НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК № 004.91

«До захисту допущено»
Завідувач кафедри
_____ Євгенія СУЛЕМА
«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою
«Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»
зі спеціальності 121 Інженерія програмного забезпечення
на тему: «Метод автоматизованої верифікації програмного коду смарт-контрактів»

Виконала:
студентка ІІ курсу, групи КП-01мп
Корунська Анна Михайлівна _____

Керівник:
Доцент кафедри ПЗКС, к.т.н., доцент,
Заболотня Тетяна Миколаївна _____

Консультант з нормоконтолю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри СПіСКС, к.т.н., доцент,
Петрашенко Андрій Васильович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Євгенія СУЛЕМА

«___»_____2020 р.

ЗАВДАННЯ

на магістерську дисертацію студенці

Корунській Анні Михайлівні

1. Тема дисертації «Метод автоматизованої верифікації програмного коду смарт-контрактів», науковий керівник дисертації Заболотня Тетяна Миколаївна, к.т.н., доцент, затверджені наказом по університету від «05» листопада 2021 р. № 3682-с.
2. Термін подання студентом дисертації «16» грудня 2021 р.
3. Об'єкт дослідження: процес автоматизованої верифікації програмного коду.
4. Предмет дослідження: методи та програмні засоби автоматизованої верифікації програмного коду смарт-контрактів
5. Перелік завдань, які потрібно розробити:
   - провести дослідження підходів до аналізу програмного коду смарт-контрактів;
   - розробити модифікований метод аналізу програмного коду смарт-контрактів, який дозволяє проводити верифікацію вимог, заданих користувачем;
   - проаналізувати переваги та недоліки запропонованого методу аналізу програмного коду смарт-контрактів;
   - розробити програмне забезпечення, що реалізує модифікований метод аналізу програмного коду смарт-контрактів;
6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
   - діаграма потоку даних модифікованого методу автоматизованої верифікації програмного коду смарт-контрактів;
   - структурна схема програмного модуля реалізації модифікованого методу;
   - діаграма потоку даних модуля модифікованого методу;
   - дерево проблем для стартап-проєкту за тематикою дослідження.;
   - приклад застосування модифікованого методу з використанням категоризації;
   - приклад застосування модифікованого методу без використанням категоризації;

7. Орієнтовний перелік публікацій:

- Тези доповіді “Метод верифікації програмного коду смарт-контрактів”

8. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент кафедри ПЗКС, к.т.н., доцент		

9. Дата видачі завдання «20» жовтня 2020 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	17.10.2020	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.12.2020	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	15.02.2021	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	05.04.2021	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	15.05.2021	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	15.06.2021	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу; підготовка матеріалів доповіді на конференції ПМК-2021	05.11.2021	
8.	Оформлення текстової і графічної частини магістерської дисертації	04.12.2021	

Студентка Анна КОРУНСЬКА

Науковий керівник Тетяна ЗАБОЛОТНЯ

РЕФЕРАТ

Актуальність теми. Смарт-контракти можна впевнено назвати одним з найбільш популярних та доступних для інтеграції застосувань децентралізованих технологій. Однак внаслідок того, що смарт-контракти засновані на технології блокчейн, вони мають специфічний недолік: після публікації смарт-контракту не можна змінити його логіку, що може призвести до викрадення цифрових активів, ситуацій, коли вони назавжди залишаються на смарт-контракті, тощо. Більшість розповсюджених нині інструментів аналізу програмного коду смарт-контракту концентруються на перебірках безпеки, тобто шукають відомі та описані вразливості або потенційно небезпечні конструкції, тоді як перевірку відповідності функціональним вимогам можуть не проводити зовсім, або ж обмежуватись ручними аудитами та unit-тестуванням.

Об'єктом дослідження є процес автоматизованої верифікації програмного коду.

Предметом дослідження є методи та програмні засоби автоматизованої верифікації програмного коду смарт-контрактів.

Мета роботи: підвищення ефективності верифікації функціональних вимог до роботи смарт-контракту, заданих користувачем, шляхом розроблення модифікованого методу автоматизованої верифікації програмного коду смарт-контрактів.

Методи дослідження. В роботі використовуються теорія програмних систем, теорія програмування, теорія алгоритмів.

Наукова новизна. Вперше запропоновано модифікований метод аналізу програмного коду смарт-контрактів, заснований на використанні проміжного представлення SlithIR, який відрізняється від існуючих методів верифікації додатковим етапом побудови графа станів смарт-контракту із врахуванням категоризації значущих змінних, що дозволяє

проводити верифікацію відповідності програмного коду смарт-контрактів заданим користувачем функціональним вимогам

Практична значущість. Розроблено програмне забезпечення, яке дозволяє перевіряти формально задані користувачем вимоги на основі модуля аналізу, інтегрованого в інструмент з відкритим вихідним кодом Slither.

Апробація роботи. Основні положення і результати роботи доповідалися та обговорювалися на XIV науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2021.

Структура та обсяг роботи. Магістерська дисертація складається з вступу, п'ятьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень.

У першому розділі проаналізовано властивості блокчейн-систем, а також децентралізовану мережу та протокол Ethereum разом із середовищем виконання смарт-контрактів. Обґрунтовано доцільність автоматизованого аналізу програмного коду смарт-контрактів. Описано можливі типи аналізу за такими характеристиками як середовище виконання або рівень абстракції. Наведено опис існуючих методів, а також інструментів з відкритим вихідним кодом, які реалізовують різноманітні підходи аналізу.

У другому розділі запропоновано модифікований метод верифікації програмного коду смарт-контракту, за основу якого взято метод обмеженої перевірки моделі на графі станів смарт-контракту з використанням проміжного представлення SlithIR та який передбачає побудову графу станів смарт-контракту. Метод передбачає категоризацію змінних користувачем, а також опис формальних вимог з використанням заданої категоризації з наступною перевіркою вимог за критерієм досяжності відповідної вершини в графі станів смарт-контракту.

У третьому розділі проведено аналіз технологій розроблення, а також аналіз інструменту Slither, обраного як основи для реалізації модифікованого методу. Модифікований метод верифікації програмного коду смарт-контрактів вбудовано в інструмент як аналізатор.

У четвертому розділі проведено аналіз переваг та недоліків запропонованого методу автоматизованої верифікації програмного коду смарт-контрактів. Розглянуто обмеження поточної реалізації, пов'язані з конструкціями мови Solitity, обмеженням на вимоги на здійснювану категоризацію. Описано проблему вибуху стану, з якою стикається модифікований метод.

У п'ятому розділі проаналізовано сферу розроблення децентралізованих додатків, визначені основі користувачі системи та їх основні проблеми. За виконаним аналізом побудовано бізнес-модель кінцевого продукту та розраховані фінансові показники системи.

У висновках проаналізовано отримані результати дослідження.

Робота виконана на 81 аркуші, містить 3 додатки та посилання на список використаних літературних джерел з 20 найменувань. У роботі наведено 10 рисунків та 3 таблиці.

Ключові слова: децентралізовані системи, блокчейн, Solidity, смарт-контракт, верифікація, Sliher, SlithIR.

ABSTRACT

Theme urgency. Smart contracts can be called one of the most popular and affordable applications of decentralized technologies. However, due to the fact that smart contracts are based on blockchain technology, they have a specific disadvantage: after the smart contract is deployed, its logic cannot be changed, which can lead to theft of cryptocurrency or locking assets on the smart contract forever. Most of the widely used smart contract analysis tools focus on finding security breaches, verifying lack of known vulnerabilities or potentially dangerous design patterns, while functional compliance checks may not be performed at all, or limited to manual audits and unit testing.

Object of research is the process of automated software verification.

Subject of research are methods and software tools for automated smart contracts verification.

Research objective is to increase the efficiency of functional requirements verification of the smart contracts, using user-defined specification, by developing a modified method of automated smart contracts verification method.

Research methods: The thesis uses the theory of software systems, theory of programming, theory of algorithms

Scientific novelty is that for the first time, a modified method of smart contract analysis based SlithIR intermediate representation is proposed, which differs from existing verification methods by an additional stage of building smart contract state graph.

Practical value: Developing software that verifies formally specified user requirements based on an analysis module integrated into the Slither open source tool.

Approbation: The basic points and outcomes of the research have been presented and discussed at the 14th scientific conference for students and postgraduates «Applied mathematics and computing» PMK-2021.

Structure and content of the thesis: The master thesis consists of the introduction, five chapters, conclusions and appendixes.

The introduction provides the general characteristic of work, state of art, urgency of research.

The first section analyzes the properties of blockchain systems, as well as the decentralized network and Ethereum protocol together with the smart contract execution environment. The expediency of automated analysis of the software code of smart contracts is substantiated. Possible types of analysis by characteristics such as environment or level of abstraction are described. The description of existing methods, as well as open source tools that implement various approaches to analysis.

The second section proposes a modified method of verification of the smart contract software code, which is based on the method of limited model verification on the smart contract status graph using the intermediate SlithIR representation and which involves building a smart contract status graph. The method involves the categorization of variables by the user, as well as a description of the formal requirements using a given categorization, followed by checking the requirements by the criterion of reaching the corresponding vertex in the graph of smart contract states.

The third section analyzes the development technologies, as well as the analysis of the Slither tool, chosen as the basis for the implementation of the modified method. A modified method of verifying the software code of smart contracts is built into the tool as an analyzer.

The fourth section. analyzes the advantages and disadvantages of the proposed method of automated verification of the software code of smart contracts. The limitations of the current implementation related to the constructions of the Solitity language, the limitations on the requirements for the performed categorization are considered. The problem of explosion of the state faced by the modified method is described.

The fifth section analyzes the scope of decentralized application development, identifies the basics of system users and their main problems. According to the analysis, a business model was built.

The results of the work are analyzed in the conclusions.

The appendices provide descriptive characteristics of logarithmic returns, characteristics for VaR and CVaR risk metrics, obtained using the bootstrap procedure.

The work is performed on 81 sheets, contains 3 appendices and links to a list of used literature sources from 20 titles. The paper presents 10 figures and 3 tables.

Key words: decentralized systems, blockchain, Solidity, smart contract, verification, Sliher, SlithIR.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Bitcoin – децентралізована електронна платіжна система, яка не має централізованого управління та емітентів і криптографічні методи для запобігання можливості подвійних витрат, а також захисту від несанкціонованих транзакцій. Bitcoin також започатковує зберігання даних в високореплікований базі даних, що має назву блокчейн.

ERC токен – вид взаємозамінного або невзаємозамінного токена, який використовується в блокчейн-мережах с підтримкою EVM.

EVM (Ethereum Virtual Machine) – віртуальна машина та середовище виконання смарт-контрактів, що опубліковані в мережі Ethereum.

Ethereum – криптовалюта а також відкрита, загальнодоступна розподілена обчислювальна платформа на основі технології блокчейн з підтримкою роботи смарт-контрактів.

Lean canvas – інструмент стратегічного управління для опису бізнес-моделей як нових, так і вже існуючих підприємств.

Python – інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією.

SlithIR – проміжне представлення, запропоноване та описане розробниками інструменту Slither.

Slither – інструмент статичного аналізу смарт-контрактів.

Solidity – об'єктно-орієнтована, предметно-орієнтована мова програмування, спроєктована для трансляції в байт код віртуальної машини Ethereum. Широко використовується при створенні смарт-контрактів та децентралізованих застосунків.

Абстрактне дерево синтаксису (Abstract Syntax Tree, AST) – це скінченна множина, позначене і орієнтоване дерево, в якому внутрішні вершини співставленні з відповідними операторами мови програмування, а листя з відповідними операндами.

Байт-код – машино-незалежний код низького рівня, що генерується транслятором і виконується інтерпретатором.

Блокчейн – розподілена база даних, що зберігає впорядкований ланцюжок записів (так званих блоків), що постійно довшає; дані захищено від підробки та спотворення, а кожен блок містить часову позначку, хеш попереднього блока та дані транзакцій, подані як хеш-дерево.

Валідація – доказ того, що вимоги конкретного користувача, продукту, послгуги чи системи є задоволеними, результатом валідації є висновок про безпечність та можливість використання продукту.

Верифікація – процес управління якістю, який перевіряє відповідність правилам, стандартам чи специфікаціям.

Взаємозамінні токени (Fungible token) – тип активу, що існує в рамках певної блокчейн-мережі, в якому кожна одиниця не відрізняється від інших та рівна їм по значенню.

Випуск подій (event emition) – у Solitiy механізм мови за платформи виконання смарт-контрактів, який дозволяє зберігати в логах транзакції певні дані про настання подій з довільними параметрами.

Газ – внутрішній механізм ціноутворення транзакцій в блокчейн-мережах, використовується для збільшення вартості спаму та стимуляції перерозподілу ресурсів у мережі.

Граф потоку виконання (Control Flow Graph, CFG) – множина всіх можливих шляхів виконання програми, представлених у вигляді графу.

Динамічний аналіз – аналіз програмного коду, який здійснюють під час виконання досліджуваних програм.

Категоризація змінних – приведення змінних до категорійних, коли кожній змінній присвоюють можливі групи значень, категорії.

Криптовалюта – різновид цифрової валюти, емісія та облік якої виконується децентралізованою платіжною системою повністю в автоматичному режимі (без можливості внутрішнього або зовнішнього

адміністрування). Принциповою особливістю криптовалют є збереження інформації у блокчейні.

Логіка Гоара – це формальна система з множиною логічних правил для строгого доведення коректності програм.

Невзаємозамінні токени (Non-fungible token) – це особливий тип криптовалюти, який відрізняється від більшості криптовалют, таких як Bitcoin, та багатьох мережевих або службових токенів, які за своїм джерелом є взаємозамінними. Представляє собою активи, які існують лише у власних криптосистемах.

ПЗ – програмне забезпечення.

Проблема вибуху станів (State explosion problem, SEP) – термін, який використовується для опису ефекту різкого («вибухового») зростання часової складності алгоритму при збільшенні розміру вхідних даних задачі.

Проміжне представлення – це структура даних або код, який використовується компілятором або віртуальною машиною для представлення вихідного коду.

Публікація смарт-контракту – спеціальна транзакція, що містить байт-код контракту і розміщує його за певною адресою в блокчейн-мережі.

Символьна верифікація – вид автономної верифікації, який перевіряє ПЗ шляхом перевірки його стану в різних точках виконання.

Смарт-контракт – згода щодо перерозподілу цінностей між контрагентами, яка передбачає суворе і однозначне задання умов, автоматизацію процесів виконання і мінімізацію включення довірених сторін.

Статичний аналіз – аналіз програмного коду, який здійснюють без реального виконання досліджуваних програм.

Транзакція – здійснення закінчених дій стосовно визначеного об'єкта, що переводить цей об'єкт з одного постійного стану в інший.

Фазинг (fuzzing) – техніка автоматизованого тестування програмного забезпечення, яка полягає в тому, що на вхід програми подаються недійсні, невідповідні або випадково згенеровані дані.

Форк – процес розділення програмного проєкту (зазвичай вільного) на два окремі проєкти.

Формальна верифікація – доказ відповідності або невідповідності системи певній формальній специфікації або характеристиці, що використовує формальні математичні методи. Дана верифікація може довести, що система, алгоритм або програма не містить помилок і дефектів та має певні властивості.

Фреймворк – програмне забезпечення, яке полегшує процес розроблення на об’єднання різних модулів програмного проєкту

Функціональні вимоги – це вимоги до програмного забезпечення, які описують внутрішню роботу системи, її поведінку: обчислення даних, маніпулювання даними, обробка даних та інші специфічні функції, які має виконувати система.

