КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки
на тему:

АРБІТРАЖ-БОТ З ВИКОРИСТАННЯМ MEMPOOL ДЛЯ ДЕЦЕНТРАЛІЗОВАНИХ ОБМІННИКІВ

Виконав студент 4-го курсу
Владислав Некряч _____ (підпис)

Науковий керівник:
професор, доктор фіз.-мат. наук
Анатолій Пашко _____ (підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань. Студент _____ (підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теоретичної кібернетики

«____»_____2023 р., протокол No ____

Завідувач кафедри
Юрій КРАК _____ (підпис)

Київ - 2023

РЕФЕРАТ

Обсяг роботи 40 сторінок, 16 ілюстрацій, 13 джерел посилань.
АРБІТРАЖ-БОТ, БЛОКЧЕЙН, MEMPOOL, СМАРТ-КОНТРАКТ
Об’єктом роботи є системи для автоматичного виконання арбітражу на децентралізованих обмінниках. Предметом роботи є реалізація арбітраж-боту з використанням mempool для блокчейнів, які працюють на основі Ethereum Virtual Machine.

Метою роботи є створення архітектури арбітраж-боту, який використовує дані з mempool для швидшого виконання арбітражу, його програмна реалізація та підбір необхідних засобів та технологій для його реалізації.

Інструменти, які були використані під час розробки: інтегровані середовища розробки JetBrains WebStorm, Microsoft VSCode, Remix IDE; провайдер доступу до блокчейн-вузлів QuickNode API; провайдер даних з mempool BlockNative MemPool API.

Результати роботи: розроблена архітектура арбітраж-боту, імплементована логіка боту, створені тести для перевірки правильності виконання дій бота. Порівняно 3 способи виконання обчислень для пошуку оптимального значення flash-loan, та знайдено недоліки в безкоштовних сервісах, які можуть використовуватися для імплементації створеної архітектури. Розроблене програмне забезпечення можу використовуватися як базовий проект для розробки ще більш оптимізованої версії арбітраж-боту, який може використовуватися для заробітку на децентралізованих біржах.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ 4
ВСТУП 5
1 ТЕОРЕТИЧНІ ВІДОМОСТІ 8
1.1 БЛОКЧЕЙН 8
1.2 СМАРТ-КОНТРАКТИ В ETHEREUM 9
1.3 КОНСЕНСУС 10
1.4 MEMPOOL 11
1.5 АРБІТРАЖ В БЛОКЧЕЙНІ 12
1.6 АВТОМАТИЗОВАНИЙ МАРКЕТ-МЕЙКЕР 13
1.7 BLOCKNATIVE MEMPOOL API 13
2 МАТЕМАТИКА АРБІТРАЖУ ДЕЦЕНТРАЛІЗОВАНИХ БІРЖ 15
2.1 ОПИС ПРОЦЕДУРИ АРБІТРАЖУ 15
2.2 МАТЕМАТИЧНЕ ФОРМУЛЮВАННЯ ЗАДАЧІ 16
3 АРХІТЕКТУРА БОТУ 19
3.1 ОПИС АРХІТЕКТУРИ 19
3.2 ОБМЕЖЕННЯ БЕЗКОШТОВНОГО MEMPOOL API 20
4 ОПТИМІЗАЦІЯ ВИКОРИСТАННЯ ГАЗУ 22
5 ДЕТАЛІ ІМПЛЕМЕНТАЦІЇ 26
5.1 TYPESCIPT-СЕРВЕР 26
5.2 СМАРТ-КОНТРАКТ 30
5.3 КОМЕНТАРІ ЩОДО АРХІТЕКТУРИ ТА РЕАЛІЗАЦІЇ АРБІТРАЖ БОТУ. 32
6 ОЦІНКА РОБОТИ 34
ВИСНОВКИ 37
ПОСИЛАННЯ 39

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

EVM (Ethereum Virtual Machine) – віртуальна машина блокчейну Ethereum, яка дозволяє виконувати код смарт-контрактів.

Mempool (скор. від memory pool) – частина програмного забезпечення блокчейн-вузлу. Основна функція цієї частини – збергіання транзакцій, котрі мають бути додані в наступний блок блокчейну.

Валідатори – віртуальні сутності, які беруть участь у протоколі консенсусу для підтримки роботи блокчейн-мережі.

AMM (automatic market makers) – автоматичні маркет-мейкери.

DEX (Decentralized Exchange) - децентралізована біржа.

Пара – пара валют/криптовалют на біржі.

Смарт-контракт – програма на блокчейні, яка може змінювати стан блокчейну.

ETH – нативний токен мережі Ethereum.

USDC - токен, який прив’язаний до вартості долару США.

PoS (Proof-of-Stake) – алгоритм консенсусу Ethereum.

Газ (eng. “gas”) – поняття для виміру кількості обчислень, які генерує виконання смарт-контракту.

ВСТУП

Оцінка сучасного стану об’єкта розробки. В квітні 2022 року, щоденний загальний обсяг проданої / купленої валюти на біржах обміну валют склав 7.5 триліонів доларів [1]. Такий великий обсяг обміну невідворотно створює можливість для арбітражу – заробітку на різниці в цінах на один і той самий актив на різних біржах валют. На жаль, більшістю подібних можливостей користуються фінансові фірми, котрі заради отримання максимального прибутку будують сервери якумога ближче до серверів централізованих бірж. Робиться це для того, щоб зменшити час надходження сигналу до біржі про купівлю/продаж активу, бо інші фірми також намагаються заробити на цій можливості. Головною проблемою в даному прикладі є те, що звичайні люди не можуть змагатися з фірмами через відсутність фінансів для побудови подібної системи, що робить подібну активність закритою для звичайних людей.

Але з 2018 року набули популярності децентралізовані біржі для обміну криптовалют, такі як Uniswap[2]. Більшість з них побудовані за однієї і тією ж архітектурою, де співвідношення активів для будь-якої пари валют виражається рівнянням x*y = k, де k – константа, а x та y позначають кількість активу для поточної пари валют. Ця архітектура розраховує на те, що невідповідності в ціні будуть виправлятися не централізовано, а саме за допомогою арбітражу. В той же час, на децентралізованих біржах кожен з учасників має приблизно однакові шанси на отримання грошової винагороди за виконання арбітражу – виграє той, хто зможе побудувати найбільш швидке програмне забезпечення для включення транзакції в блок блокчейну.

Програмне забезпечення, що дозволяє виконувати подібні операції, називають арбітраж-ботом. Через те, що подібні активності дозовляють заробити гроші, дуже часто розробники продають доступ до ботів за доволі високі суми.

Безкоштовні ж рішення часто є неефективними в плані швидкості або використання ресурсів.

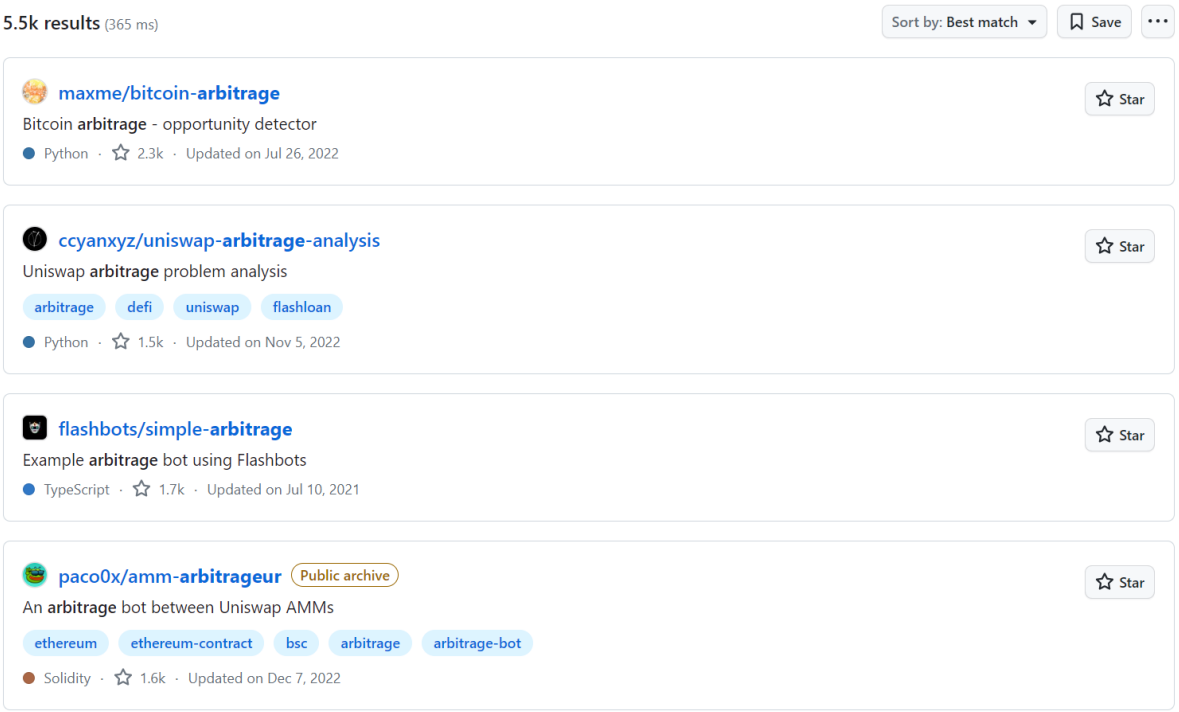

Рисунок 1 - Приклади безкоштовних ботів для арбітражу на github.com

Наша система використала ідею одного з безкоштовних ботів [3], і вона покращує використання ресурсів бота та інтегрує дані з mempool для того, щоб виконувати арбітраж, базуючись на інформації про транзакції, котрі очікують виконання.

