

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки України

Кваліфікаційна наукова
праця на правах рукопису

ЖИКІН ЮРІЙ СЕРГІЙОВИЧ

УДК 004.65:519.17

ДИСЕРТАЦІЯ

МЕТОДИ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ АНАЛІЗУ ГРАФА БІТКОЇН-ТРАНЗАКЦІЙ

121 Інженерія програмного забезпечення

12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

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

_____ Ю. С. Жикін

Науковий керівник Онай Микола Володимирович, кандидат технічних наук,
доцент

Київ – 2026

АНОТАЦІЯ

Жикін Ю. С. Методи та програмне забезпечення аналізу графа Біткоїн-транзакцій. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії в галузі знань 12 Інформаційні технології за спеціальністю 121 Інженерія програмного забезпечення. – Національний технічний університет України “Київський політехнічний інститут імені Ігоря Сікорського”, Київ, 2026.

Протягом майже двох десятиліть з моменту своєї появи Біткоїн став важливим компонентом глобальної фінансової системи. Біткоїн водночас є системою мікроплатежів та електронної готівки, активом приватних інвестицій, інструментом міжнародних розрахунків і навіть складовою державних фінансових стратегій. Незалежність Біткоїна від державних і банківських інституцій робить його привабливим фінансовим інструментом для користувачів у політично чи економічно нестабільних регіонах. Доступ індивідуального користувача до Біткоїна практично неможливо обмежити, оскільки для більшості операцій з Біткоїном достатньо навіть низькошвидкісного доступу до Інтернету, але у цьому контексті особливо важливим є питання анонімності користувачів і безпеки їхніх персональних даних.

Біткоїн, як і більшість інших криптовалютних систем, не є абсолютно анонімним. Біткоїн-протокол базується на відкритому реєстрі транзакцій, який містить всю історію операцій за весь час існування системи, і у цьому реєстрі кожен користувач представлений одним або багатьма псевдонімами, які називають адресами. Транзакції в Біткоїні утворюють складний граф переходів власності, і аналіз цього графа дає змогу встановлювати соціально-економічні зв'язки між користувачами чи організаціями, знаходити множини адрес, що належать одному й тому ж користувачу, і навіть деанонімізувати окремих користувачів. З одного боку такі можливості є корисними для правоохоронних органів у боротьбі з шахрайством, відмиванням коштів і іншою злочинною діяльністю, а також

для фінансових установ у моделюванні ринку та прогнозуванні економічних процесів. З іншого боку такі можливості ставлять під загрозу право індивідуальних користувачів на приватність їхніх фінансових даних. Втім, аналіз можливостей деанонізації користувачів допомагає виявляти слабкі місця у моделі приватності Біткоїна, що в свою чергу сприяє розробці нових протоколів та інструментів, спрямованих на покращення приватності користувачів.

Дослідження методів аналізу графа Біткоїн-транзакцій є актуальним як з точки зору окремого користувача з метою захисту персональних фінансових даних від несанкціонованого втручання, так і з точки зору правоохоронних органів та організацій з метою розпізнавання операцій, що можуть вказувати на потенційну злочинну діяльність, або прямої протидії фінансових злочинам. Розроблення відкритих програмних систем аналізу графа Біткоїн-транзакцій є корисним для підвищення безпеки всієї Біткоїн-екосистеми, як з точки зору індивідуального користувача, так і з точки зору правоохоронних органів.

Метою дисертаційної роботи є спрощення процесів розроблення та експлуатації програмного забезпечення для аналізу графа Біткоїн-транзакцій шляхом розроблення узагальненої архітектури програмної системи на основі спеціалізованої моделі транзакційних даних, яка надає уніфікований семантичний контракт для інтеграції програмних компонентів і забезпечує масштабоване оброблення історії транзакцій.

У **першому розділі** дисертаційної роботи розглянуто історичні й економічні передумови появи Біткоїна і вплив цих передумов на деякі аспекти Біткоїн-протоколу. Проведено аналіз Біткоїн-протоколу і його транзакційної моделі, розглянуто структуру транзакцій і механізм формування блоків. Проаналізовано проблему приватності користувачів і технології аналізу графа Біткоїн-транзакцій з метою деанонізації користувачів, а також розглянуто існуючі рекомендації та інструменти збереження приватності. Сформульовано загальну концепцію програмної

системи, що дає змогу будувати і розширювати граф знань про транзакції, а також здійснювати патерновий аналіз такого графа. Насамкінець, сформульовано вимоги до такої програмної системи.

У **другому розділі** розроблено спеціалізовану RDF-модель даних Біткоїн-транзакцій, адаптовану для пошуків патернів типових операцій. Для цього розглянуто підграфи графа Біткоїн-транзакцій і обрано граф, що відображає і транзакції, і транзакційні виходи, а також аргументовано вибір технології представлення великих графів, і графів знань зокрема. Розглянуто можливості використання SPARQL як універсальної мови запитів для патернового аналізу, а також розглянуто можливості розширення SPARQL спеціалізованими фільтр-функціями для спрощення складних запитів.

У **третьому розділі** запропоновано концепцію пошуку типових операцій у графі Біткоїн-транзакцій за допомогою евристичних патернів, а також її реалізацію з використанням RDF-графа знань і SPARQL. Розроблено метод кластеризації виходів транзакцій на основі пошуку евристичних патернів типових фінансових операцій та пропагації зв'язків власності в межах таких патернів. Показано, що використання RDF-графа знань дає змогу зберігати результати проміжних ітерацій, виконувати інкрементальну кластеризацію й доповнювати її новими евристиками. Також сформульовано ідеї системного захисту від патернового аналізу графа транзакцій шляхом заміни існуючих раціональних стратегій менеджменту виходів на більш випадкові, які руйнують евристичний аналіз. Запропоновано метод захисту від патернового аналізу операцій консолідації шляхом багатокрокової симуляції операцій платежу одному отримувачу.

У **четвертому розділі** розробляється архітектура програмної системи для побудови, розширення та дослідження графа знань про Біткоїн-транзакції, що складається з компонентів для збору даних, ETL-інструментів, RDF-сховища графа знань, автоматизованого компоненту

пропагації зв'язків та користувацького інтерфейсу для дослідження графа. Розроблено процес завантаження Біткоїн-блоків із підтримкою обробки реорганізацій ланцюга. Для реалізації і інтеграції решти компонентів розроблено і розміщено у відкритому доступі бібліотеку для взаємодії з Біткоїн-мережею з програмного середовища Common Lisp.

У дисертаційній роботі отримано низку **нових наукових результатів**, зокрема **вперше** розроблено спеціалізовану модель даних Біткоїн-транзакцій, характерною рисою якої є відображення транзакцій, виходів транзакцій, власників виходів та довільних зовнішніх анотацій у єдиному RDF-графі з додатковими атрибутами, обчисленими на етапі побудови, що надає уніфікований семантичний контракт для інтеграції компонентів програмної системи аналізу графа транзакцій, забезпечує зберігання усієї історії транзакцій в межах єдиного сховища даних і уможливорює класи запитів для пошуку типових операцій, що складаються з однієї і більше транзакцій і відбуваються протягом тривалих проміжків часу.

Уперше запропоновано метод евристичної кластеризації виходів Біткоїн-транзакцій, який ґрунтується на використанні спеціалізованого RDF-графа знань про Біткоїн-транзакції, що відображає повну історію транзакцій та довільні зовнішні анотації, зокрема результати попередніх аналізів, і відрізняється можливістю інкрементального виконання кластеризації з покроковою пропагацією зв'язків власності між виходами транзакцій відповідно до евристичного визначення типів операцій та збереженням проміжних результатів безпосередньо у графі знань.

Уперше запропоновано метод захисту від патернового аналізу операцій консолідації виходів транзакцій, характерною рисою якого є багатокроковий процес, який симулює платежі з одним отримувачем і створює у графі транзакцій складний, стійкий до розпізнавання підграф операції, що дає змогу знижувати ризики деанонімізації та розкриття чутливої фінансової інформації.

Уперше запропоновано узагальнену архітектуру програмної системи для аналізу графа Біткоїн-транзакцій, у якій спеціалізована модель транзакційних даних використовується як онтологічне ядро, що об'єднує транзакції, їхні виходи, власників та зовнішні метадані в межах єдиного RDF-графа, забезпечує накопичення внутрішніх і зовнішніх знань про транзакції у єдиному сховищі та надає уніфікований стандартизований інтерфейс взаємодії з графом знань.

Основні наукові результати дисертаційної роботи **опубліковано** у 7 наукових працях, зокрема у 3 наукових статтях, що належать до фахових виданнях, включених до переліку наукових фахових видань України з присвоєнням категорії «Б», та у 4 матеріалах науково-технічних конференцій.

Ключові слова: інженерія програмного забезпечення, архітектура програмного забезпечення, прикладне програмне забезпечення, криптовалюта, блокчейн, Біткоїн, соціальні мережі, виявлення шахрайства, граф залежностей, граф знань, онтологія, RDF/OWL, евристики, аналіз даних, кластеризація.

SUMMARY

Zhykin Y. Methods and Software for Bitcoin Transaction Graph Analysis.
– Qualifying scientific work, manuscript.

PhD thesis in the field of knowledge 12 Information technologies in specialty 121 Software Engineering. – National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”, Kyiv, 2026.

In nearly two decades since its emergence, Bitcoin has become an important component of the global financial system. At the same time, Bitcoin serves as a system for micropayments and electronic cash, a private investment asset, a tool for international settlements, and even a part of state financial strategies. Bitcoin's independence from governmental and banking institutions makes it an attractive financial instrument for users in politically or economically unstable regions. Access to Bitcoin for individual users is practically impossible to restrict, since most Bitcoin operations require only low-bandwidth Internet access, but the issue of user anonymity and the security of their personal data becomes especially important in this context.

Bitcoin, as well as most other cryptocurrency systems, is not absolutely anonymous. Bitcoin protocol is based on an open transaction ledger that contains the complete history of all operations since the system's inception. In this ledger, each user is represented by one or more pseudonyms, known as addresses. Bitcoin transactions form a complex graph of ownership transfers, and analyzing this graph allows to discover socio-economic links between users or organizations, identify sets of addresses belonging to the same user, and even deanonymize individual users. On the one hand, such capabilities are useful for law enforcement for combating fraud, money laundering and other criminal activities, as well as for financial institutions for market modeling and economic forecasting. On the other hand, these capabilities infringe on the rights of individual users to the privacy of their financial data. Nevertheless, studying the possibilities of user deanonymization helps discover weaknesses in Bitcoin's

privacy model, which in turn contributes to the development of new protocols and tools aimed at improving user privacy.

The study of methods of Bitcoin transaction graph analysis is relevant both from the point of view of individual users (protecting personal financial data from unauthorized access) and from the point of view of law enforcement agencies and organizations (detecting operations that may indicate potential criminal activity or directly countering financial crimes). The development of open software systems for Bitcoin transaction graph analysis is beneficial for enhancing the security of the entire Bitcoin ecosystem, both from the standpoint of individual users and from that of law enforcement authorities.

The purpose is to simplify the processes of development and operation of software for analyzing the Bitcoin transaction graph by designing a generalized software system architecture based on a specialized transaction data model that provides a unified semantic contract for integrating software components and ensures scalable processing of the transaction history.

In **the first section** of the dissertation, the historical and economic prerequisites for the emergence of Bitcoin and their influence on certain aspects of the Bitcoin protocol are examined. An analysis of the Bitcoin protocol and its transaction model is conducted, including a discussion of transaction structure and the block formation mechanism. The problem of user privacy and technologies for analyzing the Bitcoin transaction graph with the aim of user deanonymization are analyzed, and existing recommendations and tools for preserving privacy are reviewed. A general concept of a software system is formulated that enables the construction and extension of a transaction knowledge graph, as well as pattern-based analysis of such a graph. Finally, the requirements for such a software system are defined.

In **the second section**, a specialized RDF model of the Bitcoin transaction data is developed, adapted for the discovery of patterns of typical operations. To this end, subgraphs of the Bitcoin transaction graph are identified, and a graph representation that captures both transactions and transaction outputs is selected.

The choice of technologies for representing large-scale graphs, and knowledge graphs in particular, is justified. The use of SPARQL as a universal query language for pattern-based analysis is examined, and the possibilities of extending SPARQL with specialized filter functions to simplify complex queries are also considered.

In **the third section**, a concept for identifying typical operations in the Bitcoin transaction graph using heuristic patterns is proposed, along with its implementation based on an RDF knowledge graph and SPARQL. A method for clustering transaction outputs is developed based on the detection of heuristic patterns of typical financial operations and the propagation of ownership relations within such patterns. It is shown that the use of an RDF knowledge graph enables the preservation of intermediate iteration results, supports incremental clustering, and allows the incorporation of new heuristics. In addition, ideas for system-level protection against pattern-based analysis of the transaction graph are formulated by replacing existing rational output management strategies with more randomized ones that disrupt heuristic analysis. A method for protection against pattern-based analysis of consolidation operations is proposed based on a multi-step simulation of payment operations to a single recipient.

In **the fourth section**, the architecture of a software system for constructing, extending, and analyzing a knowledge graph of Bitcoin transactions is developed. The system consists of components for data collection, ETL tools, an RDF knowledge graph store, an automated component for propagating ownership relations, and a user interface for graph exploration. A process for loading Bitcoin blocks with support for handling blockchain reorganizations is developed. To implement and integrate the remaining components, a library for interacting with the Bitcoin network from the Common Lisp programming environment is developed and released as open-source software.

The dissertation has produced **a number of new scientific results**, in particular by developing **for the first time**, a specialized data model for Bitcoin transactions, whose distinguishing feature is the representation of transactions, transaction outputs, output owners, and arbitrary external annotations within a single RDF graph with additional attributes computed at the construction stage, providing a unified semantic contract for integrating components of a software system for transaction graph analysis, ensuring the storage of the entire transaction history within a single data repository, and enabling classes of queries for identifying typical operations consisting of one or more transactions occurring over extended periods of time.

For the first time, an algorithmic–software method for heuristic clustering of Bitcoin transaction outputs is proposed, which is based on the use of a specialized RDF knowledge graph of Bitcoin transactions that represents the complete transaction history and arbitrary external annotations, including the results of prior analyses, and is distinguished by the ability to perform clustering incrementally with step-by-step propagation of ownership relations between transaction outputs in accordance with heuristic definitions of operation types, while storing intermediate results directly within the knowledge graph.

For the first time, an algorithmic–software method for increasing resistance to pattern analysis of transaction output consolidation operations is proposed, whose distinguishing feature is a multi-step process that simulates payments to a single recipient and constructs a complex operation subgraph within the transaction graph that is robust to recognition, thereby enabling the reduction of de-anonymization risks and the disclosure of sensitive financial information.

For the first time, a generalized software system architecture for Bitcoin transaction graph analysis is proposed, in which a specialized transaction data model is used as an ontological core that unifies transactions, their outputs, owners, and external metadata within a single RDF graph, enables the accumulation of internal and external knowledge about transactions in a unified

data repository, and provides a unified standardized interface for interacting with the transaction knowledge graph.

The main scientific results of the dissertation **have been published** in 7 scientific works, including 3 scientific articles published in professional journals included in the list of scientific journals of Ukraine in category “B”, as well as 4 materials of scientific and technical conferences.

Keywords: software engineering, software architecture, application software, cryptocurrency, blockchain, Bitcoin, social networks, fraud detection, dependency graph, knowledge graph, ontology, RDF/OWL, heuristics, data analysis, clustering.

Список публікацій здобувача / List of publications of the applicant:

Статті у фахових виданнях, включених до переліку наукових фахових видань України з присвоєнням категорії “Б” / the articles published in scientific journals included in the list of scientific journals of Ukraine in category “Б”:

1. Жикін Ю.С. Патерновий аналіз графа Біткоїн-транзакцій. / Ю.С. Жикін, М.В. Онай // Вісник Хмельницького національного університету. Серія: Технічні науки – 2024. – №2. – С. 322-328. – DOI:10.31891/2307-5732-2024-333-2-51.

Внесок Жикіна Ю.С.: розроблення концепції аналізу графа транзакцій з використанням евристичних патернів типових операцій, розроблення RDF-словника графа транзакцій, розроблення SPARQL-запитів для пошуку патернів у графі транзакцій.

Внесок Онай М.В.: обговорення запропонованої концепції, формулювання дослідницького завдання, редагування статті.

2. Жикін Ю.С. RDF-модель графа Біткоїн-транзакцій. / Ю.С. Жикін, М.В. Онай // Вісник Хмельницького національного університету. Серія: Технічні науки – 2024. – №5. – С. 25-30. – DOI:10.31891/2307-5732-2024-341-5-3.

Внесок Жикіна Ю.С.: розроблення детальної онтології графа Біткоїн-транзакцій, розроблення аналогічної реляційної моделі для проведення експериментального порівняння швидкодії пошуку в глибину графа, аналіз та оформлення результатів.

Внесок Онай М.В.: формулювання дослідницького завдання, формулювання підходу до експериментальної перевірки результатів, редагування статті.

3. Жикін Ю.С. Метод консолідації виходів Біткоїн-транзакцій за допомогою симуляції витрачання. / Ю.С. Жикін, М.В. Онай // Комп'ютерно-інтегровані технології: освіта, наука, виробництво –

2025. – №59. – С. 113-119. –
DOI:10.36910/6775-2524-0560-2025-59-15.

Внесок Жикіна Ю.С.: аналіз відомих досліджень щодо операцій консолідації виходів транзакцій, формулювання концепції запропонованого методу, розроблення алгоритму, що реалізує метод, оцінка збільшення вартості операцій.

Внесок Оная М.В.: формулювання задачі оцінки запропонованого методу з точки зору збільшення вартості операцій, редагування статті.

Публікації у матеріалах наукових конференцій / the publications in the proceedings of the scientific conferences:

4. Жикін Ю.С. Метод групування адрес за власністю на основі патернів типових операцій у графі Біткоїн-транзакцій. / Ю.С. Жикін, М.В. Оная // Прикладна математика та комп'ютинг. ПМК, 2024: Сімнадцята наук. конф. магістрантів та аспірантів, Київ, 20-22 лист. 2024 р.: зб. тез доп. [редкол.: Дичка І. А. та ін.]. – К.: ПТФ «Просвіта», 2024.– С. 87-92.

Внесок Жикіна Ю.С.: аналіз відомих досліджень щодо кластеризації Біткоїн-адрес, формулювання концепції методу, розроблення алгоритму, що реалізує метод, побудова тестового графа для експериментальної перевірки методу, оформлення результатів.

Внесок Оная М.В.: огляд відомих досліджень щодо кластеризації Біткоїн-адрес, формулювання дослідницького завдання, редагування статті.

5. Жикін Ю.С. Метод зменшення розміру RDF-графа за допомогою непрямой типізації. / Ю.С. Жикін, М.В. Оная // Традиційні та інноваційні підходи до наукових досліджень: збірник наукових праць з матеріалами VIII Міжнародної наукової конференції (Дрогобич, 31 січня 2025 р.) / Міжнародний центр наукових досліджень. – Вінниця:

ТОВ “УКРЛОГОС Груп”, 2025. – С. 287-290. – DOI:10.62731/mcnd-31.01.2025.007.

Внесок Жикіна Ю.С.: розроблення запропонованого методу, застосування методу для тестового графа транзакцій з метою експериментальної перевірки, розроблення розширення для сховища RDF-графів, яке автоматично застосовує запропоноване перетворення у SPARQL-запитах на етапі виконання.

Внесок Оная М.В.: формулювання задачі експериментальної перевірки результатів застосування запропонованого методу, редагування тез.

6. Жикін Ю.С. Взаємодія з Біткоїн-мережею з програмного середовища Common Lisp. / Ю.С. Жикін, М.В. Оная // Наукові орієнтири: теорія та практика досліджень: збірник наукових праць з матеріалами VI Міжнародної наукової конференції (Київ, 3 жовтня 2025 р.) / Міжнародний центр наукових досліджень. – Вінниця: ТОВ “УКРЛОГОС Груп”, 2025. – С. 196-199. – DOI:10.62731/mcnd-03.10.2025.002.

Внесок Жикіна Ю.С.: розроблення структури бібліотеки, написання програмного коду, реалізація модулів тестування, підготовка та проведення експерименту для порівняння швидкодії з аналогічною бібліотекою для програмного середовища Python, оформлення результатів.

Внесок Оная М.В.: формулювання задачі оцінки швидкодії розробленої бібліотеки у порівнянні з існуючими бібліотеками для інших програмних середовищ, редагування тез.

7. Жикін Ю.С. Узагальнена архітектура програмної системи для побудови та аналізу графа знань про Біткоїн-транзакції. / Ю.С. Жикін, М.В. Оная // Міжгалузеві диспути: динаміка та розвиток сучасних наукових досліджень: збірник наукових праць з матеріалами VIII Міжнародної наукової конференції (Харків, січень

2026 р.) / Міжнародний центр наукових досліджень. – Вінниця: ТОВ “УКРЛОГОС Груп”, 2026. – С. 229-232. – DOI:10.62731/mcnd-30.01.2026.006.

Внесок Жикіна Ю.С.: формулювання та аналіз вимог до програмної системи, розроблення та обґрунтування архітектурних рішень.

Внесок Оная М.В.: формулювання і обговорення вимог до програмної системи, редагування тез.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ.....	18
ВСТУП.....	21
1. АНАЛІЗ МЕТОДІВ І ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБОРУ І ДОСЛІДЖЕННЯ ДАНИХ БІТКОЇН-ТРАНЗАКЦІЙ.....	28
1.1. Біткоїн-протокол.....	28
1.2. Проблема приватності транзакційних даних.....	39
1.3. Методи дослідження транзакційних даних.....	51
1.4. Формулювання вимог до програмного забезпечення.....	55
1.5. Висновки.....	57
2. РОЗРОБЛЕННЯ СПЕЦІАЛІЗОВАНОЇ МОДЕЛІ ДАНИХ БІТКОЇН-ТРАНЗАКЦІЙ.....	59
2.1. Підграфи графа Біткоїн-транзакцій.....	59
2.2. Вибір технології подання великих графів.....	68
2.3. Спеціалізована RDF-модель транзакційних даних.....	74
2.4. Використання SPARQL для дослідження графа транзакцій.....	90
2.5. Висновки.....	94
3. РОЗРОБЛЕННЯ МЕТОДІВ ПАТЕРНОВОГО АНАЛІЗУ ГРАФА БІТКОЇН-ТРАНЗАКЦІЙ.....	96
3.1. Евристичний пошук типових операцій у графі транзакцій.....	96
3.2. Метод евристичної кластеризації транзакційних виходів.....	98
3.3. Типові Біткоїн-операції.....	106
3.4. Методи захисту від патернового аналізу графа транзакцій.....	124
3.5. Висновки.....	128

4. РОЗРОБЛЕННЯ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ АНАЛІЗУ ГРАФА БІТКОЇН-ТРАНЗАКЦІЙ.....	130
4.1. Архітектура програмної системи.....	130
4.2. Процес побудови графа знань про транзакції.....	139
4.3. Бібліотека для взаємодії з Біткоїн-мережею.....	147
4.4. Висновки.....	151
ВИСНОВКИ.....	153
СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ.....	156
ДОДАТКИ.....	167

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Біткоїн – децентралізована система електронних грошей на базі однорангової мережі, а також умовна грошова одиниця в цій системі, що містить 100 мільйонів базових неподільних грошових одиниць, визначених протоколом. В англomовних текстах прийнято розрізняти **Біткоїн** (систему) та **біткоїн** (грошову одиницю або, як незлічуваний іменник, певну кількість грошових одиниць). У цьому тексті теж використовується подібне розрізнення.

Граф знань – це формалізоване подання знань (фактів) певної предметної області у вигляді орієнтованого графа, у якому вузли відповідають сутностям предметної області (об'єктам, поняттям, подіям), а ребра відображають семантичні відношення між ними.

Програмованість (programmability) – це здатність системи чи платформи виконувати власні програми або скрипти, що забезпечує гнучкість і адаптацію для різних завдань.

Консенсус – це процес прийняття групою учасників згоди щодо певного питання а також результат такого процесу.

Одноранговість (Peer-to-Peer, P2P) – це характеристика системи (мережі), що складається з рівноправних вузлів (вузлів одного рангу).

Патерн – форма, що відповідає певній закономірності у навколишньому середовищі, рукотворних системах чи абстрактних ідеях; у цьому тексті мається на увазі патерн у графі – форма, що відповідає повторюваному підграфу і має певну асоційовану інтерпретацію.

Приватність – це право і можливість індивіда самостійно контролювати доступ до особистої інформації, дій, думок і простору.

Скрейпінг – це процес автоматизованого збору інформації з веб-сайтів або інших цифрових джерел за допомогою спеціальних програм чи скриптів.

Стекові мови програмування – це клас мов програмування, у яких процес обчислення використовує стек в якості єдиного механізму передавання параметрів та збереження проміжних результатів обчислень.

Транзакційний вихід – структурна одиниця біткоїна в Біткоїн-протоколі, яка створюється транзакцією і представляє певну кількість біткоїна та умови її використання подальшими транзакціями.

Хеш-сума, хеш – це результат обчислення хеш-функції від певного значення; вихідне значення фіксованої довжини, що унікально (з великою ймовірністю) відповідає певному вхідному значенню.

Хеш-функція – це математична функція, яка відображає вхідні дані довільної розміру у вихідне значення фіксованого розміру.

ETL (Extract – Transform – Load, витягування – перетворення – завантаження) – це процес інтеграції даних, що складається з трьох основних етапів: витягування – отримання даних із різних джерел, перетворення – очищення, нормалізації, агрегації та узгодження даних у єдиному форматі, і завантаження – запису оброблених даних у кінцевому вигляді у цільове сховище.

RPC (Remote Procedure Call, дистанційний виклик процедур) – це протокол взаємодії в багатокомпонентних системах, який дає змогу одному компоненту викликати певну процедуру (функцію або метод), що виконується дистанційно іншим компонентом, і отримати локально результат її виконання.

RDF (Resource Description Framework, середовище опису ресурсів) – набір стандартів, розроблених консорціумом W3C для формального подання та обміну даними у вигляді графа знань, що представляє факти як трійки “суб'єкт” (початкова вершина) – “предикат” (дуга) – “об'єкт” (кінцева вершина).

RDFS (RDF Schema, RDF-схема) – це надбудова над RDF, базова онтологічна мова для RDF-графів, яка забезпечує простий механізм опису класів, властивостей і ієрархій.

REPL (Read-Eval-Print Loop, цикл читання, виконання і виведення) – це інтерактивне середовище програмування, яке дає змогу вводити фрагменти коду, виконувати їх і одразу бачити результат. Прикладами повноцінних REPL є середовища діалектів мови програмування Lisp, а прикладом примітивного REPL є інтерактивний інтерпретатор Python.

ВСТУП

Актуальність теми. Протягом майже двох десятиліть з моменту своєї появи Біткоїн став важливим компонентом глобальної фінансової системи. Біткоїн водночас є системою мікроплатежів та електронної готівки, активом приватних інвестицій, інструментом міжнародних розрахунків і навіть складовою державних фінансових стратегій. Незалежність Біткоїна від державних і банківських інституцій робить його привабливим фінансовим інструментом для користувачів у політично чи економічно нестабільних регіонах. Доступ індивідуального користувача до Біткоїна практично неможливо обмежити, оскільки для більшості операцій з Біткоїном достатньо навіть низькошвидкісного доступу до Інтернету, але у цьому контексті особливо важливим є питання анонімності користувачів і безпеки їхніх персональних даних.

Біткоїн, як і більшість інших криптовалютних системи, не є абсолютно анонімним. Біткоїн-протокол базується на відкритому реєстрі транзакцій, який містить всю історію операцій за весь час існування системи, і у цьому реєстрі кожен користувач представлений одним або багатьма псевдонімами, які називають адресами. Транзакції в Біткоїні утворюють складний граф переходів власності, і аналіз цього графа дає змогу встановлювати соціально-економічні зв'язки між користувачами чи організаціями, знаходити множини адрес, що належать одному й тому ж користувачу, і навіть деанонімізувати окремих користувачів. З одного боку такі можливості є корисними для правоохоронних органів у боротьбі з шахрайством, відмиванням коштів і іншою злочинною діяльністю, а також для фінансових установ у моделюванні ринку та прогнозуванні економічних процесів. З іншого боку такі можливості ставлять під загрозу право індивідуальних користувачів на приватність їхніх фінансових даних. Втім, вивчення можливостей деанонімізації користувачів допомагає виявляти слабкі місця у моделі приватності Біткоїна, що в свою чергу

сприяє розробці нових протоколів та інструментів, спрямованих на покращення приватності користувачів.

Дослідження методів аналізу графа Біткоїн-транзакцій є актуальним як з точки зору окремого користувача з метою захисту персональних фінансових даних від несанкціонованого втручання, так і з точки зору правоохоронних органів та організацій з метою розпізнавання операцій, що можуть вказувати на потенційну злочинну діяльність, або прямої протидії фінансових злочинам. Розроблення відкритих програмних систем аналізу графа Біткоїн-транзакцій є корисним для підвищення безпеки всієї Біткоїн-екосистеми, як з точки зору індивідуального користувача, так і з точки зору правоохоронних органів.

Зв'язок роботи з науковими програмами, планами, темами.

Дослідження за темою дисертаційної роботи проводились у Національному технічному університеті України “Київський політехнічний інститут імені Ігоря Сікорського” в рамках виконання ініціативної науково-дослідної роботи “Методи та програмне забезпечення для аналізу графа Біткоїн-транзакцій” (номер державної реєстрації 0124U000907).

Мета і задачі дослідження. Метою дисертаційної роботи є спрощення процесів проєктування, розроблення та експлуатації програмного забезпечення для аналізу графа Біткоїн-транзакцій шляхом розроблення узагальненої архітектури програмної системи на основі спеціалізованої моделі транзакційних даних, яка надає уніфікований семантичний контракт для інтеграції програмних компонентів і забезпечує масштабоване оброблення історії транзакцій.

Відповідно до поставленої мети виділено наступні основні задачі дослідження:

1. Провести аналіз відомих методів і програмного забезпечення для дослідження графа Біткоїн-транзакцій з метою кластеризації Біткоїн-адрес та деанонімізації користувачів.

2. Сформулювати узагальнену концепцію дослідження графа Біткоїн-транзакцій, поданого у вигляді графа знань.
3. Сформулювати вимоги до програмної системи для побудови, зберігання, доповнення та дослідження графа Біткоїн-транзакцій.
4. Розробити спеціалізовану модель даних Біткоїн-транзакцій, що надає уніфікований контракт для інтеграції компонентів програмного забезпечення для аналізу транзакцій.
5. Розробити алгоритмічне забезпечення процесу автоматичного доповнення графа знань про Біткоїн-транзакції новими зв'язками на основі евристичних патернів типових операцій.
6. Розробити концепцію та методи захисту транзакційних даних користувачів від патернового аналізу графа транзакцій у програмному забезпеченні для конструювання Біткоїн-транзакцій (Біткоїн-гаманцях).
7. Розробити програмне забезпечення для перетворення даних ланцюга блоків Біткоїна у граф знань про Біткоїн-транзакції, що відповідає сформульованим вимогам.
8. Розробити узагальнену архітектуру програмного забезпечення для побудови, доповнення та дослідження графа знань про Біткоїн-транзакції на основі уніфікованого контракту, який надає розроблена спеціалізована модель даних.

Об'єкт дослідження – процеси проєктування, розроблення та експлуатації програмного забезпечення для аналізу графа Біткоїн-транзакцій.

Предмет дослідження – методи, моделі даних та архітектурні рішення програмної інженерії, що застосовуються розроблення алгоритмічного та програмного забезпечення для аналізу графа Біткоїн-транзакцій.

Методи дослідження: методи системного аналізу, методи інженерії вимог, методи архітектурного проєктування програмного забезпечення,

методи моделювання даних, методи об'єктно-орієнтованого та компонентного проєктування, методи забезпечення якості програмного забезпечення, експериментальні методи.

Наукова новизна одержаних результатів полягає у наступному:

1. **Уперше** розроблено спеціалізовану модель даних Біткоїн-транзакцій, характерною рисою якої є відображення транзакцій, виходів транзакцій, власників виходів та довільних зовнішніх анотацій у єдиному RDF-графі з додатковими атрибутами, обчисленими на етапі побудови, що надає уніфікований семантичний контракт для інтеграції компонентів програмної системи аналізу графа транзакцій, забезпечує зберігання усієї історії транзакцій в межах єдиного сховища даних і уможливорює класи запитів для пошуку типових операцій, що складаються з однієї і більше транзакцій і відбуваються протягом тривалих проміжків часу.
2. **Уперше** запропоновано метод евристичної кластеризації виходів Біткоїн-транзакцій, який ґрунтується на використанні спеціалізованого RDF-графа знань про Біткоїн-транзакції, що відображає повну історію транзакцій та довільні зовнішні анотації, зокрема результати попередніх аналізів, і відрізняється можливістю інкрементального виконання кластеризації з покроковою пропагацією зв'язків власності між виходами транзакцій відповідно до евристичного визначення типів операцій та збереженням проміжних результатів безпосередньо у графі знань.
3. **Уперше** запропоновано метод захисту від патернового аналізу операцій консолідації виходів транзакцій, характерною рисою якого є багатокроковий процес, який симулює платежі з одним отримувачем і створює у графі транзакцій складний, стійкий до

розпізнавання підграф операції, що дає змогу знижувати ризики деанонізації та розкриття чутливої фінансової інформації.

4. **Уперше** запропоновано узагальнену архітектуру програмної системи для аналізу графа Біткоїн-транзакцій, у якій спеціалізована модель транзакційних даних використовується як онтологічне ядро, що об'єднує транзакції, їхні виходи, власників та зовнішні метадані в межах єдиного RDF-графа, забезпечує накопичення внутрішніх і зовнішніх знань про транзакції у єдиному сховищі та надає уніфікований стандартизований інтерфейс взаємодії з графом знань.

Практичне значення одержаних результатів. Розроблена узагальнена архітектура та уніфікована модель транзакційних даних можуть бути безпосередньо використані у реальних програмних системах аналізу графа транзакцій Біткоїна та інших криптовалютних систем, що використовують аналогічну модель транзакції, з метою спрощення процесу проєктування та інтеграції програмних компонентів.

Розроблене відкрите програмне забезпечення для накопичення інформації про Біткоїн-транзакції у вигляді графа знань може використовуватись як для аналізу графа транзакцій, зокрема для вирішення задачі кластеризації адрес за власністю, так і для дослідження можливостей захисту користувачів Біткоїна від такого аналізу, зокрема для розроблення методів здійснення типових операцій, які порушують евристичні припущення про зміст таких операцій.

Особистий внесок здобувача. Дисертаційна робота є результатом самостійного наукового дослідження. Усі подані результати досліджень належать здобувачеві.

Апробація результатів дисертації. Основні результати дисертаційного дослідження доповідалися та обговорювалися на всеукраїнських та міжнародних наукових конференціях:

1. Сімнадцята наукова конференція магістрантів і аспірантів “Прикладна математика та комп'ютинг (ПМК'2024)”, 20 – 22 листопада 2024 р., Київ, Україна.
2. VIII Міжнародна наукова конференція “Традиційні та інноваційні підходи до наукових досліджень”, 31 січня 2025 р., Дрогобич, Україна.
3. VI Міжнародна наукова конференція “Наукові орієнтири: теорія та практика досліджень”, 3 жовтня 2025 р., Київ, Україна.
4. VIII Міжнародна наукова конференція “Міжгалузеві диспути: динаміка та розвиток сучасних наукових досліджень”, 30 січня 2026 р., Харків, Україна.

Публікації. Основні наукові результати дисертаційної роботи опубліковано у 7 наукових працях, зокрема у 3 наукових статтях, що належать до фахових виданнях, включених до переліку наукових фахових видань України з присвоєнням категорії «Б», та у 4 матеріалах науково-технічних конференцій.

У роботі [1] (Жикін Ю.С., Онай М.В. “Патерновий аналіз графа Біткоїн-транзакцій”) запропоновано загальний підхід до аналізу графа транзакцій, що полягає у побудові RDF-графа всієї історії транзакцій, і пошуку евристичних патернів типових фінансових операцій у ньому за допомогою стандартизованої мови SPARQL для пошуку патернів.

У роботі [2] (Жикін Ю.С., Онай М.В. “RDF-модель графа Біткоїн-транзакцій”) розроблено RDF-модель даних Біткоїн-транзакцій та здійснено порівняння швидкості пошуку множин вершин на певній відстані від заданої вершини у RDF-графі та еквівалентній реляційній базі даних.

У роботі [3] (Жикін Ю.С., Онай М.В. “Метод консолідації виходів Біткоїн-транзакцій за допомогою симуляції витрачання”) розроблено метод багатокрокової консолідації виходів транзакцій за допомогою симуляції платежів з одним отримувачем і рештою.

У роботі [4] (Жикін Ю.С., Онай М.В. “Метод групування адрес за власністю на основі патернів типових операцій у графі Біткоїн-транзакцій”) розроблено метод групування виходів шляхом евристичного пошуку типових операцій у графі знань про транзакції і пропагації зв'язків власності з одних транзакційних виходів на інші в межах таких операцій.

У роботі [5] (Жикін Ю.С., Онай М.В. “Метод зменшення розміру RDF-графа за допомогою непрямой типізації”) запропоновано метод зменшення розміру RDF-графа шляхом видалення явних оголошень типів сутностей і заміни пошуків за типом на пошуки за наявністю властивості або зв'язку, що є однозначно вказує на тип сутності.

У роботі [6] (Жикін Ю.С., Онай М.В. “Взаємодія з Біткоїн-мережею з програмного середовища Common Lisp”) розроблено і протестовано бібліотеку інтерфейсів для взаємодії з Біткоїн-мережею з програмного середовища Common Lisp.

У роботі [7] (Жикін Ю.С., Онай М.В. “Узагальнена архітектура програмної системи для побудови та аналізу графа знань про Біткоїн-транзакції”) розроблено архітектуру програмної системи, що складається з ядра – сховища RDF-графів, що містить граф знань про транзакції, ETL-системи для початкової побудови та подальшого доповнення графа знань та аналітичної системи, яка в свою чергу складається з сервісу автоматичної пропагації зв'язків власності у графі та набору інтерфейсів користувача для здійснення досліджень графа транзакцій в ручному режимі.

Структура роботи. Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел, що включає 100 найменувань, та 6 додатків. Загальний обсяг роботи становить 220 сторінок, у тому числі 165 сторінок основного тексту, 29 рисунків, 5 таблиць та 18 лістингів.

1. АНАЛІЗ МЕТОДІВ І ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБОРУ І ДОСЛІДЖЕННЯ ДАНИХ БІТКОЇН-ТРАНЗАКЦІЙ

1.1. Біткоїн-протокол

1.1.1. Історичний та економічний контекст

Біткоїн – це однорангова система електронних грошей, початкова версія якої була розроблена невідомою особою (чи групою осіб) під псевдонімом Сатоші Накамото у 2008 році [8] і введена в експлуатацію 3 січня 2009 року.

Спочатку нова технологія зацікавила лише невелику спільноту криптографів та учасників анархічного руху “шифропанк” [9] але досить швидко стало зрозуміло, що Біткоїн – це цікава альтернатива традиційним фінансовим системам, і з тих пір ця технологія вже затвердила себе як глобальна грошова система, що використовується мільйонами людей в усіх країнах світу, і з кожним роком стає все більш важливою складовою глобальної цифрової інфраструктури. Грошову одиницю, яка використовується в системі, називають біткоїном.

Зацікавленість Біткоїном була обумовлена властивостями, яких на момент його появи не мала жодна інша грошова система: незалежність (одноранговість мережі Біткоїн означає відсутність центрального органу контролю, що робить систему непідконтрольною як локальним так і глобальним правовим системам), дефляційність (кількість одиниць біткоїна у обігу обмежена зверху, що робить його першою в історії людства грошовою системою, в якійсь повністю відсутня інфляція) та програмованість (умова подальшого використання певної кількості біткоїна у транзакції задається спеціальною мовою програмування, що дає змогу створювати так звані “розумні контракти” [10]). Окрім цього, важливим аспектом успіху Біткоїна був історичний контекст: система з’явилась у час, коли весь світ боровся з наслідками глобальної фінансової кризи 2008 року,

і дефляційність біткоїна була сприйнята багатьма людьми як потенційний інструмент захисту купівельної спроможності своїх заощаджень на випадок, якщо подібна ситуація повториться. У генезисному блоці Біткоїна розміщено заголовок з газети The Times від 3 січня 2009 року про те, що канцлер скарбниці Великої Британії готовий до чергового порятунку великих банків від банкрутства за гроші платників податків, що може вказувати на мотивацію Сатоші Накамото при створенні нової грошової системи, у якій непряме оподаткування всіх заощаджень громадян шляхом збільшення пропозиції грошової маси у обігу є неможливим.

1.1.2. Структура Біткоїн-транзакції

Весь біткоїн у обігу представляється парами значень (“кількість”, “власник”), що називаються транзакційними виходами або просто виходами (англ. transaction output, ТХО). Кожна Біткоїн-транзакція знищує певну кількість існуючих виходів і створює певну кількість нових виходів, при цьому для уникнення збільшення кількості біткоїна у обігу протокол вимагає, щоб сумарна кількість біткоїна на вході транзакції була не меншою за кількість біткоїна на виході.

Кожна транзакція містить щонайменше один вихід, що належить отримувачу, але оскільки всі виходи на вході транзакції знищуються, для того, щоб повернути собі залишок від транзакції, надсилач додає ще один вихід, отримувачем якого є він сам. Ненульова різниця між сумою біткоїна на вході транзакції і сумою біткоїна на виході є комісією за транзакцію, яку може забрати собі учасник системи, який обробив транзакцію. Комісія за транзакцію є необов’язковою, але раціональні обробники транзакції надають перевагу транзакціям з вищою комісією.

Транзакції посилаються на виходи, які вони знищують, за допомогою пар значень (“ідентифікатор виходу”, “підтвердження”). Якщо існує транзакція, яка знищує певний вихід, такий вихід називається використаним виходом. Якщо вихід не є використаним, він називається

невикористаним виходом (англ. unspent transaction output, UTXO). Множина всіх невикористаних транзакційних виходів і є активною грошовою масою в системі Біткоїн. Невикористані транзакційні виходи часто називають “монетами”, оскільки до моменту використання у транзакції вони є цілісними “шматками” біткоїна з чітко визначеним власником.

Кількість біткоїна у виході представляється цілим числом, що означає кількість базових неподільних одиниць, які прийнято називати сатоші або сатами на честь псевдонімого засновника системи. Слід зазначити, що один “цілий” біткоїн (BTC) містить 100 мільйонів базових неподільних одиниць, але є віртуальною одиницею, що використовується лише для зручності, і не існує на рівні протоколу. Натомість система оперує виключно цілочисельними значеннями, що відображають кількість неподільних базових одиниць.

Власник біткоїна у кожному виході представляється програмою, сформульованою спеціальною Тюрінг-неповною стековою мовою, що називається Біткоїн-скриптом або просто Скриптом. Саме представлення власника транзакційного виходу як програми і надає Біткоїну властивість програмованості. Програма-власник, або програма блокування (англ. lock script), що міститься у транзакційному виході, визначає умову, яку необхідно задовольнити для того, щоб використати цей транзакційний вихід.

Коли транзакція використовує певний транзакційний вихід, у транзакційному вході, що посилається на цей вихід, повинне бути розміщене підтвердження використання виходу, яке теж формулюється як програма мовою Біткоїн-скрипт. Це підтвердження називають програмою розблокування (англ. unlock script). Згідно з Біткоїн-протоколом, під час валідації кожної транзакції для кожного транзакційного входу виконується об'єднана програма, що утворюється з конкатенації програм розблокування та блокування, і якщо результатом виконання є будь-яке значення, крім

булевого значення 0, транзакційний вихід вважається успішно використаним. Якщо хоча б для одного транзакційного входу результатом виконання об'єднаної програми є булеве значення 0, то вся транзакція вважається неправильною і ігнорується протоколом.

У найпростішому варіанті програма блокування у виході – це публічний ключ власника, а програма розблокування – це криптографічний підпис, здійсненим відповідним приватним ключем, що за визначенням відомий лише власнику, і тому є достатнім підтвердженням, що власник погоджує використання даного виходу. Для простих випадків використання Біткоїна як грошової системи цього було б достатньо, але Сатоші Накамото прийняв рішення додати можливість програмування семантики власності над транзакційними виходами, через що Біткоїн отримав надзвичайно гнучку систему так званих “розумних контрактів”, тобто контрактів, що є самодостатніми комп'ютерними програмами, і не потребують зовнішнього сервісу чи правової системи для забезпечення їх виконання, а виконуються автоматично в межах платформи, в якій вони створені. Найпростішим прикладами таких контрактів є програми блокування типу “якщо виконується умова С, надіслати кошти власнику А, інакше надіслати кошти власнику В”, автоматичне передавання Біткоїна у спадщину, транзакційні канали [11], тощо. На даний момент тривають активні дослідження більш складних потенційних застосувань системи “розумних контрактів” як у Біткоїні, так і у інших криптовалютних протоколах [12 – 14].

Найбільш поширені типові програми блокування називають стандартними програмами блокування. Протокол Біткоїна не забороняє використання нестандартних програм, але більшість клієнтського програмного забезпечення працює лише зі стандартними програмами з метою уникнення випадкової втрати грошей внаслідок помилкового використання операторів Скрипту. Для більшості стандартних програм існує чітко визначений формат ідентифікатора, за допомогою якого отримувач може повідомити надсилачеві, який саме тип програми

використати і які саме унікальні значення (такі як публічний ключ) мають міститись в програмі. Такі ідентифікатори називаються Біткоїн-адресами. Отримувачу достатньо надати своєму контрагенту адресу згідно з визначеним форматом, і клієнтське програмне забезпечення контрагента зможе декодувати адресу і сконструювати транзакційний вихід з такою програмою блокування, яку очікує отримувач.

На рисунку 1.1 зображена структура Біткоїн-транзакції та її зв'язки з іншими транзакціями.

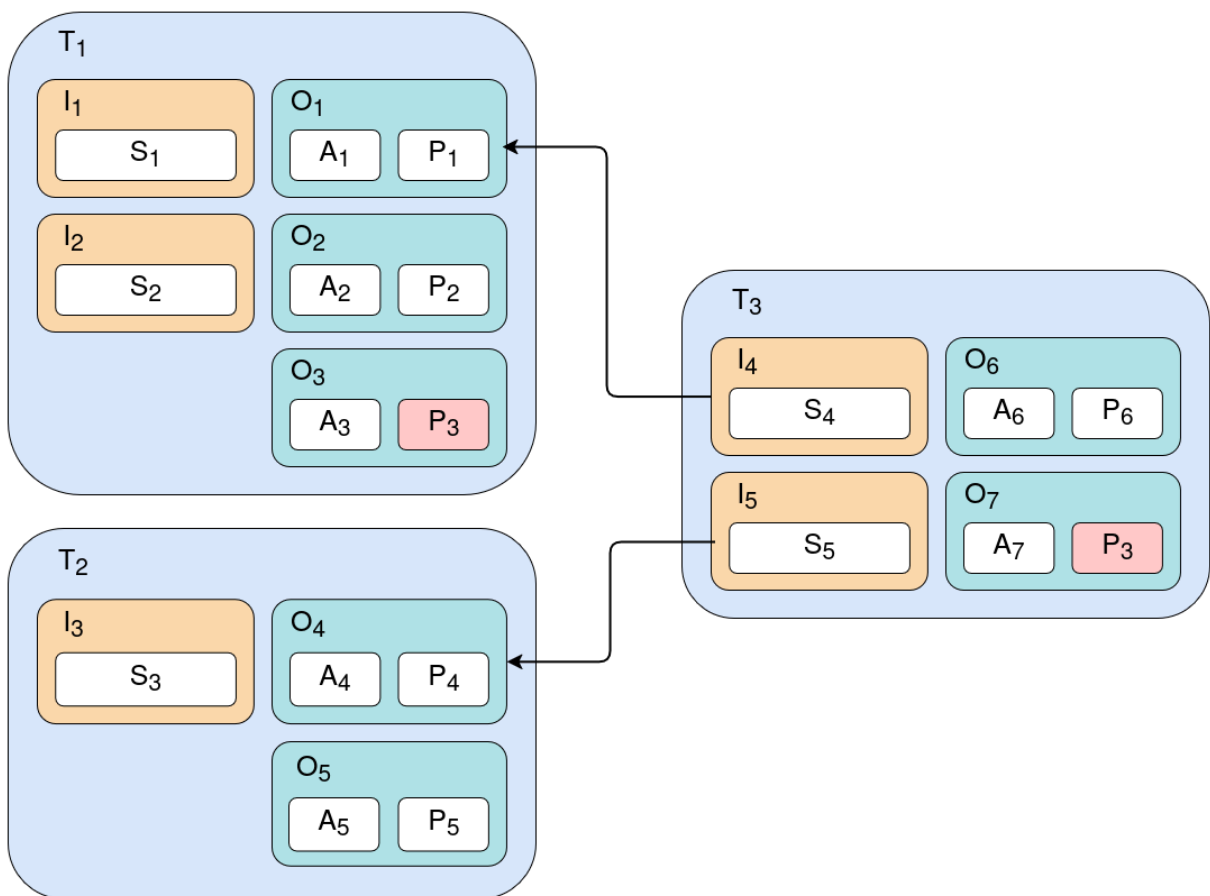


Рис. 1.1. Структура Біткоїн-транзакції

Транзакція T₃ на рисунку 1.1 містить два входи I₄ та I₅ і два виходи O₆ та O₇. При цьому вхід I₄ посилається на вихід O₁ транзакції T₁, а вхід I₅ – на вихід O₄ транзакції T₂. Отже транзакція T₃ знищує виходи O₁ та O₄ і створює два нових виходи O₆ та O₇. У кожного виходу є кількість біткоіна (A₁ – A₇) та ідентифікатор власника, який насправді є програмою

блокування ($P_1 - P_6$). У кожного входу є посилання на вихід, який він використовує (стрілками позначено посилання входу I_4 на вихід O_1 і входу I_5 на вихід O_4 ; посилання виходів $I_1 - I_3$ опущені) та підтвердження власності, тобто програма розблокування ($S_1 - S_5$). Будь-які з виходи O_2 , O_3 , O_5 та O_7 можуть бути виходами повернення решти з транзакції, але в загальному випадку це не відомо.