Хибні спрацювання (Fasle positives) – результат, який вказує на наявність заданого стану, тоді як його немає.

Час виконання (runtime) – середовище, необхідне для виконання комп'ютерної програми та доступне під час виконання комп'ютерної програми.

ВСТУП

Останнім часом у суспільстві відбуваються процеси глобальної діджиталізації та автоматизації всіх сфер життя. Поміж інших, активно розвивається сфера електронних фінансових систем, що дозволяє в реальному часі управляти власними грошовими активами та вести облік фінансів користувачів з усього світу.

Одним з найбільш помітних технологічних напрямів розвитку електронних фінансових систем є криптовалюти, тобто фінансові системи, в яких випуск та облік цифрових активів здійснюється повністю автоматично децентралізованою системою без участі будь-якого внутрішнього чи зовнішнього адміністрування. Першою криптовалютою став Bitcoin, запропонований у 2008 та запущений у 2009 році і саме мережа та криптовалюта Bitcoin принесли популярність за розповсюдили технологію блокчейн.

На сьогоднішній день технологія блокчейн широко застосовується в різноманітних публічних та приватних фінансових системах, проте широкий розголос та популярність криптовалют призвели також до появи надбудов над технологією. Смарт-контракти можна впевнено назвати одним з найбільш популярних та доступних для інтеграції застосувань децентралізованих технологій. Смарт-контракти стають ядром для різноманітних децентралізованих додатків у різних сферах, від тієї ж фінансової, до ігор або сфери верифікації авторських прав.

Однак внаслідок того, що смарт-контракти засновані на технології блокчейн, вони мають специфічний недолік: після публікації смарт-контракту не можна змінити його логіку, що може призвести до викрадення цифрових активів, ситуацій, коли вони назавжди залишаються на смарт-контракті, тощо. У випадку невідповідності поведінки смарт-контракту очікуванням, потрібно організовувати міграцію даних та

перенесення активів на новий, виправлений смарт-контракт, хоча у деяких випадках і це може виявитись неможливим.

Внаслідок такої особливості роботи із смарт-контрактами, перед публікацією смарт-контрактів в мережу блокчейн, розробники мають бути абсолютно впевненими в коректності на безпечності їхнього програмного коду.

Різноманітні методи та інструменти, які проводять аналіз програмного коду смарт-контракту до його публікації в блокчейн мережу є невід'ємним елементом процесу розроблення децентралізованих додатків. Більшість розповсюджених нині інструментів концентруються на перебірках безпеки, тобто шукають відомі та описані вразливості або потенційно небезпечні конструкції, тоді як перевірку відповідності функціональним вимогам можуть не проводити зовсім, або ж обмежуватись ручними аудитами та unit-тестуванням.

Отже, розроблення методів та інструментів, які спеціалізуватимуться на перевірці відповідності програмного коду смарт-контрактів певним функціональним вимогам, є актуальною задачею.

Для розроблення модифікованого методу автоматизованої верифікації програмного коду смарт-контрактів необхідно виконати наступні завдання:

1. Дослідити існуючі підходи аналізу програмного коду смарт-контрактів.
2. Проаналізувати можливі модифікації існуючих методів, які дозволять зробити доступними перевірки функціональних вимог відносно якомога більшої кількості сценаріїв реалізації та використання смарт-контрактів.
3. .Розробити модифікований метод аналізу програмного коду смарт-контрактів, який дозволяє проводити верифікацію вимог, заданих користувачем.

Дане дослідження пропонує модифікований метод верифікації програмного коду смарт-контрактів на основі методів статичного аналізу, який дозволяє проводити верифікацію вимог, заданих користувачем.

1. ОГЛЯД ДЕЦЕНТРАЛІЗОВАНИХ СИСТЕМ ТА МЕТОДІВ АНАЛІЗУ ПРОГРАМНОГО КОДУ

1.1. Огляд блокчейн-систем та розвитку децентралізованих рішень

Технологія розподіленого зберігання даних, відома під назвою блокчейн, з моменту свого виникнення понад десятиліття тому невпинно набирає популярності. Переважна більшість інформації про використання даної технології пов’язана із криптовалютами, проте останнім часом сфери її застосування розширились на інші галузі, що спричинило значне зростання уваги навколо технології блокчейн. Даний факт пов’язаний із виявленим потенціалом використовуваного у технології блокчейн підходу захисту кінцевого стану певної мережі або бази даних від несанкціонованих змін із використанням криптографічних методів. Таким чином, технологія блокчейн дозволяє забезпечувати надійну захищену синхронізацію та незмінність даних (з певними допущеннями) у масштабних системах із значною кількістю незалежних одна від одної сторін.

Завдяки наявності вищеописаного способу застосування технології блокчейн, її використовують для зберігання даних та їх синхронізації між певною групою користувачів без необхідності у довірі один до одного. Дане завдання досягається через збереження згрупованих у незмінні блоки даних, що пов’язані між собою спеціальними ключами, що надають використовувані у технології блокчейн методи криптографії.

Цілісність даних, що зберігаються, забезпечується шляхом перевірки значення хеша, що містить у собі кожен блок, обчислюючи його на основі попереднього блоку. Таким чином, спроба змінити один з блоків та його дані призведе до зміни значення хеша, і ланцюжок блоків буде розірвано, внаслідок чого пошкодження даних блока зможе помітити довільний

учасник системи. Дана властивість системи унеможливлює зміну даних всередині блоку та забезпечує надійність системи.

Проте, теоретична можливість зміни даних, включених до ланцюжка, не виключена. На практиці, змінювати дані вже існуючого ланцюга дійсно неможливо, але можна створити новий альтернативний ланцюг, що стане основним. Для проведення даної операції необхідно переконати більшість мережі використовувати новий альтернативний ланцюг та створювати нові блоки, беручи його за основу замість старого. Як правило, така ситуація можлива у випадку, якщо основний ланцюжок буде коротшим за альтернативний при збереженні всіх вимог протоколу. Такий тип атаки має назву атаки 51%, її проведення можливе у випадку, коли більша частина мережі об'єднається у змові про перепис даних [1]. Великі розподілені системи, такі як Bitcoin або Ethereum, захищені від такого типу атак в силу анонімності та значної географічної розподіленості їх учасників.

Технологія блокчейн дозволяє досягти таких важливих аспектів у побудові програмного забезпечення, серед яких варто виділити:

1. Цілісність історії змін бази даних.
2. Підтримка незалежної валідації даних зі сторони групи включених користувачів.
3. Мінімізація затримок синхронізації і резервного копіювання.
4. Trustless, тобто мінімальний необхідний рівень довіри до системи від користувачів завдяки мінімізації можливості людського втручання з метою шахрайства.
5. Наявність можливості проведення аудиту в реальному часі.
6. Фіксування у часі включення даних у базу.

В якості недоліків теорії блокчейн варто зазначити:

1. Захищеність інформації, що зберігається в системі, наприклад криптовалюта, залежить від безпеки та цілісності приватних ключів користувача.

2. Використання технології блокчейн накладає високу відповідальність.
3. Більшість основних публічних криптовалютних мережах береться оплата комісії за проведення транзакцій.
4. Використання даної технології може негативно вплинути на швидкодію та обчислювальну складність роботи ПЗ.

Беручи до уваги суттєвість перерахованих недоліків використання технології блокчейн, внаслідок чого її використання не завжди є доцільним. Проте, існує ряд класичних способів застосування даної технології, серед яких варто виділити [3]:

- системи голосування;
- системи зберігання медичної інформації;
- фінансові системи;
- системи верифікації унікальності прав;
- системи верифікації ланцюжків поставок;
- публічні реєстри;
- системи верифікування авторських прав;
- ігри.

Кожен з цих сценаріїв застосування має високий рівень ефективності впровадження технології блокчейн.

Виходячи з цього, можна зробити висновок, що технологія блокчейн доцільно застосовувати для протидії шахрайству, цензурі і зміні чи видаленню даних, що, в свою чергу, значно підвищує рівень довіри користувачів до розроблюваної системи.

Далі буде розглянуто інструменти, що дозволяють використовувати проаналізовані вище переваги технології блокчейн при її впровадженні у програмне забезпечення, таке як середовище виконання і мову написання смарт-контрактів, а також публікацію смарт-контрактів та фреймворк для їх тестування.

1.1.1. Мережа Ethereum

Мережа Ethereum – це загальнодоступна відкрита розподілена обчислювальна платформа на базі технології блокчейн з підтримкою роботи смарт-контрактів. Дана платформа підтримує модифікацію версії консенсусу Накамото через переходи на основі транзакцій. Ethereum посідає друге місце серед мереж за величиною капіталізації ринку одразу після мережі Bitcoin. В якості валюти мережа Ethereum використовує Ether (ETH).

Мережа Ethereum надає децентралізовану віртуальну машину Ethereum (EVM), що може виконувати смарт-контракти, використовуючи міжнародну мережу публічних вузлів.

Відмінність даної мережі від інших блокчейн-платформ, наприклад Bitcoin із Bitcoin Script, полягає в тому, що EVM дозволяє виконувати інструкції смарт-контрактів, що написані на Тюрінг-повній мові [2].

При запуску на блокчейн-платформі смарт-контракт стає схожим на окрему самостійну комп'ютерну програму, що автоматично виконується при настанні певних умов.

Смарт-контракти на блокчейні дають змогу працювати коду у точній відповідності до запрограмованого сценарію, без потенційної можливості простою, цензурування, шахрайства та іншого втручання третіх сторін. Таким чином, смарт-контракти сприяють обміну грошима, акціями, власністю та будь-яким іншим типом цінностей.

Для створення смарт-контрактів на Ethereum було розроблено наступні мови програмування:

- Solidity;
- Serpent;
- Mutan;
- LLL;
- Vyper.

Мови Serpent та Mutan вважаються застарілими та на даний момент не підтримуються.

Gas або «газ», тобто внутрішній механізм ціноутворення транзакцій, використовується для пом’якшення розподілу ресурсів у мережі та виникнення спаму.

За кожну транзакцію, яку майнерам необхідно обробити , користувач сплачує певну кількість газу, що автоматично буде придбано ним за ETH. Таким чином, користувач оплачує ресурси повних вузлів, що будуть витрачені на обробку його операції.

Інтерес до даної мережі виявляли як великі компанії (Microsoft, Acronis, IBM, банківський консорціум R3), так і невеликі бізнеси і стартапи.

1.1.2. Мова програмування Solidity

Мова Solidity – це предметно-орієнтована та об'єктно-орієнтована мова програмування, що була запропонована у серпні 2014 року Гевіном Вудом і розроблена в рамках роботи над проєктом Ethereum. Дана мова спроєктована для транслювання віртуальної машини Ethereum в байт-код. Мова Solidity широко використовується при розробці смарт-контрактів та децентралізованих застосунків.

Мова Solidity використовується для написання автоматично виконуваних смарт-контрактів для платформи Ethereum, виконання контрактів відбувається у EVM. Дана властивість дозволяє розробникам створювати самодостатні додатки, що містять власну бізнес-логіку. яка призводить до виконання зобов’язань і накладених умов та забезпечує їхню незмінність.

Після створення смарт-контракту він транслюється у байт-код для EVM та публікується в мережі Ethereum. Передбачається, що дана операція виконується шляхом надсилання транзакції із байт-кодом смарт-контракту у мережу Ethereum від імені його розробника. Крім того, користувач, який

надсилає транзакцію, зобов'язаний внести плату за публікацію смарт-контракту, використовуючи gas.

Створений смарт-контракт публікується або в основній мережі Ethereum, або в одній із додаткових текстових мереж:

- Kovan network;
- Ropsten network;
- Rinkeby network.

Перевагами Мова Solidity є:

- статична типізація;
- докладна документація;
- використання синтаксису ECMAScript;
- підтримка наслідування контрактів, в тому числі множинного наслідування;
- підтримка комплексних змінних контрактів, в тому числі довільні ієрархічні відображення (mappings) та структури;
- наявність зручних інструментів розробки та тестування смарт-контрактів – онлайн IDE Remix, фреймворк розробки та тестування контрактів Truffle, тощо;
- наявність бібліотек, що містять готові рішення (наприклад Open Zeppelin Contracts).

Недоліками даної мови можна назвати:

- особливість у використанні відображень (mappings), що не дозволяє відобразити усі записані пари ключ-значення;
- відсутність підтримки безпечної нецілочисельної арифметики
- неможливість повернення масивів даних із функцій;
- неможливість модифікації або видалення контракту після його публікації у разі, якщо така можливість не була передбачена наперед.

1.2. Задача аналізу програмного коду смарт-контрактів

Незмінюваність смарт-контракту після публікації в блокчейні дозволяє учасникам бізнес-процесу покладатись на властивість trustless, оскільки всі можливі дії, а також вже проведені транзакції видно будь-якому, тому не залишається можливості для різного роду маніпуляцій з даними або бізнес-логікою. На жаль, ця ж властивість представляє і загрозу: якщо в логіці смарт-контракту допущено помилку(випадкову або спеціально закладена на етапі розробки), після запису контракту в блокчейн її неможливо виправити. Зловмисник може скористатися вразливістю в будь-який момент для отримання вигоди або ж просто контракт не працюватиме коректно і всі активи, які йому належали, можуть бути втрачені.

Мали місце декілька показових подій, викликаних вразливостями в програмному коді смарт-контрактів. Наприклад, сумнозвісна атака на смарт-контракту TheDAO [4] призвів до втрати майже 60 мільйонів доларів. Варто сказати, що контракт The DAO перевірявся великою кількість експертів, серед яких і самі автори мови Solidity. Також можна згадати несправність в Parity Wallet, яка призвела до того, що 169 мільйонів доларів в криптовалюті ETH будуть заблоковані на смарт-контракті назавжди [5].

Таким чином, перевірка коректності коду смарт-контрактів як з точки зору безпеки, так і з точки зору відповідності бізнес-вимогам є дуже актуальною задачею, оскільки їхнє виправлення після введення коду в експлуатацію є тільки трудоємним та неймовірно дорогим, а часто просто технічно неможливим.

1.3. Огляд типів аналізу програмного коду смарт-контрактів

Враховуючи серйозність наслідків помилок та вразливостей у коді смарт-контрактів, були розроблені різноманітні методи аналізу, що відрізняються за підходами. Далі буде описано основні типи аналізу

програмного коду смарт-контрактів, а також розглянуто їхні особливості реалізації за застосування.

1.3.1. Огляд особливостей динамічного та статичного аналізу програмного коду смарт-контрактів

Методи аналізу програмного коду варіюються від статичних до динамічних.

Статичний аналіз – це процес виявлення дефектів та потенційних проблем без власне виконання програмного коду.

Переваги статичного аналізу:

1. Раннє визначення помилок та проблем, ще на етапі розробки.
2. Повне покриття програмного коду.
3. Відсутність необхідності задання вхідних даних для виконання.
4. Відсутність необхідності компілювати і запускати програмний код.
5. Добре показує себе для тих фрагментів програмного коду, що є складними для розуміння і тестування.
6. Добре показує себе при перевірках граничних ситуацій та виключень.

Варто зазначити, що статичний аналіз не може стати повною заміною тестуванню, оскільки в загальному випадку не перевіряє бізнес-логіку і специфічні функціональні вимоги програм.