Актуальність роботи. Робота актуальна через збільшення популярності децентралізованих бірж для обміну криптовалют та через неефективність існуючих безкоштовних рішень для виконання арбітражу на існуючих біржах. Також ця робота презентує архітектуру для побудови боту з використанням mempool даних.

Мета й завдання роботи. Метою роботи є ознайомлення з технологією блокчейн та смарт-контрактами, створення архітектури арбітраж-боту, який

використовує дані з mempool для швидшого виконання арбітражу, його програмна реалізація та підбір необхідних засобів та технологій для його реалізації. Розроблена система має правильно розраховувати параметри для виконання арбітражу та загальний прибуток з нього.

Об’єкт, методи й засоби розроблення. Об’єктом розробки є системи для автоматичного виконання арбітражу на децентралізованих обмінниках. Під час розробки використовувалися такі інструментальні засоби, як інтегровані середовища розробки JetBrains WebStorm, Microsoft VSCode, Remix IDE; провайдер доступу до блокчейн-вузлів QuickNode API; провайдер даних з mempool BlockNative MemPool API.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 БЛОКЧЕЙН

Блокчейн [12]- це розподілена технологія, що дозволяє створювати недвоємний журнал (ланцюжок блоків) для збереження та передачі інформації. Його основна особливість полягає у тому, що блокчейн забезпечує безпеку, надійність і недоступність для змін даних, які вже були занесені до ланцюжка.

Ланцюжок блоків складається з послідовно зв'язаних блоків, кожен з яких містить набір транзакцій. Кожен блок містить унікальний хеш, який служить ідентифікатором блоку та включає в себе хеш попереднього блоку, створюючи недвоємну зв'язність між блоками. Ця характеристика робить блокчейн майже незмінним і вразливим до змін.

Одна з головних переваг блокчейну полягає у тому, що він дозволяє безпосередню передачу цифрових активів або інформації між учасниками мережі без посередництва та додаткових витрат. Це може стосуватися фінансових транзакцій, смарт-контрактів, логістичних операцій та багатьох інших сфер. Крім того, блокчейн гарантує безпеку та непідробність даних завдяки криптографічним методам. Кожна транзакція або запис у блокчейні підписується цифровим підписом, що забезпечує його цілісність та автентичність. Це робить блокчейн надзвичайно надійним і малоймовірним для зламу або маніпуляцій.

Існує кілька видів блокчейнів, кожен з яких має свої унікальні особливості та застосування:

- Публічні блокчейни
- Приватні блокчейни
- Гібридні блокчейни

Приватні блокчейни зазвичай використовуються в комерційних та корпоративних секторах, де важлива конфіденційність даних та обмежений доступ до них. Вони контролюються обмеженим колом учасників і можуть мати різні

рівні дозволів для доступу до даних та виконання транзакцій. Приватні блокчейни надають можливість ефективно обробляти великий обсяг транзакцій, забезпечуючи при цьому конфіденційність та контроль.

Публічні блокчейни, навпаки, є відкритими та децентралізованими мережами, до яких може приєднатися будь-хто без обмежень. Вони забезпечують глобальний доступ до транзакцій та даних, а також надають можливість перевірки та підтвердження транзакцій унікальним способом, за допомогою механізму консенсусу. Публічні блокчейни використовуються для криптовалют, смарт-контрактів, голосування, відстеження ланцюга постачання та багатьох інших децентралізованих застосувань. Найвідоміший приклад публічного блокчейну - Bitcoin.

Окрім того, існують гібридні блокчейни, які поєднують в собі елементи приватних і публічних блокчейнів. Вони можуть бути використані, наприклад, в сферах, де потрібна комбінація конфіденційності та прозорості, або коли існує необхідність у співпраці між декількома організаціями з обмеженим колом учасників.

1.2 СМАРТ-КОНТРАКТИ В ETHEREUM

Смарт-контракт - це програмний код, який запускається на блокчейні і автоматично виконує угоди, засновані на певних заданих умовах. Вони вперше були впроваджені в Ethereum - одному з найбільш популярних платформ для розробки децентралізованих додатків.

Смарт-контракти в Ethereum побудовані на мові програмування Solidity і зберігаються у блокчейні. Вони визначають правила та умови, згідно з якими виконуються транзакції та розподіляються ресурси. Смарт-контракти можуть містити логіку для обробки фінансових транзакцій, управління власністю, реалізації голосування та взаємодії з іншими контрактами.

Однією з ключових особливостей смарт-контрактів є їхній автоматизований характер. Коли задані умови виконуються, контракт активується і виконує відповідні дії без необхідності довіряти посередникам або третім сторонам. Це робить процес укладення угоди більш надійним, швидким та ефективним.

Смарт-контракти в Ethereum також підтримують ідею децентралізованих програм, відомих як децентралізовані програми (DApps). Вони дозволяють розробникам створювати додатки, які працюють на основі блокчейну і можуть взаємодіяти з смарт-контрактами. Це дає можливість розробникам створювати різноманітні додатки, включаючи фінансові послуги, системи управління, розумних власність, ігри та багато іншого.

1.3 КОНСЕНСУС

Алгоритми консенсусу є ключовою складовою блокчейн технологій, які дозволяють досягати єдності та надійності в розподіленій мережі. Хоча на даний момент одними з найбільш прогресивних вважають мережі, які використовують алгоритми для розв’язку задачі Візантійських генералів, алгоритм консенсусу, використований в Ethereum, називається Proof-of-Stake, і не належить даному сімейству алгоритмів (рис. 2).

В основі PoS лежить ідея того, що вироблення нових блоків та підтвердження транзакцій залежить від долі, яку користувачі володіють у мережі. Замість використання малих комітетів для розв’язку задачі Візантійських генералів, PoS використовує стейкінг - процес, в якому учасники зберігають свої монети у спеціальних гаманцях як доказ власної відповідальності та зацікавленості у добробуті мережі. Використання алгоритму PoS дозволяє знизити витрати на споживання електроенергії, пов'язані з майнінгом, і збільшити масштабованість мережі. Зазвичай, генерація одного PoS-блоку в мережі Ethereum займає в районі 15 секунд.

Property	PoS	PBFT
Node identity management	Open	Permissioned
Energy saving	Partial	Yes
Tolerated power of adversary	< 51% stake	< 33.3% faulty replicas
Example	Peercoin	Hyperledger Fabric

Рисунок 2 - Порівняння Proof-of-Stake та BFT алгоритмів консенсусу

1.4 MEMPOOL

Mempool - це структура даних, яка використовується вузлами мережі блокчейн для зберігання непідтверджених транзакцій, які очікують на включення в наступний блок. Транзакції спочатку транслюються в мережу, а потім перевіряються вузлами перед тим, як бути доданими до пулу пам'яті. Після того, як транзакція потрапляє до пулу пам'яті, валідатори можуть включити її в наступний блок, який вони видобувають, залежно від таких факторів, як комісія за транзакцію та її розмір.

Однією з головних переваг використання даних мемпула є те, що він надає більш актуальну інформацію про транзакції, які будуть включені в майбутній блок. Розрахунки поза блокчейном, які покладаються на історичні дані або зовнішні джерела, можуть неточно відображати поточний стан мережі і призводити до неоптимальних параметрів арбітраж-транзакцій. На противагу цьому, mempool-

дані надають інформацію про поточний стан мережі в реальному часі і можуть бути використані для відповідного коригування параметрів транзакцій. Крім того, використання mempool-даних може допомогти зменшити ризик втратити можливість заробити в арбітражній торгівлі. Оскільки ринок криптовалют дуже волатильний, а ціни можуть швидко змінюватися, навіть невеликі затримки в підтвердженні транзакцій можуть призвести до втрати можливостей для прибуткової торгівлі. Включаючи mempool-дані в оптимізацію арбітражного бота, трейдери можуть підвищити ймовірність успішних угод і знизити ризик втрачених можливостей.