1.1.3. Ланцюг блоків

Основою Біткоїн-протоколу є структура даних, що називається ланцюгом блоків або блокчейном (англ. block chain) або часовим ланцюгом (англ. time chain). Ланцюг блоків – це публічний, розподілений реєстр, у якому зберігаються всі транзакції, коли-небудь здійснені в Біткоїн-мережі. Кожен блок – це “сторінка” реєстру встановленого максимального розміру, що містить певну послідовність транзакцій. Основною функцією ланцюга блоків є встановлення глобального порядку всіх транзакцій, що коли-небудь відбулись. Для кожної транзакції у ланцюгу точно відомо, що всі транзакції, що передують їй у даному блоці та транзакції, що знаходяться у блоках, що передують даному блоку, відбулись раніше, ніж дана транзакція. Саме ця її функція унеможливорює повторне використання біткоїна: жодна транзакція не може використати транзакційний вихід, якщо він вже був використаний транзакцією, що їй передує, а передування однієї транзакції іншій достеменно відомо з ланцюга блоків. І саме тому більш влучною, хоч і менш поширеною, назвою такої структури даних є назва часовий ланцюг.

Для того, щоб ланцюг блоків міг однозначно встановлювати порядок транзакцій, він повинен гарантувати неможливість додавання, видалення чи зміни порядку транзакцій в межах одного блоку, а також зміни даних у будь-якій транзакції, для чого використовуються криптографічні хеш-функції. Блок складається з заголовка та послідовності транзакцій. Заголовок блока, окрім метаданих, містить посилання на попередній блок у

вигляді криптографічної хеш-суми заголовка попереднього блока, а також посилання на всі свої транзакції у вигляді агрегованої хеш-суми послідовності транзакцій.

Агрегована хеш-сума послідовності транзакцій обчислюється за допомогою дерева Меркла [15] – бінарного дерева, у якому кожен вузол є хешем обох своїх вузлів-нащадків. Використання дерева Меркла для обчислення агрегованої хеш-суми всіх транзакцій має кілька переваг. По-перше, замість хешів усіх транзакцій, можна зберігати у блоці лише один хеш – корінь дерева, що значно зменшує розмір заголовка блока. По-друге, дерево Меркла надійно фіксує як вміст транзакцій, так і їх порядок. Насамкінець, дерево Меркла для транзакцій дає змогу надавати компактні докази присутності транзакції у блоці: для доведення присутності транзакції у дереві достатньо надати послідовність сусідніх хешів у вузлах дерева на шляху до даного елемента.

Завдяки лавинній властивості криптографічної хеш-функції зміна хоча б одного біта в двійковому представленні будь-якої транзакції чи зміна порядку транзакцій у блоці призведе до зміни кореня дерева Меркла, яка в свою чергу призведе до зміни хеш-суми заголовка блока, що призведе до зміни хеш-суми заголовка наступного блока, і так далі. Отже, для того, щоб додати, видалити, чи змінити будь-яку транзакцію, що відбулись за всю історію існування Біткоїн-мережі, необхідно переобчислити блок, у якому відбулась зміна, а також всі блоки, що слідує за ним.

Якщо існує механізм, що унеможливило б швидше створення нових блоків порівняно з іншими учасниками мережі (див. пункт 1.1.4), складність такої зміни залежить лише від сумарної обчислювальної потужності всіх учасників мережі. Для того, щоб така зміна була успішною (тобто, щоб Біткоїн-мережа прийшла до консенсусу щодо того, що така версія ланцюга блоків є правильною), необхідно “обігнати” решту мережі, тобто переобчислювати всі ці блоки з більшою швидкістю, ніж решта мережі продукує нові блоки. Отже, для успішної атаки, націленої на зміну

історії транзакцій, нападнику необхідно контролювати більшу частину сумарної обчислювальної потужності всієї мережі, але

Частина ланцюга, що складається з 3-х блоків, а також дерево транзакцій у одному з них, зображені на рисунку 1.2. Блок B_n криптографічно посиляється на всі свої транзакції та на заголовок блока B_{n-1} , який в свою чергу посиляється на всі свої транзакції ($T_0 - T_3$) та на заголовок блока B_{n-2} . Кожен блок містить криптографічні посилання як на свій вміст, так і на вміст попереднього блока.

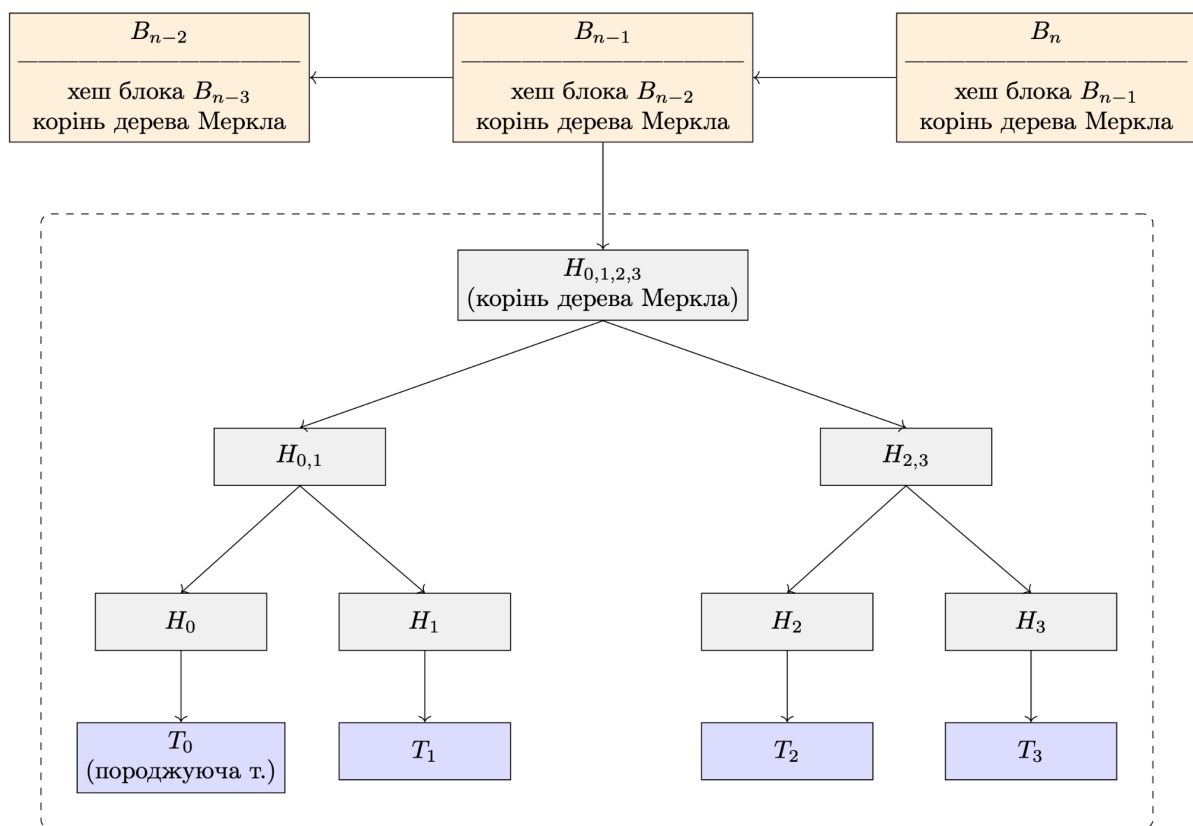


Рис. 1.2. Ланцюг блоків

Окрім встановлення глобального порядку транзакцій та гарантування незмінності цього порядку, ланцюг блоків забезпечує ще й прозорість усіх операцій. Будь-який учасник Біткоїн-мережі має повну копію всього ланцюга блоків і може швидко перевірити наявність транзакції у реєстрі та її позицію відносно інших транзакцій. Саме тут є важливим використання

дерев Меркла для транзакцій: як видно з рисунку 1.2, для доведення присутності транзакції T_2 у блоці B_{n-1} потрібно лише надати компактний доказ у вигляді послідовності хешів H_0 , $H_{2,3}$ та $H_{0,1,2,3}$. Цієї послідовності достатньо, щоб обчислити всі вузли на шляху від транзакції до кореня дерева Меркла, і якщо він співпадає з хешем $H_{0,1,2,3}$, що міститься у заголовку блока, це доводить, що транзакція T_2 дійсно присутня у даному блоці.

1.1.4. Процес формування нових блоків

Біткоїн-транзакція вважається підтвердженою, коли вона потрапляє до реєстру транзакцій і отримує однозначну позицію в часі відносно інших транзакцій. Непідтверджені транзакції зберігаються в тимчасовому сховищі в пам'яті кожного повноцінного вузла мережі, і хоча існують правила, за якими транзакції додаються чи видаляються з цього сховища, не існує механізму, що гарантує дотримання цих правил усіма вузлами в мережі. Тому непідтверджена транзакція, тобто транзакція, яка була опублікована в мережі, ще не потрапила у реєстр, часто не приймається отримувачем платежу аж до моменту підтвердження.

Для того, щоб транзакція стала підтвердженою, вона повинна бути розміщена на одній з сторінок реєстру транзакцій, тобто у одному з блоків ланцюга. Біткоїн-протокол реалізує процес кооперативного додавання транзакцій до реєстру так, щоб досягти децентралізованого консенсусу щодо множини та порядку всіх підтверджених транзакцій між усіма учасниками мережі. Для цього процес побудови ланцюга необхідно було організувати таким чином, щоб створення нового блока було надзвичайно дорогою операцією. Тоді будь-яка спроба змінити щось у реєстрі підтверджених транзакцій вимагає витрати великої кількості ресурсів для переобчислення відповідного блока і, як було описано вище, усіх блоків, що сліднують за ним. Ключовим винаходом, що зробив децентралізований консенсус можливим, є регульований механізм забезпечення високої

вартості створення нового блока, що називається доказом виконаної роботи (англ. proof of work).

Механізм доказу виконаної роботи – це криптографічний механізм, що дає змогу гарантувати, що певний обчислювальний процес був виконаний з великими витратами ресурсів [16]. З математичної точки зору доказ виконаної роботи – це задача знаходження значення N , яке у поєднанні з даними, на які накладається доказ, утворює вхід певної криптографічної хеш-функції такий, що його хеш є меншим за певне цільове значення T . Однією з основних властивостей криптографічної хеш-функції є стійкість до знаходження прообразу [17], що означає, що найкраща стратегія пошуку N – це повний перебір всіх можливих варіантів, а оскільки для даного етапу розвитку обчислювальних технологій можна оцінити мінімальну кількість енергії, необхідну на обчислення хеш-функції для одного варіанту, то можна оцінити і мінімальну середню кількість енергії, необхідну на пошук значення N , що задовольняє цільове значення T . Окрім цього, складність задачі обчислення доказу виконаної роботи можна регулювати, зменшуючи або збільшуючи цільове значення T . Ключовою властивістю цього механізму є те, що створення нового доказу може бути довільно складною задачею, на яку необхідно витратити довільно велику кількість енергії, але при цьому перевірка доказу є дуже простою – достатньо обчислити хеш-функцію один раз для комбінації числа N та даних, на які його було накладено.

Докази виконаної роботи в Біткоїн-протоколі використовують криптографічну хеш-функцію SHA-256 [18] і накладаються на блоки у ланцюгу для того, щоб для кожного блока підтвердити, що хтось з учасників мережі витратив певну кількість енергії, щоб його створити, і в той же час гарантувати, що будь-який інший учасник, який хоче створити альтернативну версію цього блока, повинен витратити для цього приблизно стільки ж енергії. Оскільки сумарна обчислювальна потужність учасників мережі, що створюють нові блоки, змінюється залежно від ринкової

вартості біткоїна, цільове значення T автоматично змінюється кожні 2016 блоків так, щоб підтримувати середній період між блоками на рівні 600 секунд. Якщо період між блоками зменшується, складність занадто низька, тому цільове значення T зменшується, і навпаки.

Процес створення мережею нового блока складається з наступних кроків:

1. Обирається підмножина непідтверджених транзакцій з тимчасового сховища, зазвичай максимізуючи сумарну комісію.
2. Транзакції складаються у дерево Меркла, після чого обчислюється його корінь.
3. Формується заголовок блока, що містить хеш попереднього блока та корінь нового дерева Меркла. На цьому етапі всі значення у заголовку блока, окрім спеціального поля, що містить доказ виконаної роботи, зафіксовані.
4. Майнер здійснює повний перебір всіх можливих варіантів значення поля доказу виконаної роботи. Якщо простір пошуку виснажено, формується новий заголовок блока.
5. Як тільки доказ виконаної роботи знайдено, майнер транслює блок у мережу.
6. Кожен учасник мережі, що отримує новий блок, перевіряє правильність та достатню складність доказу виконаної роботи, правильність кореня дерева Меркла для транзакцій і валідність кожної окремої транзакції. Якщо виявлено будь-яку невідповідність, блок відкидається. Інакше учасник додає його до своєї копії ланцюга, видаляє його транзакції з власного тимчасового сховища непідтверджених транзакцій і застосовує кожну щойно підтверджену транзакцію до власного індексу невикористаних виходів, видаляючи знищені виходи і додаючи нові.

7. Якщо два альтернативних блоки з'являються приблизно в один і той же час, мережа вирішує конфлікт між ними, підтримуючи обидві версії ланцюга протягом певного часу, і зрештою обираючи ту, у якій сумарна кількість виконаної роботи є вищою [19].

Емісія грошових одиниць – біткоїнів, реалізована як винагорода учаснику, що створив новий блок. Механізм емісії наступний: при створенні нового блока першою транзакцією у ньому розміщується спеціальна транзакція, яка називається породжуючою транзакцією (англ. *coinbase transaction*). Ця транзакція має виходи, але при цьому не має входів, генеруючи таким чином нові біткоїни. Такий підхід вирішує одразу дві проблеми – справедливе початкове розповсюдження грошових одиниць і стимулювання учасників мережі до створення нових блоків. Біткоїн-мережа “платить” своїм учасникам за енергію, необхідну для її функціонування. Максимальна кількість нових біткоїнів, які можуть бути створені у породжуючій транзакції у блоці, залежить від позиції цього блока у ланцюгу і описується спадною геометричною прогресією, що гарантує строго обмежену пропозицію біткоїнів. Кожних 210000 блоків кількість нових біткоїнів, що генеруються кожним новим блоком, зменшується вдвічі. За жартівливою алегорією, процес створення нових блоків – це “важка” одноманітна робота, в результаті якої учасник мережі “знаходить” цифрове золото, і тому цей процес традиційно називається майнінгом (від. англ. *mining* – гірнича діяльність), а учасників мережі, якій беруть участь у цьому процесі, називають майнерами (англ. *miner* – гірник).

1.2. Проблема приватності транзакційних даних

Основною інновацією Біткоїна є протокол децентралізованого консенсусу, який дає змогу будь-якому учаснику мережі у будь-який момент швидко перевірити будь-яку транзакцію на відповідність правилам консенсусу і або прийняти, або відкинути її. Крім цього, кожен новий учасник, що приєднується до мережі вперше, здійснює повну перевірку

всієї історії транзакцій відповідно до всіх правил консенсусу для того, щоб переконатись, що чинна множина невикористаних транзакційних виходів (тобто множина всіх шматків біткоїна у обігу) є правильною і не піддавалась маніпуляціям.

Для того, щоб такі перевірки транзакцій були можливими, вся інформація у транзакції є відкритою. Зокрема, основна перевірка в правилах консенсусу, а саме перевірка на повторне використання шматка біткоїна, використовує інформацію про “надсилача” (множину входів), для того, щоб довести, що жоден з виходів, що знищуються даною транзакцією, не був знищений у іншій транзакції. Ще одна важлива перевірка, а саме перевірка на відсутність інфляції, використовує інформацію про кількість біткоїна у входах і виходах транзакції, щоб гарантувати, що сумарна кількість на вході є більшою або рівною сумарній кількості на виході. Власне, самі програми блокування і розблокування повинні бути публічними, щоб кожен учасник мережі міг самостійно виконати ці програми і пересвідчитись, що результат виконання підтверджує правильність транзакції.

Втім, прозорість і відкритість транзакційних даних є одночасно і одним з найбільших недоліків Біткоїн-протоколу. Будь-який учасник мережі, що бере участь у перевірці правил консенсусу, має повну копію всієї історії транзакцій і може використовувати інформацію про транзакції у будь-який спосіб. Наприклад, якщо користувачі Еліс відомо, що користувач Боб регулярно використовує певну Біткоїн-адресу A для отримання оплати за свої товари чи послуги, то Еліс може легко знайти всі транзакції, у яких є виходи з адресою A , і таким чином дізнатись, скільки грошей Боб отримав за весь час, оскільки інформація про кількість біткоїна у кожній такому виході є відкритою. Більше того, Еліс може знайти і всі транзакції, у яких є входи, що використовують якісь з цих виходів, і таким чином може дізнатись, скільки з усіх зароблених грошей Боб вже використав (тобто надіслав на інші адреси), а отже і дізнатись, скільки з

усіх зароблених грошей у нього залишилось. За допомогою двох швидких пошуків у відкритій історії транзакцій Еліс дізналась, скільки біткоіна може бути у Боба на даний момент, і може використати цю інформацію у злочинних цілях.

Сатоші Накамото назвав Біткоїн системою електронної готівки, але однією з цікавих властивостей готівкових грошей, якої якраз і не має у Біткоіна, є абсолютна приватність транзакційних даних. Деталі готівкової транзакції (надсилач, отримувач та кількість грошей) відомі лише двом її сторонам. Навіть традиційні електронні гроші на основі національних валют можуть мати кращу транзакційну приватність, ніж Біткоїн, оскільки банківська таємниця у правових державах зазвичай строго охороняється законом саме для того, щоб інформація про фінанси приватної особи не могла бути використана в злочинних цілях.

Проблема приватності транзакційних даних активно обговорюється і досліджується спільнотою розробників Біткоїн-протоколу та інших криптовалютних проєктів. Одним з найцікавіших таких проєктів є криптовалютний протокол Monero [20], що розвинувся з узагальненого криптовалютного протоколу CryptoNote [21]. Цей протокол використовує складні криптографічні інструменти для того, щоб приховати всі три компоненти кожної транзакції. Надсилач транзакції приховується за допомогою спонтанних анонімних групових криптографічних підписів [22], тобто підписів, які здійснюються одним з ключів з певної спонтанно сформованої групи так, що можна перевірити правильність підпису, але неможливо визначити, який саме з ключів з групи його здійснив. Отримувач транзакції приховується за допомогою так званих таємних адрес. Пошук транзакцій, що надсилають кошти на таємні адреси чи з таємних адрес, можна робити лише за наявності так званого ключа перегляду, доступ до якого є лише у отримувача. Нарешті, кількість грошей у транзакції приховується за допомогою криптографічного протоколу RingCT [23], що полягає у шифруванні чисельних значень кількостей

грошей у транзакціях зі збереженням можливості перевірити суму кількостей у входах і виходах транзакції, щоб гарантувати відсутність інфляції.

Втім, жоден з методів приховування тих чи інших складових транзакцій не може бути застосований у Біткоїн-протоколі. Однією з основних цілей спільноти розробників Біткоїна є збереження або підвищення стійкості і надійності мережі, і тому одним з важливих аргументів проти реалізації цих методів у Біткоїні є те, що криптографічні інструменти, на які вони опираються, є занадто новими і недостатньо вивченими, і тому їх впровадження може бути дестабілізуючим або навіть небезпечним. Наприклад, якщо у криптографічному протоколі RingCT виявлять недолік, що дає змогу таємно генерувати додатковий біткоїн у кожній транзакції, це повністю нівелює одну з фундаментальних властивостей Біткоїна як грошей, і це може знищити його ціннісну пропозицію.

Транзакційну модель, що використовується у Біткоїні, називають UTXO-моделлю або моделлю невикористаних транзакційних виходів. Деякі інші криптовалютні системи, зокрема Ethereum [24], використовують більш інтуїтивну рахункову модель: користувач має єдиний рахунок, і транзакції віднімають певну кількість одиниць від цього рахунку і додають її до іншого рахунку. UTXO-модель має певні переваги порівняно з рахунковою моделлю, зокрема і з точки зору приватності даних користувачів:

1. UTXO-модель підтримує і навіть заохочує використання унікальних адрес для кожної транзакції, що значно покращує приватність користувачів, тоді як рахункова модель зв'язує всі транзакції з єдиним рахунком.
2. UTXO-модель має природний вбудований механізм обфускації виходів через механізм повернення решти з транзакцій.

3. UTXO-модель підтримує об'єднання операцій, оскільки Біткоїн-транзакція може містити входи та виходи різних власників. На цій можливості базується протокол децентралізованого змішування виходів CoinJoin (див. пункт 1.2.3).

Втім, ці особливості UTXO-моделі лише уможливлюють подібні типи транзакцій, але не вимагають їх використання. Зокрема, повторне використання однієї і тієї ж адреси фактично реалізує рахункову модель в межах UTXO-моделі. Тому покращення конфіденційності даних користувачів потребує систематичного підходу. На даний момент загальноприйняте бачення вирішення проблеми приватності транзакційних даних в Біткоїн-інфраструктурі полягає у впровадженні і вдосконаленні транзакційної гігієни а також використанні так званих протоколів другого рівня – протоколів-надбудов, що надають користувачам додаткову функціональність, яка не є доступною безпосередньо в Біткоїн-протоколі (який у цьому контексті називають протоколом першого рівня), і при цьому використовують Біткоїн-протокол як інструмент фіналізації і збереження цілісності інформації.

1.2.1. Транзакційна гігієна

Транзакційна гігієна – це набір практик і правил створення Біткоїн-транзакцій, що використовують можливості UTXO-моделі для мінімізації ризиків витоку персональної інформації, зокрема зв'язків між виходами, що належать одному й тому ж користувачу. Дотримання транзакційної гігієни є відповідальністю користувача. Однією з найбільш важливих практик транзакційної гігієни є уникання повторного використання псевдонімних криптографічних ідентифікаторів користувача (Біткоїн-адрес).

У перші роки існування мережі Біткоїн користувач, який хотів отримати біткоїн, надавав своєму контрагенту свій публічний ключ в якості

ідентифікатора. Цей публічний ключ безпосередньо використовувався у програмі блокування у виході відповідної транзакції. При цьому, за аналогією з криптографічними ключами, що використовувались для шифрування електронного листування, користувачі зазвичай мали єдиний публічний ключ для отримання біткоїна, і тому всі транзакції, які отримав той чи інший користувач, тривіально знаходились за його публічним ключем. Іншим методом ідентифікації отримувача, який згодом повністю витіснив публічні ключі, був метод ідентифікації за допомогою хеша публічного ключа. Програма блокування містить хеш публічного ключа, а програма розблокування – і сам публічний ключ, і відповідний підпис. Хоча цей метод має переваги, порівняно з ідентифікацією за допомогою публічного ключа (зокрема, менший розмір транзакцій та частковий захист від гіпотетичного обчислення приватного ключа з публічного за допомогою квантових комп'ютерів), але проблема пошуку всіх транзакцій за адресою залишається невирішеною: якщо використовується один і той же публічний ключ, то і адреса-хеш від цього публічного ключа буде однаковою для всіх виходів, оскільки хеш-функції є детерміністичними.

Для того, щоб уникати повторного використання адрес, користувач повинен створювати нову пару ключів для кожного окремого випадку отримання біткоїна. Іншими словами, кожен вихід, який йому належить, повинен мати іншу адресу, а отже і інший публічний ключ. Якщо користувач отримує регулярні платежі, то створення та зберігання великої кількості ключів стає все більш складною задачею. Ранні версії програмного забезпечення для зберігання ключів (так звані Біткоїн-гаманці) зазвичай використовували множини заготовлених ключів, і генерували додаткові ключі, коли їх запас доходив до певного порогового значення, але такий підхід був дуже незручним, оскільки вимагав від користувача постійно слідкувати за станом гаманця і не забувати робити резервні копії всіх нових ключів. Як наслідок, багато користувачів вдавались до повторного використання тих ключів, які вони вже надійно

зберегли, щоб уникнути ризику втрати коштів. Одним з найбільш важливих винаходів для спрощення процесу генерації та зберігання нових ключів став ієрархічний детерміністичний гаманець [25], який генерує дерево ключів практично необмеженого розміру з єдиного кореневого ключа за допомогою детерміністичних перетворень. Користувачу при цьому достатньо зберігати лише кореневий ключ, а всі інші можуть бути швидко переобчислені з нього в будь-який момент. Такий спосіб організації ключів швидко став стандартом і на сьогоднішній день використовуються у більшості програмного забезпечення для роботи з транзакціями, тому уникнення повторного використання адрес є простою і доступною практикою, якої можуть дотримуватись всі користувачі Біткоїна.

Існує багато інших практик і принципів транзакційної гігієни [26], дотримання яких є значно складнішим, наприклад уникання консолідації виходів (здійснення транзакції, що об'єднує багато малих виходів у один великий, що надсилається самому собі) чи розділення множин ключів, що використовуються для різних категорій контрагентів, тощо. У розділі 3 детально розглядаються евристичні припущення щодо транзакцій, які порушують такі правила.

1.2.2. Мережі платіжних каналів

Одним з найбільш важливих протоколів другого рівня є мережа транзакційних каналів Lightning [11], основною ціллю якої є масштабування та зменшення вартості Біткоїн-платежів.

Оскільки кожен учасник мережі повинен мати повну копію ланцюга блоків мережі, то швидкість збільшення розміру ланцюга впливає на рівень децентралізації мережі (чим більший розмір ланцюга, тим дорожче апаратне забезпечення потрібне для його зберігання). Як наслідок, Біткоїн-мережа має надзвичайно низьку пропускну здатність у порівнянні з традиційними платіжними системами – близько 5 – 7 транзакцій за секунду, тоді як платіжна система Visa зазвичай обробляє близько 2000

транзакцій на секунду, і може збільшувати пропускну здатність до 20 – 30 тисяч транзакцій за необхідності [27].

Lightning-мережа використовує спеціальні контракти (програми блокування у Біткоїн-транзакціях), що називаються платіжними каналами. Платіжний канал – це спільний транзакційний вихід, який дає змогу двом сторонам здійснювати миттєві і практично безкоштовні платежі. Початковий стан каналу – це підписана Біткоїн-транзакція, яка розподіляє спільний вихід відповідно до початкових внесків, а кожен платіж – це нова версія такої транзакції з новим розподілом коштів з вбудованим механізмом знищення попередньої версії транзакції. Проміжні платіжні транзакції не публікуються у Біткоїн-мережі, а при публікації останньої версії платіжної транзакції канал закривається і сторони отримують свої кошти відповідно до фінального розподілу. Ключова ідея Lightning-мережі полягає у тому, що множина таких каналів може використовуватись для маршрутизації платежів між каналами, навіть якщо між двома сторонами платежу немає прямого зв'язку, але існує шлях, що складається з кількох інших учасників мережі.

Оскільки всі платежі в такій мережі здійснюються без публікації в Біткоїн-мережі і задіюють лише тих учасників Lightning-мережі, які знаходяться на шляху між платником та одержувачем, то теоретична пропускну здатність такої системи є необмеженою. З цієї ж причини Lightning-платежі суттєво покращують приватність транзакційних даних користувачів: Біткоїн-мережа зберігає всю історію транзакцій, і ця історія є відкритою для всіх учасників мережі, тоді як Lightning-мережа не зберігає історію платежів, лише початковий і фінальний стан окремих платіжних каналів, а про деталі кожного платежу знають лише платник та отримувач. Завдяки цибулевій маршрутизації [11] посередники платежу, тобто ті учасники мережі, які знаходяться на шляху платежу, знають лише про своїх сусідів на шляху, але нічого не знають про інших посередників даного

платежу, зокрема і про початкову та кінцеву вершини шляху (платника та одержувача, відповідно).

На рисунку 1.3 зображено фрагмент мережі платіжних каналів Lightning та виділено шлях платежу від учасника A до учасника E , що складається з вершин A, B, C, D та E . Вершини B, C та D є посередниками платежу, оскільки між вершинами A та E не існує прямого зв'язку. Посередник B знає лише те, що попереднім кроком у платежі є посередник A , а наступним – вершина C , і при цьому не знає ні того, що A є першим кроком на шляху, ні того, яким є наступний крок після C . Вершини, що не беруть участі в даному платежі ($N_0 - N_9$), не знають навіть про факт здійснення платежу.

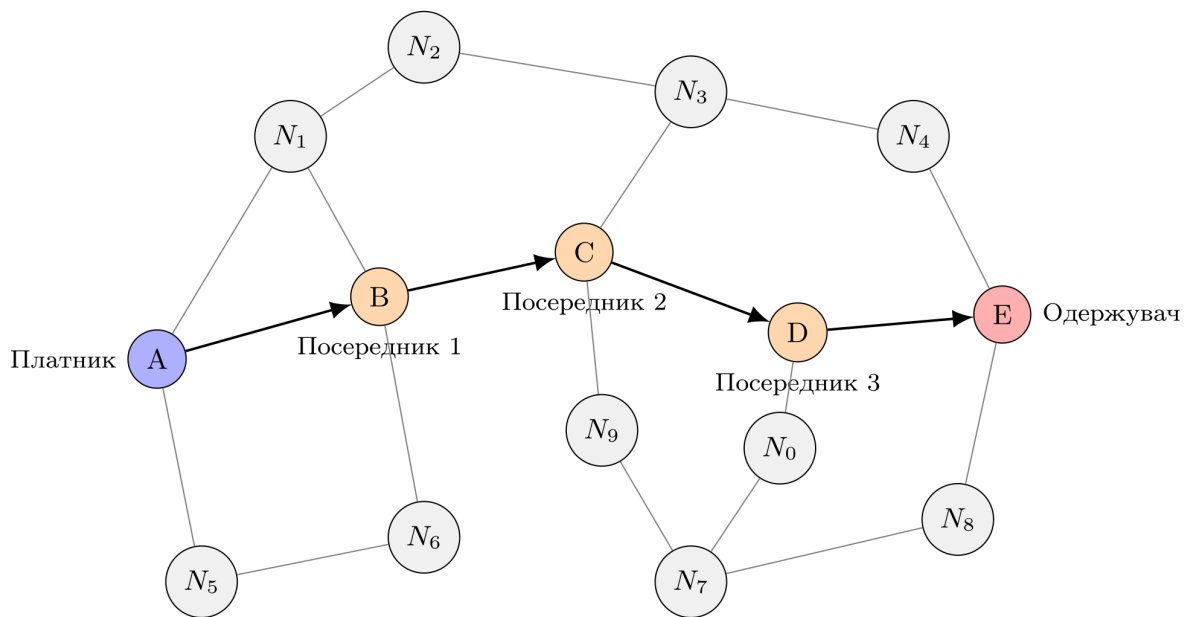


Рис. 1.3. Мережа платіжних каналів

Lightning-платежі не є повністю анонімними, адже відкриття і закриття каналів є звичайними Біткоїн-транзакціями, які потрапляють у ланцюг блоків, і їх порівняно легко відрізнити від інших транзакцій. Окрім цього, існують дослідження щодо можливостей деанонімізації Lightning-мережі зсередини шляхом встановлення вузлів-шпигунів, що

збирають інформацію про мережу загалом і про платежі, що проходять через них зокрема [28].

1.2.3. Децентралізоване змішування виходів

В аналізі Біткоїн-транзакцій часто використовується припущення про те, що типова транзакція створюється одним користувачем, а отже всі входи в ній і один з виходів (решта) належать цьому користувачу [8]. Втім, з точки зору правил Біткоїн-консенсусу всі входи та виходи транзакції є незалежними: єдиний зв'язок між входами та виходами полягає у тому, що сума біткоїна у входах транзакції повинна бути не меншою, ніж сума у виходах. Тому припущення про єдиного власника всіх входів транзакції в загальному випадку не є правильним, оскільки правила консенсусу цього не вимагають. При цьому більшість користувачів створюють свої транзакції самостійно, підвищуючи статистичну правильність такого припущення, і таким чином погіршуючи приватність даних у Біткоїн-мережі.

Метод децентралізованого змішування виходів CoinJoin був вперше запропонований Грегорі Максвелом ще у 2013 році [29]. Цей метод полягає у кооперативному об'єднанні кількох Біткоїн-транзакцій різних користувачів у єдину спільну транзакцію, що розриває емпіричний зв'язок між входами та виходами транзакцій. З тих пір CoinJoin розвинувся у сімейство протоколів, які відрізняють один від одного способом організації кооперації (пошук групи користувачів, обмін частково підписаними транзакціями, тощо).

CoinJoin-транзакція об'єднує велику кількість виходів різних користувачів і розподіляє сумарну кількість біткоїна між новими багатьма виходами, що унеможливорює співставлення надсилача коштів з одержувачем. Така транзакція розкриває лише те, що власник одного з її входів надіслав кошти власнику одного з її виходів, і ймовірність будь-якого варіанту співставлення надсилача і одержувача є однаковою.

Децентралізоване змішування виходів є менш зручним у використанні, оскільки кожна операція надсилання коштів потребує додаткового часу на пошук групи користувачів для кооперації, а також часто супроводжується підвищеними комісіями через розмір транзакції або навіть вартістю послуги, яку надає оркестратор кооперації.

CoinJoin-транзакції мають особливу структуру (велика кількість входів та виходів, однакова кількість входів та виходів, однакові кількості біткоїна у виходах та інші характерні властивості), що дає змогу легко відрізнити їх від інших транзакцій. Як наслідок, якщо такі транзакції є рідкісними, сам факт їх використання може бути маркером того, що користувач намагається приховати якусь фінансову діяльність. Щоб уникнути такого маркування, необхідна велика множина анонімності – множина користувачів, що рутинно використовують подібні транзакції для повсякденних операцій. У достатньо великій множині анонімності операції змішування виходів кожного окремого користувача “губляться” і втрачають зміст як маркери нетипової поведінки.

1.2.4. Безшумні платежі та альтернативні схеми адресації

Безшумні платежі [30] (англ. silent payments) – це протокол-надбудова, що реалізує варіацію методу утворення одноразових адрес криптовалютного протоколу Monero [20]. Основний принцип протоколу безшумних платежів полягає в тому, що одержувач публікує лише один публічний ключ, а будь-який надсилач у будь-який момент може самостійно генерувати з цього публічного ключа нову унікальну Біткоїн-адресу за допомогою обміну ключами Діффі – Геллмана [31]. Ключ відправника, необхідний для здійснення обміну, розміщується у спеціальному виході-носії даних у транзакції.

Безшумні платежі є одним з альтернативних методів формування ідентифікаторів власника у виходах транзакцій – альтернативних схем адресації. Окрім протоколу безшумних платежів, існують і інші

альтернативні схеми адресації, зокрема PayNym [32]. Безшумні платежі мають переваги як стандартною схемою адресації, так і над іншими альтернативними схемами:

1. Підвищена приватність користувачів: використання безшумних платежів розриває прямий зв'язок між адресою одержувача і виходами, що йому належать, адже публічна адреса одержувача у жодному вигляді не є частиною жодного виходу.
2. Простота використання: попередня підготовка складається лише з генерації єдиного публічного ключа, а створення нових платежів є неінтерактивним, тобто не потребує обміну повідомленнями.
3. Сумісність з існуючою інфраструктурою: протокол використовує існуючу функціональність і може бути реалізований без будь-яких змін у Біткоїн-протоколі, безпосередньо у застосунках-гаманцях.

Основним недоліком протоколу безшумних платежів є те, що він споживає більше місця у ланцюгу блоків і потребує більшої кількості обчислювальних ресурсів для пошуку транзакцій, що адресовані користувачу. В стандартній схемі адресації користувач сканує всі нові виходи, і відбирає ті, що містять один з його ключів (адрес). Для пошуку безшумного платежу користувачу потрібно для кожного виходу-носія даних, що містить публічний ключ, повторити обмін ключами Діффі – Геллмана, щоб отримати спільний ключ між ним і відправником, і лише тоді перевірити, чи саме цей ключ міститься у цільовому виході. Такий процес сканування виходів є значно більш складним з точки зору кількості обчислень.

Альтернативні схеми адресації можна розглядати як інструменти, що покращують транзакційну гігієну. Протокол безшумних платежів приховує зв'язок між адресою і виходами з точки зору зовнішнього спостерігача, але не з точки зору контрагентів. Користувачка протоколу безшумних платежів Еліс гарантовано використовує унікальну адресу у кожному виході, і при цьому адреса, яку вона надсилає своїм контрагентам, ніколи не потрапляє у

реєстр транзакцій, тому зовнішній спостерігач Боб не може знайти жодного виходу, що належить Еліс. Але якщо Боб є одним з контрагентів Еліс, то він знає, який саме вихід він створив, і тому одразу помітить інші виходи Еліс у момент, коли вона помилково чи навмисне здійснить операцію, що встановлює якісь зв'язки між виходами, наприклад консолідацію.

1.3. Методи дослідження транзакційних даних

Оскільки входи у Біткоїн-транзакціях посилаються на виходи інших транзакцій, транзакції утворюють граф залежностей: кожна транзакція залежить (у причинно-часовому сенсі) від виходів попередніх транзакцій, які вона споживає. Цей граф безпосередньо використовується у Біткоїн-протоколі, оскільки деякі правила консенсусу прямо стосуються зв'язків між транзакціями та їхніми предками. Так як весь ланцюг блоків Біткоїна та усі транзакції в ньому є відкритими, то використовуючи лише дані ланцюга, можна побудувати граф зв'язків між усіма транзакціями, що коли небудь відбулись. З цього графа можна отримати як загальну інформацію про типові операції (патерни у графі), частоту їх використання, тощо, так і інформацію про окремі випадки взаємодії між конкретними користувачами (регулярні платежі та інші складні фінансові зв'язки між групами користувачів).

Перші спроби аналізу графа Біткоїн-транзакцій з'явилися ще у 2011 – 2012 роках. Деякі з ранніх досліджень зосереджуються на зборі статистик про активність мережі [33, 34], а інші обговорюють саме можливості деанонімізації користувачів [35, 36]. У цей час формулюється задача кластеризації Біткоїн-адрес: маючи один публічний ключ (адресу), необхідно знайти якомога більше публічних ключів (адрес), що належать тому ж користувачу. Більшість описаних методів кластеризації базуються на евристичних припущеннях про поведінку примітивних програм-гаманців, що існували на той час. Деякі новіші дослідження пропонують моделі для аналізу нестандартних транзакцій [37].

Доріт Рон і Аді Шамір [33] будують граф Біткоїн-адрес для перших 180000 блоків (3 січня 2009 року – 13 травня 2012 року) і згортають його у граф користувачів за допомогою алгоритму Union-Find [38] на основі евристичного сформульованого ще Сатоші Накамото припущення, що виходи, які є входами однієї і тієї ж транзакції, ймовірно належать одному і тому ж користувачу. Користувач при цьому визначається як множина адрес, що належать одній і тій же сутності. Як зазначено у статті, станом на 2012 рік таке припущення мало дуже малу кількість винятків, бо хоча Біткоїн-протокол дає змогу групі користувачів кооперувати при створенні транзакції, на той час дуже така кооперація була надзвичайно рідкісна. Втім, нові дослідження вказують на те, що кооперація при створенні транзакцій могла бути поширена вже з перших днів існування Біткоїн-мережі [39], а з тих пір кількість транзакцій, що є винятками з такого припущення, значно зросла, оскільки з'явилися спеціальні протоколи підвищення анонімності транзакцій за допомогою кооперації спонтанно сформованих груп користувачів [29].

Фергал Рід і Мартін Харіган [35] виділяють два окремих підграфи загального графа транзакцій: граф транзакцій, який відображає рух біткоїна між транзакціями, і граф користувачів, який відображає рух біткоїна між адресами. Такий підхід уможливорює повторювані транзакції в межах спільноти користувачів, тощо. Обговорюється побудова часткового реєстру користувачів на основі добровільної публікації користувачами адрес на форумах, а також асоціацій між Біткоїн-адресами та IP-адресами користувачів з ранніх сервісів безкоштовного розповсюдження біткоїна.

Майкл Фледер та ін. [36] кластеризують граф адрес за допомогою алгоритму PageRank [40] і знаходять користувачів форумів, які взаємодіють з “відомими” вершинами, такими, як Біткоїн-казино чи сайти нерегульованої торгівлі.

Одним з найбільш важливих досліджень приватності даних в Біткоїні є дослідження Джонаса Ніка [41, 42]. У ньому детально розглядається

задача евристичної кластеризації Біткоїн-адрес, але найцікавішою частиною є каталог евристичних припущень щодо поведінки програм-гаманців, зокрема:

1. Припущення про єдиного власника всіх входів однієї транзакції; саме на цьому припущенні базуються дослідження, згадані вище.
2. Припущення про те, що більшість транзакцій, здійснених індивідуальним користувачем, мають не більше 2-х виходів.
3. Припущення про наявність виходу-решти у транзакції, здійсненій індивідуальним користувачем.
4. Припущення про те, що вихід-решта мусить бути меншим за якийсь з входів, інакше транзакція була сконструйована неоптимально (див. пункт 3.3.1).
5. Припущення про те, що вихід-решта повинен співпадати за типом адреси з входами, оскільки всі адреси були створені тією ж програмою-гаманцем (див. пункт 3.3.1).
6. Припущення про те, що вихід-решта завжди використовує адресу, яка раніше не використовувалась (див. пункт 3.3.1).

У дослідженні Джонаса Ніка продемонстровано, що деанонімізація шляхом евристичної кластеризації адрес працює ефективніше, якщо використовується декілька евристик. По-друге, наявність каталогу таких евристик дає змогу розробникам програм-гаманців адаптувати процес створення транзакцій так, щоб не порушувати приватність своїх користувачів.

Евристики, що розглядаються у згаданих вище дослідженнях, стосуються поведінки програмного забезпечення. Прості евристичні припущення для розпізнавання типів транзакцій використовується у відомих сервісах огляду ланцюга блоків. Зокрема, сервіс mempool.space [43] надає інструмент фільтрування транзакцій у блоках за типом, який називається Mempool Goggles™ [44]. Цей інструмент підтримує евристичний пошук операцій консолідації, операцій змішування виходів

(CoinJoin) та операцій платежів декільком отримувачам. Приклад використання інструменту Mempool Goggles™ для пошуку операцій консолідації у блоці 908623 зображено на рисунку 1.4. Основним недоліком цього інструменту є те, що він працює з окремими транзакціями, і не підтримує розпізнавання операцій, що складаються з 2-х і більше транзакцій. Фільтрування окремих транзакцій за допомогою подібних евристик не потребує побудови і підтримки графа транзакцій і може бути легко і ефективно реалізоване на основі реляційної бази даних з таблицею транзакцій, яка містить поля кількості входів та виходів для кожної транзакції. Розпізнавання операцій, що складаються з 2-х і більше транзакцій, потребує розгляду комбінацій з 2-х і більше транзакцій, і складність такого пошуку швидко збільшується залежно від кількості транзакцій у патерні, але все ще може бути реалізовано ефективно за допомогою графової бази даних.

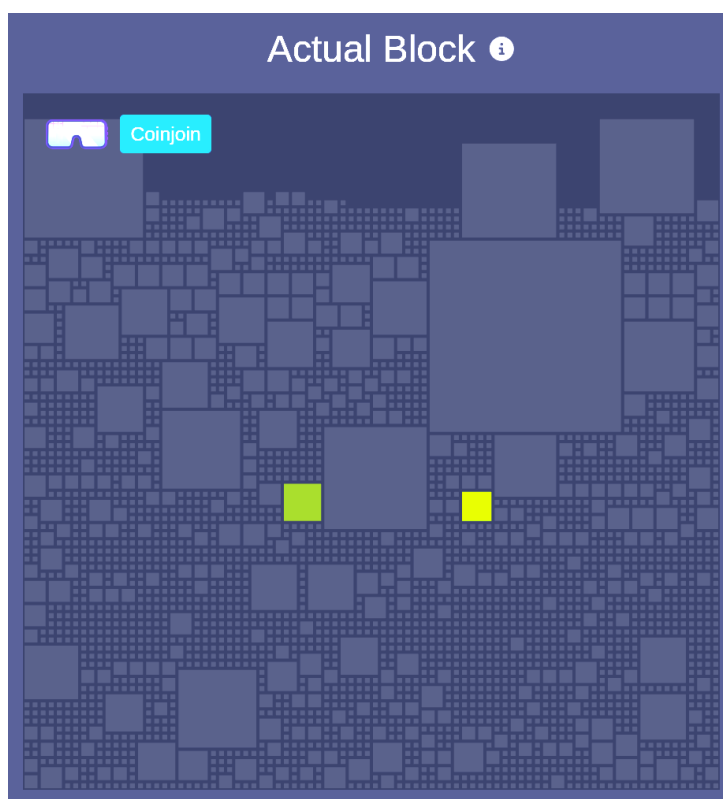


Рис. 1.4. Mempool Goggles™

У цьому дослідження розглядається підхід до аналізу графа транзакцій, що реалізує процес кластеризації виходів як ітеративну пропагацію зв'язків власності у графі, і використовує для цього евристики, що стосуються як поведінки програмного забезпечення, так і поведінки користувачів. На відміну від евристик, наведених вище, такі евристики можуть стосуватись як однієї транзакції, так і груп транзакцій. Для цього в рамках дослідження розробляється спеціалізована модель транзакційних даних і архітектура програмного забезпечення для побудови і зберігання такого графа у представленні, що дає змогу знаходити у ньому патерни груп транзакцій, а також доповнювати його додатковою інформацією, такою, як наприклад інформація про кластери, отримана з попередніх ітерацій аналізу, або зовнішньою інформацією про користувачів.

1.4. Формулювання вимог до програмного забезпечення

Для реалізації та тестування методів пошуку типових операцій та кластеризації виходів, що розглядаються у розділі 3 цього дослідження, необхідно розробити програмний комплекс, який дає змогу будувати і зберігати актуальний граф транзакцій у RDF-представленні, розробленому у розділі 2 цього дослідження, а також надає середовище виконання SPARQL-запитів і можливість створення спеціалізованих фільтр-функцій для спрощення SPARQL-запитів пошуку підграфів такого графа зі складними зв'язками.

Перед початком розроблення такої програмної системи необхідно зібрати, формалізувати і пріоретизувати функціональні та нефункціональні вимоги до програмного забезпечення.

Вимоги до програмної системи, наведені нижче, сформульовано у моделі класифікації атрибутів якості програмного забезпечення FURPS+ (Functionality – Функціональність, Usability – Зручність використання, Reliability – Надійність, Productivity – Продуктивність, Supportability – Зручність супроводу, “+” – Інше).