Крім того, статичний аналіз має певні недоліки і обмеження:

1. Відсутність даних про вхідні значення, тобто невизначені змінні на початку роботи.
2. При зростанні кількості гілок виконання програмного коду, складність аналізу зростає експоненційно.
3. Великий об’єм даних, що описують стан програми (множини можливих значень змінних).

4. Окремі гілки виконання можуть бути довгими і важкими для аналізу (за рахунок великої кількості циклів, умовних операторів).
5. Складність аналізу рекурсивних викликів.

Динамічний аналіз коду – це спосіб аналізу програмі безпосередньо під час виконання програми на реальному або віртуальному процесорі.

Переваги динамічного аналізу:

1. Можливість проведення аналізу навіть без наявності доступу до її вихідного коду.
2. Можливість виявлення помилок, які пов’язані з доступом до пам’яті, ресурсів, тощо.
3. В більшості випадків виключається ситуація хибних спрацювань (false positives), оскільки виявлена помилка реально виникла в досліджуваній програмі і констатацією факту її наявності, а не результатом передбачення на основі аналізу моделі програми.
4. Врахування особливостей середовища виконання.

Обмеження динамічного аналізу:

1. Неможливість гарантувати повне покриття програмного коду, найімовірніше кількість програмного коду, проаналізованого методами динамічного аналізу, не буде дорівнювати ста відсоткам.
2. Складність локалізації ділянки коду, в якій відбулась помилка.
3. Залежність якості аналізу від вхідних параметрів.
4. Складність виявлення помилок логічного типу.

Також, варто звернути увагу на те, що динамічний аналіз на відміну від статичного не перевіряє весь вихідний код, а тільки стійкість програми до певного набору атак чи поведінку на заданому наборі вихідних даних.

1.3.2. Огляд підходів до аналізу смарт-контрактів за рівнем абстракції

Підходи до аналізу та формалізації смарт-контрактів за рівнем абстракції, на якому проводитиметься аналіз, можна розділити на дві великі категорії: рівень смарт-контракту та рівень програми [6].

Підходи рівня смарт-контракту передбачають аналіз його високорівневої поведінки і зазвичай не враховують технічні деталі його реалізації і виконання. Ці підходи зазвичай описують взаємодію між смарт-контрактами та зовнішніми агентами, які в основному представлені користувачами та блокчейном. Для підходів рівня смарт-контракту характерно абстрагувати представлення смарт-контракту в публічний інтерфейс контракту – набір загальнодоступних функцій, які викликаються користувачами, і проводити спостереження за результатами їхнього виконання. Такими можуть бути випуск подій (events emition) або зміни в стані блокчейну. Дані підходи розглядають смарт-контракти як чорні ящики, які приймають транзакції та повідомлення ззовні, і, можливо, виконують певні обчислення на їхній основі.

Моделі аналізу, які використовуються в підходах рівня смарт-контракту покладаються на такі концепції:

1. Користувачі – а саме їхні баланси, транзакції, які вони ініціюють, та пов’язані параметри, наприклад, облікові адреси. До уваги в аналізі доцільно брати до уваги сценарії виклику методів контракту не лише користувачами, а й іншими смарт-контрактами. Деякі підходи передбачають також певний принцип підбору виконаних транзакцій на основі обставин та інформації, яку має користувач в момент здійснення транзакції, тобто такі методи беруть до уваги можливість вибору стратегії користувачем.
2. Контракти – характеризуються набором загальнодоступних функцій, тобто інтерфейсом, а також змінами стану блокчейну,

які відбуваються в результаті викликів. Такими можуть бути, наприклад, зміна власника контракту чи балансів або випущені події. Таким чином, поведінка контракту зазвичай описується набором слідів (traces), які є кінцевими послідовностями викликів функцій, а властивості є предикатами на таких слідах.

3. Стан блокчейну – включаючи глобальні змінні, на які посилаються контракти, та змінні середовища, такі як позначки часу та номери блоків. Загалом, стан блокчейну може також включати пул майнінгу, тобто транзакції ще не обрані до обробки, і стан пам'яті середовища виконання контракту.

Підходи аналізу на рівня смарт-контракту ефективні для розгляду властивостей взаємодії смарт-контракту із зовнішнім середовищем або іншими контрактами. Вони зазвичай визначаються в термінах алгебри процесів та моделі станів і переходів.

Натомість, підходи рівня програми в основному виконують аналіз реалізації контракту (на основі вихідного коду або скомпільованого байт-коду). Такі підходи дозволяють точно оцінювати низькорівневі деталі процесу виконання смарт-контракту.

Підходи аналізу рівня програми досліджують смарт-контракт за принципом білого ящика, тобто розглядають внутрішню поведінку смарт-контрактів на основі низькорівневих даних, таких як скомпільований байт-код або вихідний код, а також артефактів аналізу, якими можуть бути абстрактне дерево синтаксису (Abstract Syntax Tree, AST), графіки потоку даних та потоку керування.

Для аналізу рівня програми до уваги доцільно використовувати наступні дані:

1. Абстрактне дерево синтаксису (AST) – спосіб представлення синтаксичної структури вихідного коду смарт-контракту (наприклад, написаного мовою Solidity) у вигляді дерева. Воно

часто використовується для виконання поверхневого синтаксичного аналізу програмного коду смарт-контрактів.

2. Байт-код (opcode, код операції) – низькорівневий формат подання, призначений для виконання в конкретному середовищі, наприклад віртуальній машині Ethereum. Оскільки байт-код отримується в результаті компіляції, він може якнайкраще відображати низькорівневі особливості виконання (наприклад, прогнозована кількість газу, що буде використана під час викликів, виключні ситуації, тощо), але в той же час втрачатиметься важлива інформація з вихідного коду.
3. Граф контролю виконання (Control Flow Graph, CFG) – представлення всіх шляхів програми, які можуть бути пройдені під час виконання у вигляді графа. Граф контролю виконання є орієнтованим графом, де інструкції програми служать вершинами, а дуги з'єднують вершину A з вершиною B, якщо можливо, що блок B виконується відразу після блоку A. Дуга також може містити умовний перехід необхідний для виконання коду вершини B. Такий граф отримується з байт-коду і часто використовується в статичному аналізі смарт-контрактів, для таких методів символьне виконання (symbolic execution) та автоматизовані верифікації (automated verification).
4. Програмні сліди (Program Traces) – послідовність низькорівневих операцій (зазвичай у байт-коді) та подій, зібраних під час роботи смарт-контракту. Ці сліди відображають точну поведінку контрактів з конкретними вхідними даними, і можуть бути використані для проведення динамічного аналізу та верифікації в реальному часі (runtime verification).

Підходи аналізу, засновані на моделях рівня програми зберігають низькорівневі деталі виконання, і, отже, широко використовуються для

пошуку вразливостей та перевірки властивостей, пов’язаних з безпекою смарт-контракту.

1.4. Огляд методів аналізу програмного коду смарт-контрактів

На основі вищеописаних підходів побудовано велику кількість методів для аналізу програмного коду смарт-контракту. Нижче оглядово розглянуто основні методи аналізу, а також основні сфери їхнього застосування.

1.4.1. Методи перевірки моделі

Метод перевірки моделі це добре зарекомендована методика автоматичної перевірки моделі системи зі скінченними станами на відповідність її специфікації [7].

Для алгоритмічного розв’язання задачі перевірки моделі, модель аналізованої програмної системи та специфікація до такої системи формулюються певною точною математичною мовою, тобто надається їхнє формальне подання. Зазвичай для специфікації апаратного та програмного забезпечення застосовується так звана темпоральна логіка – спеціальна мова подання, що дозволяє описувати поведінку системи в часі.

Важливим питанням до побудови специфікації для такого методу є повнота. Методи перевірки моделі дозволяють пересвідчитись в тому, що система задовольняє вимоги, задані специфікацією, проте надати інформацію про те, чи охоплює надана специфікація всі властивості системи, такі методи не можуть.

Популярність перевірки моделі у даній сфері, можливо, викликана придатністю формального моделювання та специфікацій для опису смарт-контрактів.

Хоча метод перевірки моделі успішно застосовується для верифікації систем кількох смарт-контрактів або користувачів [8], він має обмеження пов’язані із мовою вводу систем формальної верифікації (model checker

language), а також метод стикається з проблемою вибуху кількості станів, так званою SEP (state explosion problem).

1.4.2. Методи символьного виконання

Символьне виконання – це метод аналізу ПЗ, який дозволяє визначати, які вхідні дані викликають виконання кожної з частин програми [9].

Інтерпретатор виконує програму, приймаючи символічні значення в якості вхідних даних, замість отримування фактичних вхідних даних, як при звичайному виконанні програми. Таким чином, метод дозволяє отримувати вирази в термінах цих символів для виразів і змінних у програмі, а також описувати обмеження в термінах цих символів для можливих результатів кожної умовної гілки.

Також, методом символьної верифікації можна визначити, якими вхідними даними викликається виконання умовного шляху.

Як і більшість методів статичного аналізу, даний метод стикається с SEP, а саме з експоненціальним зростанням кількості можливих шляхів програми.

1.4.3. Методи верифікації в реальному часі (runtime verification)

Верифікація в реальному часі – це обчислювально неважкий метод перевірки, який перевіряє властивості програми під час її виконання. На відміну від методів, описаних вище, верифікація в реальному часі розглядає за раз лише один шлях виконання. В контексті аналізу смарт-контрактів термін шляху виконання часто позначає послідовність інструкцій, виконуваних блокчейн платформою, в той час як він також може включати послідовність викликів функцій або подій, що випускаються смарт-контрактом. Доступ до даних часу виконання надає динамічним методам перевірки оминати одну з головних проблем аналізу

смарт-контрактів, а саме необхідність моделювання складного середовища виконання блокчейн-систем.

Методи верифікації в реальному часі зазвичай забезпечують реактивний захист від вразливостей або появи виключних ситуацій під час виконання і можуть потенційно ідентифікувати вразливі стани, які не можуть бути досягнуті методами перевірки моделі або символьної верифікації через проблеми вибуху кількості станів або кількості шляхів.

Найчастіше практичні реалізації верифікації в реальному часі використовують виконання безпосередньо в блокчейн-мережі для проведення аналізу, що додатково дозволяє виявляти вразливі транзакції і досліджувати виконання смарт-контракту в середовищі, яке повністю або майже повністю відповідає production-версії середовища мережі виконання.

Також існують підходи до верифікації в реальному часі, які мають на меті знаходження вразливостей в програмному коді шляхом вставляння додаткового захисного коду в вихідний код смарт-контракту [10]. Однак такий підхід призводить до додаткового споживання газу під час виконання модифікованого коду смарт-контракту, а також не дозволяє перевіряти складні властивості.

Крім того, до методів верифікації в реальному часі можна віднести фазинг – подаючи на вхід функцій смарт-контракту невалідні, недійсні або випадково сгенеровані дані можна виявляти та експлуатувати вразливості. Для покриття більшої кількості шляхів іноді застосовується комбінація фазингу із символьною верифікацією чи аналізом плям (taint analysis).

1.5. Огляд інструментів аналізу програмного коду смарт-контрактів

Розглянемо найбільш розповсюджені інструменти аналізу, що застосовуються в сфері блокчейн-розробки, які реалізують описані методи аналізу.

Зазначимо, що кожен із цих інструментів має відритий вихідний код, а отже є безкоштовним для користування.

1.5.1. Огляд Mythil

Mythril – це інструмент аналізу безпеки смарт-контрактів для байт-коду EVM. Він виявляє потенційні вразливості в програмному коді смарт-контрактів, розроблених для для Ethereum, Hedera, Quorum, Vechain, Roostock, Tron та інших EVM-сумісних блокчейнів [11].

Він використовує символьну верифікацію, SMT-вирішувачі та аналіз плям для виявлення різноманітних вразливостей безпеки. Він також використовується (у поєднанні з іншими інструментами та методами) у платформі аналізу безпеки MythX.

1.5.2. Огляд VeriSol

VeriSol (Verifier for Solidity) – це дослідницький проєкт Microsoft для створення прототипу системи формальної верифікації та аналізу для смарт-контрактів, написаних мовою Solidity [12].

Він заснований на переведенні вихідного коду смарт-контрактів з Solidity на в програми на проміжній мові Boogie, з наступним застосуванням аналізу. VeriSol проводить перевірку на семантичну коректність смарт-контракту завдяки переведенню машинних інструкцій в припущення, записані формальною логікою Гоара. Таким чином, інструмент може визначити неправильні переходи стану та неправильні початкові стани у смарт-контракті.

1.5.3. Огляд Manticore

Manticore – інструмент для аналізу смарт-контрактів і бінарних файлів, який може застосовуватись для EVM-сумісного байт-коду або WASM модулів [13]. Він дозволяє проводити дослідження програми завдяки символьному підбору вхідних даних і таким чином виявляти всі

стани, яких може досягати смарт-контракт. Також інструментарій Manticore з дослідження стану програми доповнюється можливістю описувати визначені користувачами обробники на настання подій або виконання інструкцій, що дозволяє проводити більш глибокий та спеціалізований аналіз стану смарт-контракту під час виконання.

1.5.4. Огляд Slither

Slither – інструмент для аналізу смарт-контрактів, який реалізує методи статичного аналізу [14]. Він містить набір аналізаторів відомих описаних вразливостей, може надавати візуалізацію інформації стосовно деталей реалізації смарт-контракту, а також надає можливість розширення інструментарію новими модулями аналізу та візуалізації, які б визначались користувачами.

Для проведення аналізу інструмент Slither застосовує власне проміжне представлення, що має назву SlithIR, Slither Intermediate Representation, що дозволяє проводити простіший та більш точний аналіз порівняно з аналізом безпосередньо вихідного коду.

1.6. Висновки до розділу 1

В цьому розділі було розглянуто основні властивості блокчейн-систем, а також одну з найбільш відомих реалізацій такої системи: децентралізовану мережу та протокол Ethereum разом із середовищем виконання смарт-контрактів.

У розділі обґрунтовано доцільність автоматизованого аналізу програмного коду смарт-контрактів. Описано можливі типи аналізу за такими характеристиками як середовище виконання або рівень абстракції. Наведено опис існуючих методів, а також інструментів з відкритим вихідним кодом, які реалізовують різноманітні підходи аналізу.

2. МЕТОД ВЕРИФІКАЦІЇ ПРОГРАМНОГО КОДУ СМАРТ-КОНТРАКТІВ

2.1. Запропонований метод автоматизованої верифікації програмного коду смарт-контрактів

На основі проведеного аналізу було складено модифікований метод, який дозволяє організувати перевірку вимог, заданих користувачем. Запропонована модифікація включає в себе наступні кроки:

1. Категоризація змінних смарт-контракту і запис вимог для перевірки з використанням цих категорійних змінних.
2. Переведення вихідного коду смарт-контракту в проміжне представлення SlithIR і побудова графу потоку виконання.
3. Створення графу станів смарт-контракту:
   3.1. Створення вершин на основі усіх комбінаторно можливих значень категорійних змінних.
   3.2. Визначення стану смарт-контракту після ініціалізації на основі CFG конструктора і позначення відповідної вершини як точки входу.
   3.3. Виконання методу обмеженої перевірки моделі на графі потоку виконання контракту.
   3.4. Встановлення дуг на основі отриманих результатів переходів станів.
4. Аналіз виконання вимог на основі досяжності відповідної вершини у графі станів смарт-контракту.

На рис. 2.1 подано діаграму потоку даних модифікованого методу аналізу програмного коду смарт-контрактів. На схемі наочно видно, вхідні дані, а також на якому етапі формуються відповідні проміжні дані.