1.5 АРБІТРАЖ В БЛОКЧЕЙНІ

Арбітраж в блокчейні - це процес використання різниці в цінах або умовах між різними ринками або обмінними платформами, щоб отримати вигоду від торгівлі активами або виконання угод. Це стратегія, яка спирається на швидку реакцію на розбіжності в цінах, що виникають між різними ринками.

Блокчейн, зокрема децентралізовані фінансові системи (DeFi), створює нові можливості для арбітражу. Це стало можливим завдяки доступу до глобальних ринків і активів через розподілені обмінні платформи та ліквідність, яка забезпечується децентралізованими біржами.

Одним з прикладів арбітражу в блокчейні є так званий "токен-токен" арбітраж. Це включає купівлю одного токена на одній обмінній платформі за низькою ціною та одночасну продажу його на іншій платформі за вищою ціною. Різниця в цінах дозволяє отримати прибуток в результаті такої операції.

Арбітраж може бути також застосований до фінансових інструментів, таких як ставки на грошовому ринку або інші фінансові контракти. Швидкий та автоматизований характер блокчейну дозволяє здійснювати розрахунки та торгівлю миттєво, що створює сприятливі умови для арбітражу.

Проте, арбітраж в блокчейні також стикається з викликами та ризиками. Наприклад, швидкість блокчейну може стати обмеженням для ефективного виконання арбітражних операцій, особливо коли велика кількість учасників намагаються скористатися розбіжностями цін. Крім того, наявність маніпуляцій на ринку або проблем з ліквідністю може ускладнити здійснення арбітражних операцій.

1.6 АВТОМАТИЗОВАНИЙ МАРКЕТ-МЕЙКЕР

Автоматизований маркет-мейкер [4] - це автономний торговий механізм, який усуває потребу в централізованій біржі для торгівлі користувачів і є основою децентралізовані біржі. Uniswap є першою платформою, яка використовує AMM. Багато з існуючих AMM, таких як SushiSwap та PancakeSwap, базуються на Uniswap [5], [6]. Ця робота фокусується на AMM, які використовують алгоритми Uniswap. Ми можемо здійснювати арбітраж між цими AMM, коли ціни на одну і ту ж пару токенів на різних AMM розходяться. Ми можемо інкапсулювати арбітражні транзакції в одну EVM транзакцію, щоб ми могли гарантувати, що ціна не зміниться протягом виконання арбітражу.

1.7 BLOCKNATIVE MEMPOOL API

BlockNative MemPool API - це сервіс, який надає дані в режимі реального часу про транзакції, які знаходяться в mempool в мережі блокчейн. Маючи доступ до даних mempool, користувачі можуть відстежувати статус своїх транзакцій, оцінювати оптимальну ціну та комісію, а також виявляти потенційні збої або реорганізації транзакцій.

BlockNative MemPool API використовує веб-хуки для доставки потоків даних про події транзакцій користувачам за допомогою POST-запитів. Користувачі можуть вказати адреси і мережі, які вони хочуть відстежувати, і отримувати сповіщення,

коли транзакція за цими адресами змінює свій статус. API підтримує різні блокчейн-мережі, такі як Ethereum та Polygon, а також різні механізми ціноутворення на газ, такі як EIP-1559 [13] та більш застарілі версії ціноутворення.

2 МАТЕМАТИКА АРБІТРАЖУ ДЕЦЕНТРАЛІЗОВАНИХ БІРЖ

2.1 ОПИС ПРОЦЕДУРИ АРБІТРАЖУ

Припустимо, ми хочемо провести арбітраж на парі токенів ETH/USDC. Пара ETH/USDC повинна існувати на декількох AMM в ланцюжку. Припустимо, USDC - це токен з реальною вартістю, в деномінації якого ми хочемо отримати прибуток. Після виконання арбітражу в нас залишиться лише USDC, в той час як токени ETH після арбітражу в нас не лишаться. Якщо обидва токени мають реальну вартість, ми можемо зарезервувати будь-який з них, але ми повинні бути послідовними і лишати лише один з них. Арбітраж можна здійснити за допомогою Flashloan [7] від UniswapV2. Також припустимо, що $pool1$ і $pool2$ - це дві пари, які мають два однакові токени на різних AMM. Як тільки ціна розходиться, ми можемо зробити арбітраж за допомогою смарт-контракту. Контракт розраховує ціну, виражену в ETH. Припустимо, що ціна ETH в $pool1$ нижче:

1. Позичаємо x ETH з $pool1$. Нам потрібно погасити борг перед $pool1$ незалежно від результату арбітражу. Заборгованість може бути деномінована в USDC. Flashloan повинен бути повернутий в тій же транзакції, що забезпечується смарт-контрактом. Якщо ми не можемо повернути гроші кредитору, мережа відхилить транзакцію і кредитор завжди отримає кошти назад.

2. Продаємо всі позичені ETH на $pool2$. Отримуємо $y2$ USDC.

3. Гасимо заборгованість перед $pool1$ за допомогою $y1$ USDC.

4. В результаті отримуємо прибуток $y_2 - y_1$ USDC. Наша мета - знайти x, при якому $y_2 - y_1$ максимізується. У наступному розділі ми покажемо, як цей x можна обчислити.

Також ми б хотіли відмітити той факт, що для виконання даного типу арбітражу не треба мати багато початкового капіталу, так як гроші для проведення арбітражу беруться в борг, і єдині витрати, які треба покрити на початку діяльності – це витрати на газ.

2.2 МАТЕМАТИЧНЕ ФОРМУЛЮВАННЯ ЗАДАЧІ

Припустимо, що спочатку $pool1$ має a_1 USDC і b_1 ETH, $pool2$ має a_2 USDC і b_2 ETH. Тоді в AMM, заснованих на Uniswap повинні виконуватися наступні 2 рівняння:

$$a_1 b_1 = k_1;\ a_2 b_2 = k_2 \qquad (1)$$

де k_1 та k_2 є константами, які залежать від кількіості криптовалюти в кожній з пар. В $pool1$, ми використовуємо USDC для того, щоб позичити ETH, тому кількість USDC збільшується на Δa_1, а кількість ETH зменшується на Δb_1. Так як формули (1) завжди мають виконуватися, маємо, що:

$$(a_1 + 0.997 * \Delta a_1)(b_1 - \Delta b_1) = k_1;$$

$$(a_1 + 0.997 * \Delta a_1) = \frac{k_1}{(b_1 - \Delta b_1)};$$

$$0.997 * \Delta a_1 = \frac{k_1}{(b_1 - \Delta b_1)} - a_1;$$

$$0.997 * \Delta a_1 = \frac{k_1 - a_1 b_1 + a_1 \Delta b_1}{(b_1 - \Delta b_1)};$$

$$\Delta a_1 = \frac{a_1 \Delta b_1}{0.997 * (b_1 - \Delta b_1)}$$

Множник 0.997 з’явився через комісію платформи у вигляді 0.3%. Аналогічно для $pool2$, кількість USDC зменшиться, а кількість ETH збільшиться. Маємо:

$$(a_2 - \Delta a_2)(b_2 + 0.997\Delta b_2) = k_2;$$

$$\Delta a_2 = \frac{0.997 * a_2 \Delta b_2}{(b_2 + 0.997 * \Delta b_2)}$$

Так як ми використовуємо весь ETH котрий ми отримуємо з $pool1$ для того, щоб обміняти їх на USDC в $pool2$, отримуємо $\Delta b_1 = \Delta b_2$. Маємо відмітити, що

$\Delta b_1 = \Delta b_2 = x$ з попереднього розділу; в той же час Δa_1 еквівалентна y_1, Δa_2 еквівалентна y_2.