1. Функціональність:

- 1.1. Система повинна отримувати блоки через прикладний програмний інтерфейс вузла Біткоїн-мережі і парсити ці блоки і їхні компоненти (транзакції та виходи).
- 1.2. Система повинна будувати граф зв'язків між транзакціями і зберігати його у базі даних.
- 1.3. Система повинна надавати можливість пошуку як всіх підграфів, що відповідають певному патерну у графі, так і конкретного підграфа з заданими параметрами.
- 1.4. Система повинна надавати можливість доповнення графа транзакцій новими даними з нових блоків у фоновому режимі.
- 1.5. Система повинна надавати можливість розширення графа транзакцій анотаціями, що містять довільну додаткову інформацію.
- 1.6. Система пошуку патернів у графі повинна мати можливість фільтрування за довільними анотаціями.

2. Зручність використання:

- 2.1. Система повинна використовувати стандартні формати даних для легкої інтеграції з зовнішніми інструментами та сервісами.
- 2.2. Система повинна надавати стандартизований гнучкий інтерфейс для пошуку у графі транзакцій.
- 2.3. Система повинна відображати статистичні дані про граф транзакцій (кількість блоків, транзакцій різних типів, тощо).

3. Надійність:

- 3.1. Система повинна гарантувати збереження цілісності та відсутності дублікації даних у випадку аварійного завершення роботи будь-якого компоненту.

- 3.2. Система повинна коректно обробляти випадки реорганізації ланцюга блоків.
- 4. Продуктивність:
 - 4.1. Система повинна мати можливість зберігати граф всієї історії транзакцій (до 1.5 мільярда транзакцій).
 - 4.2. Початкова побудова графа транзакцій повинна відбуватись за менше ніж 720 годин (30 днів).
 - 4.3. Система повинна підтримувати паралельне розширення графа та паралельний пошук у графі.
 - 4.4. Система повинна надавати можливість кешування результатів пошуку.
- 5. Зручність супроводу:
 - 5.1. Система повинна надавати можливість централізованого запуску і зупинки всіх компонентів.
 - 5.2. Запуск системи повинен підтримувати режим відтворення (видалення попереднього стану і створення нового).
 - 5.3. Система повинна забезпечувати логування всіх компонентів.

1.5. Висновки

У цьому розділі проаналізовано модель транзакцій, яка використовується у Біткоїн-протоколі, і розглянуто проблему приватності фінансових даних користувачів Біткоїна, яка є наслідком прозорості глобального реєстру транзакцій, що в свою чергу є необхідною властивістю для досягнення стійкого децентралізованого консенсусу. Також розглянуто концептуальні підходи до покращення приватності фінансових даних користувачів та їхні недоліки.

Проаналізовано переваги та недоліки методів аналізу графа Біткоїн-транзакцій у відомих дослідженнях. Зокрема, ранні дослідження розглядають лише малу частину графа транзакцій, деякі ранні евристичні припущення втратили актуальність, нові випадки використання Біткоїна

потребують формулювання нових евристик, а аналіз складних багатокрокових операцій можливий лише в межах добре спроектованого програмного комплексу, який здатний підтримувати граф всієї історії транзакцій і здійснювати швидкий пошук патернів довільної складності у ньому. Подібні програмні комплекси, ймовірно, вже існують, але є пропрієтарними і закритими, що саме по собі може бути проблемою для індивідуальних користувачів Біткоїна і інших криптовалют, якщо непрозорі висновки таких систем використовуються правоохоронними органами.

Виходячи з аналізу відомих публікацій щодо дослідження графа Біткоїн-транзакцій, сформульовано загальну ідею нового підходу до дослідження графа транзакцій, що полягає у побудові такого графа для всієї історії транзакцій з використанням моделі транзакційних даних, що є адаптованою для пошуку евристичних патернів операцій, які можуть складатись з однієї і більше транзакцій і відбуватись протягом довгих періодів часу.

Насамкінець, розроблено вимоги до програмного комплексу, що дає змогу будувати, підтримувати і розширювати граф всієї історії Біткоїн-транзакцій, що в свою чергу дає змогу розпізнавати операції незалежно від того, наскільки розкиданими в часі є їхні окремі кроки.

2. РОЗРОБЛЕННЯ СПЕЦІАЛІЗОВАНОЇ МОДЕЛІ ДАНИХ БІТКОЇН-ТРАНЗАКЦІЙ

2.1. Підграфи графа Біткоїн-транзакцій

Основною структурою даних в Біткоїні є транзакційний вихід – віртуальна “монета”, яка представляє певну кількість біткоїна та її власника. Основною подією в Біткоїн-протоколі є транзакція – акт знищення певних виходів і розподілу біткоїна, що у них містився, між новими власниками шляхом створення нових виходів. Граф Біткоїн-транзакцій може відображати як “монети” біткоїна у обігу (виходи), так і події переходу власності над цими монетами від одного користувача до іншого (транзакції).

У графі Біткоїн-транзакцій можна виділити два самостійних підграфи, що містять лише вершини одного типу: граф виходів (і його більш поширена модифікація – граф адрес) та граф транзакцій. Кожен з цих двох графів має певні переваги і недоліки, які роблять їх більш або менш придатними для тих чи інших видів аналізу. Оскільки основною задачею аналізу графа транзакцій, що розглядається у цьому дослідженні є розпізнавання патернів типових операцій, то найкращим типом графа транзакцій є комбінований граф транзакцій та виходів. Для демонстрації переваг та недоліків всіх трьох типів графів для кожного з них наведено приклад представлення однієї з найбільш поширених типових операцій – операції консолідації виходів. Операція консолідації та інші типові операції детально розглядаються у підрозділі 3.3.

2.1.1. Граф виходів

Граф виходів – це орієнтований граф, вершинами у якому є транзакційні виходи, а дугами – зв'язки “вихід – вхід – транзакція – вихід”. У графі виходів між двома вершинами-виходами O_1 і O_2 існує дуга, якщо існує транзакція, у якій O_1 є одним з входів, а O_2 – одним з виходів. Дуги у

графі виходів відображають перекази грошей між власниками і можуть мати ваги – суми відповідних переказів (кількості біткоїнів у відповідних виходах).

На рисунку 2.1 зображено граф виходів, що містить 10 виходів O_i . Зокрема, вершини O_3 та O_4 мають по одній дузі до кожної з вершин O_5 , O_6 та O_7 , що вказує на те, що існує транзакція, для якої O_3 та O_4 є входами, а O_5 , O_6 та O_7 – виходами.

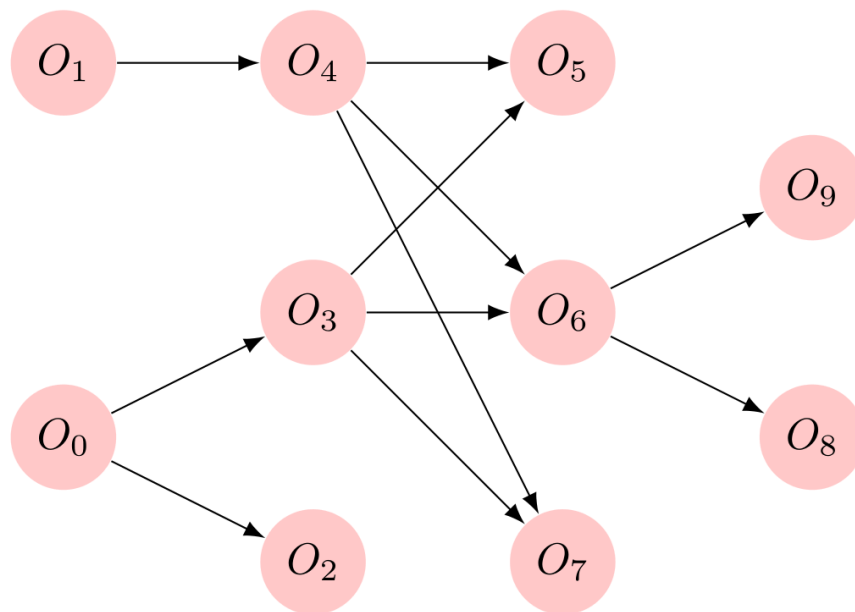


Рис. 2.1. Граф виходів

Частковим випадком графа виходів є граф адрес або граф власників, у якому вершинами є не виходи, а адреси. Основною відмінністю між графом виходів та графом адрес є те, що виходи є унікальними (ідентифікатор виходу – це ідентифікатор транзакції, яка його створила, та його порядковий номер у транзакції), тоді як адреси, які є компонентами виходів, можуть повторюватись (декілька різних виходів різних транзакцій можуть мати одну й ту ж адресу). Через це граф виходів є ациклічним, а граф адрес може містити цикли. Окрім цього, граф виходів в загальному випадку є мультиграфом, оскільки між двома адресами може бути декілька

операцій переказу коштів. Відображення кожної такої операції як окремого ребра дає змогу легко розпізнавати повторювані взаємодії між двома користувачами.

Саме граф адрес розглядається у ранніх дослідженнях, пов'язаних з аналізом Біткоїн-транзакцій [33 – 35] і досі залишається основним інструментом аналізу транзакцій завдяки наступним особливостям:

1. Граф адрес відображає взаємодію між користувачами і дає змогу виявляти групи економічно пов'язаних між собою користувачів.
2. Граф адрес можна розглядати як соціальний граф, у якому соціальний зв'язок – це фінансова взаємодія. Як наслідок, до графа адрес можна застосовувати існуючий доробок методів соціального аналізу.
3. Вершини графа адрес самі по собі є псевдонімами ідентифікаторами користувачів, тому такий граф найкраще підходить для задачі ідентифікації.

Граф адрес також має і важливі недоліки, оскільки такий граф є добре пристосованим для відображення рахункової моделі транзакцій, тоді як Біткоїн використовує UTXO-модель:

1. В Біткоїн-протоколі не існує прямого зв'язку між двома виходами чи адресами. Всі входи транзакції пов'язані з усіма виходами (див. вершини $O_3 - O_7$ на рисунку 2.1), що робить граф виходів або адрес заплутаним.
2. Для спрощення графа використовуються евристики (наприклад, припущення про спільну власність над входами однієї транзакції), які можуть бути помилковими (наприклад, для CoinJoin-транзакцій).
3. Граф адрес містить багато шуму, оскільки правила транзакційної гігієни схиляють користувачів до використання одноразових та тимчасових адрес.

На рисунку 2.2 зображено операцію консолідації у графі виходів – транзакцію, що має багато входів і один вихід.

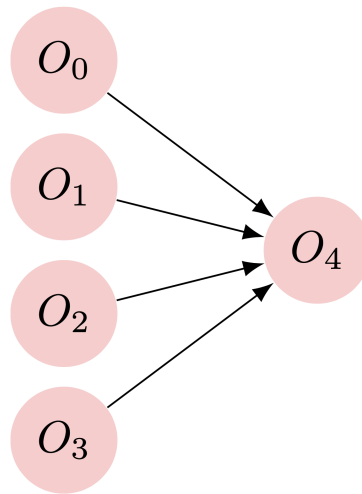


Рис. 2.2. Операція консолідації у графі виходів

Оскільки граф виходів не відображає кількості входів та виходів транзакцій, пошук патерну консолідації потребує розгляду декількох вершин для кожної вершини-кандидата: спочатку потрібно знайти вершини, у яких є багато вхідних дуг, після цього для кожної такої вершини потрібно отримати вершину, суміжну за однією з вхідних дуг, і перевірити, що вона має лише одну вихідну дугу. Граф транзакцій та виходів, про який йдеться далі, дає змогу здійснити такий пошук розглядом лише вершин-кандидатів.

2.1.2. Граф транзакцій

Граф транзакцій – це орієнтований граф, вершинами у якому є транзакції, а дугами – зв'язки “транзакція – вхід – вихід – транзакція”. У графі транзакцій між двома вершинами-транзакціями T_1 і T_2 існує дуга, якщо один з виходів транзакції T_1 є входом транзакції T_2 . Дуги у графі транзакцій відображають логічні залежності одних транзакцій від інших.

На рисунку 2.3 зображено граф транзакцій, що містить 8 транзакцій T_i та відповідні зв'язки між ними. Декілька виходів однієї транзакції

можуть бути входами іншої транзакції, тому між двома вершинами графа транзакцій може бути декілька дуг (див. вершини T_6 та T_7), а отже такий граф теж може бути мультиграфом. Наявність декількох ребер між двома транзакціями є аномалією, що може вказувати на специфічні зв'язки між виходами (наприклад, надсилання біткоїна самому собі, тощо).

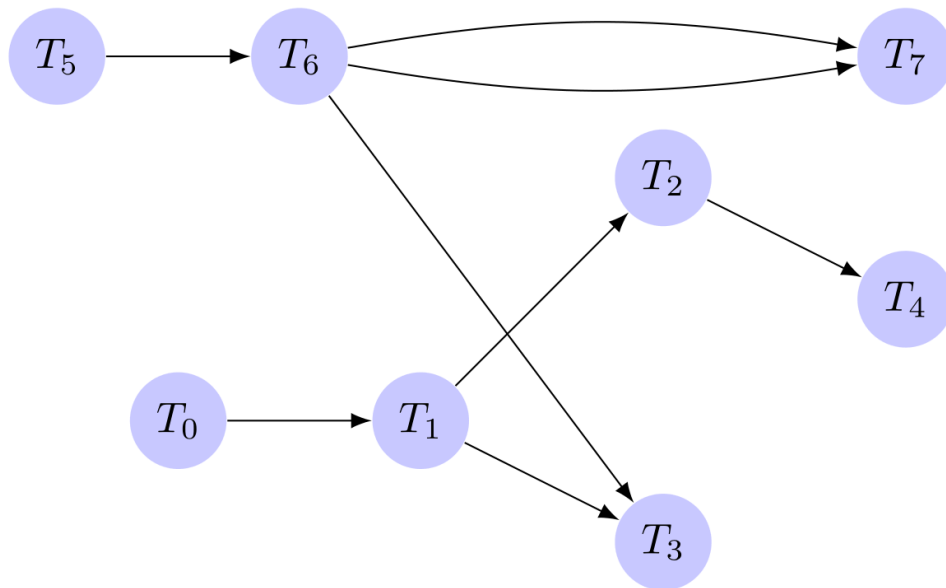


Рис. 2.3. Граф транзакцій

За винятком декількох випадків повторного використання однієї і тієї ж породжуючої транзакції на ранніх етапах існування мережі [45], кожній транзакції відповідає унікальна вершина у графі. Оскільки транзакція може використовувати лише виходи, які вже існують, то вона не може посилатись на транзакцію, що ще не відбулась, а отже граф транзакцій завжди є ациклічним.

Граф транзакцій повністю ігнорує будь-які ідентифікатори користувачів (ключі чи адреси) і зосереджується лише на послідовності та взаємозалежності транзакцій. Оскільки такий граф відображає логічну послідовність подій зміни власності, він дає змогу здійснювати аналіз потоку коштів в мережі.

Використання графа транзакцій для моделювання та аналізу Біткоїн-транзакцій має наступні переваги:

1. У такому графі використовуються вершини одного типу – транзакції. Кількість вершин у графі дорівнює кількості транзакцій у ланцюгу, і станом на червень 2025-го року становить 1.2 мільярда. Оскільки кожна транзакція в середньому містить кілька виходів, то граф транзакцій є значно меншим за граф виходів.
2. Такий граф повністю зберігає логіку переходу вартості у мережі, використовуючи при цьому мінімальну кількість вершин.
3. Такий граф є менш чутливим до використання унікальних адрес, порівняно з графом виходів.
4. Оскільки такий граф є орієнтованим ациклічним графом, під час його аналізу можна використовувати більш ефективні алгоритми обходу, сортування, пошуку патернів тощо [46].

Втім, в більшості випадків аналіз грошових потоків і пошук патернів та аномалій у транзакційних даних має на меті ідентифікацію користувачів, і для використання графа транзакцій для вирішення цієї задачі має декілька важливих недоліків:

1. На відміну від графа виходів, граф транзакцій за визначенням не містить ідентифікаторів користувачів (ключів та адрес), що одразу робить його менш придатним для задачі ідентифікації.
2. Деякі Біткоїн-операції (наприклад, повернення решти самому собі або змішування виходів, як централізоване, так і децентралізоване) значно легше розрізнити, коли у графі присутні адреси та кількості біткоїна у виходах.
3. Такий граф не відображає невикористані транзакційні виходи. Для невикористаного виходу не існує транзакції, для якої він є входом, і тому факт існування невикористаного виходу не може бути представлений у графі транзакцій. Ця особливість є важливим

недоліком з огляду на те, що Біткоїн набуває популярності в першу чергу як інструмент заощаджень, тобто все більша кількість біткоїна осідає в невикористаних виходах на довгий час.

4. Через те, що граф транзакцій не моделює невикористані виходи, кількість вихідних зв'язків транзакції не є остаточною, і в будь-який момент часу у транзакції у такому графі може з'явитися новий зв'язок. Як наслідок, аналіз такого графа повинен постійно повертатись до старих транзакцій для повторного розгляду на випадок, якщо з'явилися нові зв'язки.

На рисунку 2.4 зображено операцію консолідації у графі транзакцій. На перший погляд такий патерн легко розпізнати, але слід звернути увагу на те, що оскільки граф транзакцій не відображає виходів, ми не знаємо, скільки виходів є у транзакції T_4 . Якщо у графі присутні дві і більше вихідні транзакції (T_5 та T_6), це означає, що у транзакції T_4 щонайменше 2 виходи, тому вона не є консолідацією. У всіх інших випадках ми не знаємо точно, чи вона є консолідацією, чи ні. Цю проблему можна вирішити за допомогою атрибута вершини, що містить інформацію про кількість виходів, але це ускладнює як візуальний, так і автоматизований аналіз такого графа.

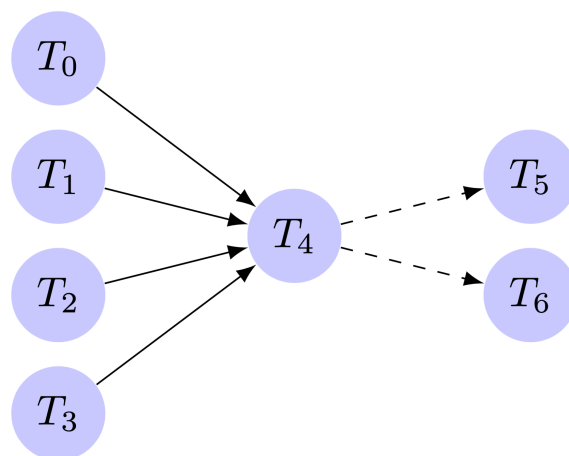


Рис. 2.4. Операція консолідації у графі транзакцій

2.1.3. Граф транзакцій та виходів

Граф транзакцій та виходів – це орієнтований біграф, що складається з вершин двох типів, вершин-транзакцій і вершин-виходів, та ребер двох типів, що представляють зв'язки “вихід – транзакція” і зв'язки “транзакція – вихід”. У такому графі між вершиною-виходом O_1 та вершиною-транзакцією T_1 існує зв'язок $O_1 - T_1$, якщо вихід O_1 є входом транзакції T_1 , або зв'язок $T_1 - O_1$, якщо вихід O_1 є виходом транзакції T_1 .

На рисунку 2.5 зображено граф транзакцій і виходів, що містить 3 транзакції T_i та 9 виходів O_i . Кожен вихід у такому графі однозначно пов'язаний і з транзакцією, яка його створила, і з транзакцією, яка його витратила.

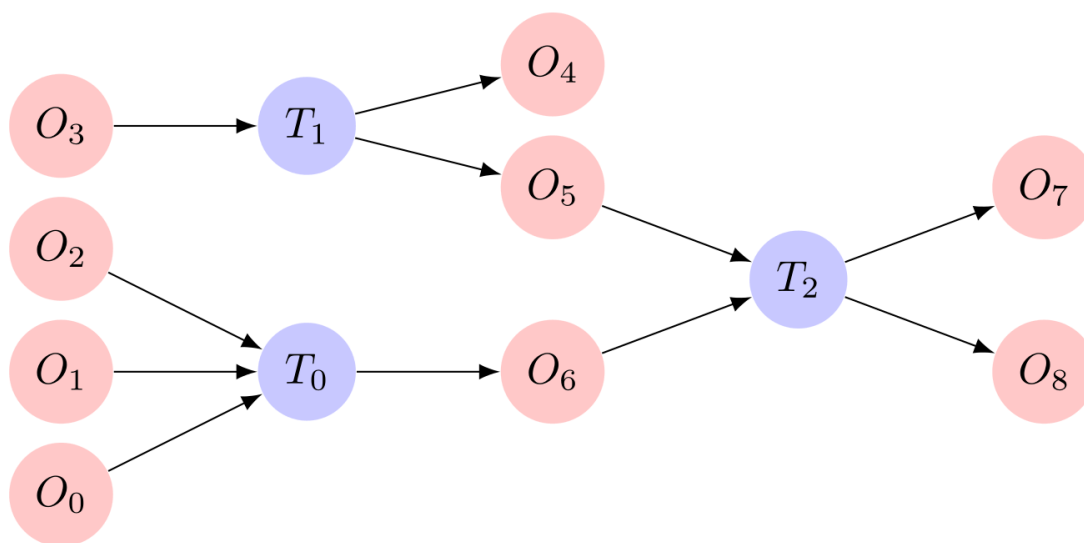


Рис. 2.5. Граф транзакцій і виходів

Граф транзакцій та виходів містить найповнішу інформацію про транзакції, оскільки він містить в собі і граф виходів і граф транзакцій. І граф виходів (адрес), і граф транзакцій можуть бути побудовані з графа транзакцій і виходів шляхом простої трансформації. Носіями інформації про тип операції є транзакції (зокрема кількості входів та виходів у них), тоді як носіями інформації про ідентифікатори власників і кількості

біткоїна у власності є виходи, тому лише граф, що містить і виходи і транзакції, відображає інформацію про типові операції і дає змогу розпізнавати патерни, що їм відповідають (див. розділ 3).

Єдиним недоліком графа транзакцій та виходів порівняно з графом виходів (адрес) чи графом транзакцій є те, що він потребує більше простору для представлення, оскільки містить в собі і граф виходів і граф транзакцій.

На рисунку 2.6 зображено операцію консолідації виходів, подану у вигляді графа транзакцій та виходів. Кількість вхідних та вихідних дуг вершини-транзакції однозначно вказує на кількість входів та виходів, і пошук патерну консолідації зводиться до пошуку всіх вершин-транзакцій, у яких багато вхідних дуг і лише одна вихідна, без необхідності розглядати суміжні вершини. Окрім того, патерни у такому графі є інтуїтивно більш зрозумілими при візуальному аналізі.

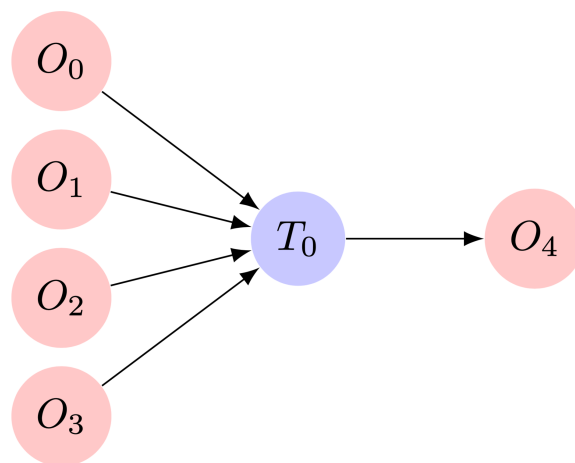


Рис. 2.6. Операція консолідації у графі транзакцій та виходів

Патерновий аналіз графа Біткоїн-транзакцій, що детально розглядається у розділі 3, використовує евристичні припущення для розпізнавання підграфів, які відповідають тим чи іншим операціям. Властивості, на які опираються дані припущення, є властивостями як транзакцій (кількості входів, виходів) так і виходів (адреса, кількість), тому

саме граф транзакцій та виходів є найкращою моделлю для такого аналізу, і саме цей граф буде розглядатись у решті тексту.

2.2. Вибір технології подання великих графів

У графі транзакцій та виходів природньо присутні дуги двох типів – дуги “транзакція – вихід” і дуги “вихід – вхід – транзакція”. Більшість виходів мають асоційовану адресу (ідентифікатор власника), і декілька виходів можуть мати одну й ту ж адресу (негігієнічна практика повторного використання адрес), що є тривіальним випадком асоціації декількох виходів з одним власником. Загалом пошук всіх вершин-виходів для певної адреси потребує обходу та фільтрування вершин-виходів за значенням відповідного атрибута, але введення додаткового типу вершини-адреси та відповідного типу дуги (“вихід – адреса”) дасть змогу швидко знаходити всі такі виходи як множину вершин, що мають дугу відповідного типу до вказаної вершини-адреси, а також автоматично візуалізує випадки повторного використання адрес. Насамкінець, абстрактні власники виходів та відповідні зв'язки “вихід – власник” теж можуть бути безпосередньо представлені у графі з аналогічних міркувань. При цьому, зв'язки “вихід – власник” можуть бути розділені на декілька типів, залежно від джерела походження інформації з метою розрізнення зовнішніх (точних) та внутрішніх (виведених) асоціацій, а зв'язки між вершинами-власниками можуть відображати певні типи фінансових відносин (“працедавець – працівник”, тощо). Як бачимо, граф транзакцій та виходів, доповнений вершинами-власниками, відображає множину фактів про транзакції, виходи, адреси, власників та різні види асоціацій між ними. Кожен з типів дуг у такому графі має зовсім інше семантичне значення, тому його зручно розглядати як граф знань про транзакції.

В загальному випадку, граф знань (англ. knowledge graph) – це модель структурування інформації, яка відображає її у вигляді графа, у якому вершини представляють об'єкти та поняття, а ребра – семантичні зв'язки

між ними. Така структура дає змогу формалізувати знання про певну предметну область і зробити їх зручними для комп'ютерної обробки, зберігаючи в той же час зручність інтерпретації людиною. Граф знань про певну предметну область складається з сутностей – реальних або абстрактних об'єктів, атрибутів – значень властивостей цих сутностей, та з семантичних зв'язків між сутностями. І атрибути і зв'язки у графі знань представляються у вигляді трійок “суб'єкт – предикат – об'єкт”, де суб'єкт – ідентифікатор сутності предметної області, об'єкт – ідентифікатор сутності або значення атрибута, а предикат – ідентифікатор зв'язку або атрибута.

Основні переваги подання інформації у вигляді графа знань:

1. Семантичність: зв'язки мають чітко визначене значення, що дає змогу і машинам і людям однозначно “розуміти” зміст відношення між сутностями чи атрибута.
2. Гнучкість: граф знань легко розширюється новими сутностями та зв'язками.
3. Сумісність: існує набір стандартів для побудови онтологій та побудови графів знань, зокрема формальних синтаксисів текстового представлення, які гарантують сумісність програмного забезпечення, яке використовує ці стандарти.

Однією з найбільш поширених технологій для створення графів знань та онтологій є технологія RDF (англ. Resource Description Framework – середовище опису ресурсів). RDF – це набір стандартів [47 – 49], розроблених консорціумом W3C [50], що визначають принципи подання інформації про об'єкти та зв'язки між ними в уніфікованому форматі у вигляді трійок “суб'єкт – предикат – об'єкт”, які називаються твердженнями.

Фундаментальним поняттям технології RDF є поняття ресурсу. Ресурс з точки зору RDF – це будь-що, про що можна зробити певне твердження. Ресурсом може бути конкретний об'єкт реального світу (місто, відома особа, тощо), цифровий об'єкт (файл, електронна пошта, тощо) або

навіть абстрактне поняття (“товаришувати”, “володіти”, тощо). Ресурси у твердженнях представляються інтернаціоналізованими ідентифікаторами ресурсів (англ. IRI – Internationalized Resource Identifier) [51] – текстовими ідентифікаторами з чітко визначеною синтаксичною формою, які є узагальненою версією уніфікованих ідентифікаторів ресурсів (англ. URI – Uniform Resource Identifier) [52]. Інтернаціоналізовані ідентифікатори відрізняються від більш звичних уніфікованих ідентифікаторів тим, що використовують повну множину символів Unicode замість множити символів ASCII. Найбільш відомим підкласом інтернаціоналізованих ідентифікаторів ресурсів є уніфіковані локатори ресурсів (англ. URL – Uniform Resource Locator) – адреси цифрових ресурсів у мережі Інтернет. Далі в тексті під ідентифікатором ресурсу матиметься на увазі саме інтернаціоналізований ідентифікатор ресурсу. Який саме ресурс позначає той чи інший ідентифікатор ресурсу, визначається такими чинниками, як схема (тип) ідентифікатора, визначений відповідальний орган чи організація, власник ресурсу, тощо.

RDF-твердження (RDF-трійка “суб’єкт – предикат – об’єкт”) відображає певний факт або зв’язок. Суб’єкт RDF-трійки представляє ресурс і може бути або ідентифікатором ресурсу або спеціальною анонімною вершиною – об’єктом, що семантично відображає існування ресурсу з певними властивостями і зв’язками з іншими ресурсами, але не називає його. Предикат RDF-трійки представляє конкретну властивість або тип зв’язку ресурсу-суб’єкта і сам по собі є ресурсом, при чому лише таким, що має ідентифікатор ресурсу, а отже семантичне значення властивості або типу зв’язку є чітко визначеним, що гарантує повну визначеність (відсутність неоднозначності) будь-якого RDF-твердження. Насамкінець, об’єкт RDF-трійки представляє значення вказаної властивості або інший ресурс, з яким існує вказаний зв’язок. Якщо об’єкт представляє ресурс, то він може бути як ідентифікатором ресурсу, так і анонімною вершиною. Натомість, для представлення значень властивостей використовуються

спеціальні об'єкти-літерали, що складаються з текстового рядка-представлення та супроводжуючого ідентифікатора ресурсу-типу. Тип літерала визначає, як саме інтерпретувати текстовий рядок-представлення. Наприклад, літерал з рядковим представленням “42” та типом “<http://www.w3.org/2001/XMLSchema#integer>” позначає ціле число 42, але у цьому випадку інтерпретація поняття “ціле число” формально визначена у параграфі 3.4.13 стандарту XML Schema [53].

Множина RDF-тверджень утворює граф, у якому вершинами є суб'єкти та об'єкти RDF-трійок (ресурси та літерали), а ребрами – предикати (ресурси). RDF-граф є орієнтованим, оскільки предикат має напрямок від суб'єкта до об'єкта, міченим, оскільки різні предикати представляють дуги різних типів, і мультиграфом, оскільки між двома вершинами може існувати декілька дуг різних типів або навіть декілька дуг одного і того ж типу.

Використання RDF для представлення великих графів знань і великих графів загалом має декілька важливих переваг:

1. Простота моделі: кожна дуга між двома вершинами у графі представляється окремою RDF-трійкою і відображає окремий факт.
2. Простота побудови: кожна дуга між двома вершинами є повністю незалежною від решти графа і може бути доданою до графа або видаленою з нього у будь-який момент.
3. Гнучкість схеми: додавання нових типів вершин та дуг у граф знань є тривіальним, оскільки кожна RDF-трійка є самодостатньою.
4. Масштабованість: існуюче програмне забезпечення для зберігання RDF-графів може працювати з великими графами, що містять 10 і більше мільярдів дуг (RDF-трійок).
5. Пошук патернів: мова запитів SPARQL [54, 55] є гнучким стандартним інструментом для пошуку патернів у RDF-графі і

має при цьому достатньо простий синтаксис для того, щоб бути частиною інтерфейсу рівня користувача.

6. Стандартизованість: існує декілька форматів експорту та архівування RDF-графів, що використовують текстове представлення даних і є стандартизованими, що гарантує сумісність з всіма інструментами, що існують в RDF-екосистемі. Зокрема, формат N-Triples [56] використовує розгорнутий синтаксис і принцип “один рядок – одне твердження”, тому добре підходить для швидкого паралельного експорту та імпорту даних, а формат Turtle [57] використовує компактний синтаксис, що робить його дуже зручним для інспекції та ручного редагування даних.

Основними недоліками використання RDF для представлення великих графів є наступні особливості:

1. SPARQL-запити можуть бути повільними при пошуку складних патернів у графах, зокрема для патернів, що потребують агрегації інформації про суміжні вершини.
2. RDF має високий ступінь надмірності. Вершина у RDF-графі представляється ідентифікатором ресурсу або літералом, тому для представлення кожного атрибуту вершини необхідна окрема RDF-трійка. Кожна вершина (ідентифікатор ресурсу) у графі дублюється стільки разів, у скількох трійках вона присутня.
3. RDF не має прямої підтримки послідовностей, а деконструкція їх у вигляді RDF-трійок є громіздкою і незручною у використанні.

Альтернативними технологіями для представлення графів є реляційні бази даних та безпосереднє представлення у вигляді об'єктної моделі.

Реляційна база даних, може представляти граф транзакцій лише опосередковано, використовуючи таблиці вершин та таблицю дуг. Реляційні бази даних є найпоширенішою технологією зберігання і індексації даних, тому їхньою основною перевагою є розвинена за декілька

десятиліть використання інфраструктура і накопичені інструменти оптимізації. Втім, недоліки використання реляційних баз даних для представлення графів є критичними:

1. Запити, що шукають складні патерни у графі, стають неефективними з кожним додатковим зв'язком, який додає ще одну операцію об'єднання.
2. Мови запитів у реляційних базах даних погано пристосовані до пошуку у графах, тому запити, що шукають патерни у графах, є громіздкими не тільки з точки зору швидкодії, а й синтаксично.
3. Додавання нових типів зв'язків потребує зміни схеми, тоді як граф знань передбачає можливість додавання будь-якого зв'язку між будь-якими сутностями у будь-який момент.

Безпосереднє представлення у вигляді об'єктної моделі полягає у тому, що кожна вершина графа представляється як об'єкт у пам'яті, що має необхідні атрибути, залежно від предметної області, а також безпосередньо містить посилання на суміжні вершини. Переваги безпосереднього представлення:

1. Об'єктна модель дає змогу представляти будь-які типи вершин без надлишковості: атрибути вершини є компактними полями у об'єктах.
2. Всі вершини та зв'язки у графі знаходяться в пам'яті, а перехід вздовж дуги відбувається за константний час, оскільки посилання на суміжну вершину міститься безпосередньо у даній вершині, тому таке представлення забезпечує найвищу швидкість обчислень.

Недоліки використання безпосереднього представлення:

1. Великі графи складно зберігати в пам'яті, тому необхідно реалізовувати механізм відвантаження частин графа на диск, що потребує значних зусиль.

2. Додавання нових типів зв'язків потребує зміни декларацій структур даних і перезавантаження програмного забезпечення.
3. Відсутній стандарт зберігання об'єктів, сумісний з зовнішніми інструментами. Для інтеграції з зовнішніми інструментами доведеться реалізовувати експорт у один з існуючих стандартів.

З аналізу переваг і недоліків технологій представлення великих графів, розглянутих вище, а також аналізу вимог до програмного забезпечення з підрозділу 1.4, можна зробити висновок, що RDF-представлення найкраще підходить для побудови графа знань про Біткоїн-транзакції. RDF-представлення має значну перевагу над реляційними базами даних при здійсненні пошуку, що потребує багатьох переходів вздовж дуг графа [2] (див. експериментальне порівняння порівняння у підрозділі 2.4). При цьому RDF-представлення є повільнішим за безпосереднє представлення, але на відміну від останнього, існуючі сховища RDF-графів можуть вмістити граф усієї історії транзакцій, автоматично надають механізми відвантаження частин графа на диск та можливості експорту графа у форматах, що сумісні з зовнішніми інструментами. Для випадків, коли пошук за допомогою SPARQL-запитів є недостатньо швидким, деякі сучасні RDF-сховища надають швидкий прямий доступ до графа через прикладний програмний інтерфейс. Відсутність підтримки послідовностей є нерелевантною для графа транзакцій, оскільки у моделі, описаній у підрозділі 2.3, вони не використовуються.

2.3. Спеціалізована RDF-модель транзакційних даних

Для побудови графа транзакцій та виходів необхідно модифікувати об'єктну модель ланцюга блоків Біткоїна, виділивши ті атрибути, які є необхідними для моделювання фінансових взаємодій між користувачами. Деякі атрибути класів існуючої моделі є зайвими у графі транзакцій, а деякі

атрибути, що є важливими для графа транзакцій, відсутні в існуючій об'єктній моделі.

Доповнення графа транзакцій та виходів довільними анотаціями, зокрема зв'язками між виходами та їх власниками передбачає існування у графі вершин інших типів, окрім вершин-транзакцій та вершин-виходів. Граф транзакцій та виходів, доповнений такими вершинами та відповідними зв'язками, називатимемо розширеним графом транзакцій. Для обговорення аналізу графа транзакцій (див. розділ 3), зокрема групування виходів за власністю (див. підрозділ 3.2), необхідно ввести у граф вершини, що відповідають власникам виходів. Ще одним корисним типом вершин у розширеному графі транзакцій є вершини-блоки, які надають розбиття множини транзакцій на логічні групи.

Формально, розширений граф Біткоїн-транзакцій – це типізований орієнтований граф

$$G = (N, L),$$

де N – множина вершин;

L – множина дуг.

Множина вершин розширеного графа транзакцій N є об'єднанням попарно неперетинних множини вершин різних типів:

$$N = N_T \cup N_O \cup N_B \cup N_E \cup N_A,$$

де N_T – множина вершин, що відображають транзакції;

N_O – множина вершин, що відображають транзакційні виходи;

N_B – множина вершин, що відображають блоки (групи транзакцій за показником часу);

N_E – множина вершин, що відображають власників транзакційних виходів;

N_A – множина вершин, що відображають анотації довільного змісту і можуть бути суміжними з вершинами будь-якого типу.

Множина дуг розширеного графа транзакцій L є об'єднанням попарно неперетинних множин дуг різних типів:

$$L = L_{TO} \cup L_{OT} \cup L_{BT} \cup L_{OE} \cup L_A,$$

де $L_{TO} \subset N_T \times N_O$ – множина дуг, що відображають зв'язки між транзакціями (N_T) та виходами (N_O), які вони створили;

$L_{OT} \subset N_O \times N_T$ – множина дуг, що відображають зв'язки між використаними транзакційними виходами (N_O) та транзакціями (N_T), які їх використали;

$L_{BT} \subset N_B \times N_T$ – множина дуг, що відображають належність транзакцій (N_T) до блоків (N_B);

$L_{OE} \subset N_O \times N_E$ – множина дуг, що відображають зв'язки між виходами (N_O) та їхніми власниками (N_E);

$L_A \subset N \times N$ – множина дуг, що відображають зв'язки довільного змісту між будь-якими вершинами (N).

У пунктах нижче наведено детальний опис класів та властивостей розробленої об'єктної моделі у RDF-поданні з використанням стандартного синтаксису Turtle. Аббревіатура простору імен “brdf” відповідає префіксу простору імен ресурсів графа транзакцій, який під час проведення цього дослідження було визначено як “<http://github.com/rodrabies/brdf#>”, а скорочена нотація ідентифікатора ресурсу “brdf:Transaction” означає ресурс “Transaction” у цьому просторі імен, повний ідентифікатор якого – “<http://github.com/rodrabies/brdf#Transaction>”.

У таблиці 2.1 наведено відповідність між множинами вершин та дуг розширеного графа транзакцій, наведеними вище, та класами і властивостями розробленої RDF-моделі, наведеними нижче. Множинам вершин відповідають RDF-класи, а множинам дуг – RDF-властивості. RDF-властивості, значеннями яких є не вершини графа транзакцій, а RDF-літерали, такі як цілі числа (“xsd:integer”) та рядки (“xsd:string”),

фактично є атрибутами вершин, і тому відсутні у таблиці, але детально обговорюються у пунктах нижче.

RDF-модель даних Біткоїн-транзакцій, подана у цьому розділі, є удосконаленою і розширеною версією моделі, описаної у статті [2]. Повний формальний RDFS-опис розробленої RDF-моделі у Turtle-синтаксисі розміщено у додатку А.

Таблиця 2.1

Відповідність між множинами вершин і дуг розширеного графа транзакцій та класами і властивостями RDF-моделі

Категорія	Множина	RDF-категорія	RDF-сутність
Вершини	N_T	Клас	brdf:Transaction
Вершини	N_O	Клас	brdf:Output
Вершини	N_O	Клас	brdf:Block
Вершини	N_E	Клас	brdf:Owner
Вершини	N_A	Клас	довільний
Дуги	L_{TO}	Властивість	brdf:transactionOutput
Дуги	L_{OT}	Властивість	brdf:outputInputTransaction
Дуги	L_{BT}	Властивість	brdf:blockTransaction
Дуги	L_{OE}	Властивість	brdf:outputOwner
Дуги	L_A	Властивість	довільна

2.3.1. Вихід

На рисунку 2.7 зображено UML-діаграму класу “Output”, що моделює транзакційний вихід.

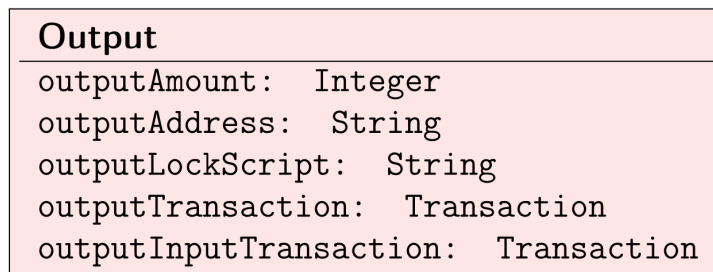


Рис. 2.7. UML-діаграма класу Output

Клас “Вихід” (“Output”) в об’єктній моделі Біткоїна має два атрибути – “кількість” (“amount”) та “скрипт блокування” (“lockScript”). Обидва ці атрибути є важливими для відображення фінансових зв’язків між користувачами: кількість біткоїна вказує на вагу фінансової взаємодії, і використовується у евристичних методах розрізнення цільових виходів та виходів решти, розпізнавання транзакцій змішування виходів (CoinJoin), тощо, а скрипт блокування є єдиним ідентифікатором власника виходу, що існує на рівні протоколу. Переважна більшість виходів у Біткоїн-транзакціях є стандартними, а отже мають чітко визначений формат адреси. Адреса, за визначенням, є спеціальним ідентифікатором, що стисло і однозначно описує відповідний скрипт блокування у легкій для людського сприйняття формі. Тому з метою зменшення кількості даних, що містяться у графі, є сенс для стандартних замінити атрибут “скрипт блокування” (“lockScript”) атрибутом “адреса” (“address”), який містить лише відповідну адресу. Нестандартні виходи все ще матимуть атрибут “скрипт блокування” замість атрибута “адреса”, що дасть змогу швидко фільтрувати виходи за ознакою стандартності, використовуючи наявність атрибута “адреса” в якості маркера. При цьому використання двох взаємовиключних атрибутів для кожного виходу є еквівалентним використанню єдиного атрибута з

точки зору розміру графа, оскільки кожен атрибут кожної вершини у RDF-графі представляється одним RDF-твердженням.

Транзакційний вхід є допоміжною структурою, яка виконує дві функції: зв'язування виходів з транзакціями, що їх споживають, та розміщення скрипта розблокування. З впровадженням модифікації протоколу під назвою SegWit (англ. Segregated Witness – Відокремлений Свідок) у 2017-му році, новий тип виходів, що стає все більш поширеним у мережі, містить порожній скрипт розблокування, а сам скрипт розміщується у окремій структурі під назвою Свідок (англ. Witness). Тому для великої частини сучасних транзакцій основна функція виходів – лише зв'язування виходів з транзакціями. У моделі даних транзакцій та виходів входи є зайвими, а зв'язок між виходом та транзакцією, що його споживає, моделюється прямим зв'язком “вихід – вхід – транзакція” (“outputInputTransaction”) між вершиною-виходом та вершиною-транзакцією. На даний момент нам не відомі дослідження, що використовують вміст скриптів розблокування у аналізі графа транзакцій, оскільки для більшості стандартних виходів вся ідентифікуюча інформація міститься у скрипті блокування. Втім, у деяких випадках публічні ключі, які містяться саме в скриптах розблокування, можуть бути більш змістовними ідентифікаторами, тому за необхідності ці скрипти або виділені з них публічні ключі можуть бути додані як додаткові атрибути вершин-виходів. Завдяки гнучкості схеми RDF-представлення графа додавання довільних додаткових атрибутів є простою операцією, яка не потребує жодних змін у решті графа, а завдяки уніфікованій семантиці RDF-трійок такі атрибути автоматично стають доступними безпосередньо у SPARQL-запитах при здійсненні пошуку у графі (див. підрозділ 2.4).

Посилання “вихід – транзакція” (“outputTransaction”) на транзакцію, яка створила даний вихід, є зворотнім посиланням на об'єкт класу “Transaction”, який в свою чергу має посилання “транзакція – вихід” (“transactionOutput”) на даний об'єкт класу “Output”. У RDF-представленні

графа ці два посилання моделюються єдиною трійкою з предикатом “brdf:transactionOutput” (див. пункт 2.3.2).

В якості ідентифікатора виходу можна використовувати поширений формат ідентифікатора виходу, який складається з ідентифікатора транзакції, у якій він був створений, та його порядкового номеру у цій транзакції, наприклад

```
brdf:a1075db5...fbf5d48d:0
```

але такий формат ідентифікатора не дає додаткових переваг при пошуку у графі, і тому з метою зменшення розміру таблиці рядків сховища RDF-трійок можна замість такого формату ідентифікатора використовувати анонімні вершини.

Скорочений онтологічний опис класу “Output” за допомогою RDFS подано у лістингу 2.1.

Лістинг 2.1. RDFS-опис класу “Output”

```
brdf:Output rdf:type rdfs:Class.

brdf:outputAmount rdf:type    rdfs:Property;
                    rdfs:domain brdf:Output;
                    rdfs:range  xsd:integer.

brdf:outputAddress rdf:type    rdfs:Property;
                    rdfs:domain brdf:Output;
                    rdfs:range  xsd:string.

brdf:outputLockScript rdf:type  rdfs:Property;
                    rdfs:domain brdf:Output;
                    rdfs:range  xsd:string.

brdf:outputInputTransaction rdf:type    rdfs:Property;
                            rdfs:domain brdf:Output;
                            rdfs:range  brdf:Transaction.
```

2.3.2. Транзакція

На рисунку 2.8 зображено UML-діаграму класу “Transaction”, що моделює транзакцію як вершину графа, що об'єднує групу транзакційних виходів на вході (знищених виходів) та групу транзакційних виходів на виході (створених виходів) у єдину операцію зміни власності над певною кількістю біткоїна.

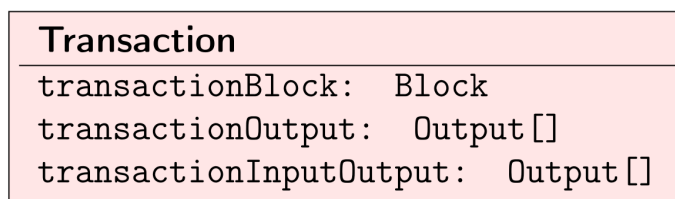


Рис. 2.8. UML-діаграма класу Transaction

Клас “Транзакція” (“Transaction”) має багато атрибутів, що містять різноманітні метадані, але у розроблюваній моделі даних вершина-транзакція виконує функцію зв'язування груп виходів, що споживаються і створюються нею. Тому єдиними важливими атрибутами транзакції є атрибути, що представляють зв'язки “вихід – вхід – транзакція” (“outputInputTransaction”) та “транзакція – вихід” (“transactionOutput”). Слід звернути увагу на те, що ці зв'язки є взаємно оберненими, і реалізовані як атрибути класу Output і класу Transaction відповідно, оскільки для швидкого пошуку у графі необхідна можливість швидко переходити вздовж відповідних ребер у будь-якому напрямку. Вершини-виходи містять посилання на відповідну вершину-транзакцію, а вершини-транзакції – на відповідний вихід. Але RDF-представлення, як і інші безпосередні представлення графа, дає змогу використовувати єдину RDF-трійку для зберігання пар атрибутів. Одна й та ж RDF-трійка представляє як зв'язок між транзакцією і виходом, так і зворотній зв'язок між виходом та транзакцією і дає змогу здійснювати швидкий перехід у будь-якому напрямку при пошуку в графі.

Евристики, що використовуються для розрізнення типів операцій, часто враховують кількість входів чи виходів у певній транзакції. Зокрема, звичайний платіж зазвичай має 2 – 3 входи і два виходи, тоді як CoinJoin-операція має багато входів і таку ж кількість виходів з однаковою кількістю біткоїна в кожному. Для кожної вершини-транзакції у графі кількість входів та кількість виходів можна легко обчислити, порахувавши кількість вхідних зв'язків “outputInputTransaction” та вихідних зв'язків “transactionOutput”, але при пошуку патернів у графі, таких обчислень доведеться робити дуже багато. Оскільки кількість входів та виходів транзакції відома під час побудови графа транзакцій і є незмінною для підтвердженої транзакції, є сенс ввести додаткові атрибути “кількість входів” (“transactionInputCount”) та “кількість виходів” (“transactionOutputCount”), які зберігатимуть цю інформацію у графі і дадуть змогу уникнути її обчислення під час пошуку патернів, оскільки такі обчислення ускладнюють SPARQL-запити і збільшують час їхнього виконання.