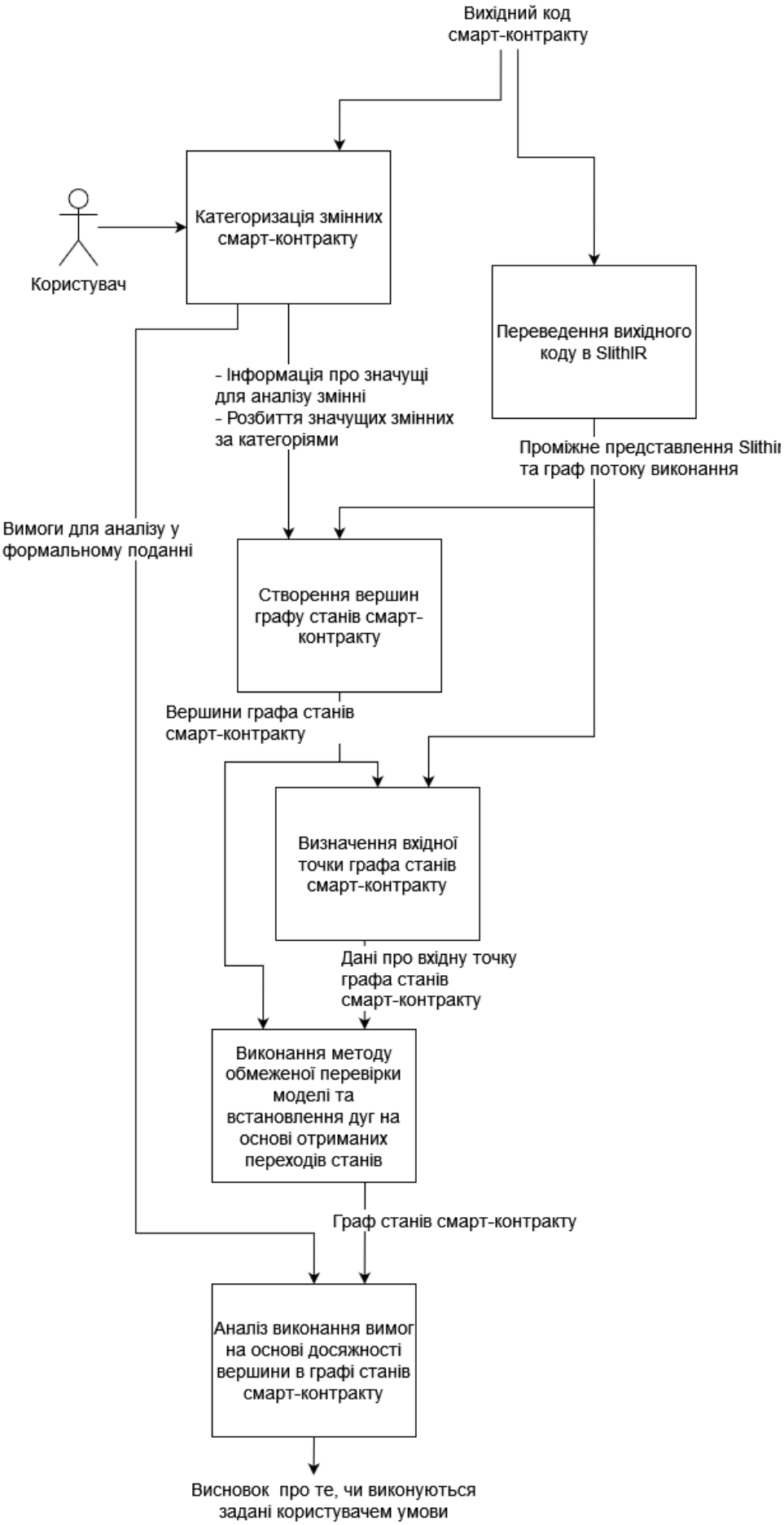


Рис. 2.1. Діаграма потоку даних методу

Детальний розбір кожного етапу роботи, теоретичне підґрунтя, а також приклади роботи для різних смарт-контрактів буде наведено у даному розділі.

2.2. Застосування категоризації змінних для запису вимог та побудови вершин графа станів смарт -контракту

На відміну від методів аналізу, орієнтованих на виявлення існуючих та описаних вразливостей безпеки, запропонована модифікація методу автоматизованої верифікації програмного коду смарт-контрактів має на меті перевірку функціональних вимог. Така зміна підходу передбачає отримання в якості вхідних даних формально заданих користувачем вимог, які б верифікувались у ході роботи модифікованого методу.

Для формулювання вимог використаємо підхід, в котрому користувач описує бажані логічні вирази з використанням назв змінних смарт-контракту і конкретних значень, або ж категорій цих значень. Для цього в модифікованому методі застосовується приведення значущих для проведення аналізу змінних смарт-контракту до категорійних.

Категорійні або дискретні змінні – це змінні, які можуть приймати значення з обмеженого або фіксованого набору значень. При цьому кожна одиниця дослідження відноситься до певної групи або категорії на підставі виконання певних вимог або за наявності деяких властивостей. [15]

Відповідно, процесом категоризації називається розбиття можливих значень певної змінної на обмежену кількість категорій, а також опис категорій розбиття.

Таким чином, для того, щоб застосувати модифікований метод автоматичної верифікації для перевірки певної функціональної властивості, користувач повинен:

1. Визначити бажану функціональну вимогу чи властивість.
2. Визначити значущі для даної властивості змінні. Для перевірки властивості можуть бути несуттєвими деякі змінні смарт-

контракту, в такому випадку користувач може виключити їх з аналізу.

3. Для кожної значущої в аналізі змінної визначити категорії, а також описати їх та подати на вхід методу. Для змінних, що не є значущими в рамках поточного аналізу, буде визначена лише одна можлива категорія – це означатиме, що в аналізі на цю змінну не потрібно зважати.
4. Власне формулювання вимоги як математичного виразу, в якості предикатів якого будуть використані назви змінних, а також назви категорій.

Важливою характеристикою задання категорій є те, що вони мають покривати всі можливі значення, які може набувати змінна цього типу. Наприклад, для змінної типу uint, тобто беззнакове ціле число, буде некоректною категоризація:

Лістинг 2.1. Приклад задання категоризації для змінної смарт-контракту

```
// @category {a} foo < 10
// @category {b} foo > 10
uint foo;
```

Така категоризація змінної foo не покриває випадок, коли значення змінної дорівнює 10.

Частковим випадком змінних, для яких не має застосовуватись ручна категоризація є змінні типу bool, оскільки вони є за визначенням дискретними і можуть набувати тільки значень true та false. Отже і при формулюванні вимог вони можуть використовуватись за значеннями, а не за категоріями, як решта змінних.

Кінцевим результатом даного етапу мають бути сформовані вершини для графу станів смарт-контракту. Кожна з таких вершин містить перелік значущих змінних, і для кожної з них визначено категорію стану. Повний перелік вершин являє собою всі комбінаторно можливі стани смарт-

контакту з врахуванням всіх можливих категорійних станів всіх значущих для аналізу змінних.

2.3. Використання проміжного представлення SlithIR

Широко розповсюдженою практикою в аналізі програмного коду смарт-контрактів є переведення їхнього вихідного або машинного коду в деяке проміжне представлення, яке б простіше піддавалося аналізу. Одним із таких проміжних представлень є SlithIR. Воно описано та реалізовано розробниками інструменту з відкритим вихідним кодом Slither, яке було розглянуто у розділі 1.

2.3.1. Використання проміжних представлень для проведення статичного аналізу програмного коду мовою Solidity

В рамках побудови мов програмування компілятори часто оперують проміжним представленням (також IR, intermediate representation), яке містить додаткові відомості про програму отримані під час обробки та розбору вихідного коду. Наприклад, компілятор створює синтаксичне дерево, яке повністю описує вихідний код програми. Однак, компілятор може також і доповнити це дерево певною метаінформацією, такою як розташування вихідних файлів, або дані про вплив на елемент потоком керування. Також, в мовах з підтримкою наслідування, як Solidity, треба враховувати, що функції та методи можуть бути визначені поза межами даного контракту. Проміжне представлення дозволяє лінеаризувати ці методи, дозволяючи додаткові перетворення та обробку вихідного коду контракту. Показовим прикладом може бути проміжне представлення LLVM у порівнянні з програмним кодом мовою C. Хоча в коді мовою C чітко прослідковується виклик функції, у ньому можуть бути відсутні деталі про стан програми, відносні шляхи, тощо. Проміжне представлення, абстрагуючи конкретні деталі команди виклику, зберігаючи при цьому змінні, стан середовища виконання та інші значення, які приводять

виконання в конкретну точку. Таким чином, LLVM може виконувати додаткову інтроспекцію отриманого коду та використовувати цей аналіз для проведення оптимізації або надавати інформацію для подальшої компіляції.

Solidity – це складна для аналізу мова з великою кількістю граничних випадків, як з точки зору синтаксису, так і з точки зору семантики. Переведення в SlithIR нормалізує велику кількість таких граничних випадків, що дозволяє проводити кращий та змістовніший аналіз програмного коду. Наприклад, граматика мови Solidity визначає операцію додавання до масиву методом push як виклик функції на об'єкті масиву. Просте представлення цієї семантики неможливо було б відрізнити від звичайного виклику функції. SlithIR, навпаки, розглядає додавання елемента до масиву як специфічну операцію, що дозволяє додатково аналізувати доступ до масивів та їхній вплив на виконання програми. Крім того, оператори в SlithIR мають ієрархію, тому, наприклад, у кількох рядках коду ви можливо відстежити всі оператори, які записують у змінну, що робить процес аналізу більш тривіальним та прозорим.

2.3.2. Особливості представлення програмного коду в SlithIR

У проміжному представленні SlithIR використовується близько 40 інструкцій [16].

Додатково проміжне представлення покладається на граф контролю виконання, для якого кожній вершині графу присвоюється код SlithIR. Нижче проаналізовано особливості представлення конструкцій мови Solidity в проміжному представленні.

Для позначення лівосторонніх (left-value) та правосторонніх (right-value) використаємо нотації LVALUE та RVALUE. LVALUE позначає присвоєння значення змінній, в той час як RVALUE використовується для позначення отримання значення змінної. Змінною може бути як

безпосередньо як змінна визначена у вихідному Solidity коді, так і змінна, створена проміжним представленням.

Присвоєння позначається через оператор «:=» і може виконуватись для правосторонніх змінних, кортежів (Tuple) та функцій (Function). Узагальнити можливі типи присвоєння можна так, як показано в лістингу 2.2.

Лістинг 2.2. Приклади присвоєнь в проміжному представленні SlithIR

```
LVALUE := RVALUE
LVALUE := Tuple
LVALUE := Function
```

Всі наявні в проміжному представленні SlithIR види бінарних арифметичних операторів можна узагальнити та представити, як показано в лістингу 2.3.

Лістинг 2.3. Приклади бінарних арифметичних операцій в проміжному представленні SlithIR

```
LVALUE = RVALUE ** RVALUE
LVALUE = RVALUE * RVALUE
LVALUE = RVALUE / RVALUE
LVALUE = RVALUE % RVALUE
LVALUE = RVALUE + RVALUE
LVALUE = RVALUE - RVALUE
LVALUE = RVALUE << RVALUE
LVALUE = RVALUE >> RVALUE
LVALUE = RVALUE & RVALUE
LVALUE = RVALUE ^ RVALUE
LVALUE = RVALUE | RVALUE
LVALUE = RVALUE < RVALUE
LVALUE = RVALUE > RVALUE
LVALUE = RVALUE <= RVALUE
LVALUE = RVALUE >= RVALUE
LVALUE = RVALUE == RVALUE
LVALUE = RVALUE != RVALUE
LVALUE = RVALUE && RVALUE
LVALUE = RVALUE -- RVALUE
```

Унарні арифметичні оператори в цій же нотації можна представити як показано на лістингу 2.4.

Лістинг 2.4. Приклади унарних арифметичних операцій в проміжному представленні SlithIR

```
LVALUE = ! RVALUE
LVALUE = ~ RVALUE
```

SlithIR має також свої особливості по відношенню до роботи з масивами та структурами. У мові Solidity звернення до масиву відбувається через розіменування вказівників. Для більшої надійності і точності аналізу, в проміжному представленні використовується особливий тип змінної REFERENCE, який зберігає результат виконання операції розіменування.

Лістинг 2.5. Приклад розіменування при зверненні по індексу масива в проміжному представленні SlithIR

```
REFERENCE -> LVALUE [ RVALUE ]
```

Також таке представлення розіменування застосовується в проміжному представленні для подання перелікових типів, структур, тощо.

Лістинг 2.5. Приклад розіменування при зверненні до member-типів в проміжному представленні SlithIR

```
REFERENCE -> LVALUE . RVALUE
REFERENCE -> CONTRACT . RVALUE
REFERENCE -> ENUM . RVALUE
```

Проміжне представлення підтримує також і оператори new та delete. При цьому оператор new має декілька зарезервованих конструкцій, кожна з яких використовується для певного типу даних:

- NEW_ARRAY використовується для створення масивів, для нього також вказується тип масиву і глибина, якщо масив має вкладення;
- NEW_CONTRACT використовується для створення нових смарт-контрактів під час виконання;

- NEW_STRUCTURE використовується для створення структур.
- NEW_ELEMENTARY_TYPE використовується для створення нових змінних елементарних типів, таких як цілі числа, булеві значення, адреси, тощо.

Подання операторів new та delete представляється наступним чином:

Лістинг 2.6. Приклад операторів new та delete в проміжному представленні SlithIR

```
LVALUE = NEW_ARRAY ARRAY_TYPE DEPTH(:int)
LVALUE = NEW_CONTRACT CONSTANT
LVALUE = NEW_STRUCTURE STRUCTURE
LVALUE = NEW_ELEMENTARY_TYPE ELEMENTARY_TYPE
DELETE LVALUE
```

Виклики функцій різних видів представляються так:

Лістинг 2.7. Приклад викликів функцій в проміжному представленні SlithIR

```
LVALUE = HIGH_LEVEL_CALL DESTINATION FUNCTION [ARG, ..]
LVALUE = LOW_LEVEL_CALL DESTINATION FUNCTION_NAME [ARG, ..]
LVALUE = SOLIDITY_CALL SOLIDITY_FUNCTION [ARG, ..]
LVALUE = INTERNAL_CALL FUNCTION [ARG, ..]
LVALUE = INTERNAL_DYNAMIC_CALL FUNCTION_TYPE [ARG, ..]
LVALUE = LIBRARY_CALL DESTINATION FUNCTION_NAME [ARG, ..]
LVALUE = EVENT_CALL EVENT_NAME [ARG, ..]
LVALUE = SEND DESTINATION VALUE
TRANSFER DESTINATION VALUE
```

Врешті, значення, яке повертає метод позначається за допомогою RETURN. RETURN None використовується для позначення такого випадку, коли метод не повертає ніякого значення.

2.3.3. Використання проміжного представлення SlithIR для визначення можливих дуг графа станів смарт-контракту

Описане проміжне представлення SlithIR включає також граф потоку виконання. Цей граф містить у вузлі певні вирази у проміжному

представленні і є придатним до обходу методом обмеженої перевірки моделі.

Метод обмеженої перевірки моделі, bounded model checking – це підвид методів перевірки моделі, основна ідея якого полягає в обході графа потоку виконання програми за k кроків таким чином, щоб виявити неприпустимий стан, який вказував би на порушення виконання формальної вимоги, заданої користувачем.