Кінцевий прибуток можна вирахувати за формулою:

$$f(x) = \Delta a_2 - \Delta a_1 = \frac{0.997 * a_2 x}{(b_2 + 0.997 * x)} - \frac{a_1 x}{0.997 * (b_1 - x)} \quad (2)$$

Ми хочемо знайти значення x, за якого прибуток (2) максимальний. Для цього вирахуємо похідну від цієї функції та прирівняємо її до нуля:

$$f'(x) = 0.997 * \frac{a_2 b_2 + a_2 * 0.997 * x - 0.997 * a_2 x}{(b_2 + 0.997 * x)^2} - \frac{1}{0.997} \frac{a_1 b_1 + a_1 x - a_1 x}{(b_1 - x)^2}$$

$$f'(x) = 0.997 * \frac{a_2 b_2}{(b_2 + 0.997 * x)^2} - \frac{1}{0.997} \frac{a_1 b_1}{(b_1 - x)^2} = 0$$

$$f'(x) = \frac{0.997 * a_2 b_2 * 0.997 * (b_1 - x)^2}{(b_2 + 0.997 * x)^2} - \frac{a_1 b_1 * (b_2 + 0.997 * x)^2}{0.997 (b_1 - x)^2} = 0$$

Опускаємо знаменник, припускаючи що він не буде дорівнювати нулю:

$$f'(x) = 0.997^2 * a_2 b_2 * \left({b_1}^2 - 2 x b_1 + x^2\right) - a_1 b_1$$
$$* \left({b_2}^2 - 2 * 0.997 * x b_2 + (0.997 x)^2\right)$$
$$= x^2 (0.997^2 * a_2 b_2 - 0.997^2 * a_1 b_1)$$
$$- 2 b_1 b_2 x (0.997^2 * a_2 - 0.997 * a_1)$$
$$+ \left(0.997^2 * a_2 b_2 * {b_1}^2 - a_1 b_1 * {b_2}^2\right) = 0$$

Отримали квадратне рівняння відносно x. Розв'язок буде виглядати наступним чином:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad (3)$$

де $a = 0.997^2 * a_2 b_2 - 0.997^2 * a_1 b_1$,

$b = -2 b_1 b_2 (0.997^2 * a_2 - 0.997 * a_1)$,

$c = b_1 b_2 (0.997^2 * a_2 * b_1 - a_1 b_2)$

за умови $0 < x < b_1$

Розв’язок, що задовольняє умові, буде кількістю токенів, які ми будемо брати в борг з $pool1$.

3 АРХІТЕКТУРА БОТУ

3.1 ОПИС АРХІТЕКТУРИ

Система складається з чотирьох компонентів: Mempool API (сторонній сервіс від Blocknative [8]), сервер,написаний на TypeScript, арбітражний смарт-контракт та провайдер доступу до блокчейну (Web3-з'єднання). Загальний дизайн системи можна знайти на рис. 3.

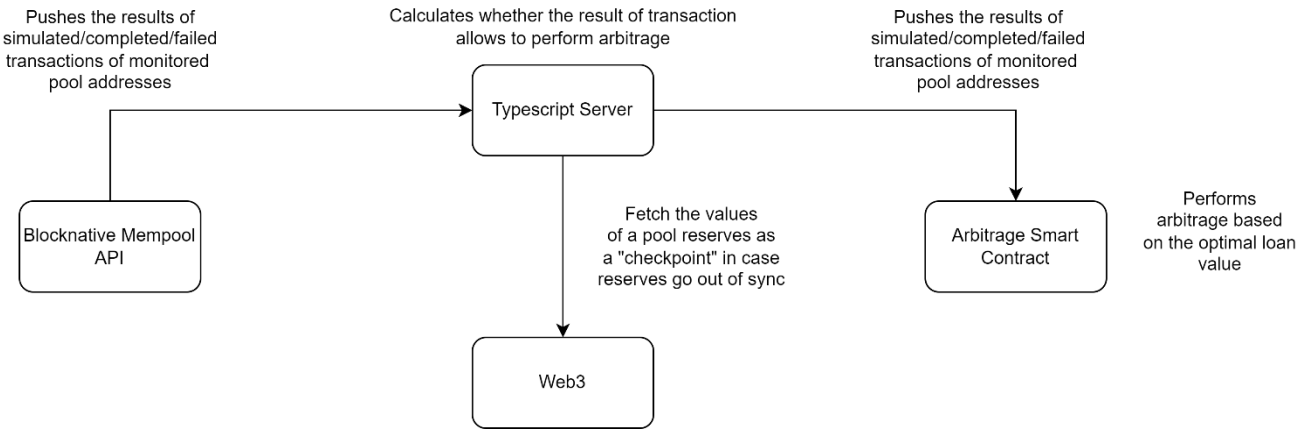

Рисунок 3 - Архітектура системи

Mempool API отримує транзакції з mempool, які впливають на пари, що відстежуються, і імітує їх виконання. Він обчислює чисті зміни в балансах токенів цих пулів після виконання транзакцій mempool. Сервер TypeScript слухає Mempool API і при кожній події зміни балансу обчислює, чи створює ця транзакція можливість для арбітражу, беручи до уваги загальну ціну газу, яку спалить арбітражна транзакція. Якщо поточні запаси дозволяють здійснити вигідний арбітраж, сервер TypeScript викликає смарт-контракт, який атомарно виконує арбітражну транзакцію.

Для того, щоб зрозуміти, чи створює подія зміни балансу прибуткову транзакцію, сервер TypeScript повинен підтримувати локальний стан балансу пулів, що відслідковуються. Однак, ми не можемо використовувати зміни балансу

від змодельованих транзакцій як єдиний засіб для оновлення станів резервів. Проблема полягає в тому, що Blocknative Mempool API виконує симуляції транзакцій на основі стану попереднього блоку. Це обмеження змушує систему оновлювати стан локальних резервів, використовуючи дані з блокчейну при створенні кожного нового блоку. Саме тоді в гру вступає з'єднання Web3, яке слугує безпечним джерелом даних про баланс смарт-контракту. Проблема більш детально обговорюється в наступному розділі “Обмеження безкоштовного mempool API”.

3.2 ОБМЕЖЕННЯ БЕЗКОШТОВНОГО MEMPOOL API

Blocknative Mempool API має обмеження: він не враховує можливість того, що деякі транзакції, які спочатку зазнають невдачі відносно стану попереднього блоку, можуть насправді бути успішними, якщо інша транзакція буде виконана до того, як буде виконана «провальна» транзакція. (рис. 4).

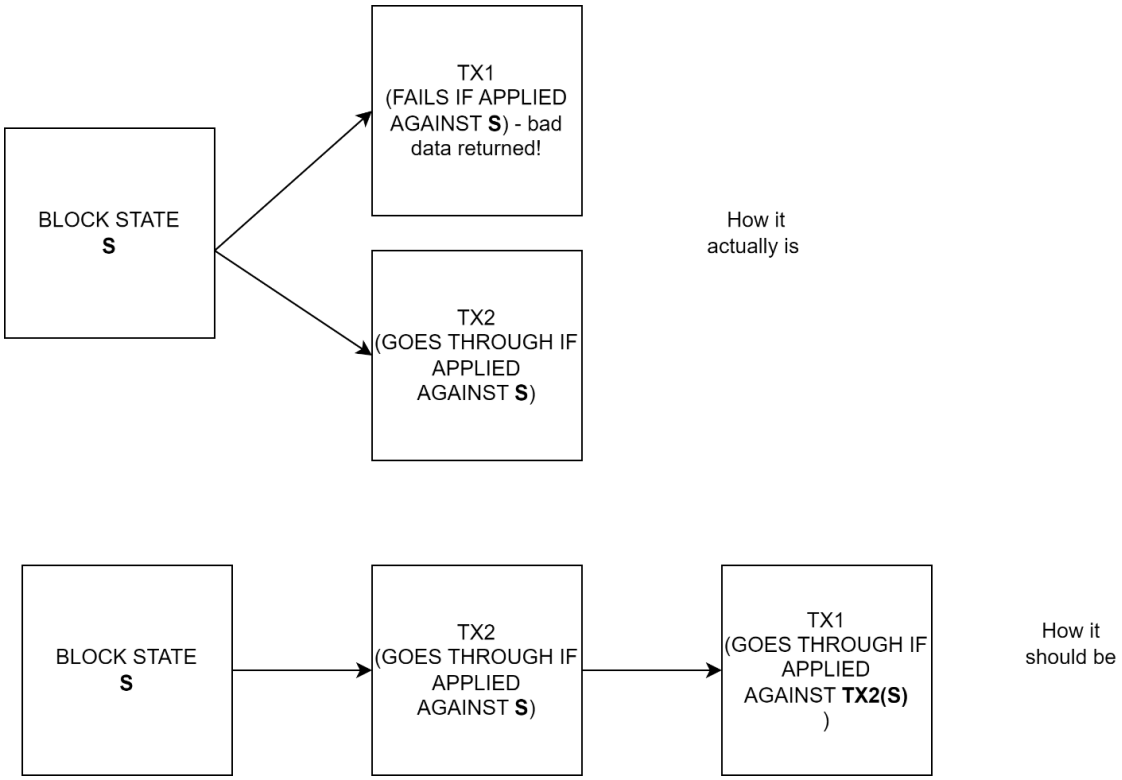

Рисунок 4 - Ілюстрація проблеми з безкоштовним mempool API

Так як Blocknative Mempool API намагається виконати кожну транзакцію відносно стану попереднього блоку, побудува арбітраж-транзакції спотворить локальну інформацію про ресурси смарт-контрактів, так як Blocknative Mempool API буде вважати її провальною і не поверне правильні значення. Через це, побудована система може виконувати арбітраж максимум один раз на блок.