Посилання “транзакція – блок” (“transactionBlock”) на блок, у якому знаходиться дана транзакція, є зворотнім посиланням на об'єкт класу “Block”, який в свою чергу має посилання “блок – транзакція” (“blockTransaction”) на даний об'єкт класу “Transaction”, і обидва цих посилання у RDF-представленні графа моделюються єдиною трійкою з предикатом “brdf:blockTransaction” (див. пункт 2.3.3). Аналогічно, пари посилань “транзакція – вхід – вихід” (“transactionInputOutput”) у об'єктах класу “Transaction” та “вихід – вхід – транзакція” (“outputInputTransaction”) у об'єктах класу “Output” моделюються єдиною трійкою з предикатом “brdf:outputInputTransaction” (див. пункт 2.3.1).

В якості ідентифікатора RDF-ресурсу транзакції використовується стандартний ідентифікатор транзакції Біткоїн-протоколу (подвійний хеш бінарного представлення транзакції у шістнадцятковому кодуванні), але як і у випадку виходу, такий формат ідентифікатора не дає додаткових переваг

при пошуку у графі, і тому з метою зменшення розміру таблиці рядків сховища RDF-трійок можна використовувати анонімні вершини.

Скорочений онтологічний опис класу “Transaction” за допомогою RDFS подано у лістингу 2.2.

Лістинг 2.2. RDFS-опис класу “Transaction”

```
brdf:Transaction rdf:type rdfs:Class.

brdf:transactionInputCount rdf:type      rdfs:Property;
                           rdfs:domain   brdf:Transaction;
                           rdfs:range    xsd:integer.

brdf:transactionOutputCount rdf:type      rdfs:Property;
                           rdfs:domain   brdf:Transaction;
                           rdfs:range    xsd:integer.

brdf:transactionOutput rdf:type      rdfs:Property;
                       rdfs:domain   brdf:Transaction;
                       rdfs:range    brdf:Output.
```

2.3.3. Блок

На рисунку 2.9 зображено UML-діаграму класу “Block”, що групує транзакції у сторінки реєстру.

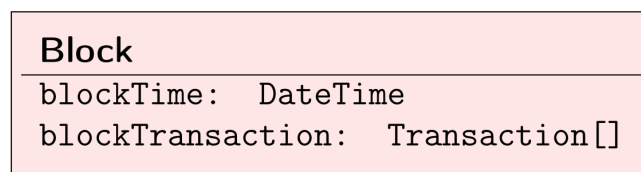


Рис. 2.9. UML-діаграма класу Block

Клас “Блок” (“Block”) відображає групування транзакцій у сторінки реєстру, для яких обчислюється доказ виконаної роботи. На момент проведення цього дослідження не відомі праці, що використовують

належність транзакції певному блоку в аналізі графа транзакцій, але завдяки тому, що кількість блоків у ланцюгу є на 3 порядки меншою, ніж кількість транзакцій (новий блок в середньому з'являється раз на 600 секунд і є обмеженим в розмірі), вершини-блоки є зручними об'єктами для зберігання деяких важливих метаданих, що є спільними для всіх транзакцій в них.

Основним атрибутом блока, що стосується всіх транзакцій і може бути корисним в аналізі графа транзакцій є атрибут “час створення” (“blockTime”), що містить інформацію про заявлений час створення блока. Кожна транзакція містить власну мітку часу створення, але правила Біткоїн-консенсусу ніяк не обмежують мітку часу створення транзакції, накладаючи при цьому певні обмеження на мітку часу створення блока. Тому мітка часу створення блока є більш надійною, ніж мітка часу створення транзакції. Використання вершин-блоків, що мають атрибут “мітка часу”, дає змогу застосовувати під час пошуку у графі фільтри, що обмежують транзакції, що розглядаються, до певного часового проміжку, використовуючи при цьому мінімальну можливу кількість RDF-трійок.

Групування транзакцій у блоки відбувається за допомогою атрибуту “блок – транзакція” (“blockTransaction”), що представляє зв'язок між вершиною-блоком та вершиною-транзакцією.

Як було зазначено вище, посилання “блок – транзакція” (“blockTransaction”) на транзакцію, що належать даному блоку, є оберненим до посилання “транзакція – блок” (“transactionBlock”) у об'єктах класу “Transaction”, і у RDF-представленні графа моделюється єдиною трійкою з предикатом “brdf:blockTransaction”.

В якості ідентифікатора RDF-ресурсу блока використовується висота блока, а не хеш блока, як у Біткоїн-протоколі, оскільки жоден з цих форматів ідентифікаторів не надає додаткових переваг при пошуку у графі, порівняно з іншим, а отже основним критерієм вибору є зручність людського сприйняття. Окрім цього, висота блока є значно коротшою

послідовністю символів, ніж шістнадцяткове кодування його хеша, тому такий формат ідентифікаторів сприяє зменшенню розміру таблиці рядків сховища RDF-трійок. Обробка реорганізацій ланцюга блоків розглядається в підрозділі 4.3.

Скорочений онтологічний опис класу “Block” за допомогою RDFS подано у лістингу 2.3.

Лістинг 2.3. RDFS-опис класу “Block”

```
brdf:Block rdf:type rdfs:Class.

brdf:blockTime rdf:type      rdfs:Property;
                rdfs:domain  brdf:Block;
                rdfs:range   xsd:dateTime.

brdf:blockTransaction rdf:type      rdfs:Property;
                      rdfs:domain  brdf:Block;
                      rdfs:range   brdf:Transaction.
```

2.3.4. Власник та інші анотації

Завдяки гнучкості RDF-моделі граф знань про транзакції може розширюватись додатковими вершинами та зв'язками, які не є частиною об'єктної моделі ланцюга блоків Біткоїна. Надалі називатимемо такі вершини та зв'язки анотаціями, оскільки вони анотують граф транзакцій зовнішніми метаданими. Анотований граф транзакцій та виходів і утворює граф знань про транзакції.

Однією з основних задач аналізу графа транзакцій є задача деанонімізації користувачів – встановлення ідентичності власників транзакційних виходів. Найбільш важливими анотаціями в контексті цієї задачі є анотації власності: зв'язки між вершинами-виходами та вершинами-власниками. Клас “Власник” (“Owner”) є абстрактним, адже в загальному випадку його атрибути можуть бути довільними (ім'я,

псевдонім в соціальній мережі, посилання на анонімну публікацію, тощо). Зв'язок “вихід – власник” моделюється трійкою з предикатом “brdf:outputOwner” та анонімною вершиною в якості об'єкта, що найкраще відображає зміст такої анотації: єдине, що відомо про ресурс-власника в загальному випадку – це лише той факт, що він існує. Якщо у процесі пошуку зовнішньої інформації про користувачів Біткоїна вдається встановити, що користувач певної соціальної мережі під певним псевдонімом публікував певну Біткоїн-адресу, то анонімна вершина, що відображає власника виходів, асоційованих з цією адресою, може бути або замінена на відповідний ідентифікатор ресурсу, або доповнена довільною анотацією, що відображає цю інформацію, наприклад RDF-трійкою з предикатом “owl:sameAs” стандартної мови опису онтологій OWL [58, 59] і об'єктом, що є посиланням (універсальним локатором ресурсу) на профіль користувача соціальної мережі.

Патерновий аналіз графа транзакцій, що розглядається у розділі 3, полягає у пропагації зв'язків “вихід – власник” з одних виходів на інші згідно з евристичною інтерпретацією того чи іншого патерну у графі транзакцій та виходів. Анотації власності, отримані з зовнішніх джерел, можна розглядати як достовірну інформацію, що має зовнішній документальний доказ, тоді як анотації власності, встановлені за допомогою евристичного аналізу, є правдивими лише з певною ймовірністю. Для того, щоб розрізняти достовірні і ймовірні анотації власності, можна використовувати властивості, відмінні від основного зв'язку власності (“brdf:outputOwner”). Сучасні сховища RDF-трійок дають змогу вводити довільні нові зв'язки у граф з мінімальними витратами обчислювальних ресурсів, тому окремий зв'язок власності може бути впроваджений для кожної окремої евристики, що застосовується під час аналізу, наприклад “brdf:outputOwner_consolidation” для операцій консолідації (див. пункт 3.3.3). Така гранулярність дає змогу здійснювати тестування експериментальних аналізів у єдиній копії графа знань. Нові

RDF-трійки зв'язків власності, що використовують новий відокремлений предикат, не можуть впливати на результати інших аналізів, оскільки множина нових RDF-трійок зв'язків власності не перетинається з множинами існуючих RDF-трійок зв'язків власності.

Інші анотації можуть використовуватись як маркери здійснених аналізів для подальшого використання у майбутніх аналізах. Наприклад, операція CoinJoin (див. пункт 3.3.4) може руйнувати деякі евристичні припущення, і при пошуку певних транзакцій може бути потрібно відсіяти транзакцію, якщо вона схожа на операцію CoinJoin. Таке відсіювання можливе за допомогою фільтрування за відсутністю підграфа у SPARQL-запиті (“FILTER NOT EXISTS”), але таке фільтрування ускладнює запит. Тому попередньо можна здійснити пошук всіх операцій CoinJoin, додати у граф анотації типу транзакції

```
brdf:a1075db5...fbf5d48d brdf:transactionType brdf:CoinJoin.
```

і у подальших аналізах швидко відсіювати CoinJoin-операції за наявністю такого маркера.

2.3.5. Об'єднана спеціалізована RDF-модель транзакційних даних

На рисунку 2.10 зображено ER/UML-діаграму класів моделі даних транзакцій з урахуванням того, що зв'язки між сутностями представляються не полями відповідних об'єктів, а дугами у RDF-графі. Класи наведено з позначенням простору імен, обраного для розробленої онтології. Типи значень атрибутів – це стандартизовані типи XML Schema [53], які використовуються в RDF-графах. Клас “brdf:Owner” – абстрактний, оскільки модель дає змогу представляти його довільним чином. У початковому стані розширеного графа транзакцій у кожного виходу є унікальний абстрактний власник, але такий власник не відображається у вигляді RDF-трійки (див. підрозділ 3.2), оскільки матеріалізація такого зв'язку у RDF-графі збільшить кількість трійок, необхідних для

представлення одного виходу на 1, що сильно вплине на розмір RDF-графа, так як вершини-виходи становлять більшість вершин розширеного графа транзакцій (див. пункт 4.2.2).

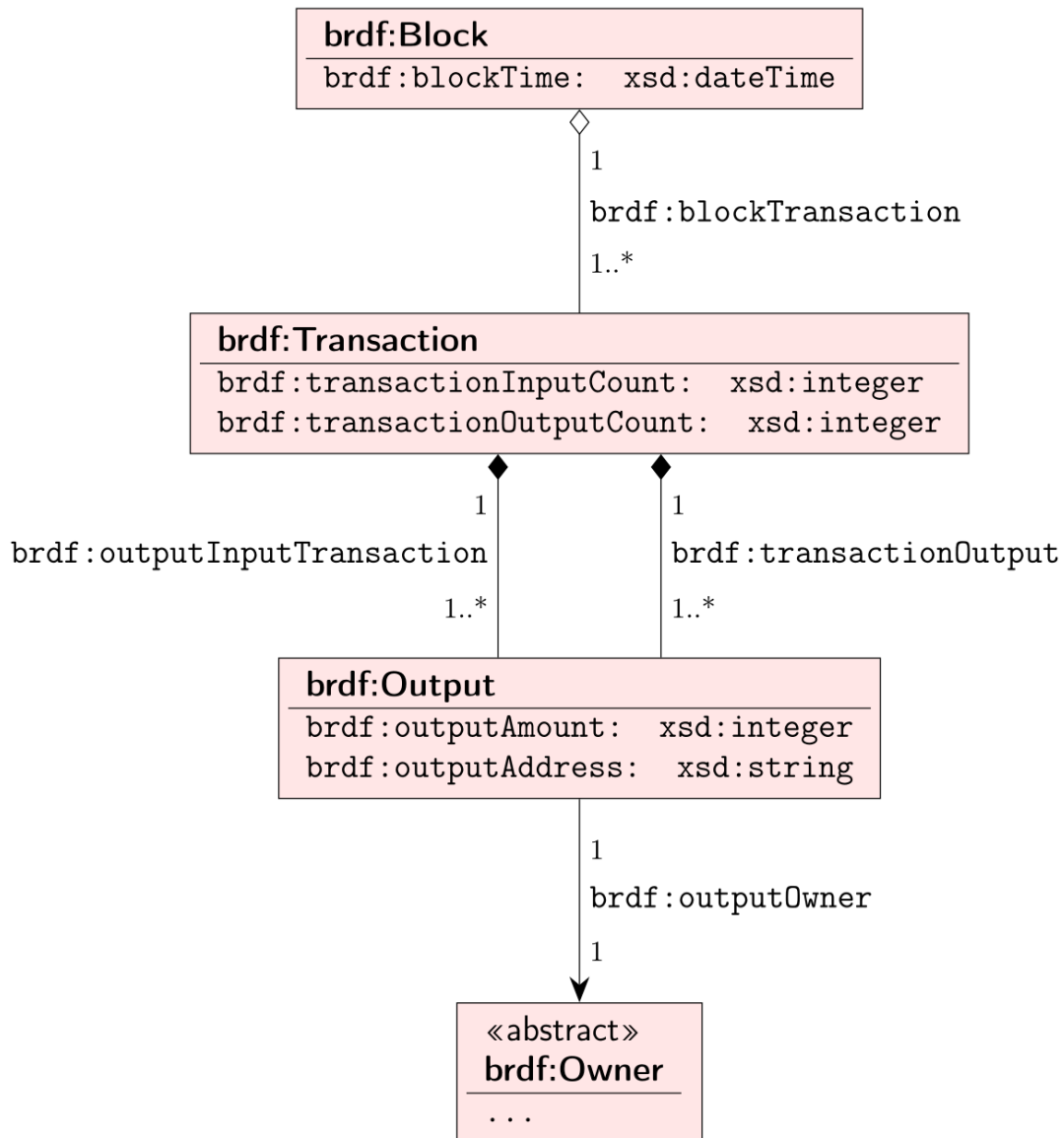


Рис. 2.10. ER/UML-діаграма класів розробленої моделі даних транзакцій

На рисунку 2.11 зображено RDF-модель окремої транзакції, її входів, виходів, анотацій власності для деяких з цих входів та виходів, та блоку, якому ця транзакція належить. Варто зазначити, що значення атрибутів, такі як кількість входів, виходів, час створення блоку, тощо (сірі вершини прямокутної форми на рисунку) теж є вершинами RDF-графа, але

відображають не сутності (ресурси), а значення їхніх властивостей (літерали).

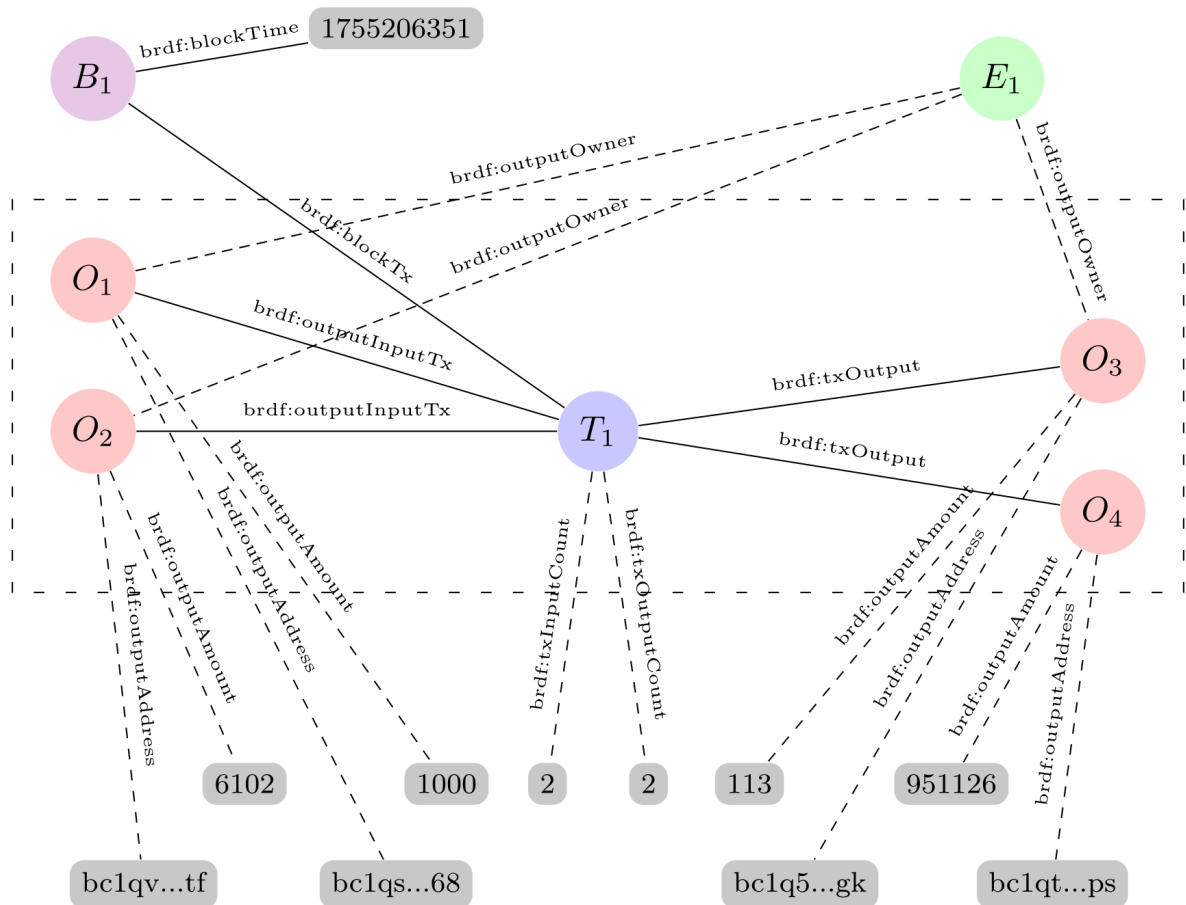


Рис. 2.11. RDF-модель транзакції з 2-ма входами та 2-ма виходами

Транзакція T_1 має два входи (O_1 та O_2) та два виходи (O_3 та O_4). Входи зв'язані з транзакцією дугами типу “`brdf:outputInputTx`”, а транзакція зв'язана з виходами дугами типу “`brdf:txOutput`”. Властивості кількості входів та кількості виходів транзакції представлені вершинами-значеннями, з якими транзакція зв'язана дугами типів “`brdf:txInputCount`” та “`brdf:txOutputCount`” відповідно. Кожен з виходів має дві властивості кількості біткоїна і адреси, з якими вихід зв'язаний дугами типів “`brdf:outputAmount`” та “`brdf:outputAddress`” відповідно. Блок B_1 , в який входить транзакція T_1 , зв'язаний з нею дугою типу “`brdf:blockTx`”, і властивість мітки часу створення блока представлена

вершиною-значенням, з яким блок зв'язаний дугою типу “brdf:blockTimestamp”. Обидва входи транзакції та один з її виходів зв'язані з анонімним власником E_1 дугами типу “brdf:outputOwner”.

2.4. Використання SPARQL для дослідження графа транзакцій

SPARQL – це стандартизована мова семантичних запитів у сховищах даних, поданих у вигляді множини RDF-графів [55]. Стандарт SPARQL описує протокол передачі запитів до сервісів-сховищ RDF-графів та зворотної передачі відповідей на запити. Основним елементом синтаксису SPARQL є трійковий патерн – трійка “суб'єкт – предикат – об'єкт”, яка за формою відповідає RDF-трійці, але кожен з її компонентів може бути як константним значенням, так і символом, що позначає змінне значення, і такий патерн семантично позначає множину всіх RDF-трійок у певному RDF-графі, таких, що константні значення співпадають, а змінні можуть бути довільними. Відповідно, група трійкових патернів семантично позначає множину всіх підграфів у RDF-графі, для яких константні значення співпадають, а змінні можуть бути довільними. Наприклад, перший SPARQL-запит з лістингу 2.4 знаходить всі входи вказаної транзакції, а другий – її виходи та кількості біткоїна у них.

Лістинг 2.4. SPARQL-запити для дослідження графа транзакцій

```
SELECT ?output {  
  ?output brdf:inputOutputTx brdf:a1075db5...fbf5d48d.  
}  
  
SELECT ?output ?amount ?address {  
  ?output brdf:outputInputTx brdf:a1075db5...fbf5d48d.  
  ?output brdf:outputAmount ?amount.  
  ?output brdf:outputAddress ?address.  
}
```

Важливим інструментом у мові запитів SPARQL є можливість швидкого переходу вздовж зв'язків на певну відстань у графі. Наприклад, запит у лістингу 2.5 знаходить множину всіх виходів в межах 3-х транзакцій від даного виходу, а також власників цих виходів, якщо вони присутні у графі. Загалом, запити пошуку всіх вершин у графі, що знаходяться на відстані N від певної вершини, є важливими при дослідженні складних патернів у графі. Подібний запит може дати відповідь на запитання, чи є хоча б один з входів даної транзакції результатом тієї чи іншої типової операції, що може пов'язати дану транзакцію з такою операцією за змістом.

Лістинг 2.5. SPARQL-запит пошуку виходів в межах 3-х транзакцій

```
SELECT ?output ?owner {  
  ?output (brdf:outputInputTx/brdf:txOutput){3} OUTPUT.  
  OPTIONAL {  
    ?output brdf:outputOwner ?owner.  
  }  
}
```

У праці [2] запропоновано ранню версію RDF-моделі, що розробляється у цьому розділі. У ній також наводиться порівняння швидкості виконання подібних запитів для графа, що містить 1.7 мільйона транзакцій, у RDF-сховищі AllegroGraph та у реляційній системі керування базами даних PostgreSQL, що містить еквівалентні дані у реляційному представленні. Це порівняння підтверджує перевагу RDF-моделі даних транзакцій над реляційною моделлю при пошуках, що потребують більше двох переходів вздовж дуг у графі. Результати цього порівняння, взяті з праці [2], наведені у таблиці 2.2. Велика різниця часу запиту для більших значень відстані N пояснюється тим, що запити у реляційній базі даних обчислюють вибірку над декартовим добутком проміжних результатів для кожного кроку вглиб графа, тоді як безпосереднє представлення графа у

графовій базі даних не потребує побудови повної множини проміжних результатів.

Таблиця 2.2

Час виконання запиту пошуку всіх виходів на відстані N від даного виходу

Відстань N	Час SPARQL-запиту, с	Час SQL-запиту, с
1	0.017	0.164
2	0.052	0.343
4	0.066	2.829
8	0.070	15.239

Фільтрування знайдених підграфів за певними властивостями здійснюється за допомогою конструкції фільтрування, і деякі сховища RDF-графів дають змогу створювати спеціальні фільтр-функції, що мають безпосередній доступ до графа, і можуть ефективно обчислювати певні перевірки, які може бути неможливо або неефективно реалізувати як частину SPARQL-запиту. Наприклад, запит у лістингу 2.6 використовує спеціальну фільтр-функцію “brdf:txInputTotal”, яка обчислює сумарну кількість біткоїна на вході транзакції, щоб знайти всі транзакції, що переводять 1 і більше BTC (10^8 базових одиниць біткоїна).

Літсинг 2.6. Приклад використання фільтр-функцій у SPARQL

```
SELECT ?output ?owner {  
  ?tx brdf:txOutputCount ?ocount.  
  FILTER (brdf:txInputTotal(?tx) > 1000000000) # 100,000,000  
}
```

Для реалізації такого пошуку в SPARQL-запиті необхідно використовувати конструкції агрегації результатів, що може бути неефективним для великих кількостей результатів, тому у таких випадках

зручно використовувати спеціальні фільтр-функції, які мають прямий швидкий доступ до графа, тобто можливість отримати множину вершин, до яких існує дуга певного типу з даної вершини, безпосередньо з індексу сховища RDF-трійок за константний час. Детальніше про підтримку спеціальних фільтр-функцій див. обговорення сховища AllegroGraph у пункті 4.2.3. Деталі реалізації фільтр-функції “`brdf:txInputTotal`” та інших подібних фільтр-функцій розміщені у додатку Б.

Частиною стандарту SPARQL є протокол операцій модифікації графа SPARQL Update [60]. Операції модифікації дають змогу знаходити патерни у графі, і на основі цих патернів додавати до графа нові підграфи або видаляти існуючі. За допомогою SPARQL Update можна, зокрема, видалити всі декларації типів (див. першу операцію у лістингу 2.6), що є одним з кроків застосування методу зменшення розміру RDF-графа транзакцій, описаного у пункті 4.2.2, або видалити результати невдалої пропagaції зв'язків власності (див. другу операцію у лістингу 2.6), як це описано у пункті 2.3.4.

Лістинг 2.6. Операції SPARQL Update для модифікації графа

```
#1
DELETE { ?s rdf:type ?p }
WHERE { ?s rdf:type ?p };

#2
DELETE { ?s rdf:outputOwner_consolidation_1 ?p }
WHERE { ?s rdf:outputOwner_consolidation_1 ?p };
```

Кроки алгоритму пропagaції зв'язків власності типових операцій з підрозділу 3.2, які видаляють частини графа або додають нові зв'язки, можуть бути реалізовані як операції SPARQL Update. Для деяких типових операцій з підрозділу 3.3 весь алгоритм може бути реалізований як єдина операція SPARQL Update, але в загальному випадку така реалізація є складною, оскільки потребує попереднього розроблення великого набору

спеціальних фільтр-функцій. У розділі 3 разом з аналізом типових операцій наводяться SPARQL-запити для розпізнавання цих операцій у графі знань про транзакції.

2.5. Висновки

У цьому розділі проведено комплексний аналіз можливостей побудови та використання графа знань для представлення і дослідження графа Біткоїн-транзакцій. Показано, що ні реляційна модель, ні безпосередня об'єктна модель не є непридатними для дослідження складних багатокрокових операцій у графі транзакцій, оскільки жодна з них сама по собі не забезпечує в повній мірі гнучкість представлення зв'язків та властивостей вершин та ефективність виконання запитів пошуку патернів, що потребують більше ніж одного-двох переходів вздовж дуг. Розглянуто переваги використання RDF-моделі, а саме можливість побудови графа усієї історії транзакції з довільними додатковими зв'язками та анотаціями, зокрема зв'язками власності та зовнішніми ідентифікаторами вершин-власників.

Окремо розглянуто використання стандартизованої мови запитів SPARQL для ефективного пошуку патернів у RDF-графах. У випадках, коли SPARQL є недостатньо експресивною для опису певного патерну, наприклад при необхідності агрегації властивостей вершин, або швидкість виконання запиту є недостатньо високою, запропоновано використовувати прямий доступ до графа, оформлений у вигляді спеціальних фільтр-функцій, які можуть використовуватись у SPARQL-запитах. Окрім цього, розглянуто можливість модифікації графа за допомогою операцій SPARQL Update, наприклад для встановлення простих зв'язків власності для деяких простих типових операцій, або для повного швидкого видалення зв'язків, встановлених якимось хибним аналізом, тощо.

Розроблена RDF-модель даних Біткоїн-транзакцій є універсальним інструментом накопичення інформації про транзакції та користувачів, що

поєднує у собі можливості зберігання графа усієї історії транзакцій і анотацій, подальшого його розширення довільними анотаціями і ефективного пошуку патернів у ньому. Саме ця модель даних є ключовим компонентом і фундаментальною відмінністю програмної системи аналізу графа транзакцій, яка може виконувати автоматизоване накопичення інформації про транзакції та користувачів як з зовнішніх джерел, так і з результатів внутрішніх аналізів.

3. РОЗРОБЛЕННЯ МЕТОДІВ ПАТЕРНОВОГО АНАЛІЗУ ГРАФА БІТКОЇН-ТРАНЗАКЦІЙ

3.1. Евристичний пошук типових операцій у графі транзакцій

Як це розкрито у розділі 1, основною структурою в Біткоїн-системі Біткоїн є індекс невикористаних транзакційних виходів. Цей індекс відображає весь біткоїн у обігу в даний момент часу, а також асоціює кожну базову одиницю біткоїна з актуальним власником. Біткоїн-транзакція є транзакцією не з фінансової точки зору, а з точки зору моделі даних: основна функція транзакції – це модифікація індексу невикористаних виходів шляхом знищення (використання) існуючих невикористаних виходів і створення нових виходів. Змістом такої модифікації є перехід власності над певною кількістю біткоїна від одної групи власників до іншої. Оскільки аналіз Біткоїн-транзакцій розглядає їх саме з фінансової точки зору, пропонуємо ввести поняття Біткоїн-операції, яке відокремлює фінансовий зміст групи транзакцій від їхнього значення з точки зору моделі даних.

Біткоїн-операція – це група Біткоїн-транзакцій, що складається з однієї або декількох Біткоїн-транзакцій, і об'єднана спільною фінансовою ціллю або змістом. Біткоїн-транзакція – це подія знищення підмножини невикористаних виходів і створення множини нових виходів. Біткоїн-операція – це група таких подій, що має єдиний фінансовий зміст. Більшість Біткоїн-операцій складаються з однієї транзакції, але узагальнення поняття саме до групи транзакцій необхідне тому, що деякі з операцій (див. підрозділ 3.3) складаються з двох і більше транзакцій, і саме такі операції часто є найбільш цікавими при здійсненні аналізу графа транзакцій.

Оскільки кожна транзакція назавжди зберігається у відкритому реєстрі транзакцій і кожен вузол Біткоїн-мережі має власну копію цього реєстру, кожна транзакція платить комісію за своє власне місце у цьому

реєстрі, тобто місце на дисках усіх вузлів мережі, і ця комісія залежить від її розміру на диску. Припускаючи, що переважна більшість користувачів Біткоїна є раціональними, можемо очікувати, що більшість користувачів зацікавлені у мінімізації комісій за свої транзакції, а отже і у мінімізації їхнього розміру. Оскільки більшість транзакцій використовують стандартні програми блокування, їхня складність, а отже й розмір, є більш-менш сталими, і тому основними чинниками, що впливають на розмір транзакції, є кількість входів та кількість виходів. З припущення про раціональність більшості користувачів випливає, що кількість входів та виходів в більшості випадків є не випадковою, а функціональною, тобто такою, що має фінансовий зміст: транзакції, з яких складається певна операція, мають мінімальну кількість входів, необхідну, щоб покрити суму кількостей у виходах, і саме стільки виходів, скільки платежів здійснюється даною операцією. З того ж таки припущення про раціональність користувачів випливає, що якщо кількості входів та виходів відхиляються від принципу мінімізації комісії за транзакції, то така операція має додатковий або зовсім інший фінансовий зміст, ніж простий переказ коштів. Отже, кількості входів та виходів транзакцій, так само як і програми блокування та кількості біткоїна у них, містять інформацію про фінансовий зміст відповідних операцій.

Основною ідеєю патернового аналізу графа Біткоїн-транзакцій є те, що кількості входів та виходів транзакцій можна використовувати для того, щоб розрізняти операції, до яких ці транзакції належать, за фінансовим змістом, і розпізнавати типові операції, тобто операції, що мають чітко визначений фінансовий зміст і є поширеним способом реалізації цього фінансового змісту в Біткоїн-мережі. Найпростішою і найпоширенішою типовою операцією є операція надсилання коштів одному отримувачу. Ще одною типовою операцією, що вже згадувалась раніше, є операція консолідації виходів графа. Деякі інші типові операції детально розглянуті у підрозділі 3.3.

Розпізнавання типів операцій за такими ознаками, як кількості входів, кількості виходів, тощо, є евристичним за принципом: ми опираємось на певні припущення (раціональність користувача, відсутність очевидних дефектів у програмному забезпеченні, що формує транзакції, тощо). Набір властивостей, за якими ми відрізнятимемо один тип операції від іншого, називатимемо евристикою типової операції. Наприклад, евристика операції надсилання коштів одному отримувачу – це єдина транзакція з декількома входами та двома виходами, а евристика операції консолідації виходів – це єдина транзакція з багатьма входами та єдиним виходом. Евристики деяких типових операцій детально розглянуті у підрозділі 3.3.

RDF-модель даних Біткоїн-транзакцій, описана у підрозділі 2.3, дає змогу формулювати евристики типових операцій як запити пошуку патернів у RDF-графі стандартизованою мовою SPARQL, яка є достатньо експресивною для опису як евристик операцій, що складаються з однієї транзакції, так і евристик для розпізнавання складних багатокрокових операцій. У підрозділі 3.3 наведено SPARQL-запити для кожної типової операції, що розглядається.

3.2. Метод евристичної кластеризації транзакційних виходів

Поняття типової Біткоїн-операції передбачає наявність чітко визначеного фінансового змісту групи транзакцій. Виходячи з фінансового змісту певної операції, можна зробити припущення про те, як пов'язані власники входів та виходів, з яких складається ця операція. Одним з найпростіших таких припущень, як згадувалось раніше, є припущення про те, що платіж одному отримувачу здійснюється одним надсилачем, а отже всі входи відповідної транзакції належать одному й тому ж власнику. Саме це припущення використовувалось в ранніх публікаціях про аналізу графа транзакцій [33 – 35]. Біткоїн-протокол підтримує кооперативні транзакції, входи в яких належать декільком власникам, але до недавнього часу такі

транзакції зустрічались надзвичайно рідко, в першу чергу через відсутність зручних програмних інструментів для створення таких транзакцій. Лише з поширенням підтримки частково підписаних транзакцій (англ. PSBT – Partially Signed Bitcoin Transactions) [61, 62] і згаданих вище протоколів децентралізованого змішування виходів (CoinJoin) такі транзакції почали з'являтися частіше, збільшуючи кількість винятків з припущення про єдиного власника виходів транзакції. Іншим типовим операціям відповідають інші подібні припущення щодо взаємозв'язків виходів, з яких вони складаються. Ці припущення можна використовувати для того, щоб пропагувати зв'язки “вихід – власник” з одного виходу у типовій операції на інші.

В рамках цього дослідження розроблено метод евристичної кластеризації (групування за власністю) транзакційних виходів. Основна ідея методу полягає у тому, що, спираючись на евристичні припущення про зміст типової операції, відомі мітки власності (зв'язки “вихід – власник”) для одних виходів поширюються на інші виходи в межах цієї операції. Таким чином, якщо у графі є хоча б один вихід з відомим власником, запропонований метод може автоматично поширити його мітку власності (зв'язок “вихід – власник”) на всі виходи, з якими він пов'язаний певним фінансовим контекстом. В основі запропонованого методу лежить ітеративний процес, кожна ітерація якого складається з 3-х основних етапів:

1. У RDF-графі транзакцій знаходяться всі підграфи, які відповідають певному патерну типової операції.
2. Для кожного знайденого підграфа виділяється група виходів, що належать одному власнику згідно з евристичним припущенням.
3. Для всіх виходів у виділеній групі евристично пов'язаних виходів встановлюється новий зв'язок “вихід – власник”, а якщо такі зв'язки вже існують, здійснюється об'єднання власників.

Процес повторюється доти, доки його ітерації додають нові зв'язки. У результаті процесу утворюються групи виходів зі спільним власником. У такому вигляді цей метод представлено на науковій конференції у праці [4].

Далі наведено удосконалену версію методу кластеризації виходів, що працює навіть за умови відсутності конкретних міток власності у графі. Для цього використовується поняття абстрактного власника, а пропagaція міток власності відбувається навіть у випадку, коли жоден з виходів у групі не має мітки власності. Якщо у майбутньому у граф додається ідентифікуюча інформація для якогось абстрактного власника, деанонімізація є автоматичною, оскільки множина його виходів вже промаркована мітками власності. Недоліком такої модифікації є потенційне додавання нової RDF-трійки для багатьох вершин-виходів, що може значно збільшити його розмір, оскільки більшість вершин у ньому є виходами.

Вважатимемо, що кожен транзакційний вихід належить якомусь користувачу, тому для кожного виходу існує унікальний анонімний власник, про якого в початковому стані графа знань про транзакції нічого не відомо. Такого власника називатимемо абсолютно анонімним абстрактним власником. Для того, щоб уникнути витрачання ресурсів на зберігання RDF-трійок, що відображають очевидну інформацію, абсолютно анонімні абстрактні власники у графі знань представляються відсутністю зв'язку (RDF-трійки). Якщо для певного виходу не існує жодного зв'язку “вихід – власник”, це означає, що власник цього виходу – абсолютно анонімний абстрактний власник.

Як тільки у граф знань додається інформація, що зв'язує з абсолютно анонімним абстрактним власником більше одного виходу, такий власник перестає бути абсолютно анонімним. Такого власника називатимемо просто абстрактним власником, і оскільки за визначенням для нього існує щонайменше два зв'язки “вихід – власник”, такого власника вже неможливо представити відсутністю зв'язку, і тому необхідно ввести в граф відповідну вершину та зв'язок, тобто додати відповідну RDF-трійку. Дію додавання у

граф знань інформації, що зв'язує декілька виходів, які раніше мали зв'язки з різними абстрактними власниками, з одним абстрактним власником, називатимемо частковою деанонімізацією виходів.

Як описано у розділі 2, граф знань про транзакції містить зовнішню інформацію про асоціації між виходами та сутностями, що відображають реальних користувачів (публікації Біткоїн-адрес у соціальних мережах від імені певного облікового запису чи на веб-сторінках певних організацій, тощо). Ця інформація у графі відображається вершинами-власниками з конкретним ідентифікатором ресурсу для відповідної сутності (власне, профіль відповідного користувача у соціальній мережі, чи ідентифікатор самої організації, тощо). Таких власників називатимемо конкретними власниками. Конкретні власники відрізняються від абстрактних власників тим, що про конкретних власників відомі якісь зовнішні факти (ім'я користувача соціальної мережі, назва організації, тощо). Дію додавання у граф знань інформації, що зв'язує вихід, який раніше мав зв'язок з абстрактним власником, з конкретним власником, називатимемо конкретизацією виходів.

Основною задачею методу пропагації зв'язків власності на основі патернів типових Біткоїн-операцій, який описано у цьому підрозділі, є часткова анонімізація і конкретизація якомога більшої кількості виходів у графі знань про транзакції. Основна ідея запропонованого методу пропагації зв'язків полягає у тому, що для кожного підграфа, що відповідає типовій операції, існуючий зв'язок “вихід – власник” для одного з виходів, з яких складається ця операція, можна також встановити для деяких інших виходів, відповідно до евристичного припущення про зміст операції. При цьому ми завжди отримуємо часткову деанонімізацію виходів, але якщо хоча б один з виходів, пов'язаних між собою евристичним припущенням про зміст операції, має зв'язок з конкретним власником, ми отримуємо ще й конкретизацію всієї групи виходів. Слід зазначити, що часткова деанонімізація без конкретизації збільшує ймовірність конкретизації цієї ж

групи виходів у майбутньому, якщо якийсь з їхніх абстрактних власників буде конкретизований новою інформацією з зовнішніх джерел.

Для прикладу на рисунку 3.1 зображено граф знань про транзакції T_1 та T_2 . До появи нової пов'язаної транзакції у графі існує 4 унікальних абсолютно анонімних абстрактних власники $E_1 - E_4$.

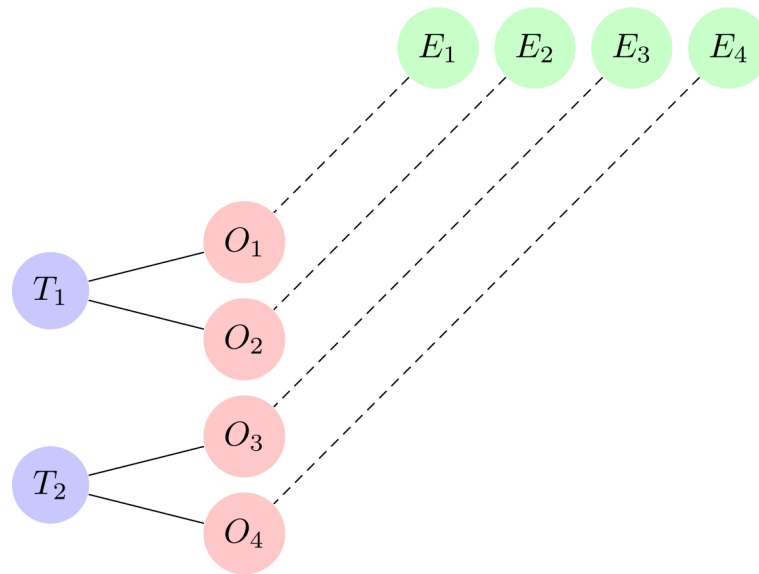


Рис. 3.1. Група транзакцій до пропагації зв'язків власності

На рисунку 3.2 зображено цей же граф після появи транзакції T_3 , яка використовує виходи O_1 та O_3 як входи, що дає змогу застосувати евристичне припущення про спільну власність входів однієї транзакції і зробити висновок, що абстрактний власник виходу O_1 (E_1) та виходу O_3 (E_3) – це один і той же власник, отже вершини-власників E_1 та E_3 можна об'єднати у єдину вершину-власника (для зручності тут – E_1) і видалити власника E_3 . Власник E_1 був частково деанонімізований: він більше не є абсолютно анонімним, оскільки для нього тепер існує 2 зв'язки “вихід – власник”. Якби при цьому один з цих двох власників був конкретним, то власника E_1 було б ще й конкретизовано. Таким чином, використовуючи евристичне припущення про зміст типової операції, зв'язок одного виходу з

його власником було перенесено на інший вихід, а у графі знань про транзакції стало на одного абстрактного власника менше.

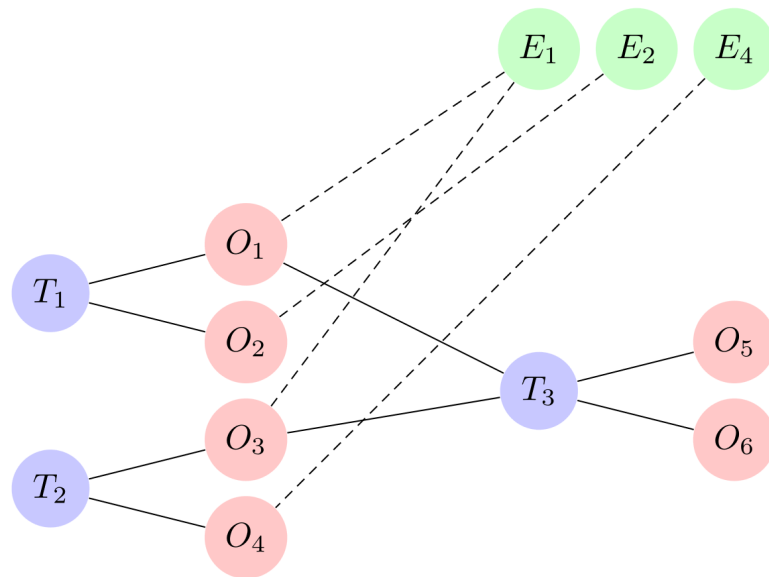


Рис. 3.2. Група транзакцій після пропagaції зв'язків власності

Загальний алгоритм, що реалізує запропонований метод пропagaції, виглядає наступним чином:

1. Знаходимо всі підграфи операцій певного типу у графі знань про Біткоїн-транзакції (наприклад “всі операції платежу одному отримувачу”; див. підрозділ 3.3).
2. Для кожного підграфа операції, знайденого у кроці 1, формуємо групу виходів, які належать одному користувачу згідно з евристичним припущенням про зміст операції (наприклад “всі входи в транзакції належать одному користувачу”).
3. Для отриманої групи виходів знаходимо всіх власників, присутніх у графі.
4. Якщо множина власників є порожньою (всі власники є абсолютно анонімними), створюємо нового абстрактного власника.
5. Якщо множина власників містить єдиного власника, обираємо його як власника для даних виходів.

6. Якщо множина власників містить більше одного власника, об'єднуємо їх всіх у єдиного нового власника, тобто замінюємо всі RDF-трійки у графі знань з предикатом “`brdf:outputOwner`” і об'єктом, що належить множині знайдених власників, на нові трійки з такими ж суб'єктом і предикатом, але новим об'єктом-власником. Повертаємо нового власника як результат вибору. Якщо деякі з знайдених власників є конкретними, на нову вершину-власника необхідно також перенести всі атрибути відповідних вершин, зокрема ідентифікатори ресурсів власників. Ідіоматичний спосіб моделювання такої ситуації в RDF – за допомогою властивості “`owl:sameAs`” [59]. Цей крок є найбільш важливим, оскільки тут може відбуватися повна деанонімізація (об'єднання конкретних власників, що розкриває ідентичність користувача).
7. Для кожного виходу в групі, сформованій у кроці 2, створюємо новий зв'язок “вихід-власник” з новим власником, обраним у кроках 4 – 6.

Остаточний вигляд алгоритму пропagaції зв'язків буде дещо іншим для різних типових операцій. У кожної з них буде свій власний крок 1, який відповідає за розпізнавання патерну у графі відповідно до евристики типової операції, та свій власний крок 2, який відповідає за виділення групи виходів з ймовірним єдиним власником згідно з евристичним припущенням про зміст операції.

Блок-схема загального алгоритму пропagaції зв'язків наведена на рисунку 3.3. Припускається, що всі підграфи операцій певного типу у графі знаходяться за допомогою SPARQL-запиту, і результат виконання такого запиту – це курсор, який дає змогу отримувати підграфи один за одним без попереднього зберігання всієї множини результатів у пам'яті.

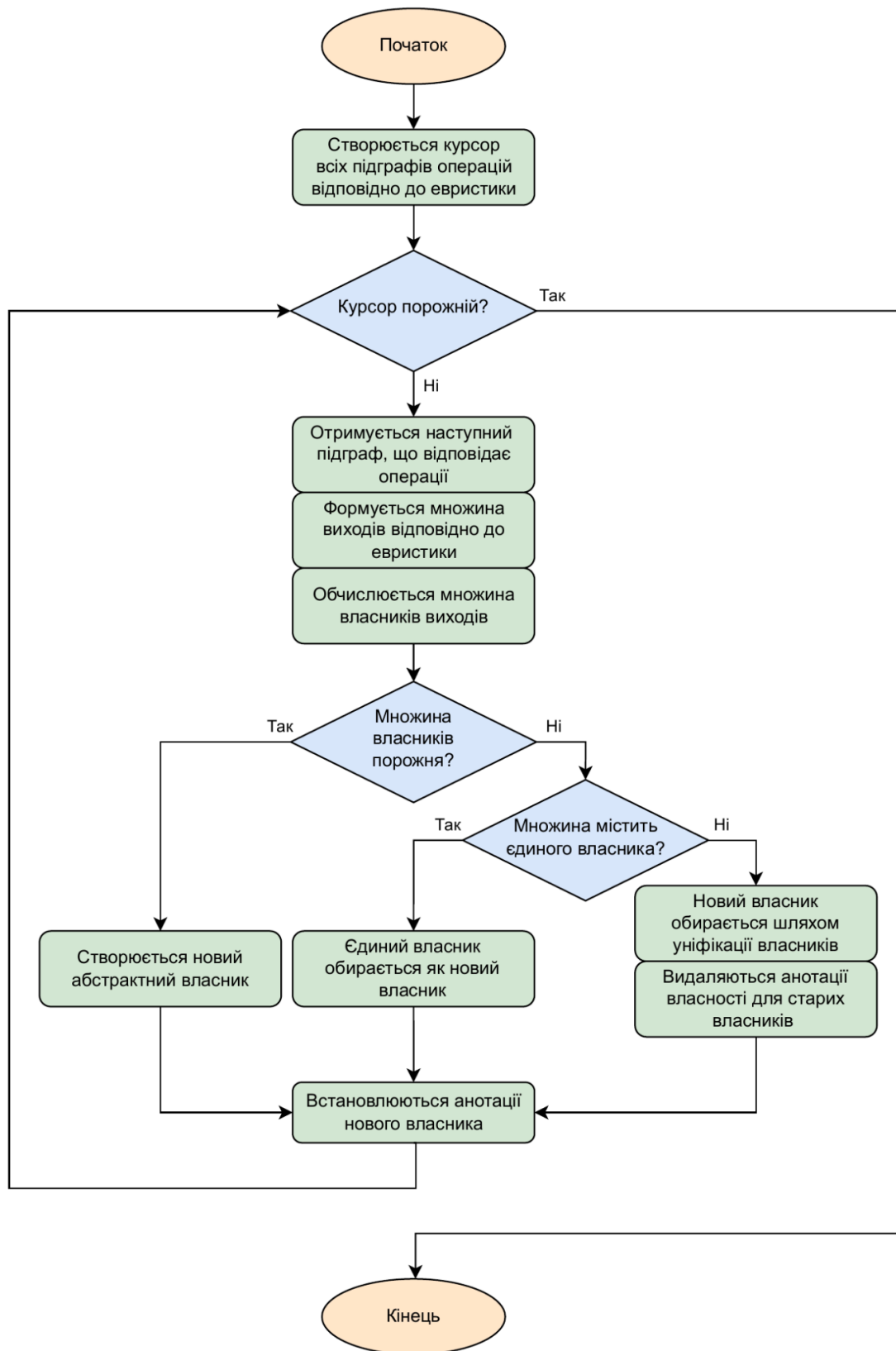


Рис. 3.3. Загальний алгоритм пропагації зв'язків власності для певної типової операції

У кроці 3 алгоритму множину власників для даної множини виходів можна знайти за допомогою SPARQL-запиту для пошуку існуючих зв'язків “вихід – власник”, наведеного у лістингу 3.1.