Існують різні підходи для визначення оптимального значення обмежуючої змінної для методу обмеженої перевірки моделі, проте будемо припускати, що значення k задається користувачем або є певним константним числом достатнім для повного обходу наданого графа потоку виконання.

Отже, при виконанні методу обмеженої перевірки моделі будемо зберігати стан виконання до і після виконання коду певного вузла. Якщо відбувається така зміна стану, яка переводить одну чи більше змінних в іншу категорію, відбувається створення дуги у відповідну вершину і присвоєння цій дузі даних про те, що який саме виклик спричинив перехід.

2.4. Приклади виконання модифікованого методу

Розглянемо покрокове виконання модифікованого методу і проміжні результати кожного кроку на прикладі двох смарт-контрактів.

2.4.1. Приклад застосування модифікованого методу без використання категоризації змінних

Маємо вихідний код смарт-контракту Light, реалізованого мовою Solidity.

Лістинг 2.8. Вихідний код смарт-контракту Light

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;
contract Light {
    bool flippedSwitch;
```

Продовження лістингу 2.8

```
    bool light;
    constructor() {
        flippedSwitch = false;
        light = true;
    }
    function turnOn() public {
        flippedSwitch = true;
        light = true;
    }
    function turnOff() public {
        flippedSwitch = true;
        light = false;
    }
}
```

Смарт-контракт містить дві змінні flippedSwitch та light які позначають, відповідно, чи натискався вимикач і чи ввімкнено світло. В конструкторі смарт-контракту задається його початковий стан – світло ввімкнено, а вимикач не натискався. Також маємо два методи turnOn та turnOff, які включають та виключають світло, а також позначають, що вимикач натискався, тобто присвоюють змінній flippedSwitch значення true.

Оскільки даний смарт-контракт оперує лише двома булевими змінними, застосування додаткової категоризації зі сторони користувача буде надлишковим. Причиною на це є те, що булеві змінні є часним випадком категорійних зміних, оскільки можуть набувати лише значень true або false, і отже мають дві категорії значень. Тому на вхід виконання методу користувач подає лише вимогу для перевірки. Оберемо твердженням для перевірки «Не можна вимкнути світло так, не натиснувши при цьому вимикач». Відповідно, формально задана умова буде сформульована так, як показано в лістингу 2.9.

Лістинг 2.9. Приклад подання формальних вимог до смарт-контракту Light

```
    !(flippedSwitch == false && light == false)
```

Виконаємо переведення вихідного коду смарт-контракту у проміжне представлення SlithIR:

Лістинг 2.10. Проміжне представлення SlithIR для смарт-контракту Light

```
Contract Light
        Function Light.constructor() (*)
                Expression: flippedSwitch = false
                IRs:
                        flippedSwitch(bool) := False(bool)
                Expression: light = true
                IRs:
                        light(bool) := True(bool)
        Function Light.turnOn() (*)
                Expression: flippedSwitch = true
                IRs:
                        flippedSwitch(bool) := True(bool)
                Expression: light = true
                IRs:
                        light(bool) := True(bool)
        Function Light.turnOff() (*)
                Expression: flippedSwitch = true
                IRs:
                        flippedSwitch(bool) := True(bool)
                Expression: light = false
                IRs:
                        light(bool) := False(bool)
```

Далі на основі проміжного представлення відбувається побудова графа потоку виконання, CFG. Можна його візуалізувати для зручності окремо для конструктора і для кожної публічної функції:

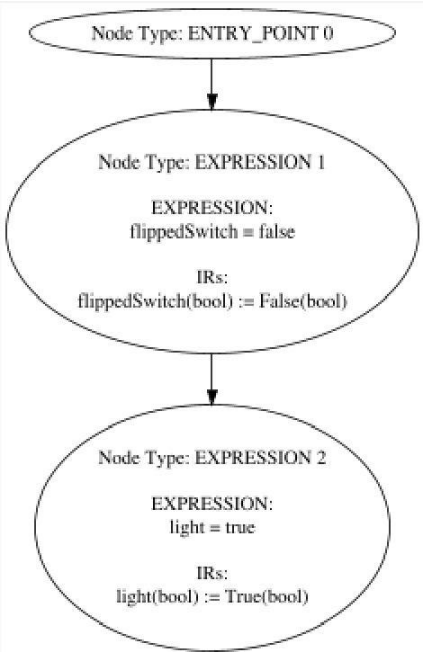


Рис. 2.1.Візуалізація графу потоку виконання для конструктора смарт-контракту

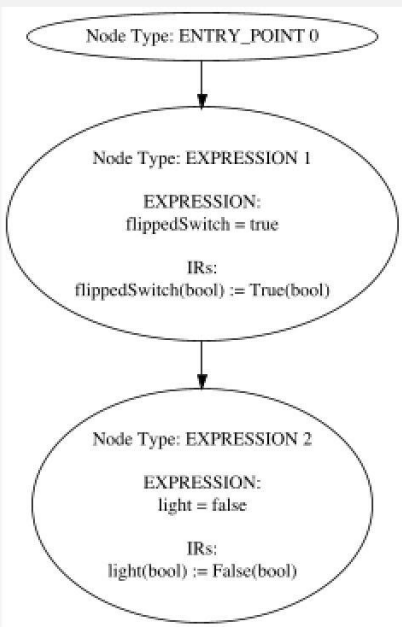


Рис. 2.2. Візуалізація графу потоку виконання для методу turnOff

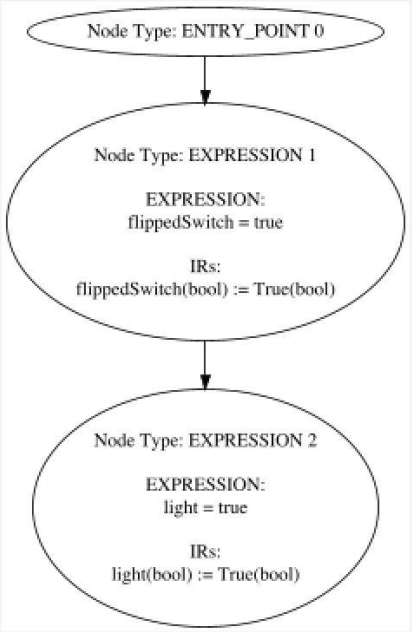


Рис. 2.3. Візуалізація графу потоку виконання для методу turnOn

Для кожного фрагменту графа потоку виконання, що відповідає функції, можемо помітити, що переходи лінійні та не мають розгалужень, що зумовлено простою логікою смарт-контракту.

Наступним кроком методу є побудова графа станів смарт-контракту. Для цього спочатку формуються всі можливі вершини, по одній на кожний

з комбінаторно можливих станів смарт-контракту. Оскільки в даному випадку маємо дві змінні із двома можливими значеннями для кожної, остаточна кількість можливих станів для смарт-контракту буде дорівнювати чотирьом.

Візуалізацію всіх чотирьох можливих станів у вигляді вершин графу станів смарт-контракту представлено на рис 2.4.

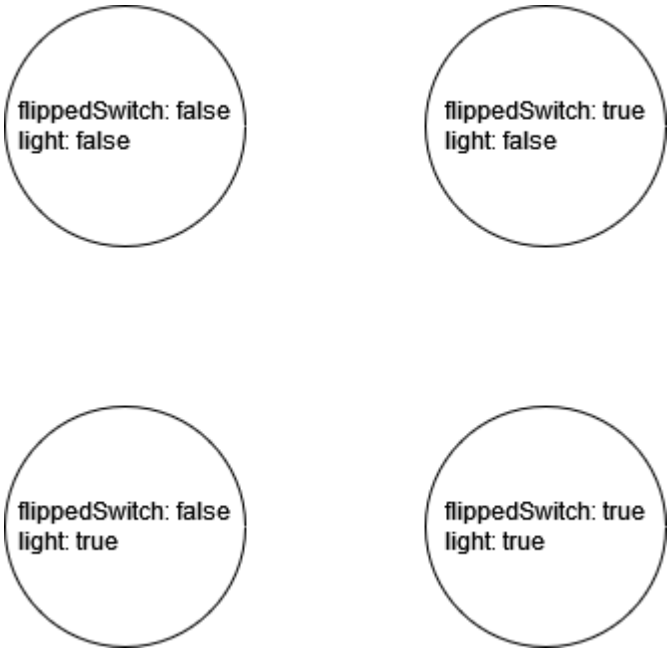


Рис 2.4. Вершини графу станів смарт-контракту

Наступний етап побудови графа станів смарт-контракту передбачає визначення точки входу на основі графу контролю виконання, що відповідає конструктору. На рис 2.5 визначену точку входу позначено блакитним.

Після цього відбувається обхід графа потоку виконання задля визначення дуг графа станів смарт-контракту. Зазначимо, що при обході ми визначатимемо зміну стану досліджуваного графу, що слугуватиме дугою і призначатимемо їй значення виклику, яке призвело до переходу.

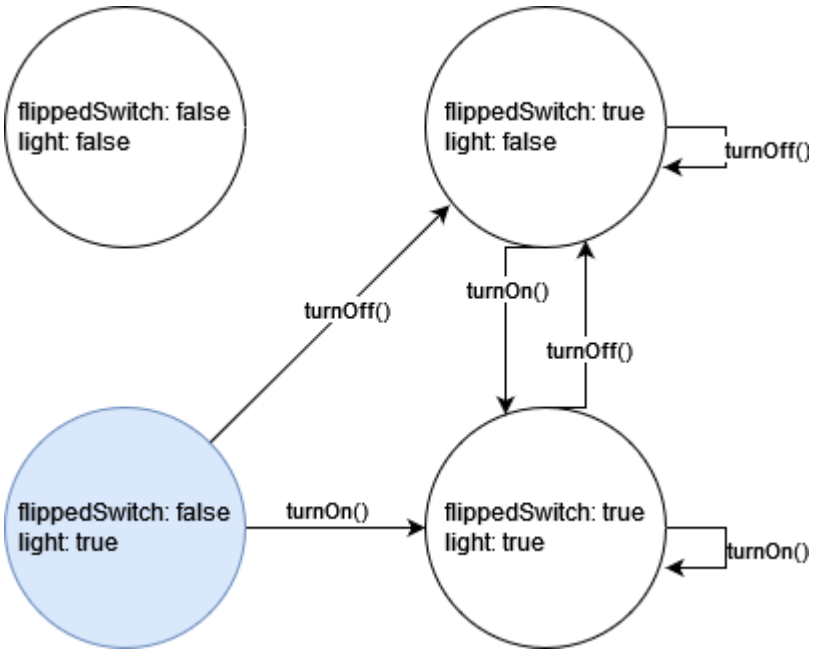


Рис 2.5. Граф станів смарт-контракту

За результатами побудованого графа можемо проводити безпосередній аналіз заданої користувачем вимоги «можна вимкнути світло так, не натиснувши при цьому вимикач». Як видно з графа, вершина, що відповідає стану, описаному вимогою, недосяжна, а отже доведено, що вимога виконується.

У випадку досяжності вершини, а отже виконання вимоги, можливим було б надання користувачеві послідовності викликів, що призвела до даного стану.

2.4.2. Приклад застосування модифікованого методу із використанням категоризації змінних

Для демонстрації роботи методу з застосуванням категоризації змінних розглянемо смарт-контракт Light2 з таким вихідним кодом:

Лістинг 2.11. Вихідний код смарт-контракту Light2

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;
contract Light2 {
    uint timesFlipped;
    bool light;
```

Продовження лістингу 2.11

```
    constructor() {
        timesFlipped = 1;
        light = false;
    }
    function turnOn() public {
        if (timesFlipped >= 3) {
            light = true;
        }
        timesFlipped += 1;
    }
    function turnOff() public {
        light = false;
        timesFlipped += 1;
    }
}
```

Смарт-контракт містить дві змінні timesFlipped та light які використовуються для позначення кількості натискань на вимикач, а також того, чи ввімкнено світло. В конструкторі присвоюються такі значення: світло вимкнено, кількість натискань дорівнює одному. Як видно з реалізації методів turnOn та turnOff, вони відповідно вмикають та вимикають світло, а також інкрементують значення змінної timesFlipped. Також метод turnOn містить умовний оператор, присвоїти змінній light значення true можна, якщо значення timesFlipped більше або дорівнює трьом.

Перед заданням умови для верифікації, користувач має присвоїти категорії для змінної timesFlipped, яка має тип unsinged integer, а отже потенційно 2^{256} можливих значень. Оскільки значущим для змінної timesFlipped в рамках поточного аналізу можна вважати значення менше та дорівнює чи більше трьох, введемо відповідні категорії big та small.

Обрану категоризацію користувач може задати коментарями, близькими по формату до jdoc, при об’явленні змінної контракту:

Лістинг 2.12. Приклад категоризації змінних для смарт-контракту Light2

```
contract Light2 {
    // @category {small} timesFlipped < 3
    // @category {big} timesFlipped >= 3
    uint timesFlipped;
```

За умову перевірки оберемо таке твердження: «Не можна ввімкнути світло так, щоб не натиснути вимикач не менш як тричі». Формальне подання умови можна сформулювати так

Лістинг 2.13. Приклад подання формальних вимог до смарт-контракту Light2

```
    !( timesFlipped == small && light == false)
```

Варто звернути увагу на те, що для формального опису умов користувач має використовувати ті самі категорії, які були присвоєнні змінним на етапі категоризації.

Виконаємо переведення вихідного коду смарт-контракту у проміжне представлення SlithIR:

Лістинг 2.14. Проміжне представлення SlithIR для смарт-контракту Light2

```
Contract Light2
        Function Light2.constructor() (*)
                Expression: timesFlipped = 1
                IRs:
                        timesFlipped(uint256) := 1(uint256)
                Expression: light = false
                IRs:
                        light(bool) := False(bool)
        Function Light2.turnOn() (*)
                Expression: timesFlipped >= 3
                IRs:
                        TMP_0(bool) = timesFlipped >= 3
                        CONDITION TMP_0
                Expression: light = true
                IRs:
                        light(bool) := True(bool)
                Expression: timesFlipped += 1
                IRs:
                        timesFlipped(uint256) = timesFlipped (c)+ 1
        Function Light2.turnOff() (*)
                Expression: light = false
                IRs:
                        light(bool) := False(bool)
                Expression: timesFlipped += 1
                IRs:
                        timesFlipped(uint256) = timesFlipped (c)+ 1
```

Візуалізацію графа потоку виконання, побудовану на проміжного представлення, приведено на рис 2.5 - 2.7.