Для того, щоб обійти дане обмеження, потрібна власна реалізація Mempool API. Подібне API має мати функціонал для послідовного виконання транзакцій, на відміну від Blocknative Mempool API, який виконує транзакції відносно стану попереднього блоку. На жаль, дана задача виходить за межі даної роботи через складність та потенційні фінансові витрати на створення власного блокчейн-вузлу.

4 ОПТИМІЗАЦІЯ ВИКОРИСТАННЯ ГАЗУ

Ми зробили три ітерації при розробці бота, намагаючись оптимізувати використання газу для відправки транзакцій і для його розгортання в наступних ітераціях. Основна ідея полягає в тому, що валідатори з більшою ймовірністю включать транзакцію в блок, якщо комісія за одиницю обчислень буде більшою, ніж в інших транзакціях.

а) Перший прохід: Початкова ідея полягала в тому, щоб включити всі обчислення (наприклад, функія getProfit(), яка використовує код для вирішення квадратного рівняння) в смарт-контракт і мати мінімум коду на стороні Typescript серверу. При цьому виникли наступні проблеми.

1. У мові програмування для смарт-контрактів Solidity немає підтримки чисел з плаваючою комою. Для розв’язання даної проблеми нам потрібно було використовувати тип *uint256* для зберігання чисел, помножених на степінь десяти, і все одно потрібно бути обережними з переповненням. Точність все одно могла бути низькою.

2. Вбудованої функції для обчислення квадратних коренів в Solidity немає, тому для наближення розв'язку потрібно використовувати метод Ньютона, що може призвести до збільшення споживання газу через велику кількість ітерацій. Перед будь-якими обчисленнями нам потрібно було розділити всі числа a_1, b_1, a_2, b_2 на велике число 10^i , щоб обчислення не переповнили 256-бітове ціле число, і нам потрібно помножити це назад у розв'язок, щоб отримати наближений цілочисельний розв'язок, досить близький до точного. Після зменшення на 10^i, щоб уникнути проблеми переповнення, ми розглянемо декілька способів розв'язання квадратного рівняння $g(x) = ax^2 + bx + c$, де a, b, c визначені в (3):

1. Використати метод Ньютона до $g(x)$, та ітеративно розв'язати рівняння за допомогою формули $x_{t+1} = x_t - \frac{g(x_t)}{g'(x_t)} = x_t - \frac{a(x_t)^2 + bx_t + c}{2ax_t + b}$ до тих пір, поки метод не зійдеться ($x_{t+1} = x_t$). Початкова умова $x_0 = \pm c$. Так як ми не знаємо, який саме з коренів задовольнить умови розв'язку, ми маємо шукати розв'язок з двох різних початкових значень.
2. Знову використати метод Ньютона, але для знаходження квадратного кореню. Знайдений квадратний корінь потім використати для того, щоб розв'язати квадратне рівняння стандартним способом. Використана для цього формула: $x_{t+1} = \frac{1}{2}(x_t + \frac{x_0}{x_t})$. Вважаємо, що метод зійшовся, якщо $x_{t+1} = x_t$.
3. Невелике покращення до 2 пункту. Замість недетермінованої кількості ітерацій, ми використовуємо властивість 256-бітових цілих чисел. Щоб отримати квадратний корінь з 256-бітових цілих чисел, ми спочатку отримуємо перше наближення за допомогою бітових зсувів вліво числа (ділення на 2). Для цього потрібно не більше 7 ітерацій зсуву бітів. Після того, як ми гарантовано зменшили число до розміру 128 бітів, використаємо метод Ньютона (рис.5). Даний метод використовують бібліотека ABDK [9]

б) Другий прохід: ми використовували компілятор для Solidity версії 0.8.7 та бібліотеку SafeMath для Uniswap V2 разом. Оскільки версія Solidity, новіша за 0.8.0 підтримує оператори переповнення, у нас є змога використати один з двох інструментів. Ми виявили, що використання вбудованої у мову

перевірки на переповнення трохи дорожче, тому ми вирішили скористатися бібліотекою SafeMath і зменшили використання газу до 206,970.

```
function sqrt2(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x == 0) return 0;
        else {
            uint256 xx = x;
            uint256 r = 1;
            if (xx >= 0x100000000000000000000000000000000) { xx >>= 128; r <<= 64; }
            if (xx >= 0x10000000000000000) { xx >>= 64; r <<= 32; }
            if (xx >= 0x100000000) { xx >>= 32; r <<= 16; }
            if (xx >= 0x10000) { xx >>= 16; r <<= 8; }
            if (xx >= 0x100) { xx >>= 8; r <<= 4; }
            if (xx >= 0x10) { xx >>= 4; r <<= 2; }
            if (xx >= 0x4) { r <<= 1; }
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1; // Seven iterations should be enough
            uint256 r1 = x / r;
            return r < r1 ? r : r1;
        }
    }
```

Рисунок 5 - Імплементація пошуку квадратного кореню

в) Третій прохід: Коли ми переглядали код Typescript, який розгортає смарт-контракт і запускає функції смарт-контракту, ми бачимо, що деякі обчислення, включаючи розв’язання квадратних рівнянь і функції UniswapV2, такі як getAmountIn і getAmountOut, не обов'язково повинні бути в смарт-контракті, і їх можна зробити на Typescript-сервері, трохи пожертвувавши при цьому швидкістю відправки транзакції. За допомогою цієї модифікації ми зменшили використання газу до 177 700 без оптимізатора смарт-контрактів та 173 394 з оптимізатором смарт-контрактів. Ми також зменшили кількість газу, що використовується для розгортання, з 4,944,282 до 1,293,338. Це також підвищує точність обчислюваних рішень, оскільки Typescript підтримує обчислення з плаваючою точкою і може

зберігати і застосовувати операції над будь-якими як завгодно великими числами за допомогою бібліотеки BigNumber.

5 ДЕТАЛІ ІМПЛЕМЕНТАЦІЇ

5.1 TYPESCIPT-СЕРВЕР

В побудованій архітектурі, Typescript-сервер виступає в ролі «спостерігача» за балансами пар, які нас цікавлять. Якщо він фіксує можливість заробити, то він відправляє сигнал смарт-контракту (який виступає в ролі «брокера»), який виконує арбітраж на блокчейні.

Щоб спостерігати за балансами токенів, Typescript-сервер оперує даними, що надаються від MemPool API для гри навипередження, а також даними з блокчейну для підтримки стабільної роботи системи. На рис. 6 можемо побачити, яким чином відбувається підключення до MemPool API. Використовується паттерн PubSub, де замість постійного опитування виробника даних ми підписуємося на нього, чекаючи на момент коли трапиться івент, який нас цікавить (в нашому випадку, нас цікавлять всі івенти що пов’язані з конкретною парою). На рис. 7 можемо побачити, як ми отримуємо дані з блокчейну з домогою бібліотеки Ethers.

```
// Watch for events on specific address, call callback using reserve changes
let watchAddress = async (pairInfo: Pool, callback: (update: ReserveUpdate, address: string) => void, network = "homestead", log = true) => {
  options.networkId = networkToChainId[network]
  // initialize and connect to the api
  const blocknative = new BlocknativeSdk(options)

  // call with the address of the account that you would like to receive status updates for
  const {
    emitter, // emitter object to listen for status updates
    details // initial account details which are useful for internal tracking: address
  } = blocknative.account(pairInfo.address)

  if (log) console.log(`Watching: ${details.address}`)

  // Register event listener, call callback with reserve changes
  emitter.on("all", (ev) => {
    let changes = handleEvent(ev, pairInfo, log);
    let update: ReserveUpdate = {
      token0Address: pairInfo.token0Address.toLowerCase(),
      token1Address: pairInfo.token1Address.toLowerCase(),
      reserve0Delta: changes.token0change,
      reserve1Delta: changes.token1change
    }
    callback(update, pairInfo.address)
  })
}
```