Лістинг 3.1. SPARQL-запит для пошуку власників множини виходів

```
SELECT ?out ?owner {  
  VALUES ?out { OUT1 OUT2 OUT3 ... }  
  OPTIONAL {  
    ?out brdf:outputOwner ?owner  
  }  
}
```

У кроці 6 алгоритму заміну RDF-трійок зв'язків виходів з старим власником на зв'язки з новим власником можна здійснити за допомогою операції SPARQL Update, наведеної у лістингу 3.2.

Лістинг 3.2. Операція SPARQL Update заміни власників виходів

```
INSERT { ?out brdf:outputOwner NEW-OWNER }  
DELETE { ?out brdf:outputOwner ?owner }  
WHERE { VALUES ?owner { OLD-OWNER-1 ... }  
        ?out brdf:outputOwner ?owner }
```

3.3. Типові Біткоїн-операції

У цьому підрозділі розглядаються типові Біткоїн-операції, і для кожної з них наводиться її опис, вигляд у графі транзакцій, узагальнений SPARQL-запит для розпізнавання її у RDF-графі знань про транзакції, а також логіка формування групи виходів, що потенційно мають єдиного власника, згідно з евристичним припущенням про зміст операції. Для зручності, на рисунках зображено лише вершини графа транзакцій. Атрибути вершин-транзакцій та вершин-виходів, які теж є вершинами у графі знань, опущені.

Каталог, наведений у цьому підрозділі, містить аналіз лише тих операцій, для яких в рамках дослідження було реалізовано і протестовано алгоритми пропагації зв'язків власності, і ні в якому разі не є повним. Зокрема, тут розглядається лише один протокол другого рівня (протокол змішування виходів CoinJoin), і не розглядаються спеціальні операції, що використовуються у інших протоколах другого рівня (наприклад, операції відкриття та закриття каналів Lightning-мережі).

Одним з напрямків подальших досліджень є виділення інших типових операцій, окреслення їхнього вигляду у графі транзакцій, формулювання евристичних припущень щодо їхнього фінансового змісту, реалізація SPARQL-запитів і допоміжних фільтр-функцій для розпізнавання цих операцій у графі, та, наостанок, реалізація алгоритмів пропагації зв'язків власності відповідно до сформульованого змісту.

3.3.1. Платіж одному отримувачу

Найбільш поширена Біткоїн-операція – це простий переказ коштів (платіж) між двома користувачами (надсилачем та єдиним отримувачем). Оскільки раціональний користувач намагається мінімізувати комісію за транзакцію, то ідеальним випадком є ситуація, коли кількість біткоїна, яка надсилається користувачем в черговій транзакції, повністю покривається лише одним з його виходів так, що різниця буде достатньою комісією за транзакцію, але не більше. Втім, типовий користувач отримує і здійснює платежі на різні суми, тому в загальному випадку ймовірність такого співпадіння є не дуже великою. Тому типовий платіж між двома користувачами має 2 виходи, один з яких є цільовим, тобто виходом, який належить отримувачу, а інший – виходом-рештою, який належить надсилачеві. Кількість входів при цьому може бути різною, але зазвичай це 1 – 4 входи. Більші кількості входів можуть вказувати на додатковий фінансовий зміст такої операції, наприклад на випадок принагідної консолідації виходів (див. пункт 3.3.3).

На рисунку 3.4 зображено два платежі між двома користувачами. Платіж T_1 має один вхід та два виходи, а платіж T_2 – три входи та два виходи.

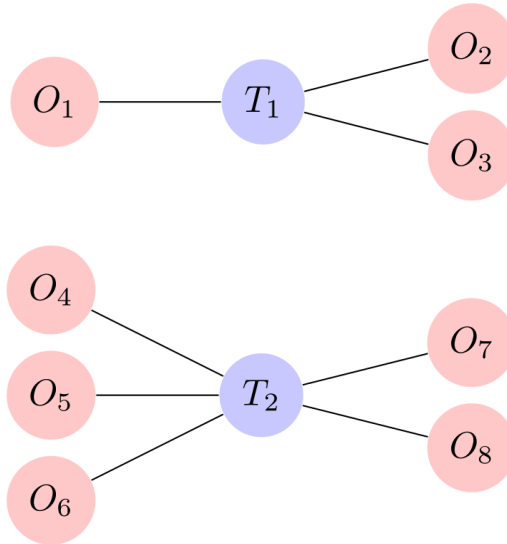


Рис. 3.4. Платежі одному отримувачу

Основною евристичною характеристикою платежу одному отримувачу є наявність двох виходів. Завдяки тому, що кількості входів та виходів включені до властивостей транзакції у розробленій RDF-моделі, достатньо знайти всі транзакції, для яких властивість “`brdf:txOutputCount`” має значення 2. Перший SPARQL-запит у лістингу 3.3 знаходить всі операції такого типу. Слід зазначити, що оскільки цей тип Біткоїн-операції є найпоширенішим, пошук усіх таких операцій у графі знань про всю історію транзакцій поверне величезну кількість результатів (сотні мільйонів операцій). Це стосується і інших типових операцій, але значно меншою мірою. Модель, розроблена у підрозділі 2.3, дає змогу швидко звужувати простір пошуку до проміжків блоків (або проміжків часу) і отримувати сотні результатів замість сотень мільйонів результатів. Другий SPARQL-запит у лістингу 3.3 знаходить всі такі операції (1062 з 1257 транзакцій) у блоці 420000.

Лістинг 3.3. Пошук платежів з одним отримувачем

```
#1
SELECT ?tx {
  ?tx brdf:txOutputCount 2.
}

#2
SELECT ?tx {
  brdf:420000 brdf:blockTx ?tx.
  ?tx brdf:txOutputCount 2.
}
```

Платіж одному отримувачу є операцією між двома користувачами, тому всі входи транзакції, якою він реалізується, належать єдиному власнику – надсилачеві. Але такі операції є особливо цікавими через те, що вони мають два виходи, один з яких є цільовим, тобто належить іншому власнику, а інший – рештою, тобто належить тому ж власнику, що й входи, і ці два входи часто можна розрізнити за допомогою евристичних припущень, наприклад:

1. Оптимальність виходу-решти. Раціональний користувач (або програма-гаманець, що намагається мінімізувати комісію) не додаватиме ще один вхід в транзакцію, якщо іншого входу достатньо, тому вихід, який є меншим за якийсь з входів транзакції з більше ніж одним входом, мусить бути виходом-рештою, інакше додавання ще одного входу було нерациональним. SPARQL-запит, що знаходить платежі одному отримувачу, а також виділяє вихід-решту транзакції, використовуючи евристику оптимальності виходу-решти, наведено у лістингу 3.4.
2. Унікальність адреси виходу-решти. При створенні виходу-решти більшість програм-гаманців генерують нову адресу, тому вихід не може бути виходом-рештою, якщо його адреса зустрічається у

попередніх транзакціях [41]. Цю евристику можна сформулювати як SPARQL-запит, але значно простіше реалізувати її як спеціальну фільтр-функцію, що має прямий доступ до графа (див. підрозділ 4.1).

3. Співпадіння типів адрес входів та виходу-решти. Оскільки адреса виходу-решти генерується у тому ж дереві ключів ієрархічного гаманця [25], що і адреси у входах, то тип адреси виходу-решти повинен співпадати з типом адреси у входах. Тому якщо всі входи транзакції і один з її виходів мають один тип адреси, а інший вихід має інший тип адреси, то саме вихід з іншим типом адреси є цільовим. Цю евристику також простіше реалізувати як спеціальну фільтр-функцію, що має прямий доступ до графа і може агрегувати дані про входи та виходи без використання громіздкої агрегації у SPARQL-запитах.

Лістинг 3.4. SPARQL-запит для розпізнавання виходу-решти

```
SELECT ?tx ?change {  
  ?tx brdf:txOutputCount 2.  
  ?tx brdf:txInputCount ?icount.  
  FILTER (?icount > 1)  
  ?tx brdf:txOutput ?change.  
  FILTER EXISTS {  
    ?in brdf:outputInputTx ?tx.  
    ?in brdf:outputAmount ?inAmount.  
    ?change brdf:outputAmount ?changeAmount.  
    FILTER (?inAmount >= ?changeAmount).  
  }  
}
```

У кроці 2 загального алгоритму пропагації зв'язків власності група виходів з єдиним власником для операції платежу одному отримувачу – це всі входи транзакції та її вихід-решта, якщо його можна відрізнити від цільового виходу за допомогою евристик, описаних вище.

3.3.2. Платіж декільком отримувачам

Досить простим і очевидним способом зменшення комісії за транзакції є групування платежів – об'єднання кількох платежів різним отримувачам у єдину транзакцію. У такому випадку надсилач платить комісію лише за один набір полів метаданих транзакції і за один вихід-решту.

На рисунку 3.5 зображено платіж трьом отримувачам: транзакція T_1 має 4 виходи, один з яких ймовірно є виходом-рештою, а три інші – цільовими.

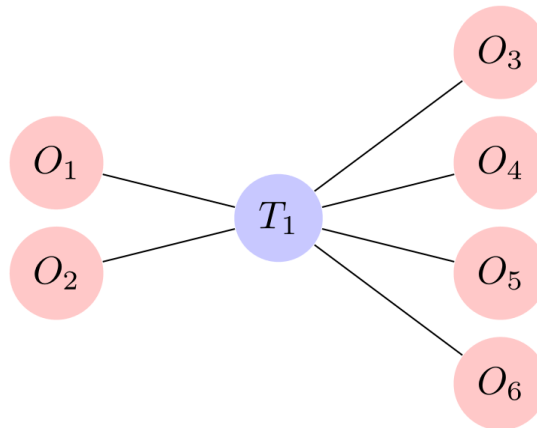


Рис. 3.5. Платіж декільком одержувачам

Евристикою для розпізнавання платежів з декількома отримувачами є наявність більш ніж двох виходів, і відповідний SPARQL-запит використовує фільтрування за допомогою нерівності (див. Перший запит у лістингу 3.5). Кількість входів, як і у платежів з одним отримувачем, може бути різною, але кількості, більші за 1 – 4 входи можуть вказувати на принагідну консолідацію виходів (див. пункт 3.3.3). На практиці групування платежів індивідуальними користувачами використовується досить рідко. Для прикладу, другий SPARQL-запит з лістингу 3.5 повертає лише 54 такі транзакції у блоці 420000, тоді як транзакцій з двома виходами (див. лістинг 3.3) у ньому – 1064. Ймовірно, важливим чинником

цього є те, що більшість популярних застосунків-гаманців не підтримують створення платежів з кількома одержувачами. Лише кілька застосунків надають таку функціональність [63 – 65].

Лістинг 3.5. Пошук платежів з декількома отримувачами

```
#1
SELECT ?tx {
  ?tx brdf:txOutputCount ?ocount.
  FILTER (?ocount > 2)
}

#2
SELECT ?tx {
  brdf:420000 brdf:blockTx ?tx.
  ?tx brdf:txOutputCount ?ocount.
  FILTER (?ocount > 2)
}
```

Групування платежів часто використовують великі сервіси, такі як біржі. Зазвичай всі внутрішні операції з біткоїном в межах сервісу відбуваються поза Біткоїн-мережею, записи про ці операції зберігаються у внутрішній базі даних сервісу, а сам біткоїн зберігається у виходах, які належать сервісу, а не окремим його користувачам. Коли користувач робить запит на виплату (наприклад, виведення біткоїна з біржі на власний гаманець), сервіс групує цей запит з іншими такими запитами, що надійшли протягом певного проміжку часу від інших користувачів, і здійснює єдиний Біткоїн-платіж, який знищує один великий вихід, що належить сервісу, і створює по одному виходу для кожного користувача. Такий підхід дає змогу сервісу значно економити на комісіях за транзакції, і це може бути достатньою мотивацією для сервісу, щоб інвестувати ресурси у розроблення власних інструментів конструювання транзакцій.

Групові виплати з сервісів можна розпізнати за допомогою евристичного припущення, що для сервісу, що має багато користувачів,

платіж з 5-ма чи 10-ма цільовими виходами може все ще бути не вигідним, і такий сервіс буде збирати групи по 100 – 200 користувачів. Подібні великі операції можна шукати за допомогою SPARQL-запитів у лістингу 3.5, але зі значно більшим пороговим значенням кількості виходів (див. лістинг 3.6).

Лістинг 3.6. Пошук групових платежів великих сервісів

```
SELECT ?tx {  
  ?tx brdf:txOutputCount ?ocount.  
  FILTER (?ocount > 100)  
}
```

У кроці 2 алгоритму пропагації зв'язків власності група виходів з єдиним власником для операції платежу декільком одержувачам – це всі входи відповідної транзакції та її вихід-решта, якщо його можливо відрізнити від решти виходів за допомогою певних евристик, описаних у пункті 3.3.1.

3.3.3. Консолідація виходів

Кожен отриманий Біткоїн-платіж додає до множини транзакційних виходів, які належать певному користувачу, ще один вихід. Кожен надісланий платіж знищує один, а рідше два, чи більше, таких виходи, але у більшості випадків додає до множини виходів користувача один новий вихід-решту. Якщо користувач отримує і надсилає платежі з приблизно однаковою частотою, то кількість виходів, яку він контролює, поступово зменшується. Втім, через те, що емісія Біткоїна зменшується і прямує до 0, все більш важливою стає його функція як інструменту накопичення цінності. Як наслідок, для все більшої кількості індивідуальних користувачів отримання платежів значно переважає надсилання, і тому кількість їхніх виходів лише зростає з часом. Для сервісів, що продають товари, чи послуги за біткоїн, вихідні платежі у біткоїні можуть бути

взагалі відсутні, і кількість їхніх виходів може зростати з великою швидкістю.

Накопичення виходів є проблемою як для окремого користувача, так і для мережі загалом:

1. Зростання розміру транзакції. Якщо користувач володіє багатьма невеликими виходами, для здійснення великого платежу йому може знадобитись велика транзакція з десятками входів і, відповідно, велика комісія за неї. При цьому такий платіж може співпасти в часі з навантаженням на мережу, коли комісія за транзакцію в десятки разів перевищуватиме мінімальну комісію. Історично такі періоди навантаження могли тривати тижнями [66, 67], що ускладнює відкладення великого платежу до періоду “затишшя”. Слід зазначити, що зменшення середнього розміру виходу в гаманці є очікуваним явищем: оскільки раціональна програма-гаманець намагається мінімізувати комісію за транзакцію, то вихід-решта буде обов'язково меншим за кожен з входів (інакше додавання меншого входу було нераціональним).
2. Обчислювальна складність підготовки транзакцій. Чим більша кількість виходів контролюється програмою-гаманцем, тим складнішими є обчислення для підбору оптимальної комбінації входів для кожної нової транзакції, що створюється.
3. Збільшення глобального індексу невикористаних виходів. Чим більше виходів контролює окремий користувач, тим більше таких виходів існує глобально, і тим більшим стає ключовий індекс Біткоїн-протоколу – індекс невикористаних транзакційних виходів. Кожен валідуючий вузол мережі змушений тримати цей індекс в оперативній пам'яті для того, щоб мати змогу швидко валідувати нові транзакції, і зростання цього індексу збільшує мінімальну вартість обчислювальних ресурсів, необхідних для підтримки вузла, що призводить до централізації мережі.

Для вирішення всіх цих проблем майже з самого початку існування мережі використовуються спеціальні Біткоїн-операції, що знищують багато дрібних виходів і створюють один вихід великого розміру, який належить тому ж користувачу. Такі операції називаються операціями консолідації виходів [68 – 70] і зазвичай здійснюються в періоди низького навантаження на мережу, коли комісії за великі транзакції, які для цього необхідні, будуть мінімальними. Таким чином користувачі уникають необхідності створення великих транзакцій в періоди великого навантаження на мережу, коли у них може не бути можливості чекати.

Приклад підграфа операції консолідації вже був наведений під час аналізу типів графа транзакцій (див. рисунок 2.6). Для зручності на рисунку 3.6 зображено ще одну операцію консолідації з більшою кількістю входів.

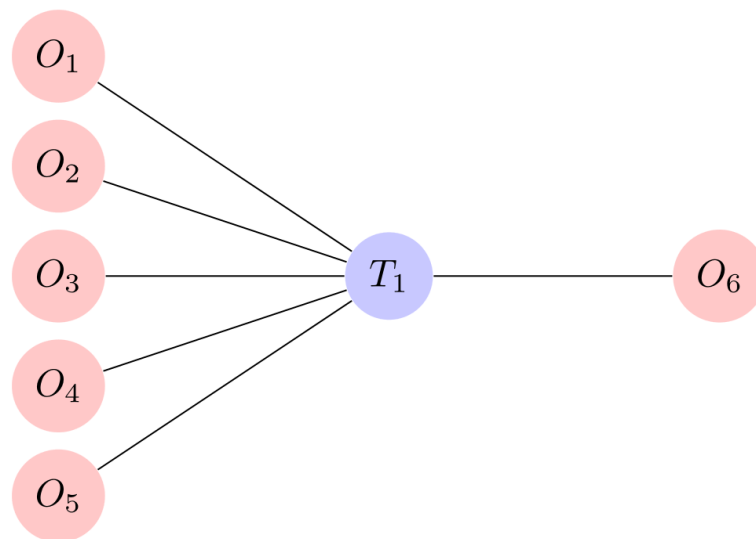


Рис. 3.6. Операція консолідації виходів

Евристикою для розпізнавання простих операцій консолідації є наявність багатьох входів та єдиного виходу у випадку простої консолідації або двох виходів у випадку принагідної консолідації. Відповідний SPARQL-запит наведено у лістингу 3.7.

Лістинг 3.7. Пошук операцій консолідації

```
SELECT ?tx {  
  ?tx brdf:txInputCount ?icount.  
  ?tx brdf:txOutputCount 1.  
  FILTER (?icount >= N)  
}
```

Операції консолідації є досить поширеними, тому подібний пошук потрібно звужувати до певного проміжку блоків або обирати порогове значення N таке, щоб відфільтрувати менші цікаві консолідації. Наприклад у блоці 420000 такий запит знаходить 32 транзакції при $N = 10$.

Операція консолідації є однією з найбільш шкідливих операцій з точки зору приватності даних користувача, оскільки, відповідно до її фінансового змісту, вона зв'язує всі свої входи та єдиний вихід з одним і тим же власником [41, 68]. У загальному алгоритмі пропагації зв'язків власності, запропонованому у підрозділі 3.2, група виходів з єдиним власником (крок 2) – це всі входи транзакції консолідації та її єдиний вихід. Відповідний SPARQL-запит наведено у лістингу 3.8.

Лістинг 3.8. Пошук всіх входів та виходів та виходів транзакції

```
SELECT ?out {  
  BIND (TXID AS ?tx)  
  ?tx (brdf:txOutput) | (^brdf:outputInputTx) ?out.  
}
```

Цікавим випадком консолідації є принагідна консолідація (англ. incidental consolidation). Принагідна консолідація – це операція що об'єднує консолідацію і платіж у єдину транзакцію. Така транзакція знищує велику кількість дрібних виходів і надсилає частину цієї суми одному чи кільком одержувачам, а залишок консолідує у один великий вихід-решту.

Принагідна консолідація може бути як вимушеною (користувачу потрібно здійснити платіж, але у нього немає інших великих виходів), або

навмисною (користувач об'єднує консолідацію і платіж, щоб зекономити на комісії і створити невизначеність щодо власності виходів), і з точки зору пропагації зв'язків власності така операція є ідентичною операції платежу одному чи кільком отримувачам: вона зв'язує всі свої входи з одним і тим же власником, але не створює однозначного зв'язку з цим власником для жодного з своїх виходів.

Однією з евристичних ознак розрізнення цільового виходу та виходу-решти тут може бути розмір сплаченої комісії. Якщо користувач заплатив велику комісію (наприклад, під час періоду навантаження в мережі), більш ймовірно, що це вимушена консолідація, і тоді вихід-решта – це менший з двох виходів такої транзакції. Втім, таке припущення є неточним і потребує статистичних досліджень подібних транзакцій.

На рисунку 3.7 зображено операцію принагідної консолідації 5-ти виходів $O_1 - O_5$, яка надсилає певну суму на один з виходів O_6 або O_7 і консолідує решту в інший.

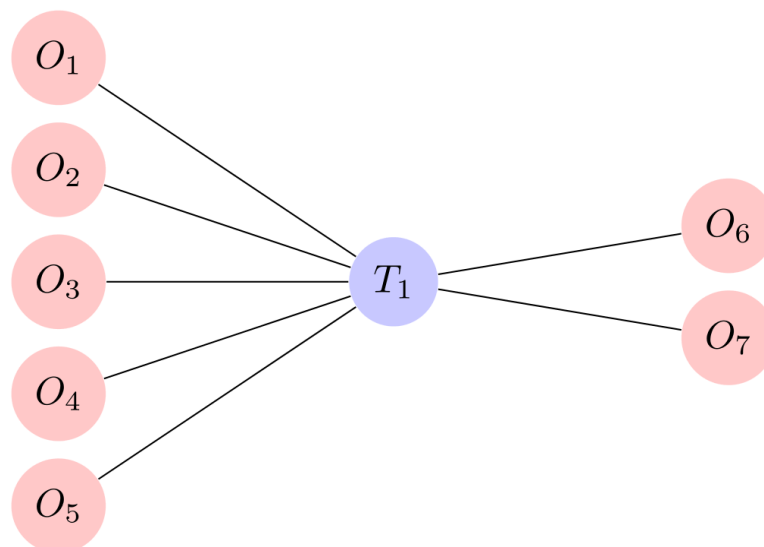


Рис. 3.7. Операція принагідної консолідації виходів

Оскільки принагідна консолідація є частковим випадком платежу одному чи кільком одержувачам, група виходів з єдиним власником такої

операції у кроці 2 алгоритму пропагації зв'язків власності є аналогічною — це всі входи відповідної транзакції та її вихід-решта, якщо його можливо відрізнити від решти виходів.

3.3.4. Змішування виходів (CoinJoin)

Однією з фундаментальних евристик, що застосовуються для аналізу графа транзакцій, є припущення про єдиного власника всіх входів однієї транзакції. Це припущення є актуальним для більшості транзакцій, в першу чергу через те, що більшість програм-гаманців не підтримує ніяких інших способів конструювання транзакцій. Втім, як вже було згадано раніше, Біткоїн-протокол підтримує використання в межах однієї транзакції входів, що належать різним власникам, оскільки кожен вхід є самостійною структурою в транзакції. Операції CoinJoin — це операції, що змішують виходи різних власників і таким чином розривають зв'язок між конкретним входом і конкретним виходом (зв'язків “платник — отримувач”), чим ускладнюють аналіз графа транзакцій і деанонімізацію користувачів. Це найбільш поширений тип операції, для якого не може бути застосована евристика щодо єдиного власника всіх входів транзакції.

Залежно від варіації протоколу координації змішування виходів, що використовується, операції CoinJoin можуть дещо відрізнитись, і конкретний патерн такої операції у графі потрібно узгоджувати з специфікацією конкретного протоколу, але в загальному випадку операції CoinJoin у графі транзакцій — це транзакції, що мають дві основні особливості:

1. Багато входів і багато виходів, що, власне, і забезпечує змішування виходів. Ця особливість є необхідною умовою класифікації певної транзакції як операції змішування виходів.
2. Кілька або багато виходів однакового розміру (ця властивість є необхідною для обфускації відповідності між входом та виходом). Ця особливість може вважатись достатньою умовою класифікації

певної транзакції як операції змішування виходів. Якщо вона не виконується (всі виходи – різного розміру), то така транзакція може бути, наприклад, платежем багатьом отримувачам, який суміщено з консолідацією виходів.

На рисунку 3.8 зображено операцію CoinJoin, що змішує виходи $O_1 - O_5$ і створює групу виходів $O_6 - O_{12}$, деякі з яких є цільовими виходами, і мають однаковий розмір, і інші – виходами-рештою.

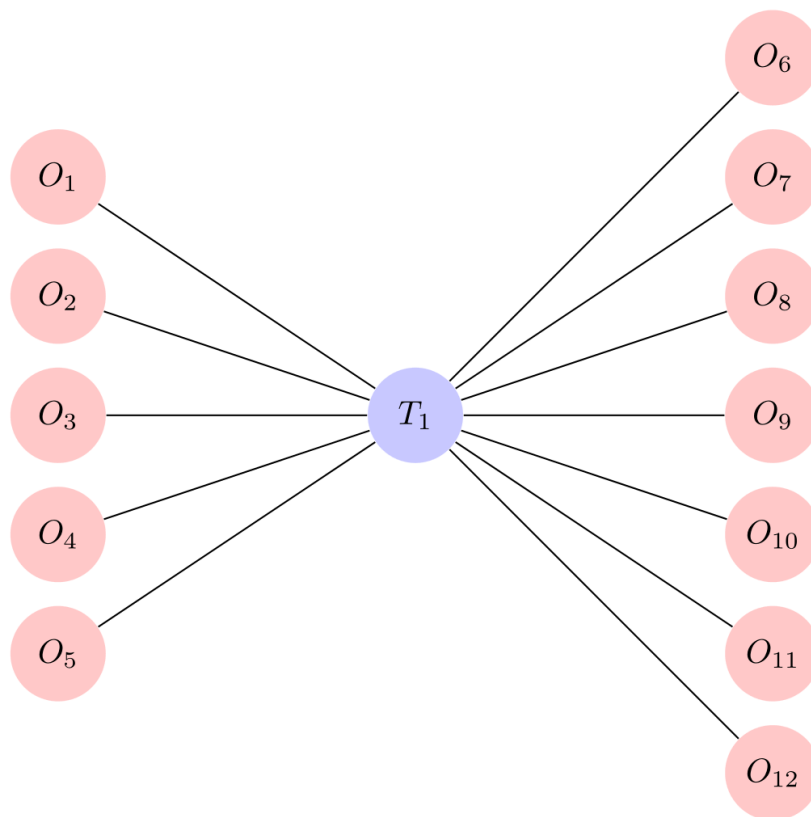


Рис. 3.8. Операція CoinJoin

SPARQL-запит для пошуку транзакцій за першою з перелічених вище ознак наведено у лістингу 3.9, але для фільтрування транзакцій за наявністю виходів однакового розміру зручніше реалізувати спеціальну фільтр-функцію, що має прямий доступ до графа, щоб уникнути використання складних підзапитів з агрегацією.

Лістинг 3.9. Простий пошук операцій CoinJoin

```
SELECT ?tx {  
  ?tx brdf:txInputCount ?icount.  
  ?tx brdf:txOutputCount ?ocount.  
  FILTER (?icount > N && ?ocount > M)  
}
```

В деяких випадках (для деяких варіацій протоколу координації змішування виходів) можливо встановити відповідність між входом і виходом-рештою. Наприклад, якщо така операція має вхід O_N розміру A_N , декілька виходів однакового розміру A_0 , і вихід O_M розміру $A_M = A_N - A_0$, можемо припустити, що вихід O_M є виходом-рештою для входу O_N , а отже належить тому ж власнику.

На відміну від попередніх типових операцій, групи виходів з єдиним власником для операції змішування виходів у кроці 2 алгоритму пропагації зв'язків власності – це лише ті пари (вхід, вихід-решта), які вдалось встановити за відповідністю розміру виходу-решти. Евристика єдиного власника всіх входів транзакції не виконується для колаборативних транзакцій. При цьому можливість розпізнавання таких транзакцій є корисною під час аналізу графа, оскільки сам факт використання протоколу змішування виходів може бути маркером того, що користувач намагається приховати частину історії транзакцій, тощо.

3.3.5. Послідовності тимчасових адрес

Примітивним способом приховування кінцевого отримувача біткоїна є використання тимчасових адрес: платіж отримується на адресу у тимчасовому мобільному гаманці, а пізніше біткоїн переводиться з цієї тимчасової адреси на кінцеву. Якщо для цього використовується симуляція платежів (тобто транзакція, що переводить біткоїн на кінцеву адресу, виглядає як справжній платіж з цільовим виходом і виходом-рештою), то такий підхід справді може мати ефект обфускації кінцевого отримувача [3],

але якщо транзакція, що переводить біткоїн з тимчасової адреси на кінцеву, має єдиний вихід, такий підхід є неефективним. Більш складним, але таким же неефективним способом приховування кінцевого отримувача є використання послідовності тимчасових адрес. Такий спосіб приховування кінцевих адрес є наївним, тому що навіть сервісу огляду ланцюга блоків достатньо, щоб у ручному режимі прослідкувати послідовність таких змін адреси аж до кінцевої, а у візуалізованому графі транзакцій такі послідовності зміни адрес є очевидними і легко помітними. Більше того, послідовність з декількох тимчасових адрес привертає додаткову увагу під час аналізу графа транзакцій, оскільки вказує на спробу, хоч і неефективну, приховати кінцеву адресу.

На рисунку 3.9 зображено послідовність тимчасових адрес $O_1 - O_3$. Оскільки O_4 поки що є невикористаним, ми не знаємо, чи він тимчасовий, чи вже кінцевий. SPARQL-запит для розпізнавання таких послідовностей необхідно конструювати програмним чином, оскільки SPARQL не підтримує циклічних та рекурсивних патернів.

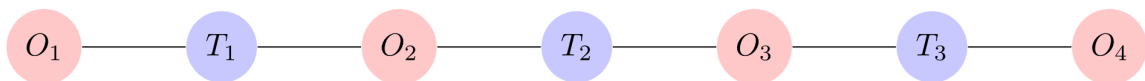


Рис. 3.9. Послідовність тимчасових адрес

Якщо у тимчасовому гаманці накопичилось декілька виходів, то транзакція переведення біткоїна на кінцеву адресу може використати їх усіх, і така операція нічим не відрізняється від консолідації. Такі операції називають операціями очищення гаманця (англ. wallet sweep), і деякі програми-гаманці навіть надають функціональність очищення гаманця, що отримує приватний ключ і забирає всі виходи, які контролюються цим ключем. Власне, операція зміни адреси є виродженим випадком консолідації з одним входом.

Приклад SPARQL-запиту для розпізнавання послідовностей в тимчасових адрес довжиною 2 наведено у лістингу 3.10.

Лістинг 3.10. SPARQL-запит для розпізнавання тимчасових адрес

```
SELECT ?tx {  
  #1  
  ?o1 brdf:outputInputTx ?tx1.  
  ?tx1 brdf:txInputCount 1.  
  ?tx1 brdf:txOutputCount 1.  
  ?tx1 brdf:txOutput ?o2.  
  #2  
  ?o2 brdf:outputInputTx ?tx2.  
  ?tx2 brdf:txInputCount 1.  
  ?tx2 brdf:txOutputCount 1.  
  ?tx2 brdf:txOutput ?o3.  
  ...  
}
```

Евристичне припущення щодо власності над входами і виходами таких операцій є аналогічним припущенню для операцій консолідації: всі входи і виходи послідовності транзакцій з одним входом і одним виходом, ймовірно, належать одному й тому ж власнику.

3.3.6. Каскадна консолідація

Каскадна консолідація — це наївний метод приховування консолідації виходів, що працює наступним чином:

1. Множина виходів, які необхідно консолідувати, розбивається на групи по декілька виходів.
2. Здійснюються окремі операції консолідації груп виходів.
3. Для виходів, отриманих в результаті кроку 2, повторюються кроки 1 та 2, доки їх кількість є більшою за 1.

Якщо проста консолідація зв'язує велику кількість виходів з єдиним власником у одній транзакції, то у операції каскадній консолідації на

перший погляд прямий зв'язок встановлюється лише у малих групах виходів, і якщо ітерації цього процесу розділені в часі, то окремі транзакції знаходитимуться у різних блоках. Насправді ж такий метод консолідації приховує інформацію про повну множину виходів, що консоліднуються, лише за умови, що система аналізу графа розглядає окремі транзакції або невеликі групи транзакцій в межах певного проміжку часу (послідовності блоків). У графі всієї історії транзакцій такі операції є очевидними, оскільки виглядають як дерева, і для кожного окремого кроку такої операції застосовується евристика щодо єдиного власника всіх входів і єдиного виходу транзакції консолідації, описана у пункті 3.3.3, і висновок про власника виходу пропагується на транзакції на наступному рівні дерева.

На рисунку 3.10 зображено каскадну консолідацію виходів $O_1 - O_5$ у єдиний вихід O_9 . Оскільки згідно з евристикою про операцію консолідації виходи O_7 , O_8 та O_9 належать єдиному власнику, то і виходи O_1 , O_2 , O_5 і O_6 належать цьому власнику, звідки в свою чергу випливає, що і виходи O_3 та O_4 теж належать цьому власнику.

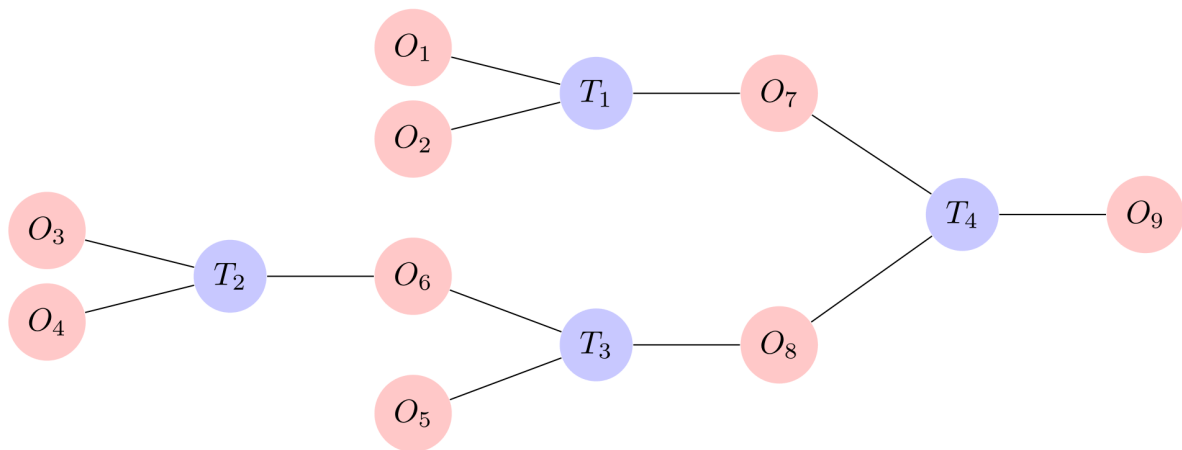


Рис. 3.10. Дерево каскадної консолідації виходів

SPARQL-запит у лістингу 3.11 знаходить всі виходи, які є коренями дерев консолідації, що містять 2 і більше транзакції, а також агрегує

транзакції, що входять до кожного такого дерева, у рядкове значення для подальшої обробки.

Лістинг 3.11. Пошук коренів дерев консолідації

```
SELECT ?out (GROUP_CONCAT (?tx; separator="; ") AS ?txs) {  
  ?tx (brdf:txOutput/brdf:outputInputTx)* / brdf:txOutput ?out.  
  FILTER NOT EXISTS {  
    ?tx (brdf:txOutput/brdf:outputInputTx)* ?branchTx.  
    ?branchTx  
      (brdf:txOutput/brdf:outputInputTx)* / brdf:txOutput ?out.  
    ?branchTx brdf:txOutputCount ?branchOutCount.  
    FILTER (?branchOutCount > 1)  
  }  
}  
GROUP BY ?out  
HAVING (COUNT(DISTINCT ?tx) > 1)
```

У кроці 2 алгоритму пропагації зв'язків власності група виходів з єдиним власником для операції платежу декільком одержувачам – це всі входи та входи всіх транзакцій, що входять в дерево каскадної консолідації (тобто усіх транзакцій, знайдених SPARQL-запитом, наведеним вище, для кожного кореневого виходу).

3.4. Методи захисту від патернового аналізу графа транзакцій

Основною ідеєю при формулюванні принципів чи розробці методів захисту від патернового аналізу графа Біткоїн-транзакцій є навмисне створення “шуму” у графі, що порушує типові евристики, які застосовуються для встановлення зв'язків між виходами, а отже ускладнює їх кластеризацію. Користувач свідомо жертвує мінімізацією комісії за транзакції на користь покращення приватності своїх транзакційних даних, що дає змогу програмі-гаманцю використовувати більшу кількість входів у транзакціях, розбивати одну транзакцію на декілька окремих транзакцій, тощо.

Евристика щодо єдиного власника всіх входів транзакції є однією з найбільш стійких, оскільки для створення операцій, які її порушують, необхідні протоколи кооперативного створення транзакцій, таких як CoinJoin. Такі протоколи стають дедалі більш популярними, а їх підтримка починає з'являтися у поширених гаманцях, але в загальному випадку їх використання потребує свідомих дій і активного контролю користувача, оскільки автоматизоване створення транзакцій змішування виходів у фоновому режимі, як це реалізовано у гаманці Wasabi [71], передбачає постійний неконтрольований доступ до криптографічних ключів користувача, що породжує низку інших безпекових ризиків.

Застосування евристики оптимальності виходу-решти, розглянутої у пункті 3.3.1, можна значно ускладнити або навіть унеможливити використанням транзакцій з більше ніж двома виходами. Платежі з декількома отримувачами є більш стійкими до евристичного аналізу, ніж платежі з одним отримувачем, оскільки відрізнити вихід-решту від цільових виходів у таких операціях значно складніше. Якщо платежі відбуваються рідко, і їх групування складно організувати, то груповий платіж можна симулювати, додаючи у транзакцію більше одного виходу-решти. Загалом будь-які транзакції, що мають декілька виходів, підвищують складність зв'язків у графі і знижують достовірність висновків з результатів евристичного патернового аналізу.

Операція консолідації виходів є однією з найбільш цікавих операцій при розробці рекомендацій і методів захисту від патернового аналізу графа транзакцій, оскільки консолідація має характерний патерн у графі, який легко розпізнається простим SPARQL-запитом (див. пункт 3.3.3). Консолідації виходів, що збігається до єдиного виходу, легко розпізнається у графі за допомогою пошуку дерев транзакцій, тобто підграфів графа транзакцій, що сходяться в єдиний вихід – корінь, і не мають “висячих” відгалужень (див. пункт 3.3.6). Для того, щоб приховати консолідацію виходів у захистити користувача від аналізу оцінки загального обсягу

біткоїна у його власності, необхідно уникати повної консолідації виходів, і зупиняти процес консолідації, як тільки досягнуто певної максимальної прийнятної для користувача кількості виходів у гаманці. Окремі операції консолідації при цьому можуть виглядати як звичайні платежі, послідовність яких досягатиме цілі консолідації за умови, що кожен симульований платіж створює менше виходів, ніж знищує. В рамках цього дослідження розроблено метод багатокрокової консолідації виходів, що складається з транзакцій, які імітують платежі з одним отримувачем [3]. Цей метод консолідації породжує підграф графа транзакцій, який складно розпізнати за допомогою патернового аналізу. Багатокроковий підхід дає змогу досягти цільової допустимої кількості виходів майже без втрати приватності, а його загальні вартість з точки зору комісії збільшуються лише в 2 – 4 рази порівняно з простою операцією консолідації однією транзакцією.

Слід зазначити, що лише деякі Біткоїн-гаманці надають користувачам можливості ручного менеджменту невикористаних виходів. Доступні інструменти контролю виходів зазвичай зводяться до можливостей ручного вибору виходів, заборони використання певних виходів, а також заміни транзакції з підвищенням комісії (англ. RBF – Replace By Fee) [72]. Основною причиною обмеженості подібної функціональності є зручність. Використання Біткоїна вимагає від користувача розуміння багатьох концепцій, пов'язаних з криптографічними ключами і принципами роботи мережі, і необхідність ручного втручання в процес конструювання транзакції для захисту приватності лише ускладнить і без того складний досвід користування. Практичним вирішенням цієї проблеми є надання можливості адаптації автоматизованої стратегії менеджменту невикористаних транзакційних виходів до потреб користувача.

Стратегія менеджменту невикористаних виходів – це частина процесу конструювання програмою-гаманцем нової транзакції, яка відповідає за вибір невикористаних виходів, що будуть знищені цією

транзакцією, та конструювання виходів, що будуть нею створені, зокрема виходу-решти. Стратегія менеджменту невикористаних транзакційних виходів у більшості сучасних гаманців зосереджена на мінімізації комісії за транзакції. З цієї очікуваної раціональної поведінки і випливає багато евристичних припущень, що застосовуються в аналізі графа транзакцій, і більшість таких припущень втрачають актуальність, якщо стратегія менеджменту невикористаних виходів у гаманцях може бути замінена на “ірраціональну”. Простим прикладом такої “ірраціональної” стратегії є випадковий вибір виходів при конструюванні транзакцій, який порушує евристику про оптимальність виходу-решти. Цікавою темою подальших досліджень є розроблення стратегій менеджменту невикористаних виходів, ціллю яких є саме руйнування евристик для захисту даних користувачів, а не мінімізація комісій за транзакції. Для деяких користувачів зменшення ймовірності деанонімізації може бути значно більш важливою ціллю, ніж економія на комісіях.

Слід також зазначити, що методи захисту від евристичного аналізу графа транзакцій, як і багато інших аспектів безпеки у різних децентралізованих технологіях, базуються на принципі “сила в кількості” (лат. *vires in numeris*). Певна технологія, націлена на захист користувача, має бажаний ефект лише якщо користувачів цієї технології достатньо багато, а інакше кожен окремий користувач виділяється з-поміж інших, і факт використання технології стає маркером того, що на цього користувача варто звернути особливу увагу. Гарним прикладом є змішування виходів CoinJoin. Якщо багато користувачів використовують протоколи змішування виходів, то кожен окремий користувач є всього лише користувачем популярної технології. Якщо ж користувачів CoinJoin дуже мало, то факт використання протоколу може бути інтерпретований як ознака того, що окремому користувачу є що приховувати. Аналогічно, “ірраціональність” конструювання транзакцій, ціллю якої є руйнування евристик патернового аналізу графа транзакцій, може мати зворотній ефект, якщо користувачів

такої стратегії менеджменту виходів мало. “Ірраціональна” транзакція може слугувати маркером, що виділяє користувача з-поміж інших.

3.5. Висновки

У цьому розділі запропоновано загальну ідею патернового аналізу графа Біткоїн-транзакцій, що полягає у пошуку у графі знань про транзакції евристичних патернів типових операцій. Сформульовано поняття Біткоїн-операції, і аргументовано доцільність використання евристичних припущень про фінансовий зміст тієї чи іншої операції для встановлення зв'язків між транзакційними виходами, що до неї входять.

Запропоновано метод кластеризації виходів, що полягає у пропагації зв'язків власності одних виходів на інші в межах однієї Біткоїн-операції, спираючись на евристичні припущення про фінансовий зміст такої операції. Цей метод дає змогу заздалегідь формувати групи виходів, що належать одному абстрактному власнику, навіть якщо у графі відсутня будь-яка інформація, що ідентифікує цього власника. Група виходів абстрактного власника буде автоматично деанонімізована у майбутньому, як тільки у графі з'явиться інформація, що ідентифікує цього власника.

Проаналізовано найбільш поширені типові операції, серед яких платежі одному та кільком отримувачам, проста та каскадна консолідація виходів, операції кооперативного змішування виходів і послідовності тимчасових адрес. Продемонстровано, що розглянуті операції утворюють у графі транзакцій чіткі патерни, які легко розпізнати за допомогою стандартних інструментів пошуку патернів у RDF-графах. Наведено деталі реалізації методу пропагації зв'язків власності для кожної розглянутої операції, для якої це можливо.

Окремо обговорено рекомендації та методи протидії патерновому аналізу графа транзакцій, що є важливим аспектом цього дослідження з точки зору індивідуального користувача Біткоїна. З аналізу типових операцій зрозуміло, що багато евристичних припущень щодо

Біткоїн-операцій спираються на припущення про раціональність поведінки програм-гаманців. На основі аналізу операцій простої та каскадної консолідації виходів розроблено метод багатокрокової консолідації, що використовує симулюючі транзакцій, і таким чином уникає утворення очевидного патерну у графі транзакцій. Запропоновано загальний підхід до захисту від патернового аналізу, який полягає у заміні стратегії менеджменту невикористаних виходів у гаманці на таку, що виглядає випадковою або навіть ірраціональною, що ускладнює або унеможлиблює застосування деяких евристик.

4. РОЗРОБЛЕННЯ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ АНАЛІЗУ ГРАФА БІТКОЇН-ТРАНЗАКЦІЙ

4.1. Архітектура програмної системи

Програмна система для аналізу графа Біткоїн-транзакцій, що розробляється у цьому розділі, базується на спеціалізованій моделі даних, розробленій у розділі 2. На основі розробленої моделі формується граф знань про Біткоїн-транзакції – семантичний граф транзакцій, їхніх виходів та власників, доповнений анотаціями власності та іншими довільними фактами про окремі транзакції і виходи у RDF-представленні. Архітектура пропонованої системи значною мірою визначається розробленою моделлю даних.

У пропонованій системі виділено двох акторів – тривіального актора “Адміністратора”, який керує сховищем даних і забезпечує безперешкодне виконання всіх автоматизованих процесів системи, та універсального актора “Дослідника”, який шукає інформацію про окремі виходи і транзакції, за необхідності здійснює модифікацію графа знань в ручному режимі за допомогою SPARQL Update, а також визначає нові евристичні типи операцій для автоматизованої пропагації зв'язків. Основна документація, необхідна “Досліднику” для взаємодії з системою – це формальний опис RDF-моделі (онтології) графа транзакцій, який визначає класи запитів, які можуть бути виконані системою (див. підрозділ 2.3, зокрема рисунок 2.10, а також додаток А). Окрім цього, оскільки пропонована система накопичує знання про Біткоїн-транзакції, які відбуваються у Біткоїн-мережі незалежно від системи, більшість компонентів системи є повністю автоматизованими процесами, вузол Біткоїн-мережі можна розглядати як окремого зовнішнього актора, який породжує нові пакети початкових даних.

На рисунку 4.1 зображено діаграму логічних компонентів розробленої програмної системи побудови і аналізу графа знань про

Біткоїн-транзакції. Система складається з наступних логічних груп компонентів: джерела даних, ETL, власне граф знань, і аналітичні інструменти. Суцільні дуги у діаграмі відображають потоки даних у системі. Логіка процесів ETL і процесів аналізу графа залежить від типів і властивостей, визначених онтологією графа Біткоїн-транзакцій, розробленою у розділі 2. Ця модель (“BRDF-модель” на діаграмі) є керуючим артефактом (контрактом між ETL, сховищем даних та аналітичними інструментами), і тому пунктирні дуги у діаграмі відображають семантичне управління моделі рештою компонентів.