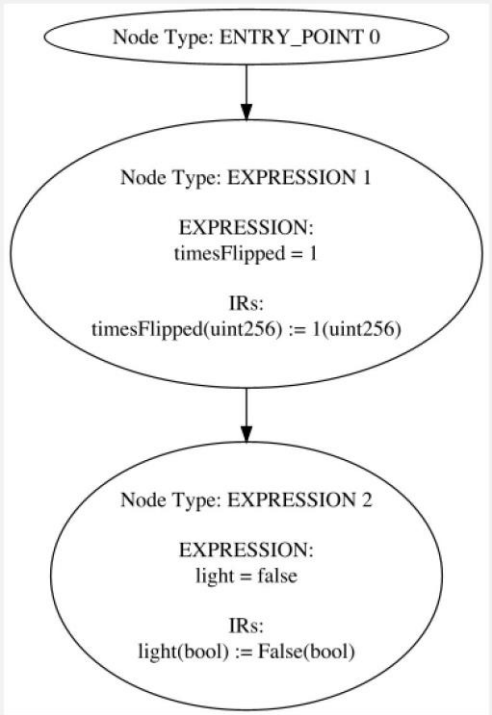


Рис. 2.5.Візуалізація графу потоку виконання для конструктора смарт-контракту

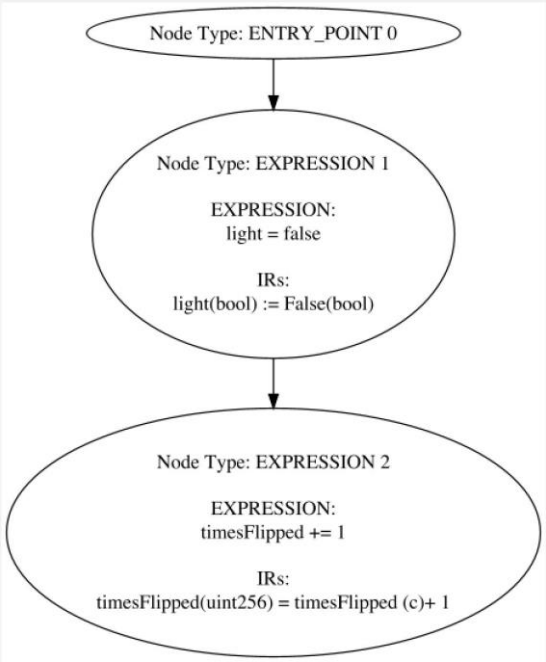


Рис. 2.6. Візуалізація графу потоку виконання для методу turnOff

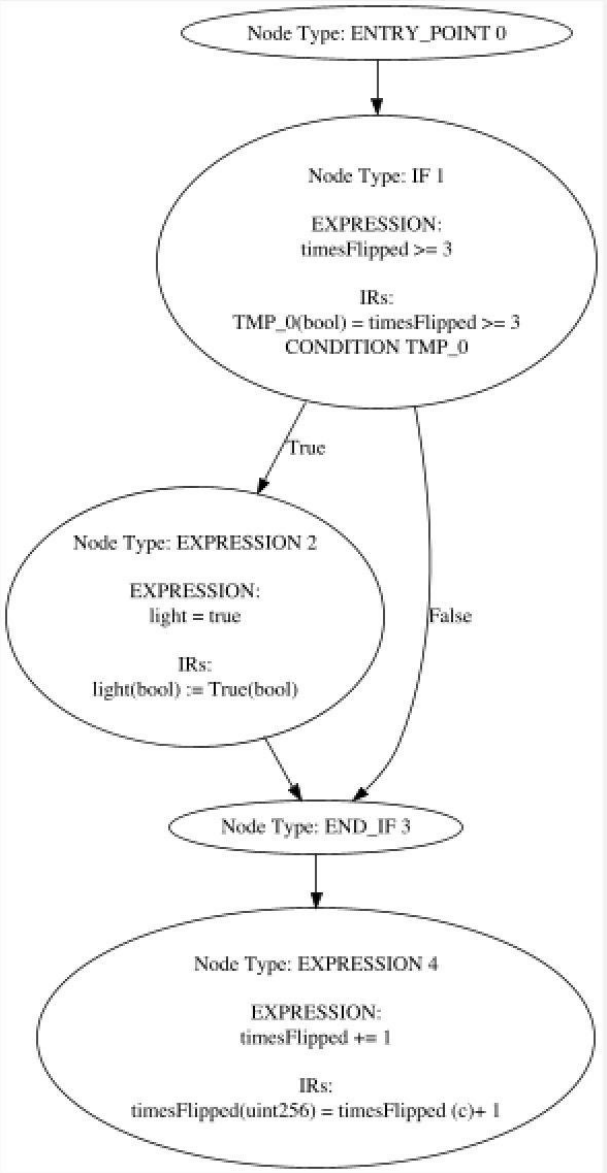


Рис. 2.7. Візуалізація графу потоку виконання для методу turnOn

Можна звернути увагу на те, як подано розгалуження умовного оператору для методу turnOn.

Для побудови вершин графу станів смарт-контракту потрібно отримати всі можливі стани смарт-контракту з оглядом на застосовану категоризацію. Оскільки одна зі значущих для аналізу змінних булева, а для другої враховуються дві категорії значень, то остаточна кількість можливих станів дорівнюватиме чотирьом. Візуалізацію чотирьох вершин можливих станів смарт-контракту наведено на рис 2.8.

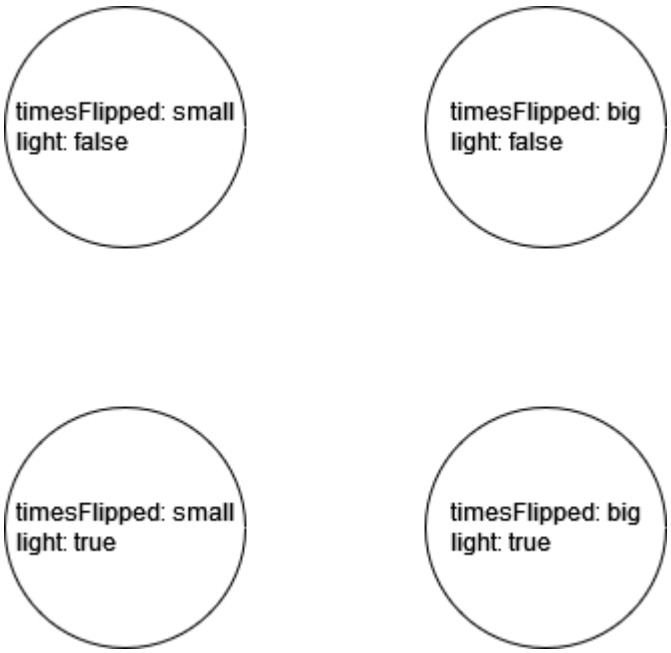


Рис 2.8. Вершини графу станів смарт-контракту

Після визначення точки входу на основі даних про вихідний стан смарт-контракту, а також виконання методу обмеженої перевірки моделі, отримуємо граф стану смарт-контракту із заповненими дугами, а також значеннями переходів викликів методів, присвоєним цим дугам.

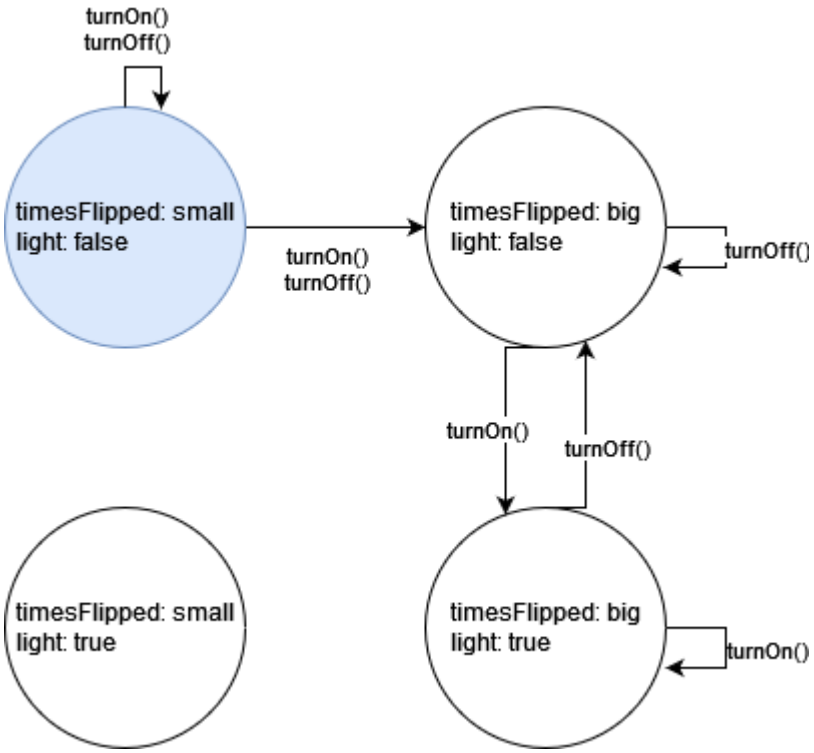


Рис 2.9. Вершини графу станів смарт-контракту

Як видно з рис. 2.9, задана для перевірки умова «Не можна ввімкнути світло так, щоб не натиснути вимикач не менш як тричі» підтвердилася, оскільки вузол, що відповідає такому стану смарт-контракту і має значення timesFlipped рівне small, a light рівне true, є недосяжним.

2.5. Висновки до розділу 2

У даному розділі сформульовано модифікований метод автоматизованої верифікації смарт-контракту, а також описано його основні кроки. Крім того, описані деталі реалізації проміжного представлення SlithIR, використаного у методі, а також розглянуте питання використання категоризації змінних, застосоване для спрощення аналізу.

У розділі також наведено приклади покрокового виконання модифікованого методу із наведенням проміжних результатів для двох смарт-контрактів. Приклади ілюструють роботу методу в разі застосування категоризації і без неї, демонструють роботу умовних конструкцій, тощо.

3. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СТАТИЧНОГО АНАЛІЗУ КОДУ СМАРТ-КОНТРАКТІВ

3.1. Обґрунтування вибору засобів реалізації ПЗ

Для розроблення програмного забезпечення, що реалізує запропонований метод автономної верифікації програмного коду смарт-контрактів, було обрано наступний стек технологій:

1. В якості основи для реалізації використано інструмент аналізу програмного коду смарт-контрактів Slihter.
2. В якості мови програмування обрано мову Python для підтримки сумісності зі Slithir та зручності інтеграції.

3.1.1. Інструмент аналізу програмного коду смарт-контрактів Slithir

Slithir – це інструмент для аналізу програмного коду смарт-контрактів, що реалізовано мовою Python. Оскільки модифікований метод аналізу програмного коду смарт-контракту використовує в якості проміжного представлення SlithIR, доцільним є практична реалізація даного метода як одного з модулів інструменту Slither.

Загальну структуру модулів інструменту Slither включно з вхідними даними подано на рис. 3.1.

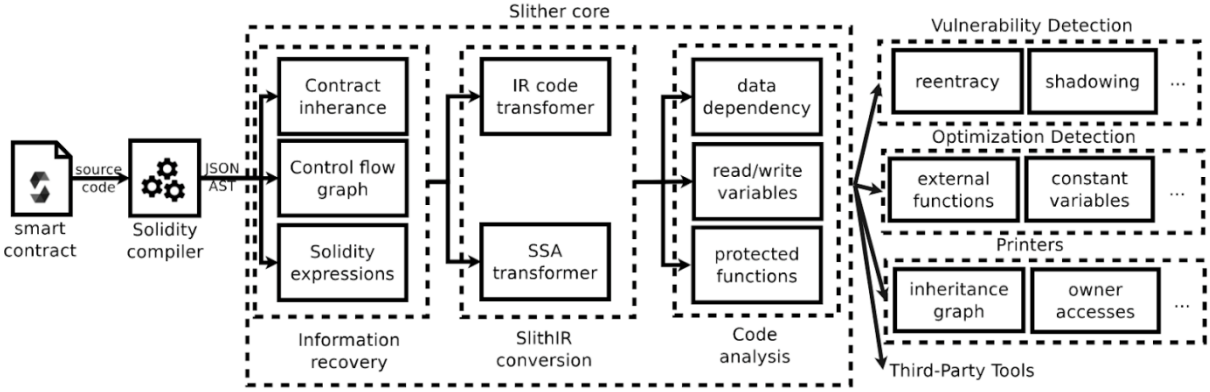


Рис. 3.1. Структура модулів Slither

Високорівнево, робота інструменту аналізу програмного коду смарт-контрактів Slither відбувається у кілька етапів [18]:

1. Slither приймає на вхід дерево абстрактне дерево синтаксису (позначене на схемі як AST), сформоване компілятором Solidity з вихідного коду контракту.
2. Slither відновлює важливу інформацію, таку як граф наслідування контракту, граф потоку керування (CFG) та перелік команд.
3. Slither перетворює весь код смарт-контракту на SlithIR, його внутрішню мову представлення.
4. Фактичний аналізу коду, Slither проводить набір визначених перевірок, які виконуються заздалегідь описаними модулями аналізу.

3.1.2. Мова програмування Python

Мова програмування Python – це інтерпретована високорівнева мова програмування, яка підтримує більшість існуючих та використовуваних на сьогодні актуальних парадигм розроблення програмного забезпечення, таких як об'єктно-орієнтоване, функціональне, процедурне, імперативне та аспектно-орієнтоване програмування. Акцент даної мови зроблено на швидкому розробленні програмного забезпечення та на підвищенні продуктивності розробника, що досягається шляхом зручності читання коду та досить простого синтаксису. Також дана мова програмування має у розпорядженні велику кількість бібліотек для вирішення задач різного типу [17].

До архітектурних рис мови програмування Python належить строга динамічна типізація, механізм обробки виключень, автоматичне керування пам'яттю, багатопоточні обчислення та повна інтроспекція часу виконання. Також дана мова підтримує механізм розбиття програми на модулі, які потім об’єднуються у пакети, та інтерактивний режим виконання коду.

Інтерпретатор для мови програмування Python, що має назву CPython, підтримується на більшості сучасних платформ, що забезпечує крос-платформність даної мови. Інтерпретатор CPython може бути розширеним функціями та типами даних, що були розроблені на будь-якій мові, котру можна викликати із мови С.

Мова програмування Python поширюється під вільною ліцензією PSFL. Це дозволяє використовувати дану мову програмування у будь-яких додатках. Тому вона широко розповсюджена для використання у великій кількості компаній з різноплановими проєктами довільного направлення. Її використовують у якості як основної мови, так і мови для розробки додаткових розширень програмних застосунків.

У якості переваг мови програмування Python можна виділити:

- зрозумілий та простий синтаксис;
- наявність багатофункціональних бібліотек, що широкий спектр засобів роботи з крос-платформними додатками та застосунками, математичними функціями та графікою;
- можливість використання на більшості відомих платформ, таких як Mac OS, UNIX, Microsoft Windows та Android;
- якісна документація до самої мови програмування та написаних нею бібліотек;
- проста інтеграція даної мови, що досягається завдяки широкому спектру можливостей керування, до яких належать виклик функцій напряму через сторонні мови С, С++ або Java. Також Python обробляє довільні мови розмітки, оскільки працює на однаковому байт-коді для всіх платформ.

Серед недоліків даної мови варто зазначити:

- відсутність статичної типізації, що забороняє реалізацію механізмів перевантаження функцій;

- низька швидкість виконання програм внаслідок інтерпретованості даної мови порівняно з компільованими мовами;
- неможливість виявлення помилок до виконання ускладнює тестування програм.

3.2. Модуль автоматизованої верифікації програмного коду смарт-контракту

Отже, модифікований метод було реалізовано програмно як один із модулів аналізу інструменту Slither. Загальну схему інтеграції в інструмент наведено на рис 3.2.