Рисунок 6 - Прослуховування MemPool API

```
async getInfoFromUniswapBasedContract(pairAddress: string): Promise<Pool> {
  const uniSwapBasedContract = new ethers.Contract(pairAddress, contractAbi, this._signer)

  const token0 = await uniSwapBasedContract.token0();
  const token1 = await uniSwapBasedContract.token1();
  const reserves = (await uniSwapBasedContract.getReserves()) as PoolReservesShort;

  return {
    address: pairAddress,
    token0Address: token0.toLowerCase(),
    token1Address: token1.toLowerCase(),
    reserve0: reserves._reserve0,
    reserve1: reserves._reserve1
  }
}
```

Рисунок 7- Отримання резервів пари з блокчейну

Через обмеження безкоштовного тарифу MemPool API, ми маємо контролювати, наскільки часто виконується арбітраж. Для цього ми використовуємо два «наївних» м'ютекси (рис. 8), котрі використовують булевий флаг як індикатор того, чи можна виконати наступний арбітраж(критична секція). Причина вибору «наївного» м'ютексу – ми хочемо, щоб асинхронна задача не очікувала входу в критичну секцію, а просто ігнорувала можливість виконати арбітраж, якщо арбітраж вже був виконаний в цьому блоці / транзакція ще не була підтверджена в блокчейні. Перший м'ютекс потрібен для того, щоб уникнути «лавинного ефекту», коли MemPool API відправляє на сервер інформацію про транзакцію, яку ми тільки що відправили як арбітражну, тим самим провокуючи сервер відправити ще одну транзакцію, яку MemPool API знову відправить на сервер, і так до тих пір, поки на гаманці не скінчаться гроші на відправку нових транзакцій або буде створений новий блок.

```
let tokensSorted = [pool.token0Address, pool.token1Address];
tokensSorted.sort()

log.debug(`Is arbitraged locked: ${arbitrageLock.locked}`)
for (const other_pool of tokensToPairs.get(tokensSorted)!) {
  if (other_pool.address != pool.address && !arbitrageLock.locked && !blockLock.locked) {
      arbitrageLock.locked = true
      blockLock.locked = true
      // Call aribtrageFunc using current reserves
      await arbitrageFunc(
          flashBot,
          baseTokens,
          isBaseTokenSmallerLocalReserves(reserves),
          getOrderedReservesLocalReserves(reserves),
          {
            symbols: "RND_PAIR",
            pairs: [pool.address.toLowerCase(), other_pool.address.toLowerCase()],
          }
      );
      arbitrageLock.locked = false
  }
}
```

Рисунок 8 - Використання "наївних" м'ютексів для перевірки на можливість виконання арбітражу

Під час виконання арбітражу, ми спочатку оцінюємо його прибутковість. Для цього ми викликаємо функцію *getProfit() (*рис. 9), котра знаходить розв'язок до квадратного рівняння (3), та виконує dry-run арбітражної транзакції для того, щоб зрозуміти, чи дійсно цей розв'язок є прибутковим. Цей крок можна прибрати для пришвидшення роботи системи, але через можливі фінансові втрати рекомендується робити подібні перевірки.

Для виконання dry-run на стороні серверу, ми переписали функції *getAmountIn()* та *getAmountOut()* смарт-контракту UniswapV2 на мову Typescript. Ці функції перевіряють, скільки токенів нам доведеться віддати як сумму боргу після виконання арбітражу та скільки грошей ми отримаємо до моменту віддачі боргу

відповідно. Якщо різниця між двома значеннями негативна, ми повертаємо нульове значення як прибуток від транзакції (рис. 10).

```
// get gross profit based on current state of DEXes
try {
  res = await getProfit(
      pair0,
      pair1,
      isBaseTokenSmallerFunc,
      getOrderedReservesFunc,
  )
  // console.log(`Profit on ${pair.symbols}: ${ethers.utils.formatEther(res.profit)}`)
  log.debug(`Profit on ${pair.symbols}: ${ethers.utils.formatEther(res.profit)}`);
} catch (err) {
  log.debug(err);
  return;
}
```

Рисунок 9 - Виклик функції розрахунку потенційного прибутку

```
const debtAmount: BigNumber = getAmountIn(
    borrowAmount,
    orderedReserves.lowerPricePoolBaseToken,
    orderedReserves.lowerPricePoolQuoteToken
)
const baseTokenGrossProfit = getAmountOut(
    borrowAmount,
    orderedReserves.higherPricePoolBaseToken,
    orderedReserves.higherPricePoolQuoteToken,
)
log.debug(`Gross profit: ${baseTokenGrossProfit.toString()}, debt amount: ${debtAmount.toString()}`)
if (baseTokenGrossProfit.lt(debtAmount)) {
    return {
        profit: BigNumber.from(0),
        baseToken
    }
} else {
    return {
        profit: baseTokenGrossProfit.sub(debtAmount),
        baseToken
    }
}
```

Рисунок 10 - Dry-run перевірка на прибутковість

Якщо транзакція вважається прибутковою, то система оцінює затрати на газ, які будуть викликані цією транзакцією (рис. 11).

```
async function calcNetProfit(profitWei: BigNumber, address: string, baseTokens: Tokens): Promise<number> {
    let price = 1;
    if (baseTokens.other.address == address) {
      price = await getEthPrice();
    }
    let profit = parseFloat(ethers.utils.formatEther(profitWei));
    profit = profit * price;

    const gasCost = price * parseFloat(ethers.utils.formatEther(config.gasPrice)) * (config.gasLimit as number);
    // const gasCost = 0
    console.log(`${profit}, ${gasCost} `)
    return profit - gasCost;
  }
```

Рисунок 11 - Розрахунок ціни газу

Якщо результуючий прибуток більше за константу, яку визначає користувач, транзакція відправляється в блокчейн на виконання.

5.2 СМАРТ-КОНТРАКТ

Смарт-контракт, котрий виконує арбітраж, написаний на мові Solidity, котра є де-факто стандартом для блокчейнів, заснованих на Ethereum Virtual Machine.

Так як робота ставила перед собою ціль оптимізувати використання газу смарт-контрактом, то в ньому лишився лише функціонал, необхідний для виконання арбітражу. Зокрема, це виклик функції для отримання займу, функція для обміну токенів, котрі ми взяли в борг, та власне повернення боргу.

Точкою входу в смарт-контракт є функція *flashArbitrage(),* котра на вхід, окрім інших параметрів, отримує параметр *data* (рис. 12). Якщо цей параметр має ненульовий розмір (у байтах), то смарт-контракт пари розуміє, що користувач хоче зайняти гроші за допомогою flashloan. В цьому випадку цей параметр кодує аргументи для callback-функції, який буде викликаний одразу після отримання грошей. Після виконання callback-функції, в якому ми маємо виконати арбітраж і

повернути позику з відсотком, ми перевіряємо, чи заробили ми гроші. Якщо ні – ми відміняємо транзакцію, і втрачаємо лише гроші, які маємо заплатити за газ.

```
/// @notice Do an arbitrage between two Uniswap-like AMM pools
/// @dev Two pools must contains same token pair
function flashArbitrage(
    address lowerPool,
    address baseToken,
    uint256 amount0Out,
    uint256 amount1Out,
    bytes calldata data
) public {
    // this must be updated every transaction for callback origin authentication
    permissionedPairAddress = lowerPool; // lower price pair
    uint256 balanceBefore = IERC20(baseToken).balanceOf(address(this));
    // borrowing (flash swap) happens here -> will callback to UniswapV2Call
    IUniswapV2Pair(lowerPool).swap(amount0Out, amount1Out, address(this), data);
    uint256 balanceAfter = IERC20(baseToken).balanceOf(address(this));

    require(balanceAfter > balanceBefore, 'Losing money');
```

Рисунок 12 - Взяття позики та фінальна перевірка

В callback-функції (*uniswapV2Call(),* рис. 13) спочатку перевіряється, чи авторизований ініціалізатор виклику функції використовувати цю callback-функцію. Після цього переводяться токени на рахунок пари за допомогою методу *SafeTransfer()*.Метод *swap()* зараховує токени на рахунок арбітраж-боту, і в кінці ми маємо повернути борг, який взяли на початку транзакції. Другий виклик до *SafeTransfer()* відповідає поверненню боргу в перший пул.