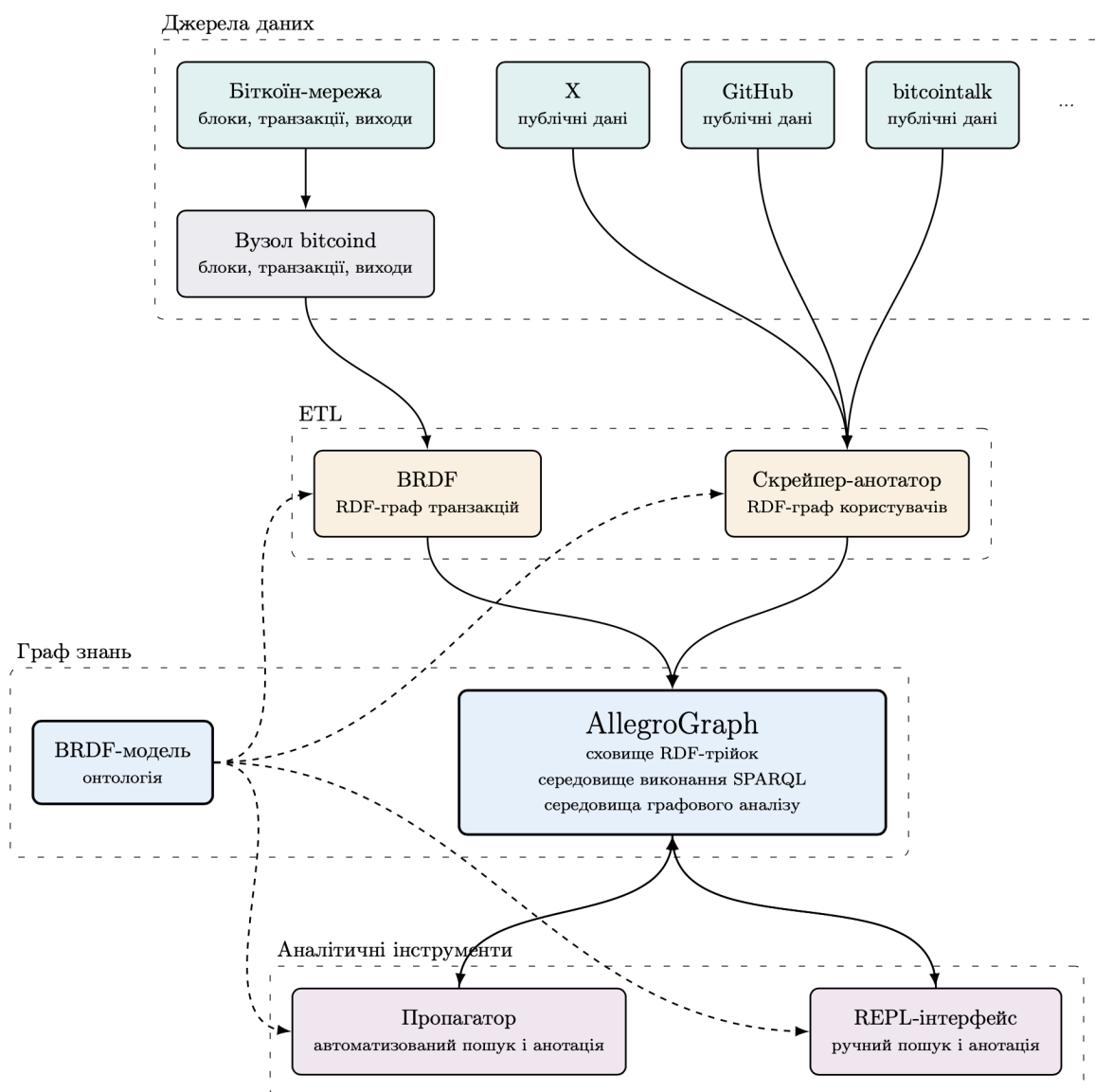


Рис. 4.1. Діаграма компонентів розробленої програмної системи

Логічні групи компонентів системи визначають її розбиття на архітектурні модулі. На рисунку 4.2 зображено модульну архітектуру розроблюваної системи, яка складається з 3-х повністю незалежних компонентів: ETL-системи, сховища RDF-графів та системи аналізу графа. ETL-система складається з автоматизованих процесів, що безперервно збирають інформацію про транзакції і формують RDF-граф транзакцій згідно з розробленою онтологією. Сховище RDF-графів містить RDF-представлення розробленої онтології графа транзакцій, яке формується процесами ETL-системи, а також надає програмні інтерфейси для взаємодії з графом (перегляду, пошуку та модифікації). Саме сховище RDF-графа транзакцій є ядром розроблюваної програмної системи. Система аналізу складається з системи автоматизованої пропagaції зв'язків та інтерфейсів користувача для перегляду, пошуку та модифікації RDF-графа.

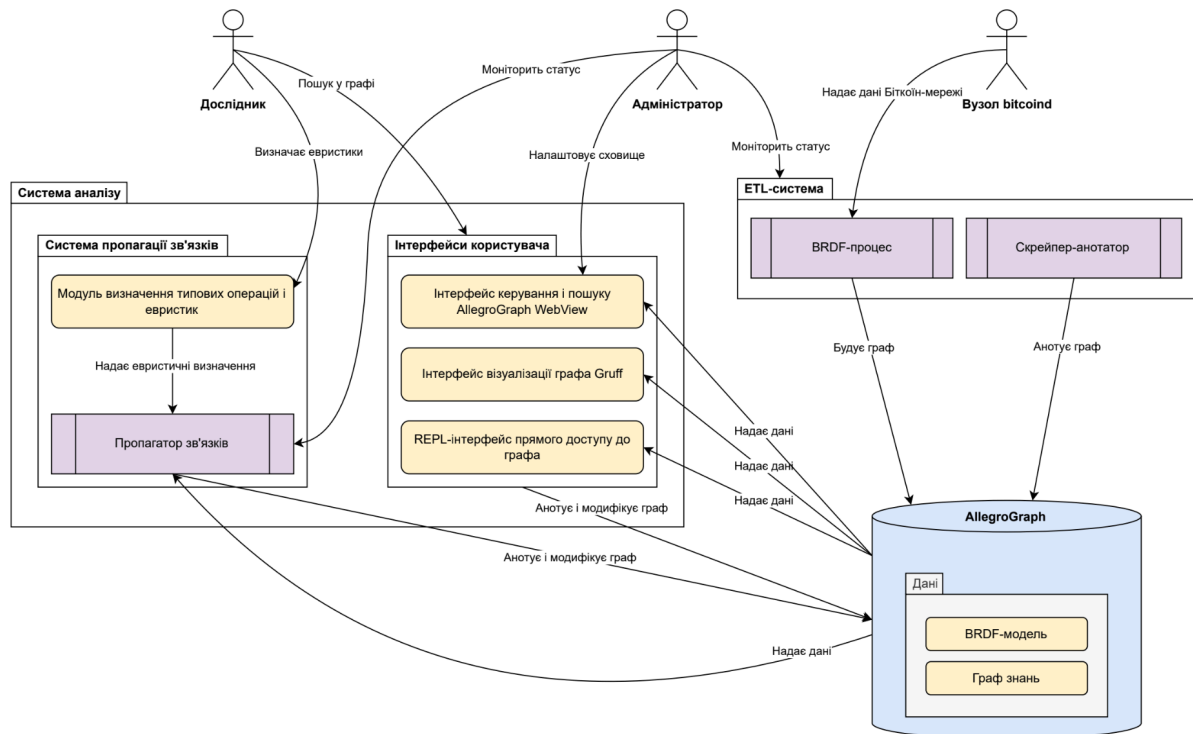


Рис. 4.2. Архітектура розробленої програмної системи

В наступних пунктах розглянуто окремі модулі розроблюваної системи і зазначено виконання вимог до програмного забезпечення, сформульованих у підрозділі 1.4.

4.1.1. ETL-система

Основним першоджерелом даних в системі є Біткоїн-мережа, яка надає інформацію про ланцюг блоків, транзакції у блоках, вміст сховища непідтверджених транзакцій і входи та виходи транзакцій. Оскільки Біткоїн-мережа є одноранговою, найбільш зручним та ефективним способом отримання інформації про транзакції у мережі є налаштування окремого вузла мережі, сервера Bitcoin Core [73] (компонент “Вузол bitcoind” на рисунку 4.1 та відповідний актор на рисунку 4.2), який підтримуватиме локальну копію ланцюга блоків та сховища непідтверджених транзакцій взаємодією з рештою мережі через P2P-протокол [74], і надаватиме блоки та транзакції на запит ETL-процесу через швидкий локальний RPC-інтерфейс [75] (вимога 1.1).

Інформація про асоціації між користувачами та Біткоїн-адресами, а також інформація, що ідентифікує користувачів (наприклад, ім'я користувача у соціальній мережі, тощо), у системі, що розробляється, називається зовнішніми даними. Будь-які сервіси чи платформи, на яких потенційно може розміщуватись інформація про такі асоціації, є джерелами зовнішніх даних для системи, що розробляється. Більшість з таких джерел даних є централізованими сервісами (компоненти “X”, “GitHub” та “bitcointalk” у групі “Джерела даних” на рисунку 4.1), тому взаємодія з ними реалізована через прикладний програмний інтерфейс безпосередньо у ETL-процесі, але деякі з цих джерел є децентралізованими (наприклад, Nostr [76]), і для ефективного збору інформації з таких джерел, як і у випадку Біткоїн-мережі, необхідне налаштування власного вузла для відповідної мережі.

ETL-система відповідає за збирання інформації з джерел даних, конвертацію цих даних у RDF-трійки відповідно до RDF-моделі даних транзакцій, і завантаження отриманих RDF-трійок у сховище RDF-графів (вимога 1.2). Ця система складається з 2-х процесів: процесу завантаження даних ланцюга блоків (компонент “BRDF” на рисунках 4.1 та 4.2; вимога 1.4) і процесу збору зовнішніх даних і анотації графа транзакцій та виходів (компонент “Скрейпер-анотатор” на рисунках 4.1 та 4.2; вимога 1.5).

Процес завантаження даних ланцюга блоків відповідає за побудову ядра графа знань про транзакції – підграфа транзакцій та виходів. Цей процес детально розглядається у пункті 4.3.3, і є найбільш важливим процесом в системі, оскільки від деталей його реалізації залежить повнота даних, ефективність представлення і навіть швидкість пошуку у графі.

Процес збору зовнішніх даних і анотації графа транзакцій та виходів у цьому дослідженні детально не розглядається. Загальні ідеї реалізації збору зовнішніх даних описані ще у ранніх публікаціях щодо аналізу графа транзакцій [33, 35, 36]. В найпростішому варіанті – це автоматизований сканер певного переліку публічних ресурсів (соціальних мереж, форумів, тощо), який безперервно шукає опубліковані ідентифікатори (транзакції та адреси) і додає у граф анотації з посиланнями на ресурси, у яких вони опубліковані.

Як було зазначено вище, з точки зору розроблюваної системи Біткоїн-мережу або вузол, що представляє мережу в межах ETL-системи, можна розглядати як зовнішнього актора (актор “Вузол bitcoind” на рисунку 4.2). Поява нового блока є системним сценарієм, який ініціюється актором “Вузол bitcoind” і є тригером для чергових ітерацій BRDF-трансформації і пропагації зв'язків. Оскільки вузол мережі працює за визначеним протоколом, окремий процес “BRDF-планувальник” в межах загального BRDF-процесу ініціює цей сценарій, надсилаючи запит на вузол щодо наявності нових блоків. Процес пропагації зв'язків отримує інформацію про зміни у графі з сховища RDF-графів, і в свою чергу ініціює

обмін даними з сховищем для встановлення нових зв'язків. Перед тим, як пропагатор зможе здійснювати ітерації пропагації, актор “Дослідник” повинен зареєструвати набір евристичних визначень типових операцій. На рисунку 4.3 зображено діаграму активності системи відносно системного сценарію появи нового блока.

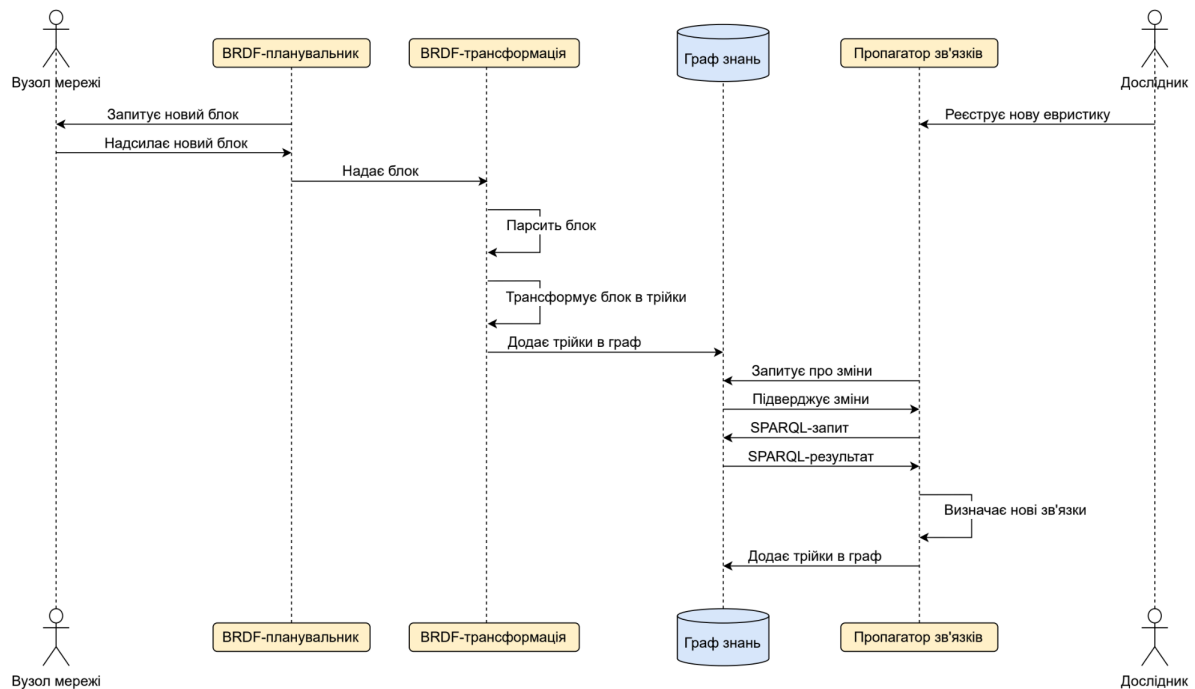


Рис. 4.3. Діаграма активності програмної системи

4.1.2. Сховище RDF-графів

Як було зазначено у розділі 2, саме граф знань є ключовим компонентом розробленої програмної системи, а використання RDF-моделі даних є її фундаментальною відмінністю від систем, описаних у відомих публікаціях [33, 35 – 37]. Унікальність такого підходу полягає у тому, що RDF-граф знань про транзакції містить як граф транзакцій та виходів, побудований з даних ланцюга блоків (див. пункт 2.1.3), так і довільні сутності і зв'язки, що слугують анотаціями вершин графа транзакцій та виходів (див. пункт 2.3.4). При цьому такі довільні анотації одразу стають доступними у запитах пошуку патернів у графі знань завдяки

уніфікованому представленню всіх без винятку вершин та зв'язків у вигляді тверджень (RDF-трійок). Окрім цього, в екосистемі RDF існують інструменти, які можуть працювати з графами знань дуже великих розмірів (10 – 20 мільярдів RDF-трійок), що дає змогу будувати граф знань про всі транзакції від початку роботи Біткоїн-мережі. Стандартна мова SPARQL дає змогу ефективно шукати патерни навіть у таких великих графах, а у випадках, коли ефективність стандартних SPARQL-запитів є недостатньою, деякі сховища RDF-трійок надають можливість реалізації спеціалізованих фільтр-функцій – розширень стандартної мови SPARQL, що мають прямий доступ до індексів RDF-трійок графа і можуть значно пришвидшувати деякі обчислення під час пошуку патернів.

Отже, сховище RDF-трійок для побудови графа знань про Біткоїн-транзакції повинне підтримувати зберігання великих графів (до 20 мільярдів RDF-трійок; вимога 4.1), надавати ефективний SPARQL-інтерфейс для пошуку патернів у великих графах і можливості розширення SPARQL-інтерфейсу спеціалізованими фільтр-функціями (вимоги 1.3, 1.6, 2.2). Для здійснення складних досліджень графа та реалізації необхідних графових алгоритмів, сховище також повинне надавати прикладний програмний інтерфейс для прямого доступу до вершин та дуг графа.

Виходячи з перелічених вимог, в рамках цього дослідження вирішено використовувати сховище RDF-трійок AllegroGraph компанії Franz Inc [77]. Це сховище підтримує репозиторії з десятками мільярдів RDF-трійок і надає обширний прикладний програмний інтерфейс для доступу до графа з середовища програмування Common Lisp. Унікальною особливістю сховища AllegroGraph є те, що сервер сховища містить компілятор мови Common Lisp, що дає змогу створювати спеціальні фільтр-функції, описані цією мовою, які при завантаженні на сервер AllegroGraph автоматично компілюються у машинний код і їхня швидкодія не відрізняється від вбудованих стандартних фільтр-функцій.

4.1.3. Система аналізу графа

Система аналізу дає змогу здійснювати автоматизовані чи ручні дослідження графа, результатами яких можуть бути як таблиці даних і статистики, так і множини нових сутностей та зв'язків у графі, які стають частиною графа і впливають на подальші дослідження.

Перший підсистема системи аналізу графа – це система автоматизованої кластеризації виходів з використанням методу пропагації зв'язків власності, розробленого у підрозділі 3.2, на основі евристичних патернів типових Біткоїн-операцій з переліку, наведеного у підрозділі 3.3.

Друга підсистема – це набір інтерфейсів інтерактивної взаємодії з графом знань про транзакції, зокрема SPARQL-інтерфейс для пошуку патернів у графі, інтерфейс візуалізації підграфів графа транзакцій, та програмний інтерфейс прямого доступу до RDF-графа з середовища Common Lisp.

AllegroGraph надає прикладні програмні інтерфейси для взаємодії з графом з Python [78] та Java [79] (вимога 2.1). Обидві ці бібліотеки використовують HTTP-протокол для комунікації з сервером, і є достатньо швидкими для більшості звичайних операцій на зразок виконання SPARQL-запитів та імпорту та експорту даних, але є повільними, якщо потрібний прямий доступ до графа. Якщо для збільшення швидкості виконання певних операцій чи аналізів необхідна можливість для однієї вершини в графі швидко отримати множину її сусідів за певним типом зв'язку, то кожен такий запит на отримання множини сусідніх вершин здійснюватиме HTTP-комунікацію, яка має великі накладні витрати. Для задач, що потребують максимально можливої ефективності доступу до графа, AllegroGraph надає програмний інтерфейс для програмного середовища Allegro Common Lisp [80], що дає змогу прямо взаємодіяти з сховищем RDF-трійок через регіон спільної пам'яті [81].

Інтерактивна система взаємодії з графом складається з текстового редактора Emacs [82] та популярного модуля SLIME/SLY [83, 84], який є

обгорткою для REPL-середовища Common Lisp. Для використання цієї інтерактивної системи користувачу необхідно мати поверхневі знання мови Common Lisp. Перевагою використання тандему Emacs та SLY є те, що, на відміну від графічного середовища для виконання SPARQL-запитів, клієнт AllegroGraph для середовища Common Lisp надає і SPARQL-інтерфейс і набір функцій прикладного програмного інтерфейсу прямої взаємодії з графом, і дає змогу використовувати об'єкти внутрішнього представлення RDF-трійок, які повертають SPARQL-запити, як аргументи у цих функціях. Окрім того, клієнт AllegroGraph для Common Lisp дає змогу створювати спеціальні фільтр-функції без необхідності зупинки і перезапуску сервера.

В рамках цього дослідження також розроблено розширення для AllegroGraph, яке дає змогу швидко візуалізувати знайдений підграф на екрані безпосередньо з REPL-середовища Common Lisp. Подібна функціональність присутня у вбудованому в AllegroGraph середовищі візуального розроблення запитів Gruff [85], але є занадто громіздким для швидких експериментів. Для цього реалізовано серіалізацію результатів SPARQL-запитів у популярному форматі текстового опису графів Graphviz [86]. Візуалізація результатів SPARQL-запиту здійснюється виведенням результату у цьому форматі у програму генерації зображення з текстового представлення “dot” [87], яка постачається разом з програмним пакетом Graphviz, і подальшого перенаправлення бінарних даних згенерованого зображення у програму візуалізації “display” програмного пакету ImageMagick [88]. Деталі реалізації розширення наведені у додатку В, а приклади візуалізації результатів SPARQL-запитів типу “CONSTRUCT”, що знаходять і реконструюють підграфи транзакцій – у додатку Г.

4.2. Процес побудови графа знань про транзакції

4.2.1. Аналіз *RDF*-моделі транзакційних даних

Оцінимо розмір *RDF*-графа (кількість *RDF*-трійок) всієї історії транзакцій, побудованого відповідно до моделі, розробленої у підрозділі 2.3. Для спрощення оцінки вважатимемо, що кожен вихід має зв'язки з двома транзакціями (як власне вихід, і як вхід), і це дасть нам змогу використати фіксовані значення кількості трійок на один об'єкт. У найгіршому з точки зору кількості необхідних ресурсів випадку, коли кожен вихід має асоційованого власника, кількість трійок, необхідна для опису одного виходу, $S_O = 6$:

1. Декларація типу (“*rdf:type*”).
2. Кількість біткоїна (“*brdf:outputAmount*”).
3. Адреса або програма блокування (“*brdf:outputAddress*”, “*brdf:outputScript*”).
4. Зв'язок з транзакцією, що створила цей вихід (“*brdf:txOutput*”).
5. Зв'язок з транзакцією, що вже використала або використає цей вихід у майбутньому (“*brdf:outputInputTx*”).
6. Власник “*brdf:outputOwner*”.

Кількість трійок, необхідна для опису однієї транзакції, $S_T = 4$:

1. Декларація типу (“*rdf:type*”).
2. Кількість входів (“*brdf:txInputCount*”).
3. Кількість виходів (“*brdf:txOutputCount*”).
4. Зв'язок з блоком (“*brdf:blockTx*”).

Кількість трійок, необхідна для опису одного блока, $S_B = 2$:

1. Декларація типу (“*rdf:type*”).
2. Мітка часу блока (“*brdf:blockTimestamp*”).

Загальну кількість трійок, необхідну для представлення графа всієї історії транзакцій згідно з моделлю, розробленою у підрозділі 2.3:

$$S = S_O N_O + S_T N_T + S_B N_B,$$

де N_B – кількість блоків, N_T – кількість транзакцій, а N_O – кількість виходів, створених за весь час існування мережі. Значення N_B та N_T можна отримати безпосередньо з вузла Біткоїн-мережі, оскільки вони входять до переліку статистичних даних, які збираються вузлом в процесі синхронізації з мережею. Обчислення кількості виходів N_O є нескладним, але тривалим, оскільки потребує обходу всіх транзакцій у ланцюгу, тому використано приблизну середню кількість виходів у транзакції. Підсумовуючи, можемо оцінити кількість RDF-трійок, необхідних для представлення всієї історії транзакцій станом на вересень 2025-го року:

$$S_{2025.09} = 6 \cdot 3.0 \cdot 10^9 + 4 \cdot 1.2 \cdot 10^9 + 2 \cdot 9 \cdot 10^5 \approx 22.8 \cdot 10^9.$$

Сховище RDF-трійок AllegroGraph представляє одну трійку 56-ма октетами, тому розмір сховища такого графа на диску становить 1.19 Тб на кожен індекс.

4.2.2. Метод зменшення розміру RDF-графа транзакцій

У розробленій RDF-моделі для кожного типу сутності існує зв'язок, який є унікальним лише для цього типу сутностей, тому до цієї моделі можна застосувати метод зменшення розміру RDF-графа за допомогою непрямої типізації [5]. Основна ідея цього методу полягає у тому, що для сутностей, тип яких зрозумілий з наявності певного характеризуючого зв'язку, декларація типу є надлишковою, і якщо таких сутностей у графі багато, то видалення трійок, що декларують їх тип може значно зменшити розмір графа, а пошук сутностей за типом можна вручну або автоматично перетворювати у пошук сутностей за наявністю відповідного зв'язку.

Характеризуючий зв'язок потрібно обирати таким, щоб для кожної сутності певного типу існувала лише одна RDF-трійка з відповідним предикатом. Тоді запит, що знаходить всі сутності з таким зв'язком (запит 1 у лістингу 4.1) буде еквівалентним запиту пошуку всіх сутностей певного

типу (запит 2 у лістингу 4.1) і не потребуватиме використання дедублікації (“DISTINCT”) при обчисленні кількостей сутностей певних типів у графі.

Лістинг 4.1. Запити з прямою і непрямою типізацією

```
#1
SELECT ?e { ?e <property> ?t }

#2
SELECT ?e { ?e rdf:type <Type> }

#3
SELECT (COUNT(?output) AS ?outputCount) {
    ?tx brdf:txOutput ?output.
}

#4
SELECT (COUNT(?output) AS ?outputCount) {
    ?output rdf:type brdf:Output.
}
```

У розробленій моделі даних транзакцій будь-яка властивість однозначно ідентифікує типи об'єктів, які вона пов'язує, тому використання характеризуючих властивостей потрібне лише для підрахунку кількостей об'єктів різних типів (виходів, транзакцій та блоків) під час збору статистики про стан графа у даний момент. Наприклад, областю визначення відношення “brdf:txOutput” є множина транзакцій, а областю значень – множина виходів, і для кожного виходу існує лише один такий зв'язок. Тому пошук всіх сутностей типу “Вихід” є еквівалентним пошуку всіх сутностей, які є об'єктами трійок з предикатом “brdf:txOutput”. А отже декларацію типу для кожного виходу можна опустити, а пошук сутностей за типом “brdf:Output” автоматично перетворювати у пошук сутностей за наявністю зв'язку “brdf:txOutput”. Модифікований запит для обчислення кількості виходів у кінцевому графі (запит 3 у лістингу 4.1) є еквівалентним стандартному запиту (запит 4 у лістингу 4.1).

Натомість, властивість “brdf:outputInputTx” не може бути характеристичною для типу “brdf:Output”, тому що для невикористаних виходів такого зв'язку за визначенням не існує, а властивість “brdf:txOutput” не може бути характеристичною для типу “brdf:Transaction”, тому що більшість транзакцій мають більше одного такого зв'язку. Властивість “brdf:blockTx” не може бути характеристичною для типу “brdf:Transaction”, тому що непідтверджені транзакції не мають асоціації з блоком (див. пункт 4.3.3). У таблиці 4.1 наведено характеристичні властивості для кожного з типів розробленої моделі.

Таблиця 4.1

Характеристичні властивості типів RDF-моделі даних транзакцій

Тип	Властивість
brdf:Output	brdf:outputAmount brdf:txOutput
brdf:Transaction	brdf:transactionInputCount brdf:transactionOutputCount
brdf:Block	brdf:blockTimestamp

Застосувавши цей принцип до усіх типів у розробленій моделі, ми зменшуємо кількість трійок на представлення кожної сутності на 1: $S_O = 5$, $S_T = 3$, $S_B = 1$. Нова оцінка розміру RDF-графа

$$S_{2025.09} = 5 \cdot 3.0 \cdot 10^9 + 3 \cdot 1.2 \cdot 10^9 + 9 \cdot 10^5 \approx 18.6 \cdot 10^9,$$

а розмір сховища на диску – 0.97 Тб на один індекс, тобто таке перетворення зменшує кількість необхідного простору на диску на 18.5%. Зменшення розміру графа впливає також і на швидкість запитів, оскільки пошук у індексі меншого розміру є швидшим.

4.2.3. Завантаження даних ланцюга блоків

Для того, щоб система аналізу графа знань про Біткоїн-транзакції стала придатною для здійснення аналізів чи встановлення зв'язків власності, необхідно побудувати RDF-граф транзакцій. Тільки після цього в системі з'являється інформація про зв'язки між транзакціями і стає можливим пошук патернів у графі. RDF-граф транзакцій – це підграф графа знань про транзакції, розробленого у підрозділі 2.3, який відображає інформацію з ланцюга блоків Біткоїна, і не містить анотацій власності та інших зовнішніх даних про транзакції. Цей підграф складається з вершин-транзакцій, вершин-виходів, і зв'язків між ними. Також до цього підграфа входять вершини-блоки, які є механізмом співставлення транзакцій з приблизним, але закріпленим правилами консенсусу, часом, коли вони відбулись. Такий RDF-граф будується ітерацією вздовж ланцюга блоків і додавання у початково порожнє сховище RDF-трійок, що описують кожен блок, всі транзакції у ньому і всі входи та виходи цих транзакцій.

Основною складністю початкової побудови RDF-графа транзакцій є те, що ланцюг блоків Біткоїна станом на вересень 2025-го року містить більше 900000 блоків, і послідовна конвертація їх у RDF-представлення займає багато часу. Окрім цього, під час дослідження у розроблену RDF-модель вносились і продовжують вноситись зміни, тому час від часу виникала потреба перебудувати RDF-граф транзакцій з початкового стану. Для того, щоб початкове завантаження даних у граф відбувалось швидко, було розроблено окремий програмний інструмент ETL (Extract, Transform, Load) під назвою “BRDF” (скор. англ. Bitcoin RDF) який дає змогу завантажувати ланцюг блоків Біткоїна у сховище RDF-графів AllegroGraph, і підтримує контрольований паралелізм та безпечне відновлення після зупинки чи помилки.

Інструмент завантаження ланцюга блоків Біткоїна використовує два типи процесів-працівників, процес-планувальник, який існує в єдиному екземплярі, і набір процесів-завантажувачів, яких може бути декілька,

залежно від наявних обчислювальних ресурсів та бажаної швидкості завантаження.

Процес-планувальник відповідає за координацію процесу завантаження блоків. Він створює підключення до сховища RDF-графів AllegroGraph і до вузла Біткоїн-мережі, співставляє перелік блоків, присутніх у RDF-графі з повним переліком блоків, наданим вузлом Біткоїн-мережі і формує чергу блоків до завантаження. Якщо процес завантаження запущено без верхнього обмеження на висоту блока, процес-планувальник відповідає також за періодичне опитування вузла Біткоїн-мережі про появу нових блоків і додавання їх у чергу, якщо вони з'являються. Блоки у черзі представляються цілими числами, що відповідають висоті блока у ланцюгу. Як наслідок, кожен процес-завантажувач змушений підтримувати власне підключення до вузла Біткоїн-мережі, але це дає змогу відкласти читання блока та побудову його об'єктної моделі аж до моменту, коли вона буде потрібна. Окрім цього, процес-планувальник звітує про статус завантаження, виводячи на екран інформацію про кількість блоків та транзакцій у RDF-графі і у Біткоїн-мережі для порівняння.

Процес-завантажувач відповідає за трансформацію об'єктної моделі блока у RDF-трійки відповідно до моделі з підрозділу 2.3 та додавання цих трійок у сховище трійок AllegroGraph. Для цього процес-завантажувач ініціює підключення до сховища RDF-графів AllegroGraph і до вузла Біткоїн-мережі, отримує з черги, сформованої процесом-планувальником наступний блок, який необхідно завантажити, будує об'єктну модель цього блока, формує відповідні RDF-трійки, що описують вершини та зв'язки графа (блок, його транзакції та їхні виходи) відповідно до моделі з підрозділу 2.3, і додає повний набір RDF-трійок для даного блока атомарною операцією додавання RDF-трійок в AllegroGraph. Таким чином, якщо під час обробки блока стається якась помилка, жодна з RDF-трійок блока не попаде у граф окремо. Або всі RDF-трійки, що описують блок,

будуть додані у граф єдиною операцією, або жодна з них не буде додана. Завдяки цьому при відновленні завантаження після зупинки чи помилки процесу-планувальнику достатньо додати у чергу всі блоки, вершини яких відсутні у графі.

Оскільки кожна транзакція, що публікується у мережі, містить інформацію про зв'язки між виходами, яка може бути корисною у графі знань, то є сенс також додавати у граф транзакції зі сховища непідтверджених транзакцій. Коли підмножина непідтверджених транзакцій потрапляє у черговий блок, достатньо додати у граф лише ті транзакції з нього, які були відсутні у сховищі непідтверджених транзакцій, а ті транзакції з нього, які вже присутні у графі як непідтверджені, не потрібно видаляти, достатньо лише проставити для них відповідний зв'язок “блок – транзакція”. Цей механізм є схожим за принципом на механізм компактного поширення блоків (англ. Compact Block Relay) [89]. Додавання у граф непідтверджених транзакцій дає змогу відображати події, які не є видимими у ланцюгу блоків, наприклад, заміну транзакцій з підвищенням комісії (RBF). Так, якщо початкова транзакція і серія транзакцій, що її замінюють, використовують різні множини входів, то евристику про єдиного власника входів транзакції можна застосувати до об'єднання цих множин, а не лише до входів кінцевої транзакції. Після пропагації цих зв'язків всі замінені транзакції можна видаляти з метою мінімізації розміру графа.

Особливу увагу потрібно приділити подіям реорганізації ланцюга (вимога 3.2), оскільки неправильна обробка реорганізацій під час побудови графа може призвести до значних порушень достовірності інформації в системі. Обробка реорганізації відбувається наступним чином:

1. Процес-планувальник додає у чергу всі блоки нового ланцюга, починаючи з точки розгалуження і до кінця ланцюга.
2. Якщо процес-завантажувач отримує з черги блок, який вже присутній у графі, він видаляє існуючий блок-сироту (англ. orphan

block) і всі його зв'язки “блок – транзакція”, і додає новий блок, уникаючи дублювання транзакцій.

3. Всі транзакції блока-сироти, що відсутні у новому блоці, стають непідтвердженими транзакціями, оскільки для них немає зв'язку “блок – транзакція”.
4. Якщо якась з утворених непідтверджених транзакцій використовує вже використаний вихід, вважаємо, що відбулась заміна транзакції, виконуємо пропагацію зв'язків для заміненних транзакцій, описану вище, і видаляємо замінені транзакції з графа.

Наявність зв'язку “блок – транзакція” є зручним маркером підтвердженості транзакції у графі. Пошук всіх непідтверджених транзакцій є швидкою операцією фільтрування за наявністю зв'язку, і завдяки цьому можна швидко видалити з графа всі непідтверджені транзакції, якщо їх накопичилось занадто багато.

Таблиця 4.2

Швидкість завантаження 100 блоків

Кількість завантажувачів N	Час завантаження 100 блоків, с
1	142.173
2	79.668
4	58.739
8	62.131

У таблиці 4.2 наведено вимірювання швидкості завантаження 100 блоків проміжку 840000 – 840099 для різних значень N кількості процесів-завантажувачів (вимога 4.3). Оскільки RDF-трійки є самодостатніми елементами даних, RDF-опис одного блоку є повністю незалежним від RDF-опису будь-якого іншого блоку, навіть якщо вони містять пов'язані транзакції, тому задача побудови RDF-графа ланцюга

блоків є приголомшливо паралельною (англ. *embarrassingly parallel*). Зменшення швидкості 8 завантажувач і лінійне збільшення швидкості завантаження зі збільшенням кількості процесів-завантажувачів є очікуваним.

Блоки 840000 – 840099 створені в період найвищого навантаження на Біткоїн-мережу за весь час її існування, тому консервативна оцінка часу завантаження повного ланцюга блоків (вимога 4.2) становить

$$(900000 / 100) \cdot 58.7 \text{ с} \approx 53000 \text{ с} \approx 150 \text{ год}$$

Програмний інструмент “BRDF” розміщений у відкритому доступі на сервісі колаборативного розроблення програмного забезпечення GitHub [90], і використовує розроблену бібліотеку для взаємодії з Біткоїн-мережею для програмного середовища Allegro Common Lisp [80] (див. підрозділ 4.4). Програмний код інструменту наведено у додатку Г.

4.3. Бібліотека для взаємодії з Біткоїн-мережею

Як було зазначено у пункті 4.2.4, найбільш ефективним клієнтом для взаємодії з сховищем RDF-трійок AllegroGraph є клієнт для програмного середовища Common Lisp, тому більшість програмних компонентів, розроблених в рамках цього дослідження, розроблені саме у цьому програмному середовищі. У момент зародження основних ідей цього дослідження не існувало якісних прикладних програмних інтерфейсів для взаємодії з Біткоїн-мережею для середовищ Common Lisp, тому було прийнято рішення розробити такий інтерфейс. Розроблення відкритих бібліотек загального застосування для взаємодії з різноманітними відкритими системами, такими, як Біткоїн-мережа, є важливою сферою інженерної діяльності.

Розроблена бібліотека має назву “bp” (англ. BP – Bitcoin Protocol) і розміщена у відкритому доступі на сервісі GitHub [91], а також

обговорювалась на науковій конференції [6]. Вона складається з декількох модулів, фрагменти програмного коду яких наведені у додатку Д.

4.3.1. Модуль-ядро

Модуль “bp.core” [92], який також має короткий ідентифікатор, що співпадає з назвою бібліотеки (“bp”), надає визначення класів об’єктної моделі транзакцій та блоків, абстрактні інтерфейси для отримання транзакцій та блоків з мережі, серіалізація та десеріалізації транзакцій та блоків з бінарного представлення, що використовується у Біткоїн-мережі, а також рудиментарна реалізації правил Біткоїн-консенсусу. Окрім цього, тут реалізовані додаткові інструменти на зразок покрокового виконання програм блокування та розблокування.

Основна функціональність, необхідна для реалізації процесу побудови графа знань про транзакції, – це інтерфейс “bp:chain-supplier” та клас “bp.rpc:node-rpc-connection”, що надає цей інтерфейс, зображені на рисунку 4.4.

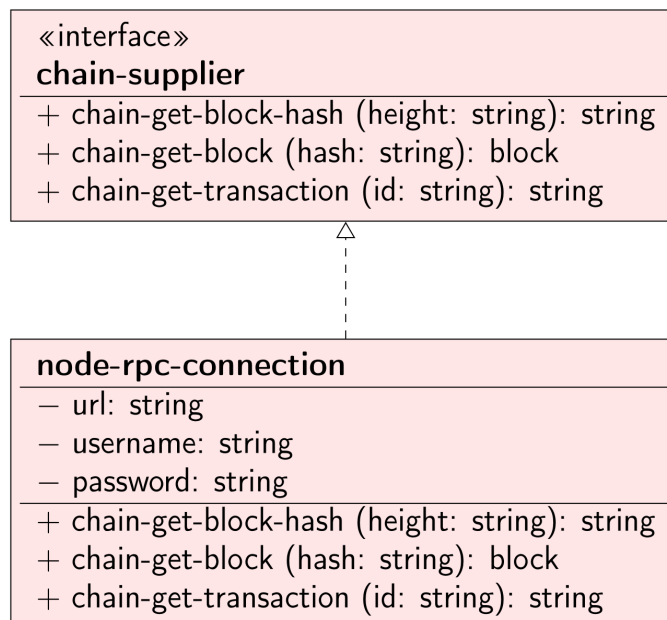


Рис. 4.4. Інтерфейс постачальника блоків та його реалізація

Метод “bp:chain-get-block-hash” повертає ідентифікатор (хеш) блока за його позицією (висотою) у ланцюгу. Метод “bp:chain-get-block” повертає об’єкт “bp:cblock”. Як описано у пункті 4.3.3, потоки-завантажувачі отримують з черги невідомих блоків висоту блока, і кожен потік може за допомогою цих двох методів за висотою блока отримати об’єкт блока, якого достатньо, щоб додати всі транзакції та їхні входи і виходи у RDF-граф транзакцій.

4.3.2. Модуль криптографічних інструментів

Модуль “bp.crypto” [93] містить повний набір криптографічних інструментів, які використовуються у Біткоїн-протоколі. Зокрема, цей модуль містить процедури для обчислення хеш-функцій, що використовуються в усіх інших модулях бібліотеки. Найважливішим компонентом тут є самостійний підмодуль “bp.crypto.secp256k1” – інтерфейс зовнішніх функцій (англ. Foreign Function Interface) для криптографічної бібліотеки libsecp256k1 [94], яка використовує еліптичну криву secp256k1 і вважається еталоном інженерії бібліотек асиметричної криптографії. Через високу швидкість, безпеку і якість коду ця бібліотека поступово починає використовуватись і за межами Біткоїн-мережі, зокрема у проєкті Nostr [76] – децентралізованому протоколі обміну повідомленнями для розподілених соціальних мереж, а також у багатьох інших криптовалютних проєктах, тому цей підмодуль може бути корисним в якості самостійної криптографічної бібліотеки загального застосування у екосистемі Common Lisp.

4.3.3. Модуль RPC-інтерфейсу

Модуль “bp.rpc” [95] – реалізація повного переліку методів JSON-RPC-інтерфейсу вузла Біткоїн-мережі Bitcoin Core. Основним компонентом цього модуля є клас “bp.rpc:node-rpc-connection”, об’єкт якого реалізує інтерфейс “bp.core:chain-supplier” і надає блоки та транзакції, які

отримує через виклики відповідних RPC-методів вузла Bitcoin Core, до якого цей об'єкт має доступ.

4.3.4. Модуль однорангової мережі

Модуль “bp.net” [96] – рудиментарна реалізація протоколу прямої взаємодії з вузлами Біткоїн-мережі. Основним компонентом цього модуля є клас “bp.net:simple-node”, який реалізує інтерфейс “bp.core:chain-supplier”, і надає блоки та транзакції безпосередньо з однорангової мережі Біткоїн, без необхідності RPC-доступу до окремого вузла мережі.

4.3.5. Тестування

Частиною бібліотеки є модуль “bp.tests” [97] – система модульного тестування з налаштуваннями автоматичного запуску через систему безперервної інтеграції та розгортання GitHub Actions [98].

Таблиця 4.3

Порівняння швидкості читання і десеріалізації блоків

Операція, 100 повторень	Час виконання, с	
	bp	python-bitcoinlib
Десеріалізація блока	7.501	10.184
Читання блока через JSON-RPC	49.906	37.013

У таблиці 4.2 наводиться порівняння швидкості виконання двох основних операцій, які використовуються при перетворенні даних ланцюга блоків у RDF-граф: десеріалізації бінарного представлення блока у шістнадцятковому кодуванні та читання блока через JSON-RPC-інтерфейс. Порівнюється розроблена бібліотека для програмного середовища Common Lisp та популярна аналогічна бібліотека для середовища Python – python-bitcoinlib [99]. Для підвищення наглядності порівняння, обидві

операції виконуються для однакового набору зі 100 блоків. Як видно з таблиці, операція десеріалізації у розробленій бібліотеці є швидшою за аналогічну операцію у бібліотеці `python-bitcoinlib`, але операція читання блока через JSON-RPC-інтерфейс є повільнішою. Операція читання блока здійснює обмін HTTP-повідомленнями, отримує серіалізований блок і виконує операцію десеріалізації. Тому можна зробити висновок, що HTTP-бібліотека, що використовується у `python-bitcoinlib`, і значно швидшою за HTTP-бібліотеку, яка була обрана для розробленої бібліотеки (Portable AllegroServe [100]). У майбутньому планується заміна HTTP-бібліотеки на більш швидку.

4.4. Висновки

У цьому розділі розглядається програмна система для побудови графа знань про Біткоїн-транзакції, подальшого розширення цього графа, та його дослідження шляхом пошуку у ньому патернів типових операцій та пов'язаних з ними користувачів.

Розроблено архітектуру програмної системи, що складається з ETL-модуля завантаження даних ланцюга блоків і зовнішньої інформації, RDF-графа знань про транзакції, що зберігається у сховищі AllegroGraph, та аналітичних інструментів автоматизованої пропагації зв'язків власності на основі патернів типових операцій і інтерактивного дослідження графа знань за допомогою SPARQL-запитів, візуалізації результатів цих запитів та функцій прямого обходу графа. Дані ланцюга блоків отримують процесами ETL-модуля з локального вузла Біткоїн-мережі, а зовнішні асоціації між користувачами та Біткоїн-адресами і інші анотації – з публічно доступної інформації на зовнішніх сервісах.

Проаналізовано розроблену RDF-модель транзакційних даних з точки зору розміру відповідного RDF-графа на диску і запропоновано метод зменшення розміру графа шляхом видалення явних оголошень типів вершин і застосування непрямой типізації через характеристичні

властивості. Також здійснено оцінку розміру RDF-графа знань про всю історію транзакцій і часу побудови такого графа. Розроблено ETL-інструмент для паралельного завантаження блоків, завантаження непідтверджених транзакцій і обробки реорганізацій ланцюга.

В рамках дослідження створена бібліотека взаємодії з мережею Біткоїн для програмного середовища Common Lisp, яка розміщена у відкритому доступі на сервісі колаборативного розроблення програмного забезпечення GitHub під ліцензією MIT.

ВИСНОВКИ

У дисертаційній роботі вирішено актуальну науково-технічну задачу спрощення процесів проектування, розроблення та експлуатації програмного забезпечення для аналізу графа Біткоїн-транзакцій шляхом розроблення узагальненої архітектури програмної системи на основі спеціалізованої моделі транзакційних даних, яка надає уніфікований семантичний контракт для інтеграції програмних компонентів і забезпечує масштабоване оброблення історії транзакцій. Окрім цього, зроблено висновки щодо удосконалення процесів конструювання та публікації Біткоїн-транзакцій.

Вирішено всі задачі, поставлені на початку дослідження, і отримано наступні результати:

1. Проаналізовано відомі методи і програмне забезпечення для аналізу графа Біткоїн-транзакцій, транзакційну модель Біткоїна і проблему приватності користувачів в умовах відкритості реєстру транзакцій у Біткоїні.
2. Сформульовано загальну концепцію дослідження графа Біткоїн-транзакцій шляхом накопичення інформації про транзакції у вигляді графа знань у централізованому сховищі і пошуку патернів у такому графі з метою подальшого встановлення нових зв'язків у ньому.
3. Сформульовано вимоги до програмної системи для побудови, зберігання, доповнення та дослідження графа Біткоїн-транзакцій.
4. Розроблено спеціалізовану модель даних Біткоїн-транзакцій, яка об'єднує транзакції, виходи транзакцій, власників виходів та довільні зовнішні анотації у єдину онтологію. Розроблена модель надає уніфікований семантичний контракт для інтеграції компонентів програмної системи для аналізу графа транзакцій, дає змогу зберігати граф усієї історії транзакцій в межах єдиного

сховища даних, а також спрощує запити для пошуку типових операцій.

5. Запропоновано метод кластеризації виходів Біткоїн-транзакцій за допомогою пошуку евристичних патернів типових операцій у графі знань про Біткоїн-транзакції. Цей метод відрізняється від відомих методів тим, що використовує спеціалізований RDF-граф знань про Біткоїн-транзакції, який відображає всю історію транзакцій та довільні анотації, зокрема результати попередніх аналізів. Це дає змогу ефективно знаходити всі підграфи графа транзакцій, що відповідають певному типу операції, та пропагувати зв'язки власності з одних виходів таких підграфів на інші згідно з евристичним визначенням змісту операції. Процес кластеризації може бути розбитий на окремі кроки і здійснений інкрементально зі збереженням проміжних результатів безпосередньо у графі знань.
6. Сформульовано концепцію ірраціональної стратегії менеджменту транзакційних виходів у програмному забезпеченні для конструювання Біткоїн-транзакцій (Біткоїн-гаманцях). Запропоновано метод захисту від патернового аналізу операцій консолідації виходів транзакцій за допомогою багатокрокового процесу, який симулює платежі з одним отримувачем і створює у графі транзакцій складний підграф, стійкий до розпізнавання. Це дає користувачу змогу знижувати ризики розкриття чутливої персональної фінансової інформації.
7. Розроблено ETL-інструмент для завантаження даних ланцюга блоків Біткоїна у сховище RDF-трійок для побудови графа знань про транзакції, що підтримує завантаження непідтверджених транзакцій і обробку реорганізацій ланцюга. Завдяки самодостатності одиниці представлення даних, що використовується (RDF-трійки), цей інструмент масштабується

через довільну кількість процесів-завантажувачів, що отримують блоки для завантаження з центрально планованої черги, і дає змогу здійснити початкову побудову графа всієї історії транзакцій за 150 годин. Запропоновано метод зменшення розміру RDF-графа шляхом видалення RDF-трійок явних оголошень типів сутностей і заміни явної типізації на неявну через характеризуючі властивості. Оцінено розмір графа знань про всю історію транзакції до і після перетворення і встановлено, що таке перетворення дає змогу зменшити розмір RDF-графа, який розглядається у цьому дослідженні, на 18.5%. Оцінка місця на диску, необхідного для зберігання всієї історії транзакцій у вигляді графа знань після перетворення – 0.97 Тб на один індекс, що узгоджується з вимогою зберігання графа в межах однієї машини. Розроблено бібліотеку для взаємодії з Біткоїн-мережею з програмного середовища Common Lisp, що складається з модуля-ядра, який реалізує частини Біткоїн-протоколу, модуля криптографічних інструментів, а також модулів взаємодії з вузлами мережі через P2P-комунікацію, або з окремим вузлом через RPC-протокол.