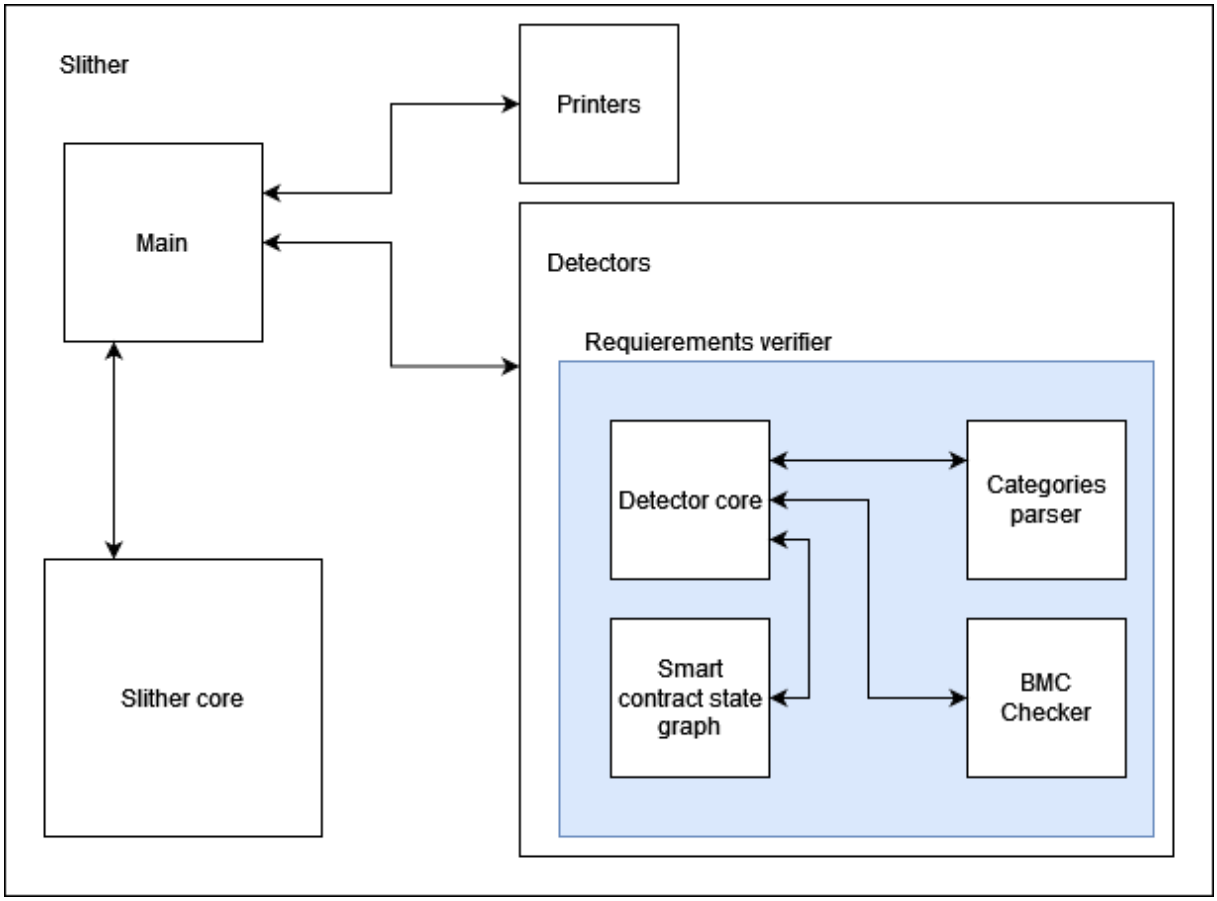


Рис 3.2. Схема інтеграції модулю модифікованого методу в інструмент Slither

Як видно зі схеми інтеграції, робота інструменту Slither покладається на декілька основних типів модулів, позначених на схемі як printers та detectors.

Модуль модифікованого методу верифікації програмного коду смарт-контрактів реалізовано як один з детекторів, що дозволяє користуватись модулем на рівні з всіма іншими доступними аналізаторами Slither.

Модуль модифікованого методу верифікації програмного коду смарт-контрактів складається з таких основних класів:

1. Detector core – основний клас, який наслідує AbstractDetector. У ньому визначені такі поля як назва аналізатору, описові тексти для виклику допомоги, тощо. Основним методом цього класу є _detect, який на приймає на вхід граф потоку виконання і повертає масив результатів. Він керує високорівневим ходом алгоритму модифікованого методу верифікації.
2. Categories parser – це допоміжний клас, який виконує збір даних про значущі для аналізу змінні, а також їхні категорії. В якості вхідних даних даний метод приймає вихідний код смарт-контракту і з коментарів позначених @category збирає задану користувачем категоризацію і динамічно будує функції визначення категорії для кожної із значущих змінних. Також в цьому класі відбуваються і парсинг формальних умов, які задаються безпосередньо у файлі з вихідним кодом смарт-контракту як коментар, позначений @requiement.
3. Smart contract state graph – допоміжний клас, який відповідає за побудову графу станів смарт-контракту. Спочатку він створює набір вершин на основі наданих змінних та категоризації, а потом встановлює між вершинами зв’язки. Він також відповідає за поточну вершину у аналізі та моніторинг зміни категорій значущих змінних досліджуваного смарт-контракту.

4. BMC checker – це допоміжний клас, в якому виконується власне метод обмеженої перевірки моделі на графі станів смарт-контракту. Наразі використане статичне значення обмежуючої змінної *k* рівне 80.

Діаграму потоку даних модуля представлено на рис. 3.3.

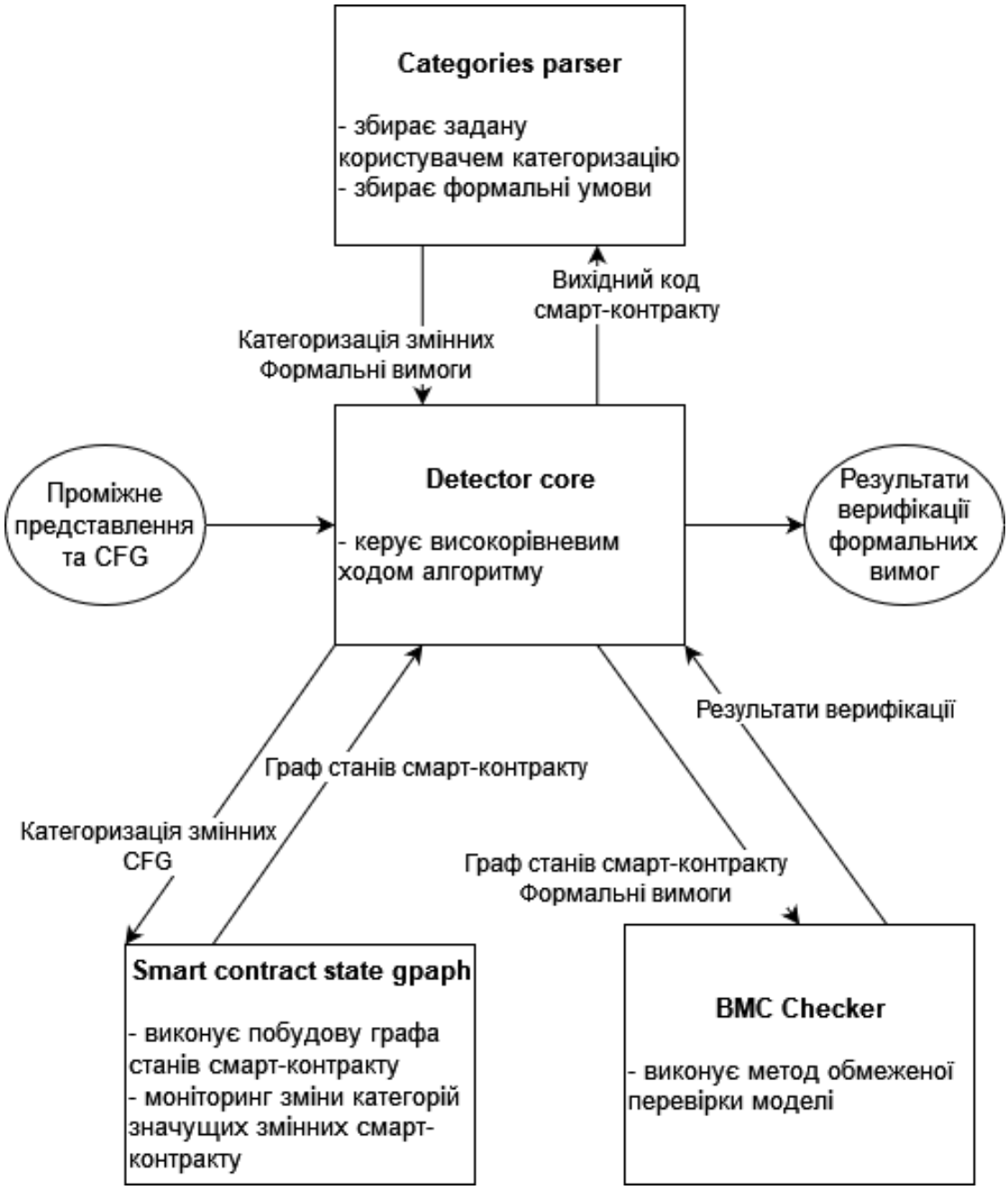


Рис. 3.3. Діаграма потоку даних модуля

3.3. Особливості розробленої програмної системи

Для запуску розробленої програмної системи необхідно встановити пакет python3 та встановити залежності проєкту за допомогою менеджера пакетів pip.

Оскільки модуль реалізовано в рамках інструменту Slither, вся інша функціональність інструменту підтримується і доступна для модуля автоматизованої верифікації. Це також передбачає доступ до модуля автоматизованої верифікації за допомогою консольного інтерфейсу, який дозволяє, наприклад, безпосередньо запускати верифікацію вимог або викликати довідку. Запустити аналізатор можна виконавши команду, як показано в лістингу 3.1. В якості параметрів необхідно передати ідентифікатор аналізатора, який відповідає реалізованому модулю автоматизованої верифікації – requirements-verifier, а також шлях до вихідного коду смарт-контракту, що буде аналізуватись.

Лістинг 3.1. Приклад виклику модуля автоматизованої верифікації

```
python -m slither path/to/Contract.sol --detect requirements-verifier
```

Також даний детектор спрацює разом з усіма іншими детекторами інструменту, якщо буде викликаний повний аналіз.

Очікується, що програмний код смарт-контракту міститиме коментарі спеціального вигляду, що вказують на вимоги для перевірки, а також задану категоризацію. Формат подання вихідного коду з відповідними коментарями наведено в лістингу 3.2.

Лістинг 3.2. Приклад формату подання категоризації змінних та вимог для перевірки

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.10;

contract Light2 {
    bool light;
```

Продовження лістингу 3.2

```
    // @category {small} timesFlipped < 3
    // @else {big}
    uint timesFlipped;

    @requirement !( timesFlipped == small && light == false)

    constructor() {
        timesFlipped = 1;
        light = false;
    }
    function turnOn() public {
        if (timesFlipped >= 3) {
            light = true;
        }
        timesFlipped += 1;
    }
    function turnOff() public {
        light = false;
        timesFlipped += 1;
    }
}
```

У наведеному прикладі застосовано спосіб подання категорії за допомогою коментарів, що починаються з спеціальних префіксів @category та @else, після у фігурних дужках наводяться назви категорій та задаються умови належності до категорії.

Після опису всіх категорій повинна бути наведена задана для перевірки функціональна вимога, задана формальною мовою. Спеціальний префікс такого коментаря @requirement, і після нього слідує математичний вираз, який представляє формалізовану функціональну вимогу. Вираз може включати в якості предикатів константи, змінні смарт-контракту, та задані для змінних категорії.

За результатами аналізу формується консольний вивід з результатами верифікації. У разі, якщо не було знайдено порушень функціональних вимог, модуль виведе лише формально сформульовані вимоги. Якщо ж заборонених вершин за результатом обходу графа станів смарт-контракту можна досягти, вивід включатиме послідовність викликів, яка призводить до порушення вимоги.

Приклад виводу програми, якщо в смарт-контракті наведено одну вимогу, яка не була порушена, наведено на лістингу 3.3.

Лістинг 3.3. Приклад виводу програми у разі відсуності порушень вимог

```
> python -m slither path/to/Contract.sol --detect requirements-verifier

!( timesFlipped == small && light == false)
```

Якщо ж порушення вимог виявлені, вивід міситиме послідовність викликів, що призвела до порушення, як наведено в лістингу 3.4.

Лістинг 3.4. Приклад виводу програми у разі виявлення порушень вимог

