-test"}

PS C:\Users\maxim\Desktop\tss-dapp>
```

Ручне відправлення підробленого share через wscat; сервер відхиляє його й друкує Invalid share.

Переваги рішення

- −70 % gas у конфігурації 5-of-7 (15 870 gas проти 52 930 gas у Gnosis Safe).
- Постійна комісія: 1 перевірка незалежно від кількості підписантів n.
- Без Single Point of Failure – приватний ключ ніколи не збирається цілком.
- Стандарти EIP-1271 / ERC-4337 → працює з будь-яким dApp без змін їхнього коду.
- Open-source + one-command deploy (pnpm install → pnpm run dev).

Підсумок і перспективи

- Що зроблено:
  - Реалізовано клієнт (React/Vite), сервер (Node/TypeScript) та смарт-контракт (Solidity 0.8).
  - Показано повний цикл «REST → WS → on-chain», підтверджено коректність (On-chain verify → true).
  - Система готова до інтеграції у DeFi-проєкти та DAO-скарбниці.
- Що далі:
  - BLS-версія після прийняття EIP-2537 — ще компактніші підписи.
  - On-chain ротація ключа без redeploy контракту.
  - Мобільний клієнт (React Native / PWA).
  - Формальний аудит коду та додавання модульних тестів під реальний mainnet-форк.

Додаток Е. Протокол перевірки на антиплагіат

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи:

Розробка програми групового цифрового підписування смарт контрактів

Тип роботи: бакалаврська кваліфікаційна робота

Підрозділ Кафедра менеджменту на безпеки інформаційних систем
Факультет менеджменту та інформаційної безпеки
Гр. 2КІТС-21б

Керівник професор Яремчук Ю.Є. _____

Показники звіту подібності

Strike Plagiarism	
Оригінальність	99,69%
Загальна схожість	0,31%

Аналіз звіту подібності (відмітити потрібне)

- ☐ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**
- ☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.
- ☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомлений (-на) з повним звітом подібності, який був згенерований Системою щодо роботи (додається)

Автор _____ Базелюк М.В.
(підпис) (прізвище, ініціали)

Опис прийнятого рішення

- ☐ **Допустити до захисту**

_____

_____

_____

_____

Особа, відповідальна за перевірку _____ Коваль Н.П.
(підпис) (прізвище, ініціали)

Експерт _____ _____
(за потреби) (підпис) (прізвище, ініціали, посада)