Після закінчення виконання функції *uniswapV2Call(),* ми повертаємося в функцію *flashArbitrage(),* і перевіряємо баланс рахунку смарт-контракту.

```
function uniswapV2Call(
    address sender,
    uint256 amount0,
    uint256 amount1,
    bytes memory data
) public {
    // access control
    require(msg.sender == permissionedPairAddress, 'Non permissioned address call');
    require(sender == address(this), 'Not from this contract');

    uint256 borrowedAmount = amount0 > 0 ? amount0 : amount1;
    CallbackData memory info = abi.decode(data, (CallbackData));

    IERC20(info.borrowedToken).safeTransfer(info.targetPool, borrowedAmount);

    (uint256 amount0Out, uint256 amount1Out) =
        info.debtTokenSmaller ? (info.debtTokenOutAmount, uint256(0)) : (uint256(0), info.debtTokenOutAmount);
    // ordinary swap -> $$$ done here
    IUniswapV2Pair(info.targetPool).swap(amount0Out, amount1Out, address(this), new bytes(0));
    // pay your debt
    IERC20(info.debtToken).safeTransfer(info.debtPool, info.debtAmount);
}
```

Рисунок 13 - Callback-функція

5.3 КОМЕНТАРІ ЩОДО АРХІТЕКТУРИ ТА РЕАЛІЗАЦІЇ АРБІТРАЖ БОТУ.

Наведена архітектура та приклади імплементації, які були обговорені вище, можуть бути адаптовані під будь-який блокчейн, який базується на EVM, тому за необхідності наступні роботи можуть перевикористати дану архітектуру. Відкритим питанням лишається порівняння середнього часу на відправку транзакції в залежності від мови програмування, на якій написаний сервер.

Треба також відмітити, що мінусом даної архітектури є залежність від провайдеру даних про ціну газу, так як можливі ситуації, коли цей сервіс буде недоступним, і через це прибуткові арбітраж-транзакції не будуть відправлені.

Фінальну діаграму послідовностей для виконання арбітражу можна знайти на рисунку 14.

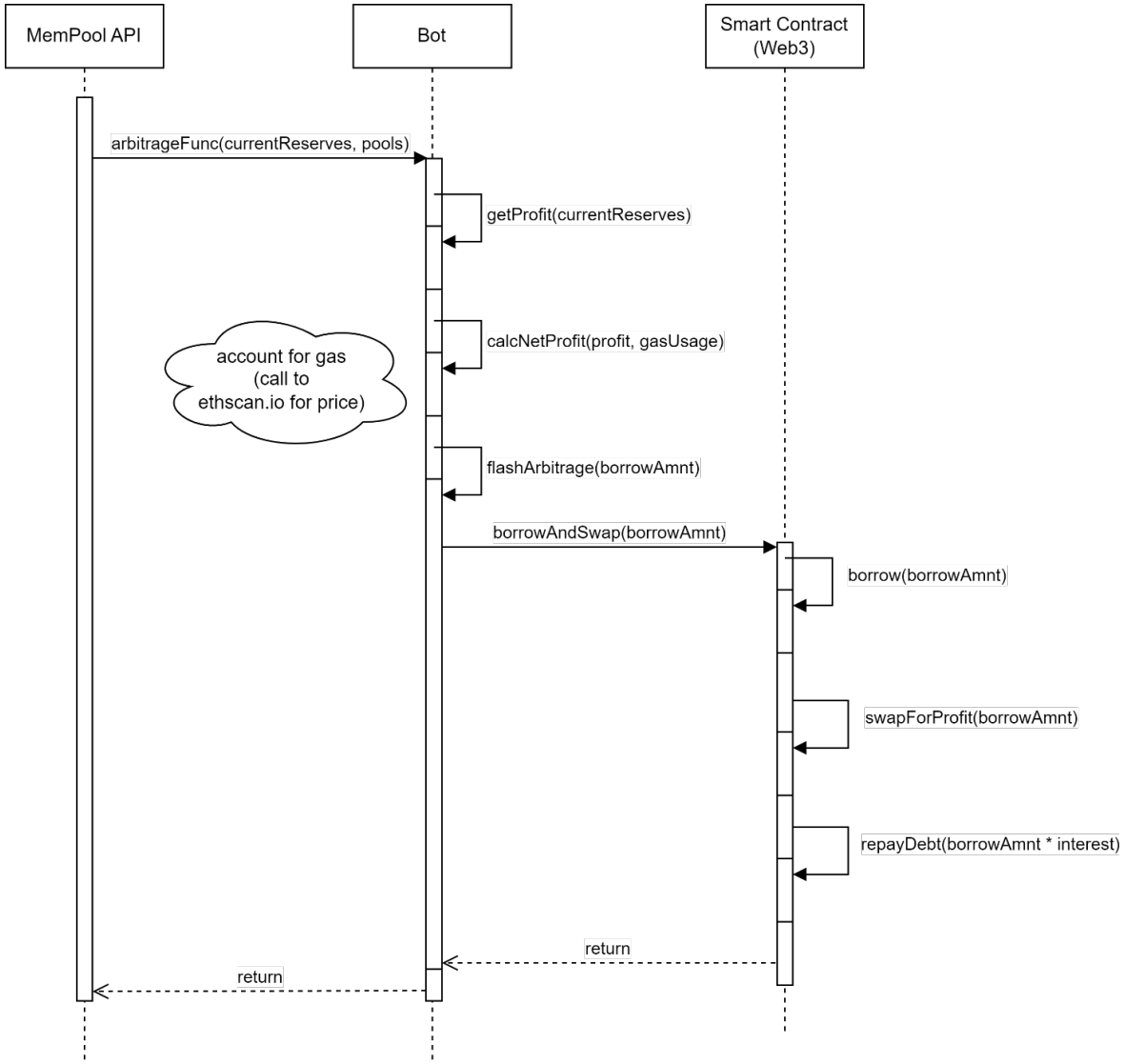

Рисунок 14 - Діаграма послідовностей виконання арбітражу

6 ОЦІНКА РОБОТИ

Ми протестували нашого бота на тестовій мережі Ethereum Goerli. Бот зміг зафіксувати арбітражні можливості, які були штучно створені (приклад арбітражної транзакції [10], рис. 7). Бот зміг зафіксувати транзакції, коли вони перебували в mempool, що дозволило зафіксувати можливості для арбітражу до того, як вони були записані в блокчейн.

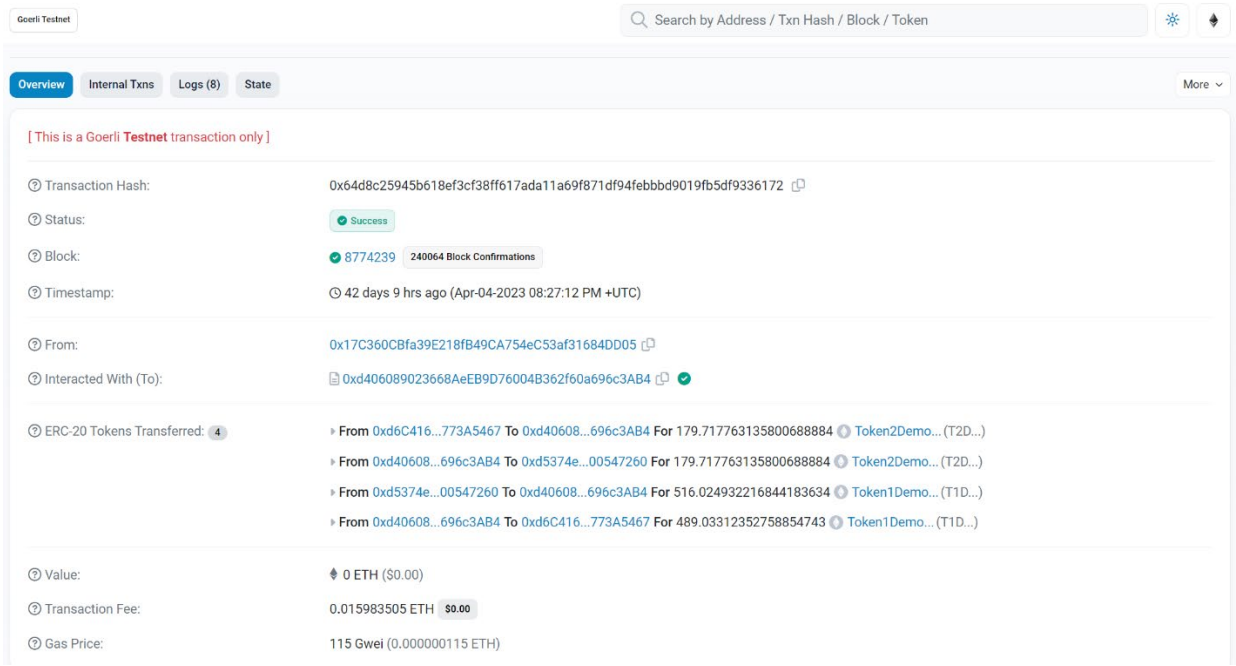

Рисунок 15 - Успішний результат виконання арбітражу

Ми також оцінили перевагу використання API Mempool Blocknative, спостерігаючи за 100 пулами UniswapV2 в основній мережі Ethereum для транзакцій, обмежених максимальною щоденною кількістю змодельованих транзакцій у безкоштовному тарифі Blocknative Mempool API. Зі 152 зареєстрованих транзакцій 57% були спочатку виявлені в Mempool, а потім успішно підтверджені в тому ж блоці, 37% були вперше помічені вже в блоці, а решта транзакцій зазнали невдачі. Час, що минув між симуляцією та

підтвердженням, можна побачити на рис. 8. Дві транзакції з великою затримкою в 32 і 152 секунди не показані для кращої наочності.