8. Розроблено узагальнену архітектуру програмної системи для аналізу графа Біткоїн-транзакцій на основі спеціалізованої моделі даних транзакцій, яка об'єднує транзакції, виходи, власників та інші зовнішні метадані у єдину онтологію. Це дає змогу накопичувати в єдиному сховищі внутрішні і зовнішні знання про транзакції у вигляді RDF-графа і надає уніфікований інтерфейс взаємодії з графом знань про транзакції через стандартизований RDF-протокол.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Жикін Ю.С. Патерновий аналіз графа Біткоїн-транзакцій. / Ю.С. Жикін, М.В. Онай // Вісник Хмельницького національного університету. Серія: Технічні науки – 2024. – №2. – С. 322-328. DOI:10.31891/2307-5732-2024-333-2-51.
2. Жикін Ю.С. RDF-модель графа Біткоїн-транзакцій. / Ю.С. Жикін, М.В. Онай // Вісник Хмельницького національного університету. Серія: Технічні науки – 2024. – №5. – С. 25-30. – DOI:10.31891/2307-5732-2024-341-5-3.
3. Жикін Ю.С. Метод консолідації виходів Біткоїн-транзакцій за допомогою симуляції витрачання. / Ю.С. Жикін, М.В. Онай // Комп'ютерно-інтегровані технології: освіта, наука, виробництво – 2025. – №59. – С. 113-119. – DOI:10.36910/6775-2524-0560-2025-59-15.
4. Жикін Ю.С. Метод групування адрес за власністю на основі патернів типових операцій у графі Біткоїн-транзакцій. / Ю.С. Жикін, М.В. Онай // Прикладна математика та комп'ютинг. ПМК, 2024: Сімнадцята наук. конф. магістрантів та аспірантів, Київ, 20-22 лист. 2024 р.: зб. тез доп. [редкол.: Дичка І. А. та ін.]. – К.: ПТФ «Просвіта», 2024.– С. 87-92.
5. Жикін Ю.С. Метод зменшення розміру RDF-графа за допомогою непрямой типізації. / Ю.С. Жикін, М.В. Онай // Традиційні та інноваційні підходи до наукових досліджень: збірник наукових праць з матеріалами VIII Міжнародної наукової конференції (Дрогобич, 31 січня 2025 р.) / Міжнародний центр наукових досліджень. – Вінниця: ТОВ “УКРЛОГОС Груп”, 2025. – С. 287-290. – DOI:10.62731/mcnd-31.01.2025.007.
6. Жикін Ю.С. Взаємодія з Біткоїн-мережею з програмного середовища Common Lisp. / Ю.С. Жикін, М.В. Онай // Наукові орієнтири: теорія та практика досліджень: збірник наукових праць з матеріалами VI Міжнародної наукової конференції (Київ, 3 жовтня 2025 р.) / Міжнародний центр наукових досліджень. – Вінниця: ТОВ “УКРЛОГОС Груп”, 2025. – С. 196-199. – DOI:10.62731/mcnd-03.10.2025.002.

7. Жикін Ю.С. Узагальнена архітектура програмної системи для побудови та аналізу графа знань про Біткоїн-транзакції. / Ю.С. Жикін, М.В. Онай // Міжгалузеві диспути: динаміка та розвиток сучасних наукових досліджень: збірник наукових праць з матеріалами VIII Міжнародної наукової конференції (Харків, 30 січня 2026 р.) / Міжнародний центр наукових досліджень. – Вінниця: ТОВ “УКРЛОГОС Груп”, 2026. – С. 229-232. – DOI:10.62731/mcnd-30.01.2026.006.
8. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System [Електронний ресурс] / S. Nakamoto. – 2008. – Режим доступу: <https://bitcoin.org/bitcoin.pdf>.
9. Cypherpunks Mailing List Archives [Електронний ресурс]. – Режим доступу: <https://cypherpunks.venona.com/>.
10. Szabo N. Smart Contracts: Building Blocks for Digital Markets [Електронний ресурс] / N. Szabo. – 1996. – Режим доступу: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.
11. Poon J., Dryja T. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments [Електронний ресурс] / J. Poon, T. Dryja. – 2016. – Режим доступу: <https://lightning.network/lightning-network-paper.pdf>.
12. Dryja T. Discreet Log Contracts [Електронний ресурс] / T. Dryja. – 2017. – Режим доступу: <https://adiabat.github.io/dlc.pdf>.
13. Orlovsky M., Todd P., Zucco G., Tenga F., Ukolova O. RGB Black Paper [Електронний ресурс] / M. Orlovsky, P. Todd, G. Zucco, F. Tenga, O. Ukolova. – 2023. – Режим доступу: <https://black-paper.rgb.tech/>.
14. Lightning Labs. Taproot Assets Protocol [Електронний ресурс] / Lightning Labs. – 2023. – Режим доступу: <https://docs.lightning.engineering/the-lightning-network/taproot-assets>.

15. Merkle R. C. A Digital Signature Based on a Conventional Encryption Function / R. C. Merkle // Advances in Cryptology – CRYPTO '87. – Lecture Notes in Computer Science. – 1988. – Vol. 293. – P. 369–378. – DOI: 10.1007/3-540-48184-2_32. – ISBN 978-3-540-18796-7.
16. Back A. Hashcash – a denial of service counter-measure [Электронный ресурс] / A. Back. – 1997. – Режим доступа: <http://www.hashcash.org/papers/announce.txt>.
17. Menezes A. J., van Oorschot P. C., Vanstone S. A. Hash functions // Handbook of Applied Cryptography. – CRC Press, 2018. – P. 33–. – ISBN 978-0-429-88132-9.
18. National Institute of Standards and Technology. Secure Hash Standard (SHS). FIPS PUB 180-4 [Электронный ресурс] / NIST. – 2015. – Режим доступа: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.
19. StackExchange. What is Chainwork? [Электронный ресурс] / Bitcoin StackExchange. – 2014. – Режим доступа: <https://bitcoin.stackexchange.com/questions/26869/what-is-chainwork/26894#26894>.
20. Alonso K. M. Zero to Monero: First Edition [Электронный ресурс] / K. M. Alonso. – 2018. – Режим доступа: <https://www.getmonero.org/library/Zero-to-Monero-1-0-0.pdf>.
21. Saberhagen N. CryptoNote v 2.0 [Электронный ресурс] / N. Saberhagen. – 2013. – Режим доступа: <https://www.semanticscholar.org/paper/CryptoNote-v-2.0-Saberhagen/5bafdd891c1459ddfd22d71412d5365de723fb23>.
22. Rivest R. L., Shamir A., Tauman Y. How to Leak a Secret [Электронный ресурс] / R. L. Rivest, A. Shamir, Y. Tauman // Advances in Cryptology – ASIACRYPT 2001. – Lecture Notes in Computer Science, Vol. 2248. – Springer, 2001. – P. 552–565. – Режим доступа: <https://www.iacr.org/archive/asiacrypt2001/22480554.pdf>.

23. Noether S. Ring Signature Confidential Transactions [Электронный ресурс] / S. Noether. – Monero Research Lab Report MRL-0005. – 2015. – Режим доступа: <https://www.getmonero.org/resources/research-lab/pubs/MRL-0005.pdf>.
24. Buterin V. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform [Электронный ресурс] / V. Buterin. – 2014. – Режим доступа: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.
25. Pieter Wuille. BIP 32: Hierarchical Deterministic Wallets [Электронный ресурс] / P. Wuille. – 2012. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0032.
26. Bitcoin Wiki. Privacy [Электронный ресурс]. – 2011. – Режим доступа: <https://en.bitcoin.it/wiki/Privacy>.
27. Visa Inc. VisaNet Technology: About Visa Fact Sheet [Электронный ресурс] / Visa. – 2023. – Режим доступа: <https://www.visa.co.uk/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf>.
28. Kappos G., Yousaf H., Piotrowska A. M., Kanjalkar S., Delgado-Segura S., Miller A., Meiklejohn S. An Empirical Analysis of Privacy in the Lightning Network [Электронный ресурс] / G. Kappos, H. Yousaf, A. M. Piotrowska, S. Kanjalkar, S. Delgado-Segura, A. Miller, S. Meiklejohn // Financial Cryptography and Data Security (FC 2021). – Lecture Notes in Computer Science, vol. 12674. – Springer, 2021. – P. 167–186. – DOI: 10.1007/978-3-662-64322-8_8. – Режим доступа: <https://smeiklej.com/files/fc21.pdf>.
29. Maxwell G. CoinJoin: Bitcoin privacy for the real world [Электронный ресурс] / G. Maxwell // Bitcointalk forum. – 2013. – Режим доступа: <https://bitcointalk.org/index.php?topic=279249.0>.

30. Jonas Nick, Andrew Poelstra, Tim Ruffing. BIP 352: Silent Payments [Электронный ресурс] / J. Nick, A. Poelstra, T. Ruffing. – 2022. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0352.
31. Merkle R. C. Secure Communications Over Insecure Channels / R. C. Merkle // Communications of the ACM. – 1978. – Vol. 21, № 4. – P. 294–299. – DOI: 10.1145/359460.359473. – CiteSeerX: 10.1.1.364.5157. – S2CID 6967714.
32. Justus Ranvier. BIP 47: Reusable Payment Codes for Hierarchical Deterministic Wallets [Электронный ресурс] / J. Ranvier. – 2015. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0047.
33. Ron D., Shamir A. Quantitative Analysis of the Full Bitcoin Transaction Graph / D. Ron, A. Shamir // Financial Cryptography and Data Security (FC 2013). – Lecture Notes in Computer Science, vol. 7859. – Springer, 2013. – P. 6–24. – DOI: 10.1007/978-3-642-39884-1_2.
34. Hamacher K., Katzenbeisser S. Bitcoin – An Analysis [Электронный ресурс] / K. Hamacher, S. Katzenbeisser. – 2011. – Режим доступа: <http://www.youtube.com/watch?v=hlWyTqL1hFA>.
35. Reid F., Harrigan M. An Analysis of Anonymity in the Bitcoin System [Электронный ресурс] / F. Reid, M. Harrigan. – 2012. – Режим доступа: <https://arxiv.org/abs/1107.4524v2>.
36. Fleder M., Kester M. S., Pillai S. Bitcoin Transaction Graph Analysis [Электронный ресурс] / M. Fleder, M. S. Kester, S. Pillai. – 2014. – Режим доступа: <https://people.csail.mit.edu/spillai/data/papers/bitcoin-transaction-graph-analysis.pdf>.
37. Caprolu M., Pontecorvi M., Signorini M., Segarra C., Di Pietro R. A Novel Framework for the Analysis of Unknown Transactions in Bitcoin: Theory, Model, and Experimental Results [Электронный ресурс] / M. Caprolu, M. Pontecorvi, M. Signorini, C. Segarra, R. Di Pietro. – 2021. – Режим доступа: <https://arxiv.org/pdf/2103.09459>.

38. Cormen T. H., Leiserson C. H., Rivest R. L., Stein C. Introduction to Algorithms. 2nd ed. – MIT Press; McGraw-Hill, 2001. – Chapter 21: Data structures for Disjoint Sets. – P. 498–524.
39. Reiter J. An Empirical Study Of Early Coinjoins: Placing Limits On Blockchain Tracing In The Early Years Of Bitcoin [Электронный ресурс] / J. Reiter. – 2024. – Режим доступа: <https://ssrn.com/abstract=4887962>.
40. Wikipedia. PageRank [Электронный ресурс]. – 2025. – Режим доступа: <https://en.wikipedia.org/wiki/PageRank>.
41. Nick J. D. Data-Driven De-Anonymization in Bitcoin: Master's Thesis [Электронный ресурс] / J. D. Nick. – 2015. – Режим доступа: <https://jonasnick.github.io/papers/thesis.pdf>.
42. Nick J. Bitcoin Privacy: Theory and Practice [Электронный ресурс] / J. Nick. – 2016. – Режим доступа: <https://www.youtube.com/watch?v=HScK4pkDNds>.
43. Mempool Space. Bitcoin block explorer and mempool visualizer [Электронный ресурс]. – 2025. – Режим доступа: <https://mempool.space>.
44. Mempool Space. FAQ: How do Mempool Goggles work? [Электронный ресурс]. – 2025. – Режим доступа: <https://mempool.space/docs/faq#how-do-mempool-goggles-work>.
45. Wuille P. BIP 0030: Duplicate transactions [Электронный ресурс] / P. Wuille. – 2012. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0030.
46. Sedgewick R., Wayne K. Algorithms, 4th edition: Graphs [Электронный ресурс] / R. Sedgewick, K. Wayne. – Princeton University, 2011. – Режим доступа: <https://algs4.cs.princeton.edu/40graphs/>.
47. World Wide Web Consortium (W3C). RDF 1.1 Concepts and Abstract Syntax [Электронный ресурс] / W3C. – 2014. – Режим доступа: <https://www.w3.org/TR/rdf11-concepts/>.
48. World Wide Web Consortium (W3C). RDF 1.1 Semantics [Электронный ресурс] / W3C. – 2014. – Режим доступа: <https://www.w3.org/TR/rdf11-mt/>.

49. World Wide Web Consortium (W3C). RDF 1.1: On Semantics of Datasets [Электронный ресурс] / W3C. – 2014. – Режим доступа: <https://www.w3.org/TR/rdf11-datasets/>.
50. World Wide Web Consortium (W3C). Official Website [Электронный ресурс] / W3C. – 2025. – Режим доступа: <https://www.w3.org>.
51. Duerst M., Suignard M. Internationalized Resource Identifiers (IRIs) [Электронный ресурс] / M. Duerst, M. Suignard // RFC 3987. – IETF, 2005. – Режим доступа: <https://www.rfc-editor.org/rfc/rfc3987>.
52. Berners-Lee T., Fielding R., Masinter L. Uniform Resource Identifier (URI): Generic Syntax [Электронный ресурс] / T. Berners-Lee, R. Fielding, L. Masinter // RFC 3986. – IETF, 2005. – Режим доступа: <https://www.rfc-editor.org/rfc/rfc3986>.
53. World Wide Web Consortium (W3C). W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes [Электронный ресурс] / W3C. – 2012. – Режим доступа: <https://www.w3.org/TR/xmlschema11-2/#integer>.
54. World Wide Web Consortium (W3C). SPARQL 1.1 Overview [Электронный ресурс] / W3C. – 2013. – Режим доступа: <https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/>.
55. World Wide Web Consortium (W3C). SPARQL 1.1 Query Language [Электронный ресурс] / W3C. – 2013. – Режим доступа: <https://www.w3.org/TR/sparql11-query/>.
56. World Wide Web Consortium (W3C). RDF 1.1 N-Triples [Электронный ресурс] / W3C. – 2014. – Режим доступа: <https://www.w3.org/TR/n-triples/>.
57. World Wide Web Consortium (W3C). RDF 1.1 Turtle: Terse RDF Triple Language [Электронный ресурс] / W3C. – 2014. – Режим доступа: <https://www.w3.org/TR/turtle/>.
58. World Wide Web Consortium (W3C). OWL 2 Web Ontology Language: Document Overview [Электронный ресурс] / W3C. – 2012. – Режим доступа: <https://www.w3.org/TR/owl2-overview/>.

59. World Wide Web Consortium (W3C). OWL Web Ontology Language Reference. 5.2.1 owl:sameAs [Электронный ресурс] / W3C. – 2004. – Режим доступа: <https://www.w3.org/TR/owl-ref/#sameAs-def>.
60. World Wide Web Consortium (W3C). SPARQL 1.1 Update [Электронный ресурс] / W3C. – 2013. – Режим доступа: <https://www.w3.org/TR/sparql11-update/>.
61. Ava Chow. BIP 174: Partially Signed Bitcoin Transaction Format [Электронный ресурс] / A. Chow. – 2017. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0174.
62. Ava Chow. BIP 370: PSBT Version 2 [Электронный ресурс] / A. Chow. – 2022. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0370.
63. Bitcoin Core – Wallet Source Code [Электронный ресурс] / Bitcoin Core Developers. – 2025. – Режим доступа: <https://github.com/bitcoin/bitcoin/tree/master/src/wallet>.
64. Electrum Documentation [Электронный ресурс] / Electrum Developers. – 2025. – Режим доступа: <https://electrum.readthedocs.io/en/latest/>.
65. BlueWallet: Bitcoin Wallet [Электронный ресурс] / BlueWallet Developers. – 2025. – Режим доступа: <https://github.com/bluewallet/bluewallet>.
66. Bier J. The Blocksize War: The Battle over Who Controls Bitcoin's Protocol Rules. / J. Bier. – Independently published, 2021. – 228 p. – ISBN: 979-8721895609.
67. What is Bitcoin Network Congestion? Key Drivers and Solutions [Электронный ресурс] / Bitcoin Magazine Pro. – 2023. – Режим доступа: <https://www.bitcoinmagazinepro.com/blog/what-is-bitcoin-network-congestion-key-drivers-and-solutions/>.
68. Bitcoin Wiki. Techniques to reduce transaction fees. Consolidation [Электронный ресурс] / Bitcoin Wiki. – 2024. – Режим доступа: https://en.bitcoin.it/wiki/Techniques_to_reduce_transaction_fees#Consolidation.

69. Bitcoin Wiki. How to cheaply consolidate coins to reduce miner fees [Электронный ресурс] / Bitcoin Wiki. – 2024. – Режим доступа: https://en.bitcoin.it/wiki/How_to_cheaply_consolidate_coins_to_reduce_miner_fees.
70. Харо UTXO Consolidation [Электронный ресурс] / Bitcoin Ops. – 2020. – Режим доступа: <https://bitcoinops.org/en/харо-utxo-consolidation/>.
71. Ghesmati S. Usability of Cryptocurrency Wallets Providing CoinJoin Transactions [Электронный ресурс] / Simin Ghesmati, Walid Fdhila, Edgar Weippl. – USEC 2022, Network and Distributed System Security Symposium. – 2022. – Режим доступа: https://www.ndss-symposium.org/wp-content/uploads/usec2022_23037_paper.pdf.
72. Harding D. A. , Todd P. BIP 0125: Opt-in Full Replace-by-Fee Signaling [Электронный ресурс] / D. A. Harding, P. Todd. – 2015. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0125.
73. Bitcoin Core [Электронный ресурс] / Bitcoin Core Developers. – 2009. – Режим доступа: <https://github.com/bitcoin/bitcoin>.
74. P2P network [Электронный ресурс] / Bitcoin Developer Guide. – 2021. – Режим доступа: https://developer.bitcoin.org/devguide/p2p_network.html.
75. RPCs [Электронный ресурс] / Bitcoin Developer Reference. – 2021. – Режим доступа: <https://developer.bitcoin.org/reference/rpc/>.
76. Nostr Implementation Possibilities (NIPs) [Электронный ресурс] / Nostr Protocol. – 2020. – Режим доступа: <https://github.com/nostr-protocol/nips>.
77. AllegroGraph Introduction [Электронный ресурс] / Franz Inc. – 2023. – Режим доступа: <https://franz.com/agraph/support/documentation/agraph-introduction.html>.
78. AllegroGraph Python Client [Электронный ресурс] / Franz Inc. – 2017. – Режим доступа: <https://github.com/franzinc/agraph-python>.
79. AllegroGraph Java Client [Электронный ресурс] / Franz Inc. – 2017. – Режим доступа: <https://github.com/franzinc/agraph-java-client>.

80. Allegro CL version 11.0 documentation [Электронный ресурс] / Franz Inc. – 2023. – Режим доступа: <https://franz.com/support/documentation/contents.html>.
81. AllegroGraph Lisp Quickstart [Электронный ресурс] / Franz Inc. – 2023. – Режим доступа: <https://franz.com/agraph/support/documentation/lisp-quickstart.html>.
82. GNU Emacs [Электронный ресурс] / Free Software Foundation. – 1985. – Режим доступа: <https://www.gnu.org/software/emacs/>.
83. SLIME: Superior Lisp Interaction Mode for Emacs [Электронный ресурс] / SLIME Developers. – 2003. – Режим доступа: <https://github.com/slime/slime>.
84. SLY: Sylvester the Cat's Common Lisp IDE [Электронный ресурс] / João Távora. – 2013. – Режим доступа: <https://github.com/joaotavora/sly>.
85. Gruff: AllegroGraph Graph Visualization [Электронный ресурс] / Franz Inc. – 2023. – Режим доступа: <https://franz.com/agraph/support/documentation/gruff.html>.
86. Graphviz Documentation [Электронный ресурс] / Graphviz Project. – 2024. – Режим доступа: <https://graphviz.org/documentation/>.
87. DOT Language [Электронный ресурс] / Graphviz Project. – 2024. – Режим доступа: <https://graphviz.org/doc/info/lang.html>.
88. ImageMagick [Электронный ресурс] / ImageMagick Studio LLC. – 1990. – Режим доступа: <https://imagemagick.org/>.
89. Corallo M. Compact Block Relay [Электронный ресурс] / M. Corallo. – 2016. – Режим доступа: https://en.bitcoin.it/wiki/BIP_0152.
90. BRDF: Bitcoin chain data represented as RDF [Электронный ресурс] / Y. Zhykin. – 2025. – Режим доступа: <https://github.com/rodrabies/brdf>.
91. BP: Bitcoin Protocol Library [Электронный ресурс] / Y. Zhykin. – 2025. – Режим доступа: <https://github.com/rodrabies/bp>.
92. BP: Bitcoin Protocol Library, Core Module [Электронный ресурс] / Y. Zhykin. – 2025. – Режим доступа: <https://github.com/rodrabies/bp/tree/master/core>.

93. BP: Bitcoin Protocol Library, Crypto Module [Электронный ресурс] / Y. Zhykin. – 2025. – Режим доступа: <https://github.com/roentrabies/bp/tree/master/crypto>.
94. libsecp256k1 [Электронный ресурс] / Bitcoin Core Developers. – 2013. – Режим доступа: <https://github.com/bitcoin-core/secp256k1>.
95. BP: Bitcoin Protocol Library, Net Module [Электронный ресурс] / Y. Zhykin. – 2025. – Режим доступа: <https://github.com/roentrabies/bp/tree/master/net>.
96. BP: Bitcoin Protocol Library, RPC Module [Электронный ресурс] / Y. Zhykin. – 2025. – Режим доступа: <https://github.com/roentrabies/bp/tree/master/rpc>.
97. BP: Bitcoin Protocol Library, Test Subsystem [Электронный ресурс] / Y. Zhykin. – 2025. – Режим доступа: <https://github.com/roentrabies/bp/tree/master/tests>.
98. Understand GitHub Actions [Электронный ресурс] / GitHub Docs. – 2024. – Режим доступа: <https://docs.github.com/en/actions/get-started/understand-github-actions>.
99. Todd P. python-bitcoinlib [Электронный ресурс] / P. Todd. – 2013. – Режим доступа: <https://github.com/petertodd/python-bitcoinlib>.
100. Portable AllegroServe [Электронный ресурс] / sharplispers. – 2014. – Режим доступа: <https://github.com/sharplispers/portableaserve>.

ДОДАТКИ

Додаток А. RDFS-опис RDF-моделі даних Біткоїн-транзакцій

Повна актуальна версія цього документу доступна за посиланням <https://github.com/rodrabies/brdf/blob/master/brdf.ttl> [90]. Інструмент BRDF використовує лише підмножину наведених властивостей класів.

```
@base <http://github.com/rodrabies/brdf#>.
```

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
```

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
```

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
```

```
# Block is a class of entities that aggregate Transactions.
```

```
<#Block>
```

```
    rdf:type      rdfs:Class.
```

```
<#blockHeight>
```

```
    rdf:type      rdfs:Property;
```

```
    rdfs:domain   <#Block>;
```

```
    rdfs:range    xsd:integer.
```

```
<#blockHash>
```

```
    rdf:type      rdfs:Property;
```

```
    rdfs:domain   <#Block>;
```

```
    rdfs:range    xsd:string.
```

```
<#blockVersion>
```

```
    rdf:type      rdfs:Property;
```

```
    rdfs:domain   <#Block>;
```

```
    rdfs:range    xsd:integer.
```

```
<#blockMerkleRoot>
```

```
    rdf:type      rdfs:Property;
```

```
    rdfs:domain   <#Block>;
```

```
    rdfs:range    xsd:string.
```

```
<#blockDifficulty>
```

```

        rdf:type      rdfs:Property;
        rdfs:domain   <#Block>;
        rdfs:range    xsd:integer.

<#blockTime>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Block>;
        rdfs:range    xsd:dateTime.

<#blockTxCount>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Block>;
        rdfs:range    xsd:integer.

<#blockPreviousBlock>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Block>;
        rdfs:range    <#Block>.

<#blockNextBlock>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Block>;
        rdfs:range    <#Block>;
        owl:inverseOf <#prevBlock>.

<#blockTx>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Block>;
        rdfs:range    <#Tx>.

# Tx is a class of entities that contains transaction metadata and sets of
# transaction inputs and outputs.

<#Tx>
        rdf:type      rdfs:Class.

<#txIndex>
        rdf:type      rdfs:Property;
        rdfs:range    xsd:integer.

<#txID>

```

```

        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    xsd:string.

<#txVersion>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    xsd:integer.

<#txSize>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    xsd:integer.

<#txVSize>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    xsd:integer.

<#txWeight>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    xsd:integer.

<#txLockTime>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    rdf:integer.

<#txInputCount>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    xsd:integer.

<#txInput>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    <#Input>.

<#txOutputCount>

```

```

        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    xsd:integer.

<#txOutput>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Tx>;
        rdfs:range    <#Output>.

# Output represents an abstract chunk of value in the Bitcoin system
# and consists of the integer amount of atomic bitcoin units and a
# locking script.

<#Output>
        rdf:type      rdfs:Class.

<#outputIndex>
        rdf:type rdfs:Property;
        rdfs:range xsd:integer.

<#outputAmount>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Output>;
        rdfs:range    xsd:integer.

<#outputScript>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Output>;
        rdfs:range    xsd:string.

<#outputType>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Output>;
        rdfs:range    xsd:string.

<#outputAddress>
        rdf:type      rdfs:Property;
        rdfs:domain   <#Output>;
        rdfs:range    xsd:string.

<#outputInput>

```

```
    rdf:type      rdfs:Property;  
    rdfs:domain  <#Output>;  
    rdfs:range   <#Input>.
```

<#outputInputTx>

```
    rdf:type      rdfs:Property;  
    rdfs:domain  <#Output>;  
    rdfs:range   <#Tx>.
```

Додаток Б. Реалізація та приклади використання спеціалізованих фільтр-функцій SPARQL

```
(in-package :db.agraph.sbqe)

(ag:register-namespace "bp" "http://github.com/rodentrabies/brdf#")

(ag:enable-!-reader)

;;; bp:txInputTotal

(defsparql-filter-function txInputTotal (tx)
  (:uri !bp:txInputTotal
   :language :sparql
   :category :bp
   :eager? t
   :document? nil)
  bp-tx-input-total)

(defun bp-tx-input-total (tx)
  (let ((total nil)
        (input-triples (ag:get-triples-list :p !bp:outputInputTx :o tx)))
    (when input-triples
      (setf total 0)
      (with-temp-upis (output amount)
        (dolist (input-triple input-triples)
          (%subject input-triple output)
          (let ((amount-triple (ag:get-triple :s output :p
!bp:outputAmount)))
            (if* amount-triple
              then (%object amount-triple amount)
              (incf total (ag:part->value amount)))
            else (setf total nil)
            (return))))))
    (or total ag.base:+unbound-value-upi+)))
```

```
;;; bp:txOutputTotal
```

```
(defspargl-filter-function txOutputTotal (tx)
```

```
  (:uri !bp:txOutputTotal
```

```
   :language :sparql
```

```
   :category :bp
```

```
   :eager? t
```

```
   :document? nil)
```

```
  bp-tx-output-total)
```

```
(defun bp-tx-output-total (tx)
```

```
  (let ((total nil)
```

```
        (output-triples (ag:get-triples-list :s tx :p !bp:txOutput))))
```

```
  (when output-triples
```

```
    (setf total 0)
```

```
    (with-temp-upis (output amount)
```

```
      (dolist (output-triple output-triples)
```

```
        (%object output-triple output)
```

```
          (let ((amount-triple (ag:get-triple :s output :p  
!bp:outputAmount))))
```

```
            (if* amount-triple
```

```
                  then (%object amount-triple amount)
```

```
                    (incf total (ag:part->value amount))
```

```
                    else (setf total nil)
```

```
                    (return))))))
```

```
  (or total ag.base:+unbound-value-upi+))
```

```
;;; bp:txFee
```

```
(defspargl-filter-function txOutputTotal (tx)
```

```
  (:uri !bp:txFee
```

```
   :language :sparql
```

```
   :category :bp
```

```
   :eager? t
```

```
   :document? nil)
```

```
  bp-tx-fee)
```

```
(defun bp-tx-fee (tx)
```

```
  (let ((input-total (bp-tx-input-total tx))
```

```
        (output-total (bp-tx-output-total tx)))
```

```
    (if* (and (numberp input-total) (numberp output-total))
```

```

        then (- input-total output-total)
        else ag.base:+unbound-value-upi+)))

;;; Testing and examples

#+test
(sparql:run-sparql "

PREFIX bp: <http://github.com/rodrabries/brdf#>

SELECT ?itotal ?ototal ?fee {
  VALUES ?tx {
    bp:4ed9b30b20af3e3af03631da33dad38378ceec9633f33dae5eb5b37f987a381b
  }
  BIND (bp:txInputTotal(?tx) AS ?itotal)
  BIND (bp:txOutputTotal(?tx) AS ?ototal)
  BIND (bp:txFee(?tx) AS ?fee)

}

" :output-format :table)

;; -----
;; | itotal    | ototal    | fee    |
;; =====
;; | 101464606 | 101059726 | 404880 |
;; -----

```

Додаток В. Реалізація розширення для візуалізації результатів SPARQL-запитів типу “CONSTRUCT”

```
(in-package :sparql.results)

(defparameter *custom-dot-attributes*
  '(:labelmask))

(defparameter *default-dot-attributes*
  '(:graph      (:rankdir :LR)
    :resource   (:shape :ellipse :style :filled :fillcolor :aquamarine)
    :blank      (:shape :ellipse :style :filled :fillcolor :grey)
    :literal    (:shape :box :style :filled :fillcolor :bisque)
    :edge       (:arrowhead :normal)))

(defvar *dot-attributes* nil)
(defparameter *dot-renderer* "dot")
(defparameter *dot-viewer* "display")
(defparameter *dot-pipeline* nil)

(defun dot-attributes (category)
  (let ((standard-attributes (copy-list (getf *default-dot-attributes*
category))))
    (custom-attributes nil))
  (loop :for (name value) :on (getf *dot-attributes* category) :by #'cddr
    :do (setf (getf standard-attributes name) value))
  (loop :for name :in *custom-dot-attributes*
    :for value := (getf standard-attributes name)
    :do (when value
        (setf (getf custom-attributes name) value)
        (remf standard-attributes name)))
  (values standard-attributes custom-attributes)))

(defun make-dot-display-stream (&key
                                (renderer *dot-renderer*)
                                (viewer *dot-viewer*)
                                (pipeline *dot-pipeline*)
                                if-does-not-exist)
  (unless pipeline
    (when (eq if-does-not-exist :error)
      (unless (excl:osi:find-in-path renderer :executable t)
```

```

(error "DOT renderer executable (`~a`) not found." renderer))
(unless (excl:osi:find-in-path viewer :executable t)
  (error "DOT image viewer executable (`~a`) not found." viewer)))
(let ((command (or pipeline (format nil "~a -Tpng | ~a -" renderer
viewer))))
  (excl:run-shell-command command :input :stream :output nil :wait nil)))

(in-package :db.agraph.sbqe)

(defun output-dot-from-triple-cursor (cursor output-stream)
  (bind (((:values graph-attributes _) (dot-attributes :graph))
        ((:values resource-attributes custom-resource-attributes)
         (dot-attributes :resource))
        ((:values literal-attributes custom-literal-attributes)
         (dot-attributes :literal))
        ((:values blank-attributes _) (dot-attributes :blank))
        ((:values edge-attributes custom-edge-attributes) (dot-attributes
:edge)))
    (generic-node-template " ~a [label=~\"~a\", ~{~a=~a~^, ~}];~%"
      (generic-edge-template " ~a -> ~a [label=~\"~a\", ~{~a=~a~^,
~}];~%"
        (resource-template (format nil generic-node-template
resource-attributes))
        (literal-template (format nil generic-node-template
literal-attributes))
        (blank-template (format nil generic-node-template blank-attributes))
        (edge-template (format nil generic-edge-template edge-attributes))
        (resource-mask (getf custom-resource-attributes :labelmask))
        (literal-mask (getf custom-literal-attributes :labelmask))
        (edge-mask (getf custom-edge-attributes :labelmask))
        (node-map (make-upi-hash-table))
        (node-id-counter -1))
      (labels ((node-label (node)
        (db.agraph:part->concise node))
        (apply-mask (mask label)
          (if* mask
            then (let ((llen (length label))
                      (nmask (max mask 8))
                      (gap "..."))
              (if* (> llen (+ nmask (length gap)))
                then (let* ((l (floor nmask 2))
                           (r (- (length label) 1)))

```

```

                                (string+ (subseq label 0 1) gap (subseq
label r)))

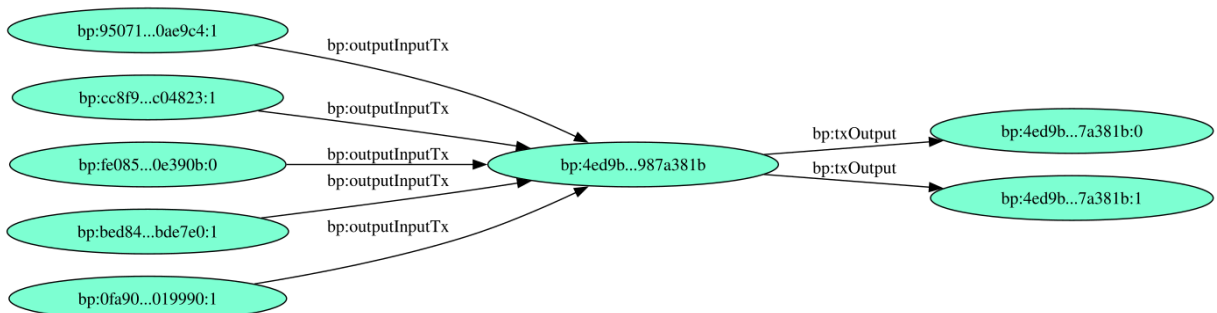
                                else label))
    else label))
  (maybe-output-node (node)
    (unless (gethash node node-map)
      (let ((node-id (incf node-id-counter)))
        (setf (gethash (copy-upi node) node-map) node-id)
        (multiple-value-bind (template mask)
          (if* (resource-upi-p node)
            then (values resource-template resource-mask)
            elseif (db.agraph:blank-node-upi-p node)
            then (values blank-template nil)
            else (values literal-template literal-mask))
          (format output-stream template node-id (apply-mask mask
(node-label node)))))))
    (output-edge (from edge to)
      (format output-stream
        edge-template
        (gethash from node-map)
        (gethash to node-map)
        (apply-mask edge-mask (node-label edge))))))
(with-temp-upis (s p o)
  (format output-stream "~&digraph {~%")
  (format output-stream "  ~{~a=~a~^;~%  ~};~%" graph-attributes)
  (iterate-cursor (triple cursor)
    (%subject triple s)
    (%predicate triple p)
    (%object triple o)
    (maybe-output-node s)
    (maybe-output-node o)
    (output-edge s p o))
  (format output-stream "~&}~%"))))

```

Додаток Г. Приклади використання розширення для візуалізації результатів SPARQL-запитів з REPL-інтерфейсу середовища Common Lisp

Візуалізація вказаної транзакції, її входів та виходів (у цьому випадку запит типу “CONSTRUCT” може повністю відтворити оригінальну структуру підграфа):

```
CL-USER> (let ((sparql.results:*dot-attributes*  
                '(:resource (:labelmask 16)  
                  :literal (:labelmask 32))))  
          (sparql:run-sparql "  
  
PREFIX bp: <http://github.com/rodrabies/brdf#>  
CONSTRUCT {  
  ?tx bp:txOutput ?out.  
  ?in bp:outputInputTx ?tx.  
} WHERE {  
  VALUES ?tx {  
    bp:4ed9b30b20af3e3af03631da33dad38378ceec9633f33dae5eb5b37f987a381b  
  }  
  ?tx bp:txOutput ?out.  
  ?in bp:outputInputTx ?tx.  
}  
  
" :output-format :dot  
  :output-stream :dot/display))
```



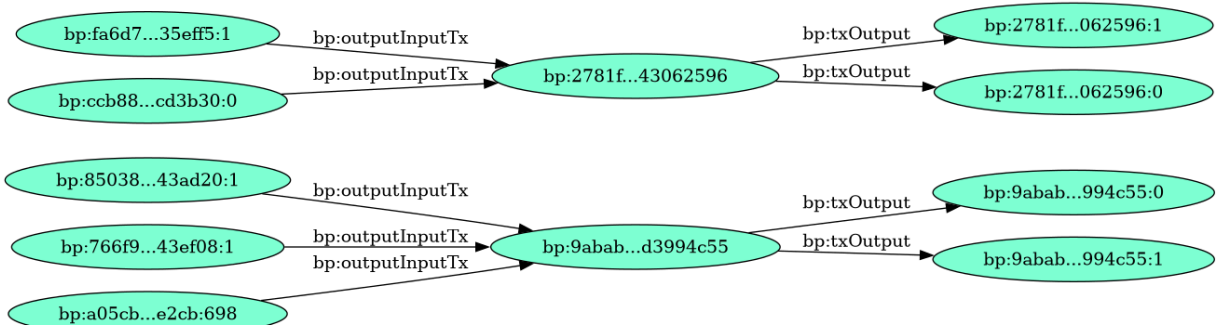
Візуалізація всіх транзакцій, що використовують вказану множину виходів, та їхніх власних виходів (візуалізація нових власників вказаних виходів):

```
CL-USER> (let ((sparql.results:*dot-attributes*
                '(:resource (:labelmask 16)
                  :literal (:labelmask 32))))
  (sparql:run-sparql "
```

```
PREFIX bp: <http://github.com/rodentrabies/brdf#>
```

```
CONSTRUCT {
  ?in bp:outputInputTx ?tx.
  ?tx bp:txOutput ?out.
} WHERE {
  ?in bp:outputInputTx ?tx.
  ?tx bp:txOutput ?out.
  FILTER EXISTS {
    VALUES ?given {
      bp:85038971e9235667f779f6d5d0f9cc3277e1b886469064a9a50c794a9843ad20:1
      bp:ccb88d04b23d2aedef77c772151424c752f8d5d7ef25f908fb9a43d01ddcd3b30:0
    }
    ?given bp:outputInputTx ?tx.
  }
}

" :output-format :dot
:output-stream :dot/display))
```



Додаток Г. Програмний код ETL-інструменту BRDF

Повна актуальна версія програмного коду і супроводжуючих даних інструменту BRDF доступна за посиланням <https://github.com/rodrabies/brdf> [90].

```
;;; BRDF - Bitcoin chain data represented as RDF

;; Fail here if the Lisp image is not an AllegroCL one.
#-allegro
(error "Direct AllegroGraph interface only works in AllegroCL.")

;; Load `agraph.fasl' dependency unless it's already present.
#-allegrograph
(load "agraph.fasl")

;; Finally, load external dependencies `jsown' and `bp' using Quicklisp.
(ql:quickload "jsown")
(ql:quickload "bp")

(defpackage :brdf
  (:use :cl :bp :db.agraph)
  (:import-from :bp.core.script
    #:*print-script-as-assembly*)
  (:import-from :db.agraph.triple-store-spec
    #:parse-triple-store-specification
    #:ground-store-address-p
    #:ground-store-address-scheme
    #:ground-store-address-host
    #:ground-store-address-port
    #:ground-store-address-user
    #:ground-store-address-password
    #:ground-store-address-catalog
    #:ground-store-address-repo)
  (:import-from :db.agraph.part-constants
    #:+rdf-type-uri+
    #:+xsd-string-uri+
    #:+xsd-integer-upi+)
  (:import-from :db.agraph.http.store
    #:@client))
```

```

(:import-from :db.agraph.http.client
              #:define-namespace
              #:repository-data)
(:import-from :st-json
              #:jso
              #:write-json-to-string)

(:export #:prepare-repository
          #:start-load
          #:stop-load
          #:*status-check-period*))

(in-package :brdf)

;;; Model

(eval-when (:compile-toplevel :load-toplevel :execute)
  (defun source-relative-pathname (path)
    (let ((defaults (or *load-truename* *default-pathname-defaults*)))
      (merge-pathnames path (make-pathname :name nil :type nil :defaults
                                              defaults)))))

(defconstant +brdf-schema+ (source-relative-pathname "brdf.ttl"))

(defconstant +brdf-queries+ (source-relative-pathname "queries/"))

(defconstant +brdf-namespace+ "http://github.com/rodrabies/brdf#")

;; Function that converts a Bitcoin block with hash `block-hash' at
;; height `block-height' into triples according to the model given by
;; `graph'. Assumes a triple store is available in `db.agraph:*db*'
;; and a chain supplier is available in `bp:*chain-supplier*'.
(defgeneric load-block (graph block-height block-hash))

(defun type-to-rdf-datatype (type)
  (ecase type
    (:integer +xsd-integer-uri+)
    (:string +xsd-string-uri+)
    (:datetime "http://www.w3.org/2001/XMLSchema#dateTime")))

(defun unix-to-date-time (timestamp)
  (excl:universal-time-to-string (excl:osi:unix-to-universal-time timestamp)
    :format :iso8601))

```

```

(defmacro %r (name)
  `(resource ,name "bp"))

(defmacro %l (value type)
  `(literal (format nil "~a" ,value) :datatype (type-to-rdf-datatype ,type)))

(defmethod load-block ((graph (eql :chain)) block-height block-hash)
  (let* ((block-hex (get-block block-hash :encoded t))
        (block (decode 'cblock block-hex))
        (block-version (block-version block))
        (block-size (/ (length block-hex) 2))
        (block-time (unix-to-date-time (block-timestamp block)))
        (block-transactions (block-transactions block))
        (block-transactions-count (length block-transactions))
        (block-resource (%r (format nil "~a" block-height))))
    ;; Add block information triples.
    (add-triple block-resource +rdf-type-uri+ (%r "Block"))
    (add-triple block-resource (%r "blockHeight") (%l block-height :integer))
    (add-triple block-resource (%r "blockHash") (%l block-hash :string))
    (add-triple block-resource (%r "blockVersion") (%l block-version :integer))
    (add-triple block-resource (%r "blockSize") (%l block-size :integer))
    (add-triple block-resource (%r "blockTime") (%l block-time :datetime))
    (add-triple block-resource (%r "blockTxCount") (%l block-transactions-count :integer))
    ;; Link previous/next blocks.
    (when (plusp block-height)
      (let ((previous-block-resource (%r (format nil "~a" (1- block-height)))))
        (add-triple block-resource (%r "blockPreviousBlock") previous-block-resource)
        (add-triple previous-block-resource (%r "blockNextBlock") block-resource)))
    ;; Load block's transactions.
    (loop :for tx-index :below block-transactions-count :do
      (let* ((tx (aref block-transactions tx-index))
            (tx-id (tx-id tx))
            (tx-version (tx-version tx))
            (tx-size (/ (length (encode tx)) 2))
            (tx-locktime (tx-locktime tx))
            (tx-inputs (tx-inputs tx)))

```

```

(tx-inputs-count (length tx-inputs))
(tx-outputs (tx-outputs tx))
(tx-outputs-count (length tx-outputs))
(tx-resource (%r tx-id))
;; Link transaction with the block.
(add-triple block-resource (%r "blockTx") tx-resource)
(add-triple tx-resource (%r "txIndex") (%l tx-index :integer))
;; Add transaction data.
(add-triple tx-resource +rdf-type-uri+ (%r "Tx"))
(add-triple tx-resource (%r "txID") (%l tx-id :string))
(add-triple tx-resource (%r "txVersion") (%l tx-version
:integer))
(add-triple tx-resource (%r "txSize") (%l tx-size :integer))
(add-triple tx-resource (%r "txLockTime") (%l tx-locktime
:integer))
(add-triple tx-resource (%r "txInputCount") (%l tx-inputs-count
:integer))
(add-triple tx-resource (%r "txOutputCount") (%l tx-outputs-count
:integer))
;; Load transaction inputs.
(loop :for input-index :below tx-inputs-count :do
  (let* ((input (aref tx-inputs input-index))
         (input-prevout-id (txin-previous-tx-id input))
         (input-prevout-index (txin-previous-tx-index input))
         (input-sequence (txin-sequence input))
         (input-script-sig (txin-script-sig input))
         (input-resource (new-blank-node)))
    ;; Link input with the transaction.
    (add-triple tx-resource (%r "txInput") input-resource)
    (add-triple input-resource (%r "inputIndex") (%l input-index
:integer))
    ;; Add input data.
    (add-triple input-resource +rdf-type-uri+ (%r "Input"))
    (add-triple input-resource (%r "inputSequence") (%l
input-sequence :integer))
    (add-triple input-resource (%r "inputScript") (%l
input-script-sig :string))
    ;; Link input with the output that it spends unless this is a
    coinbase transaction.
    (when (not (every #'zerop input-prevout-id))
      (let* ((input-prevout-id-hex (hex-encode (reverse
input-prevout-id)))

```



```

        (tx-output-count (length (tx-outputs tx))))
;; Create transaction resource.
(add-triple block-resource (%r "blockTx") tx-resource)
(add-triple tx-resource +rdf-type-uri+ (%r "Tx"))
        (add-triple tx-resource (%r "txInputCount") (%1 tx-input-count
:integer))
        (add-triple tx-resource (%r "txOutputCount") (%1 tx-output-count
:integer))
;; Link transaction with input outpoints.
(loop :for input-index :below tx-input-count :do
  (let* ((input (aref (tx-inputs tx) input-index))
         (input-prevout-id (txin-previous-tx-id input))
         (input-prevout-index (txin-previous-tx-index input)))
    (when (not (every #'zerop input-prevout-id))
      ;; We expect outputs to be described by their generating
transactions,
      ;; so only a link here.
      (let* ((input-prevout-id-hex (hex-encode (reverse
input-prevout-id)))
             (input-previous-output (format nil "~a::~~a"
input-prevout-id-hex input-prevout-index)))
        (add-triple (%r input-previous-output) (%r "outputInputTx")
tx-resource))))))
;; Link transaction with outputs.
(loop :for output-index :below tx-output-count :do
  (let* ((output (aref (tx-outputs tx) output-index))
         (output-amount (txout-amount output))
         (output-script (txout-script-pubkey output))
         (output-resource (%r (format nil "~a::~~a" tx-id
output-index))))
    ;; Link output with the transaction.
    (add-triple tx-resource (%r "txOutput") output-resource)
    (add-triple output-resource +rdf-type-uri+ (%r "Output"))
    (add-triple output-resource (%r "outputAmount") (%1 output-amount
:integer))
    ;; Output address will be non-NIL only for standard scripts.
    (multiple-value-bind (_ output-address)
      (script-standard-p output-script :network (network))
      (declare (ignore _))
      (if output-address
        (add-triple output-resource (%r "outputAddress") (%1
output-address :string))

```

```

                                (add-triple output-resource (%r "outputScript") (%l
output-script :string))))))))))

;;; Loader scaffolding

(defvar *workers* nil
  "A place to store loader processes to be able to conveniently stop
them at once.")

(defvar *status-check-period* 10)

(define-condition stop-worker () ())

(defmacro with-open-remote-triple-store ((var spec &rest args) &body body)
  "Hack-macro that forces a triple store specified by SPEC to be
opened as REMOTE-TRIPLE-STORE."
  (let ((parsed-spec (gensym "parsed-spec")))
    `(let ((,parsed-spec (parse-triple-store-specification ,spec)))
      (assert (ground-store-address-p ,parsed-spec))
      (with-open-triple-store (,var (ground-store-address-repo ,parsed-spec)
                              :triple-store-class 'remote-triple-store
                              :scheme (ground-store-address-scheme
,parsed-spec)
                              :server (ground-store-address-host
,parsed-spec)
                              :port (ground-store-address-port
,parsed-spec)
                              :user (ground-store-address-user
,parsed-spec)
                              :password (ground-store-address-password
,parsed-spec)
                              :catalog (ground-store-address-catalog
,parsed-spec)
                              :expected-size 1000000000 ;; 1B
                              :string-table-size 512000000 ;; 512M
                              ,@args)
        ,@body))))

```

```

(defun prepare-repository (dst &key include-schema-p include-queries-p)
  (with-open-remote-triple-store (db dst :if-does-not-exist :create)
    ;; Perform reinitialization *only* if no triples in the triple
    ;; store.
    (when (= (triple-count :db db) 0)
      (drop-index :gposi :db db)
      (drop-index :gspoi :db db))
    ;; Only add schema triples if explicitly asked.
    (when include-schema-p
      (load-turtle +brdf-schema+ :db db))
    ;; These require HTTP API access hence the forced remote triple store.
    (let ((client (@client db)))
      ;; Persistently set the main namespace.
      (define-namespace client "bp" +brdf-namespace+ :type :repository)
      ;; Add a bunch of useful saved queries.
      (when include-queries-p
        (flet ((save-query (file)
                  (let* ((name (pathname-name file))
                        (query (excl:file-contents file))
                        (saved-query (jso "title" name "query" query
                                           "language" "SPARQL"))
                        (setf (repository-data client (format nil "vv.query.~a"
                                                                name))
                              (write-json-to-string saved-query))))
                  (query-file-p (file)
                                (equal (pathname-type file) "rq"))))
          (excl:map-over-directory #'save-query +brdf-queries+ :filter
                                   #'query-file-p))))
      ;; Commit and close remote triple store interface.
      (commit-triple-store :db db)))

(defun start-load (graph src dst &key workers from-height to-height)
  (when *workers*
    (error "Load process with ~a workers is already running." (length
                                                                *workers*)))
  ;; Prepare repository for load.
  (prepare-repository dst :include-schema-p nil :include-queries-p t)
  ;; Set local namespace abbreviation.
  (register-namespace "bp" +brdf-namespace+)
  ;; Now open repository for operation.

```

```

(with-open-triple-store (db dst :if-does-not-exist :error)
  (let ((block-queue (make-instance 'mp:queue :name "BRDF block queue
lock"))))
    ;; Print starting heights and number of workers.
    (format t "[~a] Starting load with ~a worker~p~%"
      (excl:universal-time-to-string (get-universal-time))
      workers
      workers)

    ;; Start planner/status thread.
    (push (mp:process-run-function
      "BRDF planner"
      #'start-planner
      src dst block-queue from-height to-height workers)
      *workers*)

    ;; Start workers.
    (dotimes (i workers)
      (push (mp:process-run-function
        (format nil "BRDF worker ~a" i)
        #'start-worker
        graph src dst block-queue)
        *workers*))))))

(defun stop-load ()
  (dolist (worker *workers*)
    (ignore-errors
      (mp:process-interrupt worker (lambda () (signal 'stop-worker))))
    (mp:process-join worker))
  (setf *workers* nil)
  (remove-namespace "bp"))

(defun start-worker (graph src dst block-queue)
  (handler-case
    (with-open-triple-store (db dst :if-does-not-exist :error)
      (unwind-protect
        (with-chain-supplier (bprpc:node-rpc-connection :url src)
          (with-buffered-triple-adds (db)
            (load-blocks graph block-queue)))
        (rollback-triple-store :db db)))
    (stop-worker ())))

```

```

(defun load-blocks (graph block-queue)
  (loop
    :for block-height := (mp:dequeue block-queue :wait t)
    :while block-height
    :do
      (tagbody retry-block
        ;; Load block at a height `HEIGHT'.
        (let ((*print-script-as-assembly* t)
              (block-hash (get-block-hash block-height)))
          (when (null block-hash) ;; no known block at that height
            (go retry-block))
          (load-block graph block-height block-hash)))
        ;; Commit all the triples for current block and continue to the
        ;; next one. It will also flush the buffered triples.
        (commit-triple-store)))

(defun loaded-blocks-table ()
  (let ((table (make-hash-table :test '=)))
    (dolist (result (sparql:run-sparql "SELECT ?b { ?b a bp:Block }"
                                       :output-format :lists))
      (let* ((block (part->value (first result)))
              (height (parse-integer (subseq block (length
+brdf-namespace+))))
        (setf (gethash height table) t)))
      table))

(defun start-planner (src dst block-queue from-height to-height workers)
  (with-open-triple-store (db dst :if-does-not-exist :error)
    (let* ((node-connection (make-instance 'bprpc:node-rpc-connection :url
src))
            (initial-chain-stats (bprpc:getchaintxstats node-connection))
            (best-height (jsown:val initial-chain-stats
"window_final_block_height")))
      (loaded-blocks-table (loaded-blocks-table))
      (from-height (or from-height 0))
      (initially-enqueued-blocks 0))
    ;; Populate the block queue with all blocks in the specified
    ;; range unless they are already in the DB.
    (loop :for i :from from-height :to (or to-height best-height)
      :do (unless (gethash i loaded-blocks-table)

```

```

        (incf initially-enqueued-blocks)
        (mp:enqueue block-queue i)))

(format t "[~a] Initially enqueued blocks: ~a~%"
  (excl:universal-time-to-string (get-universal-time))
  initially-enqueued-blocks)
;; Start the status check loop.
(handler-case
  (loop
    (let* ((chain-stats (bprpc:getchaintxstats node-connection))
           (new-best-height (jsown:val chain-stats
                                         "window_final_block_height")))
      (txcount (jsown:val chain-stats "txcount"))
      (graph-blocks (sparql:run-sparql "SELECT DISTINCT ?b { ?b
a bp:Block }"
                                         :output-format :count))
      (graph-txs (sparql:run-sparql "SELECT DISTINCT ?b { ?b a
bp:Tx }"
                                     :output-format :count))
      (progress (/ graph-txs txcount)))
    ;; Enqueue new blocks if needed.
    (when (> new-best-height best-height)
      (loop
        :for i :from (1+ best-height) :to (or to-height
new-best-height)
        :do (format t "[~a] Adding block ~a to the queue~%"
          (excl:universal-time-to-string (get-universal-time))
          i)
          (mp:enqueue block-queue i))
      (setf best-height new-best-height)
      (when (and to-height (> best-height to-height))
        (loop :repeat workers
          :do (mp:enqueue block-queue nil))))
    (format t "[~a] Blocks: ~a/~a, txs: ~a/~a, progress: ~5$~%"
      (excl:universal-time-to-string (get-universal-time))
      graph-blocks
      best-height
      graph-txs
      txcount
      progress)
    (sleep *status-check-period*)
    (rollback-triple-store)))
  (stop-worker ())))))

```

```
;;; Tests and examples
```

```
#+test
```

```
;; On first invocation, choose the number of workers that will utilize  
;; the underlying machine to the optimum.
```

```
(brdf:start-load :chain
```

```
    "http://user:password@127.0.0.1:8332"
```

```
    "http://user:password@127.0.0.1:10035/repositories/brdf"
```

```
    :workers 4)
```

```
#+test
```

```
;; In order to stop the load, do:
```

```
(brdf:stop-load)
```

Додаток Д. Фрагменти програмного коду бібліотеки для взаємодії з Біткоїн-мережею з програмного середовища Common Lisp

Повна актуальна версія програмного коду бібліотеки доступна за посиланням <https://github.com/rodrabies/bp> [91].

```
;;; Copyright (c) BP Developers & Contributors
;;; See the accompanying file LICENSE for the full license governing this
code.

(uiop:define-package :bp.core.chain (:nicknames :bp/core/chain)
  (:use :cl)
  (:use :bp.core.block
        :bp.core.transaction
        :bp.core.encoding)
  (:export
   ;; Chain supplier API:
   #:*chain-supplier*
   #:chain-supplier
   #:with-chain-supplier
   #:chain-testnet-p
   #:chain-get-block-hash
   #:chain-get-block
   #:chain-get-transaction
   #:network
   #:testnet-p
   #:get-block-hash
   #:get-block
   #:get-transaction
   ;; Chain supplier helpers:
   #:with-chain-supplier-normalization
   ;; Chain supplier conditions:
   #:unknown-entity-error
   #:unknown-block-hash-error
   #:unknown-block-height
   #:unknown-block-error
   #:unknown-block-hash
   #:unknown-transaction-error
   #:unknown-transaction-id))
```

```

(in-package :bp.core.chain)

;;; Generic declarations

(defclass chain-supplier ()
  ((network
    :accessor chain-supplier-network
    :initarg :network
    :initform :mainnet
    :documentation "Network marker (one of :MAINNET, :TESTNET, :REGTEST)."))))

(define-condition unknown-entity-error (simple-error)
  ())

(define-condition unknown-block-hash-error (unknown-entity-error)
  ((height
    :initarg :height
    :accessor unknown-block-height))
  (:report
    (lambda (e s)
      (format s "Unknown block at height ~a." (unknown-block-height e)))))

(define-condition unknown-block-error (unknown-entity-error)
  ((hash
    :initarg :hash
    :accessor unknown-block-hash))
  (:report
    (lambda (e s)
      (format s "Unknown block ~a." (unknown-block-hash e)))))

(define-condition unknown-transaction-error (unknown-entity-error)
  ((id
    :initarg :id
    :accessor unknown-transaction-id))
  (:report
    (lambda (e s)
      (format s "Unknown transaction ~a." (unknown-transaction-id e)))))

(defgeneric chain-network (supplier)
  (:documentation "Return keyword representing the given SUPPLIER's
network (one of :MAINNET, :TESTNET, :REGTEST).")
  (:method (supplier)
    (chain-supplier-network supplier)))

```

```

(defgeneric chain-testnet-p (supplier)
  (:documentation "Return NIL if SUPPLIER's network is :MAINNET and T
otherwise.")
  (:method (supplier)
    (not (eq (chain-supplier-network supplier) :mainnet))))

(defgeneric chain-get-block-hash (supplier height &key encoded errorp)
  (:documentation "Get the hash of the block from SUPPLIER by its
HEIGHT in the chain. HEIGHT must be an integer. If ENCODED is non-NIL,
returns a hex-encoded string, otherwise returns a raw id represented
as byte array. If there is no known block at the given HEIGHT, return
NIL or signal an UNKNOWN-BLOCK-HASH-ERROR error, depending on the
ERRORP value."))

(defgeneric chain-get-block (supplier hash &key encoded errorp)
  (:documentation "Get raw block data from SUPPLIER by its HASH. HASH
can be either a hex-encoded string or a byte array. If ENCODED is
non-NIL, returns a hex-encoded string, otherwise returns CBLOCK
object. If there is no block with the given HASH, return NIL or signal
an UNKNOWN-BLOCK-ERROR error, depending on the ERRORP value."))

(defgeneric chain-get-transaction (supplier id &key encoded errorp)
  (:documentation "Get raw transaction data from SUPPLIER by its
ID. ID can be either a hex-encoded string or a byte array. If ENCODED
is non-NIL, returns a hex-encoded string, otherwise returns TX
object. If there is no transaction with a given ID, return NIL or
signal an UNKNOWN-TRANSACTION-ERROR error, depending on the ERRORP
value."))

;;; Helper macros

(defmacro with-chain-supplier-normalization ((id-var encoded-var errorp-var
                                              &key entity-type id-type
                                              body-type error-type)
                                             &body body)
  "Helper macro for generating the normalization if the entity
identifier (block height, block hash and transaction id) and
post-processing (encoding, decoding and error signalling) for the
chain supplier API implementations.

ID-VAR is an entity identifier variable, which will be normalized to a
hex-string, byte array or left unchanged if the value of ID-TYPE is

```

:ENCODED, :DECODED or :AS-IS respectively.

ENCODED-VAR corresponds to ENCODED chain supplier parameter - it will be used in combination with BODY-TYPE argument to determine if the result of the BODY should be encoded, decoded (as an ENTITY-TYPE entity in the latter case) or left as-is.

ERROPR-VAR corresponds to the ERRORP chain supplier parameter - it will be used to either return NIL or signal a corresponding error if the body returns NIL. If ERROR-TYPE is non-NIL, it will be used instead of the default error type."

```
(let* ((result (gensym "chain-supplier-result"))
      (id-form
        (ecase id-type
          (:encoded `(if (stringp ,id-var)
                        ,id-var
                        (hex-encode (reverse ,id-var))))
          (:decoded `(if (stringp ,id-var)
                        (reverse (hex-decode ,id-var))
                        ,id-var))
          (:as-is `(,id-var))))
      (encode-form
        (ecase entity-type
          (:block-hash `(hex-encode (reverse ,result)))
          ((:block :transaction) `(encode ,result))))
      (decode-form
        (ecase entity-type
          (:block-hash `(reverse (hex-decode ,result)))
          (:block `(decode 'cblock ,result))
          (:transaction `(decode 'tx ,result))))
      (error-form
        (ecase entity-type
          (:block-hash `(error ',(or error-type 'unknown-block-hash-error)
                                :height ,id-var))
          (:block `(error ',(or error-type 'unknown-block-error)
                            :hash ,id-var))
          (:transaction `(error ',(or error-type
                                'unknown-transaction-error)
                                :id ,id-var)))))
  `(let* ((,id-var ,id-form)
        (,result (progn ,@body)))
    (or ,(ecase body-type
          (:encoded
```

```

        `(cond ((and ,result ,encoded-var) ,result)
              (,result ,decode-form)))
      (:decoded
       `(cond ((and ,result ,encoded-var) ,encode-form)
             (,result ,result)))
      (:as-is
       result))
      (and ,errorp-var ,error-form))))))

;;; Context-dependent API

(defvar *chain-supplier* nil
  "Global chain supplier bound by the WITH-CHAIN-SUPPLIER context manager.")

(defmacro with-chain-supplier ((type &rest args &key &allow-other-keys) &body
  body)
  `(let ((*chain-supplier* (make-instance ',type ,@args)))
    ,@body))

(defun network ()
  (chain-network *chain-supplier*))

(defun testnet-p ()
  (chain-testnet-p *chain-supplier*))

(defun get-block-hash (height &key errorp)
  (chain-get-block-hash *chain-supplier* height :errorp errorp))

(defun get-block (hash &key encoded errorp)
  (chain-get-block *chain-supplier* hash :encoded encoded :errorp errorp))

(defun get-transaction (id &key encoded errorp)
  (chain-get-transaction *chain-supplier* id :encoded encoded :errorp
    errorp))

;;; Copyright (c) BP Developers & Contributors
;;; See the accompanying file LICENSE for the full license governing this
code.

(uiop:define-package :bp.core.transaction (:nicknames :bp/core/transaction)
  (:use :cl)
  (:use

```

```

:bp.core.encoding
:bp.core.script
:bp.crypto.hash)
(:export
;; Transaction API:
#:tx
#:tx-version
#:tx-inputs
#:tx-input
#:tx-outputs
#:tx-output
#:tx-witnesses
#:tx-witness
#:tx-locktime
#:tx-hash
#:tx-id
#:tx-wid
;; Transaction input API:
#:txin
#:txin-previous-tx-id
#:txin-previous-tx-index
#:txin-script-sig
#:txin-sequence
;; Transaction output API:
#:txout
#:txout-amount
#:txout-script-pubkey
;; Witness API:
#:witness
#:witness-items))

;;; Copyright (c) BP Developers & Contributors
;;; See the accompanying file LICENSE for the full license governing this
code.

(uiop:define-package :bp.core.block (:nicknames :bp/core/block)
  (:use :cl)
  (:use :bp.core.encoding
        :bp.core.transaction
        :bp.crypto.hash)
  (:export
   ;; Block header API:
   #:block-header

```

```

#:block-version
#:block-previous-block-hash
#:block-merkle-root
#:block-timestamp
#:block-bits
#:block-nonce
#:block-hash
#:block-id
;; Complete block API:
#cblock
#:block-transactions
#:block-transaction))

(in-package :bp.core.block)

;;; Generic API

(defgeneric block-version (block))

(defgeneric block-previous-block-hash (block))

(defgeneric block-merkle-root (block))

(defgeneric block-timestamp (block))

(defgeneric block-bits (block))

(defgeneric block-nonce (block))

(defgeneric block-hash (block))

(defun block-id (block)
  (hex-encode (reverse (block-hash block))))

;;; Data structures and API implementation

(defstruct block-header
  version
  previous-block-hash
  merkle-root
  timestamp
  bits
  nonce)

```

```

(defmethod serialize ((block-header block-header) stream)
  (let ((version (block-version block-header))
        (previous-block-hash (block-previous-block-hash block-header))
        (merkle-root (block-merkle-root block-header))
        (timestamp (block-timestamp block-header))
        (bits (block-bits block-header))
        (nonce (block-nonce block-header)))
    (write-int version stream :size 4 :byte-order :little)
    (write-bytes previous-block-hash stream 32)
    (write-bytes merkle-root stream 32)
    (write-int timestamp stream :size 4 :byte-order :little)
    (write-bytes (reverse bits) stream 4)
    (write-bytes nonce stream 4)))

(defmethod parse ((entity-type (eql 'block-header)) stream)
  (let ((version (read-int stream :size 4 :byte-order :little))
        (previous-block-hash (read-bytes stream 32 ))
        (merkle-root (read-bytes stream 32))
        (timestamp (read-int stream :size 4 :byte-order :little))
        (bits (reverse (read-bytes stream 4)))
        (nonce (read-bytes stream 4)))
    (make-block-header
     :version version
     :previous-block-hash previous-block-hash
     :merkle-root merkle-root
     :timestamp timestamp
     :bits bits
     :nonce nonce)))

(defmethod print-object ((block-header block-header) stream)
  (print-unreadable-object (block-header stream :type t)
    (format
     stream "~& id: ~a~& prev: ~a"
     (block-id block-header)
     (hex-encode (reverse (block-previous-block-hash block-header))))))

(defmethod block-version ((block-header block-header))
  (block-header-version block-header))

(defmethod block-previous-block-hash ((block-header block-header))
  (block-header-previous-block-hash block-header))

```

```

(defmethod block-merkle-root ((block-header block-header))
  (block-header-merkle-root block-header))

(defmethod block-timestamp ((block-header block-header))
  (block-header-timestamp block-header))

(defmethod block-bits ((block-header block-header))
  (block-header-bits block-header))

(defmethod block-nonce ((block-header block-header))
  (block-header-nonce block-header))

(defmethod block-hash ((block-header block-header))
  (hash256
   (with-output-to-byte-array (stream)
     (serialize block-header stream))))

(defstruct (cblock (:conc-name block-))
  header
  transactions)

(defun block-transaction (cblock index)
  (aref (block-transactions cblock) index))

(defmethod serialize ((cblock cblock) stream)
  (let* ((header (block-header cblock))
         (transactions (block-transactions cblock))
         (num-transactions (length transactions)))
    (serialize header stream)
    (write-varint num-transactions stream)
    (loop
     :for i :below num-transactions
     :do (serialize (aref transactions i) stream))))

(defmethod parse ((entity-type (eql 'cblock)) stream)
  (let* ((header (parse 'block-header stream))
         (num-transactions (read-varint stream))
         (transactions
          (loop
           :with transaction-array
           := (make-array num-transactions :element-type 'tx)
           :for i :below num-transactions

```

```

        :do (setf (aref transaction-array i) (parse 'tx stream))
        :finally (return transaction-array))))
(make-cblock
 :header header
 :transactions transactions)))

(defmethod print-object ((cblock cblock) stream)
  (print-unreadable-object (cblock stream :type t)
    (format
      stream "(~a txs)~& id: ~a~& prev: ~a"
      (length (block-transactions cblock))
      (block-id (block-header cblock))
      (hex-encode (reverse (block-previous-block-hash cblock))))))

(defmethod block-version ((cblock cblock))
  (block-version (block-header cblock)))

(defmethod block-previous-block-hash ((cblock cblock))
  (block-previous-block-hash (block-header cblock)))

(defmethod block-merkle-root ((cblock cblock))
  (block-merkle-root (block-header cblock)))

(defmethod block-timestamp ((cblock cblock))
  (block-timestamp (block-header cblock)))

(defmethod block-bits ((cblock cblock))
  (block-bits (block-header cblock)))

(defmethod block-nonce ((cblock cblock))
  (block-nonce (block-header cblock)))

(defmethod block-hash ((cblock cblock))
  (block-hash (block-header cblock)))

;;; Copyright (c) BP Developers & Contributors
;;; See the accompanying file LICENSE for the full license governing this
code.

(uiop:define-package :bp.core.transaction (:nicknames :bp/core/transaction)
  (:use :cl)
  (:use
   :bp.core.encoding

```

```

:bp.core.script
:bp.crypto.hash)
(:export
;; Transaction API:
#:tx
#:tx-version
#:tx-inputs
#:tx-input
#:tx-outputs
#:tx-output
#:tx-witnesses
#:tx-witness
#:tx-locktime
#:tx-hash
#:tx-id
#:tx-wid
;; Transaction input API:
#:txin
#:txin-previous-tx-id
#:txin-previous-tx-index
#:txin-script-sig
#:txin-sequence
;; Transaction output API:
#:txout
#:txout-amount
#:txout-script-pubkey
;; Witness API:
#:witness
#:witness-items))

(in-package :bp.core.transaction)

(defstruct tx
  version
  inputs
  outputs
  witnesses
  locktime)

(defmethod serialize ((tx tx) stream)
  (let* ((version (tx-version tx))
        (inputs (tx-inputs tx))
        (num-inputs (length inputs)))

```

```

    (outputs (tx-outputs tx))
    (num-outputs (length outputs))
    (witnesses (tx-witnesses tx))
    (locktime (tx-locktime tx)))
(write-int version stream :size 4 :byte-order :little)
(when witnesses
  (write-byte #x00 stream) ;; write SegWit marker
  (write-byte #x01 stream)) ;; write SegWit flag
(write-varint num-inputs stream)
(loop
  :for i :below num-inputs
  :do (serialize (aref inputs i) stream))
(write-varint num-outputs stream)
(loop
  :for i :below num-outputs
  :do (serialize (aref outputs i) stream))
(when witnesses
  (loop
    ;; Witness array length equals to input array length.
    :for i :below num-inputs
    :do (serialize (aref witnesses i) stream)))
(write-int locktime stream :size 4 :byte-order :little)))

(defun read-num-inputs-or-segwit-flag (stream)
  "If currently parsed transaction is a SegWit one, it will have SegWit
marker
and flag fields (0x00 0x01) bytes after its version field instead of the
number
of inputs, so this function reads a varint and if it is 0x00, verifies that
the
next byte is 0x01. Returns list (<number-of-inputs> <segwit-flag>), where
<segwit-flag> is the value of the flag (0x01) if transaction is SegWit, or
NIL
otherwise."
  (let ((v (read-varint stream)))
    (if (= v #x00)
      ;; SegWit marker found, read flag and then the actual number
      ;; of inputs.
      (let ((flag (read-byte stream)))
        ;; Currently flag *must* be 0x01.
        (ecase flag
          (#x01 (list (read-varint stream) flag))))
      (list v nil))))

```

```

(defmethod parse ((entity-type (eql 'tx)) stream)
  (let* ((version (read-int stream :size 4 :byte-order :little))
         (num-inputs/segwit-flag (read-num-inputs-or-segwit-flag stream))
         (num-inputs (first num-inputs/segwit-flag))
         (segwit-flag (second num-inputs/segwit-flag))
         (inputs
          (loop
           :with input-array := (make-array num-inputs :element-type 'txin)
           :for i :below num-inputs
           :do (setf (aref input-array i) (parse 'txin stream))
           :finally (return input-array)))
         (num-outputs (read-varint stream))
         (outputs
          (loop
           :with output-array := (make-array num-outputs :element-type
'txout)

           :for i :below num-outputs
           :do (setf (aref output-array i) (parse 'txout stream))
           :finally (return output-array)))
         (witnesses
          ;; Check if not NIL, ignore the value for now.
          (when segwit-flag
            (loop
             :with witnesses := (make-array num-inputs :element-type
'twitness)

             :for i :below num-inputs
             :do (setf (aref witnesses i) (parse 'witness stream))
             :finally (return witnesses))))
         (locktime (read-int stream :size 4 :byte-order :little)))
    (make-tx
     :version version
     :inputs inputs
     :outputs outputs
     :witnesses witnesses
     :locktime locktime)))

(defun tx-hash (tx)
  "Raw transaction ID is a double SHA256 of its binary serialization."
  (hash256
   (with-output-to-byte-array (stream)
     (serialize tx stream))))

```

```

(defun tx-id (tx)
  "Return hex-encoded txid - little-endian hash of the transaction
serialization
without witness structures."
  (let ((legacy-tx (copy-tx tx)))
    (setf (tx-witnesses legacy-tx) nil)
    (hex-encode (reverse (tx-hash legacy-tx)))))

(defun tx-wid (tx)
  "Return hex-encoded wtxid - little-endian hash of the transaction
serialization including witness structures."
  (hex-encode (reverse (tx-hash tx))))

(defun tx-output (tx index)
  "Return INDEXth output of the given transaction."
  (aref (tx-outputs tx) index))

(defun tx-input (tx index)
  "Return INDEXth input of the given transaction."
  (aref (tx-inputs tx) index))

(defun tx-witness (tx index)
  "Return INDEXth witness of the given transaction if it is a SegWit
transaction, otherwise return NIL."
  (when (tx-witnesses tx)
    (aref (tx-witnesses tx) index)))

(defmethod print-object ((tx tx) stream)
  (print-unreadable-object (tx stream :type t)
    (format stream "~a" (tx-id tx))))

(defstruct txin
  previous-tx-id
  previous-tx-index
  script-sig
  sequence)

(defmethod serialize ((txin txin) stream)
  (let ((previous-tx-id (txin-previous-tx-id txin))
        (previous-tx-index (txin-previous-tx-index txin))
        (script-sig (txin-script-sig txin))
        (sequence (txin-sequence txin)))
    (write-sequence previous-tx-id stream)

```

```

(write-int previous-tx-index stream :size 4 :byte-order :little)
(serialize script-sig stream)
(write-int sequence stream :size 4 :byte-order :little)))

(defmethod parse ((entity-type (eql 'txin)) stream)
  (let ((previous-tx-id (read-bytes stream 32))
        (previous-tx-index (read-int stream :size 4 :byte-order :little))
        (script-sig (parse 'script stream))
        (sequence (read-int stream :size 4 :byte-order :little)))
    (make-txin
      :previous-tx-id previous-tx-id
      :previous-tx-index previous-tx-index
      :script-sig script-sig
      :sequence sequence)))

(defmethod print-object ((txin txin) stream)
  (print-unreadable-object (txin stream :type t)
    (format
      stream "~a:~a"
      (hex-encode (reverse (txin-previous-tx-id txin)))
      (txin-previous-tx-index txin))))

(defstruct txout
  amount
  script-pubkey)

(defmethod serialize ((txout txout) stream)
  (let ((amount (txout-amount txout))
        (script-pubkey (txout-script-pubkey txout)))
    (write-int amount stream :size 8 :byte-order :little)
    (serialize script-pubkey stream)))

(defmethod parse ((entity-type (eql 'txout)) stream)
  (let ((amount (read-int stream :size 8 :byte-order :little))
        (script-pubkey (parse 'script stream)))
    (make-txout
      :amount amount
      :script-pubkey script-pubkey)))

(defmethod print-object ((txout txout) stream)
  (print-unreadable-object (txout stream :type t)
    (format stream "amount: ~a" (txout-amount txout))))

```

```

(defstruct witness
  items)

(defmethod serialize ((witness witness) stream)
  (let* ((items (witness-items witness))
         (num-items (length items)))
    (write-varint num-items stream)
    (loop
     :for i :below num-items
     :for item      := (aref items i)
     :for item-size := (length item)
     :do
       (write-varint item-size stream)
       (write-bytes item stream item-size))))

(defmethod parse ((entity-type (eql 'witness)) stream)
  (let* ((num-items (read-varint stream))
         (items
          (loop
           :with items
             := (make-array num-items :element-type '(array (unsigned-byte
8) *))
           :for i :below num-items
           :do (setf (aref items i) (read-bytes stream (read-varint
stream))))
         :finally (return items))))
    (make-witness :items items))

(defmethod print-object ((witness witness) stream)
  (print-unreadable-object (witness stream :type t)
    (format stream "items: ~a" (length (witness-items witness)))))

;;; Copyright (c) BP Developers & Contributors
;;; See the accompanying file LICENSE for the full license governing this
code.

(uiop:define-package :bp.core.consensus (:nicknames :bp/core/consensus)
  (:use :cl)
  (:use :bp.core.encoding
        :bp.core.chain
        :bp.core.merkletree
        :bp.core.block
        :bp.core.transaction))

```

```

        :bp.core.script
        :bp.core.parameters
        :bp.crypto.hash)
(:export
  #:validate
  #:validp
  #:validation-context
  #:make-validation-context))

(in-package :bp.core.consensus)

;;; The definitions in this file contain Bitcoin consensus rules. The
;;; top-level consensus API consists of two functions: VALIDATE, which
;;; returns T if an entity from the model is valid, and signals an
;;; explanatory error otherwise, and a wrapping VALIDP predicate,
;;; which ignores the errors and returns NIL in the latter case.
;;;
;;; Bitcoin Consensus rules are particularly hard to reimplement,
;;; since there is a lot of corner cases (caused by engineering
;;; decisions, errors and semantic peculiarities of the C++, etc),
;;; some of which are known, and some are not, and failure to
;;; reproduce these corner cases may easily put this implementation
;;; out of sync with the network.
;;;
;;; Hence, THIS BITCOIN CONSENSUS RULES IMPLEMENTATION IS NOT, AND
;;; WILL PROBABLY NEVER BE FULLY COMPLIANT WITH BITCOIN CORE
;;; IMPLEMENTATION. DO NOT RELY ON IT FOR VALIDATING YOUR MAINNET
;;; TRANSACTIONS.

;;; Validation API

(defgeneric validate (entity &key context)
  (:documentation "Validate entity according to the Bitcoin Protocol
consensus rules, throw an error if an entity is invalid for any reason."))

(defun validp (entity &key context)
  "Return T if the ENTITY is valid, NIL otherwise."
  (ignore-errors (apply #'validate entity context)))

(defclass validation-context ()
  ((height      :initarg :height      :accessor @height      :initform nil)
   (cblock      :initarg :block       :accessor @block       :initform nil)
   (tx          :initarg :tx          :accessor @tx          :initform nil)))

```

```

(tx-index      :initarg :tx-index      :accessor @tx-index      :initform nil)
(txin          :initarg :txin          :accessor @txin          :initform nil)
(txin-index    :initarg :txin-index    :accessor @txin-index    :initform nil)
(txout         :initarg :txout         :accessor @txout         :initform nil)
(txout-index   :initarg :txout-index   :accessor @txout-index   :initform nil))
(:documentation "Structure for storing additional information needed
during entity validation."))

(defmacro ensure-validation-context ((context-sym) &body body)
  "Ensure CONTEXT-SYM is bound to the VALIDATION-CONTEXT object before
executing the BODY."
  (assert (symbolp context-sym))
  `(let      ((,context-sym      (or      ,context-sym      (make-instance
'validation-context))))
    ,@body))

(defun extend-validation-context (context &key height block tx tx-index txin
                                txin-index txout txout-index)
  "Create a new VALIDATION-CONTEXT object from the given one and
extend it with additional data if supplied."
  (make-instance
'validation-context
:height      (or height      (@height context))
:block       (or block       (@block context))
:tx          (or tx          (@tx context))
:tx-index    (or tx-index    (@tx-index context))
:txin        (or txin        (@txin context))
:txin-index  (or txin-index  (@txin-index context))
:txout       (or txout       (@txout context))
:txout-index (or txout-index (@txout-index context))))

;;; Blocks

(defun bits-to-target (bits)
  (let* ((f (byte-array-to-integer bits :start 1))
        (e (aref bits 0)))
    (integer-to-byte-array
      (* f (expt 2 (* 8 (- e 3))))
      :n-bits 256
      :big-endian nil)))

(defun block-target (block)
  (bits-to-target (block-bits block)))

```

```

(defun block-difficulty (block)
  (let ((current-target (bits-to-target (block-bits block)))
        (max-target (bits-to-target (make-byte-array 4 #(#x1d #x00 #xff
#xff))))))
    (float
     (/ (byte-array-to-integer max-target :big-endian nil)
        (byte-array-to-integer current-target :big-endian nil)))))

(defmethod validate ((block-header block-header) &key context)
  (declare (ignore context))
  (unless (< (byte-array-to-integer
              (block-hash block-header) :big-endian nil)
            (byte-array-to-integer
              (block-target block-header) :big-endian nil))
    (error "Block hash does not satisfy Proof-of-Work target."))
  t)

(defmethod validate ((cblock cblock) &key context)
  (declare (ignore context))
  (let* ((merkle-leaves (coerce (block-transactions cblock) 'list))
        (merkle-tree (build-merkle-tree merkle-leaves)))
    (unless (equalp (merkle-tree-node-hash merkle-tree)
                    (block-merkle-root cblock))
      (error "Block merkle root and transaction tree merkle root differ."))
    (validate (block-header cblock)))
  t)

;;; Transactions

(defconstant +sighash-all+          #x01)
(defconstant +sighash-none+         #x02)
(defconstant +sighash-single+       #x03)
(defconstant +sighash-anyonecanpay+ #x80)

(defun tx-sighash-base (tx txin-index script-code sighash-type)
  (let* ((tx (decode 'tx (encode tx))) ;; copy
        (txin (tx-input tx txin-index)))
    ;; SIGHASH_ALL: sign ALL of the outputs.
    (loop
     :for i :below (length (tx-inputs tx))
     :do (setf (script-commands (txin-script-sig (tx-input tx i))) #()))
    (setf (script-commands (txin-script-sig txin))

```

```

        (script-commands script-code))
;; SIGHASH_NONE: sign NONE of the outputs.
(when (= (logand sighash-type #x1f) +sighash-none+)
  ;; Empty output vector.
  (setf (tx-outputs tx) (make-array 0 :element-type 'txout))
  ;; Set each input's (except current one) sequence to 0.
  (loop
    :for i :below (length (tx-inputs tx))
    :if (/= i txin-index)
    :do (setf (txin-sequence (tx-input tx i)) 0)))
;; SIGHASH_SINGLE: sign ONE of the outputs.
(when (= (logand sighash-type #x1f) +sighash-single+)
  ;; Bitcoin Core's edge case; see
  ;; https://bitcointalk.org/index.php?topic=260595.0
  ;; for an explanation.
  (when (< (length (tx-outputs tx)) (length (tx-inputs tx)))
    (let ((fhash (make-byte-array 32)))
      (setf (aref fhash 31) 1)
      (return-from tx-sighash-base fhash)))
  ;; Resize output vector to TXIN-INDEX + 1 outputs.
  (setf (tx-outputs tx) (subseq (tx-outputs tx) 0 (1+ txin-index)))
  ;; Set each output's (except the TXIN-INDEXth one) to empty
  ;; script and value of -1.
  (loop
    :for i :below (length (tx-outputs tx))
    :if (/= i txin-index)
    :do
      (let ((txout (tx-output tx i)))
        (setf (script-commands (txout-script-pubkey txout)) #())
        (setf (txout-amount txout) -1)))
  ;; Set each input's (except current one) sequence to 0.
  (loop
    :for i :below (length (tx-inputs tx))
    :if (/= i txin-index)
    :do (setf (txin-sequence (tx-input tx i)) 0)))
;; SIGHASH_ANYONECANPAY: anyone can add inputs to this
;; transaction.
(when (/= (logand sighash-type +sighash-anyonecanpay+) 0)
  ;; Set input vector to single (current) input.
  (setf (tx-inputs tx)
    (make-array 1 :element-type 'txin :initial-element txin)))
(hash256
  (with-output-to-byte-array (stream)

```

```

        (serialize tx stream)
        (write-int sighash-type stream :size 4 :byte-order :little))))))

(defun tx-previous-outputs-hash (tx sighash-type)
  (if (zerop (logand sighash-type +sighash-anyonecanpay+))
      (hash256
        (with-output-to-byte-array (stream)
          (loop
            :for txin :across (tx-inputs tx)
            :for prev-id := (txin-previous-tx-id txin)
            :for prev-index := (txin-previous-tx-index txin)
            :do
              (write-bytes prev-id stream 32)
              (write-int prev-index stream :size 4 :byte-order :little))))
        (make-byte-array 32)))

(defun tx-sequence-hash (tx sighash-type)
  (if (and (zerop (logand sighash-type +sighash-anyonecanpay+))
            (/= (logand sighash-type #x1f) +sighash-single+)
            (/= (logand sighash-type #x1f) +sighash-none+))
      (hash256
        (with-output-to-byte-array (stream)
          (loop
            :for txin :across (tx-inputs tx)
            :for sequence := (txin-sequence txin)
            :do (write-int sequence stream :size 4 :byte-order :little))))
        (make-byte-array 32)))

(defun tx-outputs-hash (tx txin-index sighash-type)
  (cond
    ((and (/= (logand sighash-type #x1f) +sighash-single+)
           (/= (logand sighash-type #x1f) +sighash-none+))
      (hash256
        (with-output-to-byte-array (stream)
          (loop
            :for txout :across (tx-outputs tx)
            :do (serialize txout stream))))))
    ((and (= (logand sighash-type #x1f) +sighash-single+)
           (< txin-index (length (tx-outputs tx))))
      (hash256
        (with-output-to-byte-array (stream)
          (serialize (tx-output tx txin-index) stream))))))
  (t

```

```

(make-byte-array 32))))

(defun tx-sighash-witness-v0 (tx txin-index amount script-code sighash-type)
  "Compute sighash for version 0 witness transactions in BIP-0143 (see the
  spec at https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki)."
  (let ((previous-outputs-hash (tx-previous-outputs-hash tx sighash-type))
        (sequence-hash         (tx-sequence-hash tx sighash-type))
        (outputs-hash          (tx-outputs-hash tx txin-index sighash-type))
        (txin                  (tx-input tx txin-index)))
    (hash256
      (with-output-to-byte-array (stream)
        (write-int (tx-version tx) stream :size 4 :byte-order :little)
        (write-bytes previous-outputs-hash stream 32)
        (write-bytes sequence-hash stream 32)
        (write-bytes (txin-previous-tx-id txin) stream 32)
        (write-int (txin-previous-tx-index txin) stream :size 4 :byte-order
          :little)
        (serialize script-code stream)
        (write-int amount stream :size 8 :byte-order :little)
        (write-int (txin-sequence txin) stream :size 4 :byte-order :little)
        (write-bytes outputs-hash stream 32)
        (write-int (tx-locktime tx) stream :size 4 :byte-order :little)
        (write-int sighash-type stream :size 4 :byte-order :little))))))

(defun tx-sighash (tx txin-index amount script-code sighash-type sigversion)
  (ecase sigversion
    (:base
      (tx-sighash-base tx txin-index script-code sighash-type))
    (:witness-v0
      (tx-sighash-witness-v0 tx txin-index amount script-code sighash-type))))

;; TODO: maybe make this part of chain supplier API.
(defun get-transaction-output (id index)
  "Get txout described by its transaction ID and output index.
  Will signal an error if the transaction is unknown or does not have
  the output with given index. Assumes chain supplier context."
  (let ((prev-tx (get-transaction id :errorp t)))
    (when prev-tx
      (if (>= index (length (tx-outputs prev-tx)))
        (error "Unknown previous output ~a:~a." (tx-id prev-tx) index)
        (tx-output prev-tx index))))))

(defun tx-coinbase (tx)

```

```

"Assume transaction TX is a coinbase transaction and extract its
coinbase field (first input). Signal error if there is more than one
input."
  (if (> (length (tx-inputs tx)) 1)
      (error "Coinbase transaction must have a single input.")
      (tx-input tx 0)))

;; TODO: maybe make this part of transaction data API.
(defun txin-amount (txin)
  "Get the amount of the txout corresponding to the given txin.
Assumes chain supplier context."
  (let* ((previous-tx-id (txin-previous-tx-id txin))
         (previous-tx-index (txin-previous-tx-index txin))
         (previous-output (get-transaction-output previous-tx-id
previous-tx-index)))
    (txout-amount previous-output)))

(defun tx-fee (tx)
  (- (apply #' + (map 'list #'txin-amount (tx-inputs tx)))
     (apply #' + (map 'list #'txout-amount (tx-outputs tx)))))

(defun block-fee (block)
  (apply #' + (map 'list #'tx-fee (subseq (block-transactions block) 1))))

(defmethod validate ((tx tx) &key context)
  (ensure-validation-context (context)
    (let ((coinbase-p (and (@tx-index context) (= (@tx-index context) 0))))
      (flet (;; Total amount of all (which is one) coinbase inputs.
              (%coinbase-in-amount ()
                (let ((height (@height context))
                      (block (@block context)))
                  (unless (and height block)
                      (error "Block data and height required for coinbase tx
validation."))
                  (let* ((halvings (truncate (@height context)
+halving-period+))
                          (base (truncate +initial-block-reward+ (expt 2
halvings))))
                      (fee (block-fee block)))
                    (+ base fee))))
        ;; Verify the amount restriction.
        (%validate-amounts (in-amount out-amount)
          (unless (>= in-amount out-amount)

```

```

        (error "Output total is larger then input total.)))
;; Validate every output.
(%validate-outputs (outputs)
  (loop
    :for txout-index :below (length outputs)
    :for txout-context
      := (extend-validation-context context :tx tx :txout-index
txout-index)
      :do (validate (tx-output tx txout-index) :context
txout-context)))
;; Validate every input.
(%validate-inputs (inputs)
  (loop
    :for txin-index :below (length inputs)
    :for txin-context
      := (extend-validation-context context :tx tx :txin-index
txin-index)
      :do (validate (tx-input tx txin-index) :context
txin-context))))
(if coinbase-p
  (let ((in (%coinbase-in-amount))
        (out (apply #'(map 'list #'txout-amount (tx-outputs
tx)))))
    (%validate-amounts in out)
    (%validate-inputs (list (tx-coinbase tx))))
  (let ((in (apply #'(map 'list #'txin-amount (tx-inputs tx)))
        (out (apply #'(map 'list #'txout-amount (tx-outputs
tx)))))
    (%validate-amounts in out)
    (%validate-inputs (tx-inputs tx)))
  (%validate-outputs (tx-outputs tx)
t))))

(defmethod validate ((txin txin) &key context)
  (ensure-validation-context (context)
    (let ((coinbase-p (and (@tx-index context) (= (@tx-index context) 0))))
      (if coinbase-p
        (let ((height (@height context))
              (block (@block context)))
          (unless (and height block)
            (error "Block data and height required for coinbase
validation."))
          ;; Verify BIP-0034 restriction.

```

```

    (when (= (block-version block) #x02)
      ;; TODO: too much details of script implementation leak here.
      (let* ((bip34-height-command (aref (script-commands
(txin-script-sig txin)) 0))
              (bip34-height
              (decode-integer (cdr
bip34-height-command))))
        (unless (= bip34-height height)
          (error "BIP-0034: coinbase must include block height.")))
        (unless (equalp (txin-previous-tx-id txin) (make-byte-array 32))
          (error "Coinbase previous tx id must be 0."))
        (unless (= (txin-previous-tx-index txin) #xffffffff)
          (error "Coinbase previous tx index must be 0xffffffff."))
        t)
    ;; Regular txin.
    (let* ((prev-out (get-transaction-output
      (txin-previous-tx-id txin)
      (txin-previous-tx-index txin)))
           (script-pubkey (txout-script-pubkey prev-out))
           (amount (txout-amount prev-out))
           (script-sig (txin-script-sig txin))
           (witness (tx-witness (@tx context) (@txin-index context)))
           (*bip-0016-active-p*
            ;; If block information is available, detect whether
            ;; BIP-0016 is active using the known switch
            ;; timestamps.
            ;; TODO: this is too hacky - need to come up with a
            ;; better upgrade indicators.
            (if (@block context)
              (>= (block-timestamp (@block context))
                (if (testnet-p)
                  +testnet-bip-0016-switch-time+
                  +mainnet-bip-0016-switch-time+))
              *bip-0016-active-p*)))
           (sighashf
            (lambda (script-code sighash-type sigversion)
              (tx-sighash (@tx context)
                          (@txin-index context)
                          amount
                          script-code
                          sighash-type
                          sigversion)))
           (script-state
            (make-script-state

```

```

        :witness (when witness (witness-items witness))
        :sighashf sighashf)))
    (unless (execute-scripts script-sig script-pubkey :state
script-state)
        (error "Script execution failed."))
    t))))))

(defmethod validate ((txout txout) &key context)
  (declare (ignore context))
  ;; Assume all txouts are valid for now.
  t)

;;; Copyright (c) BP Developers & Contributors
;;; See the accompanying file LICENSE for the full license governing this
code.

(uiop:define-package :bp.rpc (:nicknames :bprpc :bp/rpc/all)
  (:use :cl)
  (:import-from :aserve)
  (:import-from :jsown)
  (:use :bp.core)
  (:export
    ;; Classes and conditions:
    #:node-connection ;; will be removed soon
    #:node-rpc-connection
    #:rpc-error
    ;; Node RPC methods:
    #:getblockhash
    #:getblock
    #:getrawtransaction
    #:getchaintxstats))

(in-package :bp.rpc)

;;; Conditions

(define-condition connection-error (simple-error)
  ((code
    :initarg :code
    :reader connection-error-code)
   (message
    :initarg :message
    :reader connection-error-message)
   (format-string

```

```

:allocation :class
:initform "Connection error: code ~a~@[ (~a)~]"
(:report
(lambda (e s)
  (let* ((code (connection-error-code e))
         (message (connection-error-message e))
         (message/nil (when (plusp (length message)) message))
         (format-string (slot-value e 'format-string)))
    (format s format-string code message/nil))))

(define-condition http-error (connection-error)
  ((format-string :initform "HTTP connection error: HTTP status ~a (~a)")))

(define-condition rpc-error (http-error)
  ((format-string :initform "RPC connection error: RPC code ~a (~a)")))

;;; RPC connection
(defclass rpc-connection ()
  ((url
    :accessor rpc-connection-url
    :initarg :url)
   (username
    :accessor rpc-connection-username
    :initarg :username
    :initform nil)
   (password
    :accessor rpc-connection-password
    :initarg :password
    :initform nil)))

(:documentation "RPC-CONNECTION is a simple generic client for the
RPC server.))

;;; Generic RPC utilities

(defun do-simple-rpc-call (connection method &rest arguments)
  (let* ((user (rpc-connection-username connection))
         (password (rpc-connection-password connection))
         (authorization (cons user password))
         (content
          (format nil
                   "{
                    \"jsonrpc\": \"1.0\",
                    \"method\": \"~a\",

```

```

        \"params\":  [{~{~s~^, ~}], ~
        \"id\":      \"bp\"      ~
    }"
    method
    arguments)))
(multiple-value-bind (response status)
  (net.aserve.client:do-http-request (rpc-connection-url connection)
    :basic-authorization authorization
    :method :post
    :content content
    :content-type "text/plain")
  (cond ((= status 200)
    (jsown:val (jsown:parse response) "result"))
    ((= status 500)
    (let* ((errorinfo (jsown:val (jsown:parse response) "error"))
      (code (jsown:val errorinfo "code"))
      (message (jsown:val errorinfo "message")))
      (error 'rpc-error :code code :message message)))
    (t
    (error 'http-error :code status :message response))))))

(defmacro ignore-rpc-errors (&body body)
  `(handler-case
    (progn ,@body)
    (rpc-error (e)
      (values nil e))))

;;; Node RPC connection

(defclass node-rpc-connection (chain-supplier rpc-connection) ())

;;; Key RPC method implementations

(defun getblockhash (connection height)
  (do-simple-rpc-call connection "getblockhash" height))

(defun getblock (connection hash)
  ;; Second argument (0) tells Bitcoin RPC handler to return raw
  ;; hex-encoded block.
  (do-simple-rpc-call connection "getblock" hash 0))

(defun getrawtransaction (connection id)
  (do-simple-rpc-call connection "getrawtransaction" id))

```

```

(defun getchaintxstats (connection)
  (do-simple-rpc-call connection "getchaintxstats"))

;;; CHAIN-SUPPLIER interface implementation for NODE-RPC-CONNECTION

(defmethod chain-get-block-hash ((supplier node-rpc-connection) height &key
  (encoded t) errorp)
  (with-chain-supplier-normalization (height encoded errorp
    :entity-type :block-hash
    :id-type      :as-is
    :body-type    :encoded)
    (ignore-rpc-errors (getblockhash supplier height))))

(defmethod chain-get-block ((supplier node-rpc-connection) hash &key encoded
  errorp)
  (with-chain-supplier-normalization (hash encoded errorp
    :entity-type :block
    :id-type      :encoded
    :body-type    :encoded)
    (ignore-rpc-errors (getblock supplier hash))))

(defmethod chain-get-transaction ((supplier node-rpc-connection) id &key
  encoded errorp)
  (with-chain-supplier-normalization (id encoded errorp
    :entity-type :transaction
    :id-type      :encoded
    :body-type    :encoded)
    (ignore-rpc-errors (getrawtransaction supplier id))))

;;; Redefine BP:NODE-CONNECTION here for backward compatibility.
;;; These will be removed in one of the upcoming 0.0.* releases.
(in-package :bp.core)

(defclass node-connection (bp.rpc:node-rpc-connection) ())

(defmethod initialize-instance :after ((object node-connection) &rest args)
  (declare (ignore args))
  (warn "BP:NODE-CONNECTION has been deprecated in favor of ~
    BP.RPC:NODE-RPC-CONNECTION."))

```