Медіана часу, що минув, становить 8,4 секунди, середній час, що минув, - 10,6 секунди. Середній час на блок - 12,2 секунди станом на 17 квітня 2023 року

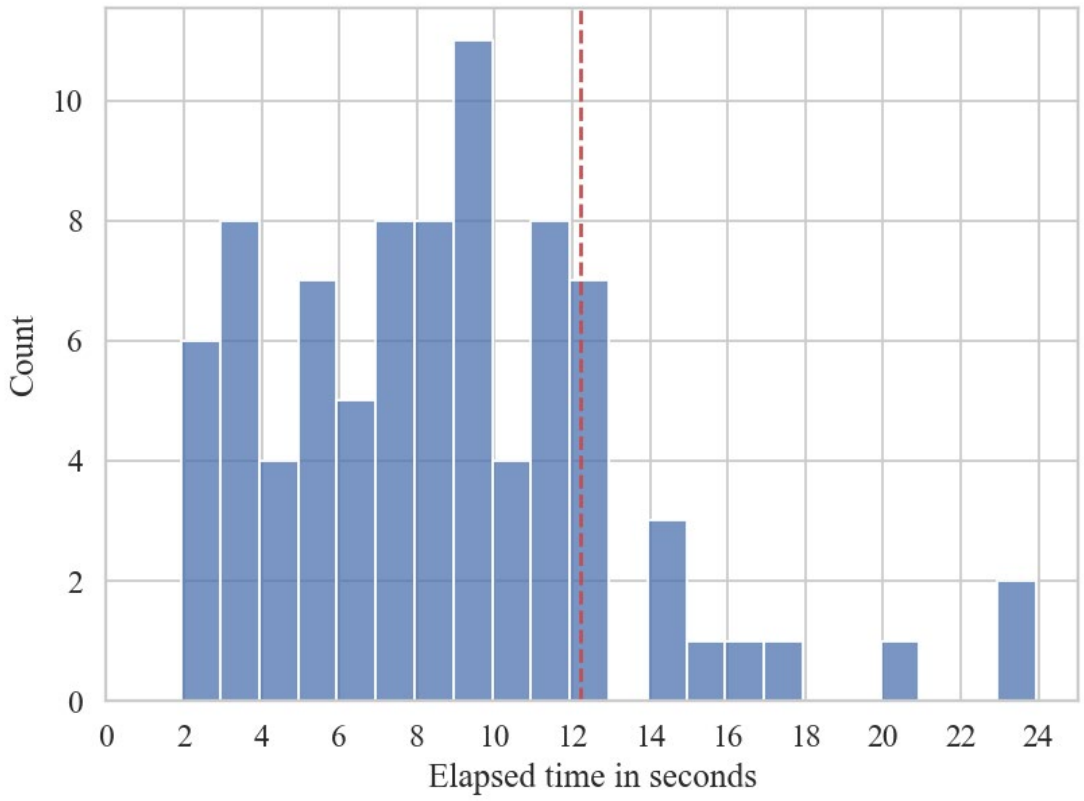

Рисунок 16 - Кількість часу між симуляцією транзакції та її включенням в блок. Пунткрина червона лінія показує середній час для генерації блоку.

Враховуючи той факт, що більш ніж половина арбітраж-транзакцій були успішно додані в той же блок, в якому їх знайшов BlockNative Mempool API, автори вважають результати задовільними.

В наступних роботах для покращення цих метрик має сенс імплементувати низькорівневі оптимізації для пришвидшення модулю розв’язання квадратичного рівняння (наприклад, використати більш швидку мову програмування, як C++ [11]), так як більшість сучасних мов програмування дозволяє підключити вже

зкомпільовані модулі та використовувати їх разом з кодом цієї мови програмування.

ВИСНОВКИ

У цій роботі ми представили реалізацію арбітраж-боту для протоколу Uniswap V2 на тестовому блокчейні Ethereum Goerli з використанням Solidity, TypeScript та Blocknative MemPool API.

Наша система складається з чотирьох компонентів: API Mempool, сервер TypeScript, арбітражний смарт-контракт та он-лайн провайдер даних. Бот прослуховує події зміни балансу, обчислює, чи існує арбітражна можливість, і атомарно виконує арбітражну операцію за допомогою смарт-контракту, якщо вона є вигідною.

Наша оцінка Mempool API показала, що ми можемо отримати значну перевагу в часі для більшості транзакцій. Одним з основних обмежень є те, що API виконує симуляцію транзакцій на основі стану попереднього блоку, що може призвести до повернення невірних змін балансу та відправки невдалих транзакцій. Ми визначили потребу в кастомній реалізації Mempool API, яка імітує транзакції на основі стану, отриманого після виконання попередніх транзакцій. Однак ця реалізація виходить за рамки нашого проекту.

Під час розробки нашого бота ми також зіткнулися з проблемою оптимізації використання газу. Наша початкова ідея полягала в тому, щоб включити всі необхідні обчислення в смарт-контракт, але такий підхід мав недоліки через відсутність нативної підтримки чисел з плаваючою комою і вбудованих функцій для обчислення квадратних коренів в Solidity. Ми ітерували наш дизайн, щоб мінімізувати використання газу для відправки транзакцій і розгортання бота в блокчейні.

Подальша робота включає пошук альтернативних Mempool API та подальшу оптимізацію використання газу. Ми також плануємо оцінити прибутковість нашого бота в реальних сценаріях і порівняти його з існуючими арбітражними ботами, щоб оцінити його ефективність. Нарешті, ми плануємо розгорнути блокчейн-вузол для оптимізації роботи бота. Використання власного вузла дозволить нам мати

прямий доступ до даних блокчейну, оминаючи потребу в зовнішньому постачальнику даних. Це потенційно може зменшити затримку і підвищити надійність даних, що призведе до більш точної ідентифікації арбітражних можливостей і більш швидкого виконання арбітражу.

ПОСИЛАННЯ

[1] OTC foreign exchange turnover in April 2022 [Електронний ресурс] – Режим доступу до ресурсу: https://www.bis.org/statistics/rpfx22_fx.htm

[2] Uniswap [Електронний ресурс] – Режим доступу до ресурсу:- https://uniswap.org/

[3] *AMM-arbitrageur* [Електронний ресурс] – Режим доступу до ресурсу: - https://github.com/paco0x/amm-arbitrageur

[4] What Is an Automated Market Maker (AMM)? [Електронний ресурс] – Режим доступу до ресурсу: https://www.gemini.com/cryptopedia/amm-what-are-automated-market-makers

[5] SushiSwap [Електронний ресурс] – Режим доступу до ресурсу: https://www.sushi.com/

[6] PancakeSwap [Електронний ресурс] – Режим доступу до ресурсу: https://pancakeswap.finance/

[7] Wang, D., Wu, S., Lin, Z., Wu, L., Yuan, X., Zhou, Y., Wang, H. and Ren, K., 2021, травень. Towards a first step to understand flash loan and its applications in defi ecosystem. У матеріалах Дев'ятого міжнародного семінару з безпеки в блокчейні та хмарних обчисленнях (с. 23-28).

[8] Blocknative Mempool Explorer. [Електронний ресурс] – Режим доступу до ресурсу: https://explorer.blocknative.com

[9] ABDK library. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/abdk-consulting/abdk-libraries-solidity

[10] Приклад успішного успішної транзакції виконаної створеним арбітражний ботом [Електронний ресурс] – Режим доступу до ресурсу: https://goerli.etherscan.io/tx/0x64d8c25945b618ef3cf38ff617ada11a69f871df94febbbd9019fb5df9336172

[11] C++ (мова програмування) [Електронний ресурс] – Режим доступу до ресурсу:. https://en.wikipedia.org/wiki/C%2B%2B

[12] Блокчейн [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Blockchain

[13] EIP-1559 [Електронний ресурс] – Режим доступу до ресурсу: https://eips.ethereum.org/EIPS/eip-1559