```
> python -m slither path/to/Contract.sol --detect requirements-verifier

!( timesFlipped == big && light == false)FAIL

turnOff()
turnOn() -> timesFlipped == big && light == false
```

3.4. Висновки до розділу 3

В даному розділі наведено аналіз технологій розроблення програмної реалізації. Проаналізовано архітектуру інструменту Slither, інтеграцію з яким було вирішено провести задля реалізації запропонованого модифікованого методу автоматизованої верифікації програмного коду смарт-контрактів, що наклало обмеження на вибір мови програмування. Внаслідок цього було вирішено використати для розроблення запропонованого модифікованого методу мову програмування Python.

Також у розділі запропонована програмна реалізація модуля модифікованого методу в якості модуля аналізу, інтегрованого в інструмент Slither. Наведено особливості реалізації, такі як необхідні для роботи програмної системи пакети, формат подання смарт-контрактів для проведення аналізу, приклади роботи.

4. АНАЛІЗ РЕАЛІЗАЦІЇ МЕТОДУ АВТОМАТИЗОВАНОЇ ВЕРИФІКАЦІЇ ПРОГРАМНОГО КОДУ СМАРТ-КОНТРАКТІВ

4.1. Переваги та недоліки запропонованого методу автоматизованої верифікації програмного коду смарт-контрактів

Основною перевагою запропонованого методу автоматизованої верифікації програмного коду смарт-контрактів безперечно можна назвати те, що запропонована модифікація дозволяє перевіряти вимоги, сформульовані користувачем, а не лише аналізувати розповсюджені вразливості. Вимоги можуть бути будь-якими логічними виразами, які включають в себе назви змінних контракту і назви визначених категорій, а також константи, окрім деяких обмежень, що будуть розглянути нижче. Це надає користувачам величезний простір для можливих перевірок функціональних вимог. Даний метод буде доцільним доповненням до широко розповсюдженого у сфері розробки програмного коду смарт-контрактів unit-тестування, оскільки доводить або спростовує вимогу як таку, а не лише у певному сценарії використання.

До переваг можна віднести також і генерацію графу станів смарт-контракту, що відбувається в процесі роботи, а також той факт, що цей граф може бути візуалізовано і надано як окремий артефакт і результат проведеного аналізу. Граф станів смарт-контракту з переходами та категоризацією може слугувати для покращення розуміння реалізації смарт-контракту, документування, тощо.

З точки зору практичної розробки можна відмітити і те, що категоризація змінних сприятиме ранньому та детальному документуванню смарт-контрактів з ранніх етапів розробки. За рахунок постійної підтримки актуальності стану категоризації змінних для аналізу та функціональних вимог, розробники та керівники проєктів можуть бути впевненими у формулюванні поточних вимог. Також результати

категоризації та сформульовані вимоги можуть бути застосовані для документування програмного коду смарт-контрактів, а також сприятимуть розумінню програмного коду всіма розробниками проєкту.

До недоліків запропонованого методу автоматизованої верифікації програмного коду смарт-контрактів можна віднести необхідність категоризація значущих змінних і формулювання вимог розробником вручну. Такий підхід є затратним по часу і може призводити до помилок категоризації, які відповідно, спричинять некоректні результати аналізу

Недоліком є і те, що вибір числа k для оптимальної роботи методу обмеженої перевірки моделі не проводиться на основі поданого на вхід програмного коду смарт-контракту або ж вимог. В k реалізації емпірично підібране значення $k = 80$, проте воно може бути надлишковим для деяких випадків і недостатнім в інших, в залежності від розмірів графа контролю виконання смарт-контракту.

Крім того, як і більшість методів формальної верифікації, запропонована модифікація має експоненціальну складність та стикається з проблемами вибуху кількості станів, що детальніше розглянуто далі у розділі.

4.2. Обмеження поточної реалізації методу автоматизованої верифікації програмного коду смарт-контрактів

Поточна реалізація методу автоматичної верифікації програмного коду смарт-контрактів має ряд обмежень, пов’язаних із різними чинниками, до яких належать складні для аналізу конструкції мови Solidity, обмеження категоризації змінних, проблема вибуху станів.

4.2.1. Конструкції мови Solidity

Поточна реалізація методу автоматичної верифікації програмного коду смарт-контрактів покладається на проміжне представлення SlithIR, яке може обробити більш як 99% контрактів мовою Solidity [18] в

контексті підтримки синтаксису мови. Втім, для спрощення поточний метод використовує деякі припущення щодо обмеженого синтаксису смарт-контрактів, що будуть аналізуватися, а також об’єму аналізованої програмної системи.

До таких конструкцій належать:

1. Системи багатьох смарт-контрактів. Ситуації взаємодії декількох смарт-контрактів не були реалізовані з декількох причин. Перша з них полягає в складності формалізації вимог для такого випадку, оскільки є природнім, що при розширенні методу на системи з міжконтрактною взаємодією, має підтримуватись і задання умов, які одночасно включають дані стану більш ніж одного контракту. Друга полягає у складності реалізації виконання методу обмеженої перевірки моделі для обходів графу стану декількох смарт-контрактів.
2. Публікація контрактів під час виконання (в рантаймі) і перевірка вимог на цих новоопублікованих контрактах. Симуляція нових контрактів і динамічне введення їхнього графу потоку виконання в поточний аналіз є складною задачею і може призводити до різноманітних сценаріїв неочевидної поведінки.
3. Підтримка умов, орієнтованих на перерозподіл активів між адресами. До таких умов можна віднести отримання або відправку токенів мережі безпосередньо методами transfer чи send. Дане обмеження пояснюється тим, що симулювання перерозподілу криптовалюти під час виконання вимагатиме додатково представлення сутностей з балансами та хоча б базового представлення зміни балансів при формулюванні дуг переходів стану для графа стану смарт-контракту. Варто зазначити, що перерозподіл активів, що представлені програмно, наприклад, токенів ERC-стандартів, як

взаємозамінних так і невзаємозамінних, буде можливим, оскільки вони представляються як змінні стану контракту.

4. Відсутня підтримка умов, пов’язаних виключно з випуском подій смарт-контрактом. Це пов’язане з тим, що сам по собі випуск подій не може змінити стан смарт-контракту, а отже і не відслідковується графом стану смарт-контракту. Отже, для підтримки перевірки подій потрібно реалізовувати паралельний аналіз і окремо задавати формат опису вимог такого характеру.

Описані вище припущення слугують для спрощення можливих випадків обробки та слугують для вдалішого прототипування, яке підтримує базис мовних конструкцій та особливих сценаріїв використання смарт-контрактів. Дані обмеження можуть бути зняті і для кожного з них реалізація методу є можливою в подальшій розробці.

4.2.2. Обмеження категоризації

Також поточна реалізація висуває і певні очікування відносно того, як буде проведення категоризація змінних для аналізу смарт-контракту, та, відповідно, як будуть побудовані функціональні умови для перевірки смарт-контракту.

Поточна категоризація проводиться вручну користувачем і не покладається на подальшу програмну обробку. Це накладає наступні припущення:

1. Категоризація, надана користувачем, має бути повною для кожної змінної, тобто покривати всі можливі значення змінної цього типу. Такого можна досягти за допомогою визначення проміжків або інших умовних конструкцій, або ж спеціальною директивою else при заданні категорії.
2. Назви категорій мають бути унікальними як в рамках задання категорій для однієї змінної, так і в рамках всієї категоризації смарт-контракту. Така вимога дозволяє уникнути

неоднозначних ситуацій при формулюванні вимог для перевірки.

3. Категоризація проведена для всіх значущих змінних в рамках даного смарт-контракту. Якщо певна змінна пропущена, буде вважатись, що вона має тільки одну категорію і тому її недоцільно включати до графа станів смарт-контракту. Відповідно, така змінна прирівнюється до константи, не впливає на граф станів, а отже і на хід аналізу.

Також варто зазначити, що категоризація змінних типу address не підтримується. Це пов’язано із тим, що поточна реалізація не має можливості симулювати наявність сутностей із балансами і адресами, а отже не можна при заповненні дуг переходів графу станів смарт-контракту використовувати виклик від імені конкретного користувача. Отже, саму по собі категоризацію провести можливо, проте вона не буде доцільною, оскільки симуляція створення транзакцій різними користувачами ще не підтримується.

4.2.3. State explosion problem

State explosion problem або проблема вибуху кількості станів це термін, що використовується для позначення ефекту «вибухового» зростання складності алгоритму зі зростанням обсягу вхідних даних. Зазвичай це означає, що складність є не поліноміальною, а експоненціальною або надекспоненціальною. Ця проблема актуальна і для методу верифікації моделі [19].

Оскільки в поточному формулюванні метод передбачає, що для розв’язування задачі перевірки вимог потрібно перевірити всі шляхи виконання програми, тобто не існує іншого способу перевірки вимоги окрім як повним перебором всіх можливих варіантів обходу. Виходить, що час, та оперативна пам’ять, потрібні для доведення функціональної вимоги пропорційні до кількості всіх можливих конфігурацій.

У розрізі застосованого підходу категоризації, на складність аналізу впливатимуть як кількість значущих для аналізу змінних, так і кількість можливих категорій, описаних для кожної змінної, оскільки вони з комбінаторних міркувань збільшують кількість вершин графа стану смарт-контракту, а також графа потоку виконання смарт-контракту.

4.3. Напрямки подальшої роботи

Поточна реалізація методу дозволяє перевіряти формальні функціональні вимоги до роботи смарт-контракту в зручному для користувача поданні. Даний метод є перспективним з теоретичної та практичної точок зору, тому доцільно розглянути можливі шляхи його покращення та подальшого дослідження.

Одним з перспективних напрямів дослідження є створення підходу автоматизованої категоризації. Поточний підхід повністю покладається за категоризацію користувача, що означає, що категоризація може бути неповною, некоректною або не найбільш оптимальною з можливих. Автоматизація категоризації для аналізу допомогла б вірити ці проблеми і зробити такий метод суттєво більш привабливим з точки зору прикладного використання. Можливим підходом для впровадження автоматизованої категоризації змінних смарт-контракту в модифікованому методі може слугувати аналіз вихідного коду смарт-контракту, можливо поєднаний із аналізом функціональних вимог, наданих для перевірки, методами машинного навчання.

Одним із можливих напрямів дослідження та розширення методу є введення додаткової категоризації змінних, що мають тип address, та створення особливого виду категорій – «акторів». Такий підхід дозволить проводити симуляції викликів транзакцій деякими окремо взятими користувачами і перевіряти сценарії доступу до певних публічних функцій смарт-контракту, що є дуже поширеним сценарієм використання.

Врешті в подальшій роботі доцільним є дослідження вибору числа k для оптимізації роботи методу обмеженої верифікації моделі при побудові графу станів смарт-контракту [20].

4.4. Висновки до розділу 4

В даному розділі детально проаналізовано переваги та недоліки запропонованого методу автоматизованої верифікації програмного коду смарт-контрактів. Розглянуто обмеження поточної реалізації, пов’язані з певними конструкціями мови Solitity, обмеженням на вимоги на здійснювану категоризацію. Описано проблему вибуху стану, з якою стикається модифікований метод.

Також у розділі приведені можливі напрями подальшої роботи над модифікованим методом.

5. СТАРТАП-ПРОЄКТ ЗА ТЕМАТИКОЮ ДОСЛІДЖЕННЯ

5.1. Опис проблеми

Кількість проєктів, які покладаються на децентралізовані технології, такі як криптовалюти, взаємозамінні або невзаємозамінні токени, невпинно зростає. Застосування блокчейн-технологій є найбільш розповсюдженим у фінансовій сфері, проте існує і велика кількість проєктів в сфері ігор, страхування, підтвердження авторських прав, тощо.

Однією з основних технологій для інтеграції децентралізованих рішень в бізнес є смарт-контракти, а особливо – смарт-контракти, що можуть виконуватись в мережах з підтримкою EVM. Найчастіше смарт-контракти для мереж з підтримкою EVM розробляються мовою Solidity. Вони мають широку технічну підтримку, велику кількість якісної документації, середовища для розробки і тестування, бібліотеки з інтеграції взаємодії із смарт-контрактами для клієнтської та серверної частини. Крім того, смарт-контракти загалом і Solidity зокрема добре зарекомендували себе як надійне рішення, що добре підходить як для невеликих проєктів, так і для технічно зрілих протоколів чи фінансових систем.

Втім, використання смарт-контрактів як однієї з основних технологій розробки проєкту тягне за собою і певні проблеми. В якості кореневої можна виділити той факт, що смарт-контракти часто є складними та неочевидними для розуміння. Не зважаючи навіть на те, що смарт-контракти зазвичай невеликі за об’ємом коду, можна стверджувати, що із зростанням кількості вихідного коду збільшується і складність для оцінки людиною поведінки такого смарт-контракту.

Також, децентралізовані додатки, які покладаються на код смарт-контрактів мають і іншу особливу проблему: неможливість зміни логіки смарт-контракту після його публікації в блокчейн. Це поглиблює проблему складності розуміння смарт-контракту, оскільки впевненість у коректності

смарт-контракту перед публікацією має бути повною, і це накладає певні обмеження на процес випуску продукту.

Найбільшою проблемою, що виходить з попередньої є можливість ситуації виявлення невідповідності смарт-контракту певним функціональним вимогам, виявлена вже після публікації в блокчейн. По-перше, сам пошук такої проблеми займає багато часу спеціалізованих розробників і виливається у витрати для бізнесу та незаплановане збільшення бюджету проєкту. По-друге, найчастіше технічно неможливо скоригувати поведінку смарт-контракту в випадку відхилень від функціональних вимог, якщо це не було передбачено на етапі проєктування, вже не кажучи про випадки суттєвого відхилення від очікуваної поведінки. Така ситуація найчастіше призводить до необхідності мігрування даних та перенесення активів на новий, виправлений смарт-контракт, хоча у деяких випадках і це може виявитись неможливим. Процедура перенесення потребує часу, великої кількості кваліфікованих ресурсів і оплати вартості газу транзакцій, що виконуватимуть перенесення даних на новий смарт-контракт. Врешті, бізнес у такому випадку несе репутаційні витрати, витрачає час через відстрочення дати запуску і втрачає у кінцевій якості продукту.

Також актуальною для бізнесу, який покладається на децентралізовані розробки буде і проблема часових витрат на документування програмного коду смарт-контрактів. Нерідкою є ситуація динамічного розвитку продукту децентралізованого додатку і документування чи формалізація вимог мають занадто низький пріоритет для того, щоб на нього регулярно виділялись кадрові ресурси. Це призводить до випадків відсутності формально сформульованих функціональних вимог до смарт-контракту, що може ще більше поглибити проблеми, описані вище.

Все вищеописане можна узагальнити за допомогою дерева проблем, зображеного на рис. 5.1.

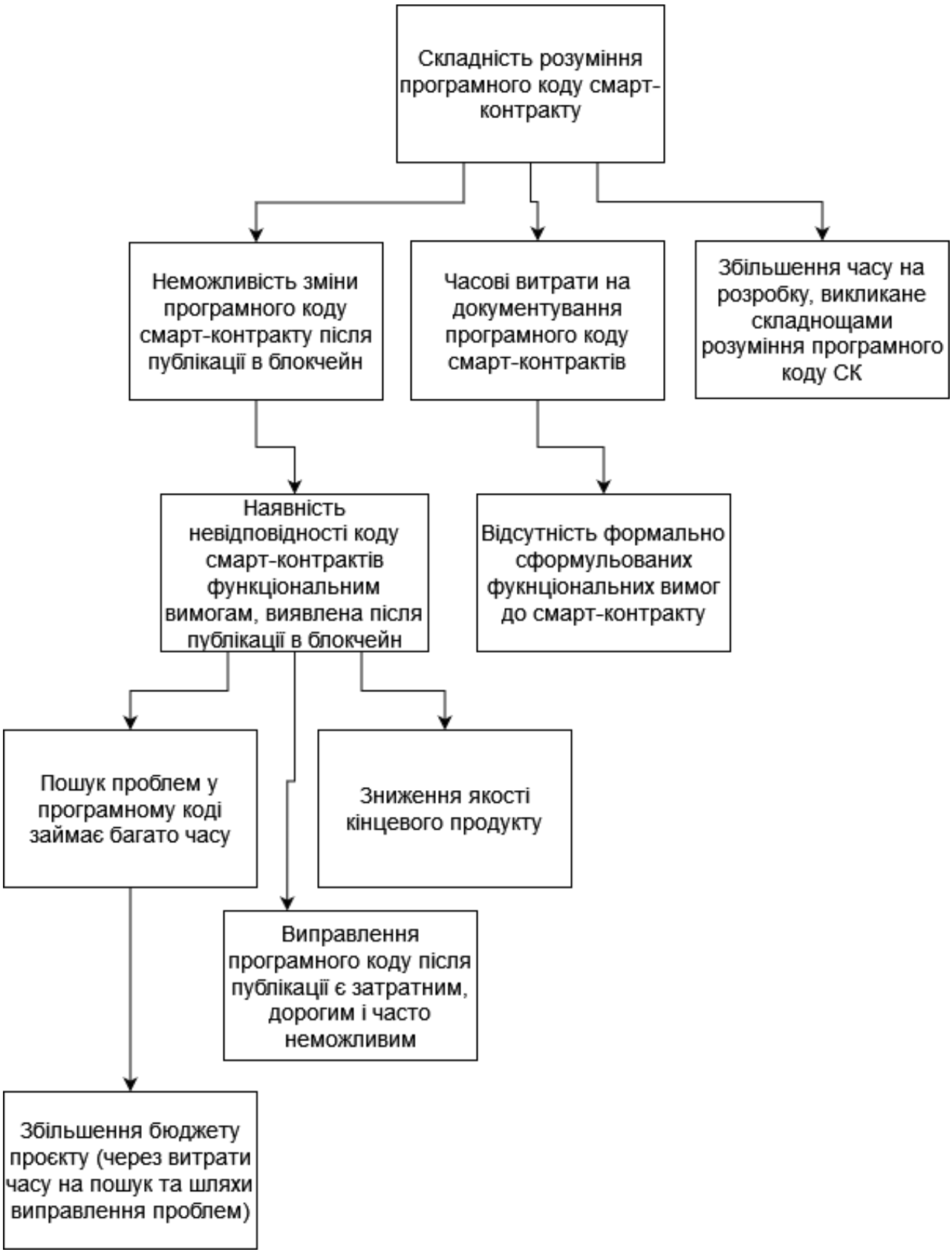


Рис. 5.1. Дерево проблем

5.2. Опис зацікавлених сторін

У вирішенні описаної вище проблеми зацікавленою стороною в першу чергу виступають компанії, що розробляють програмне забезпечення децентралізованих додатків. Для досягнення максимальної ефективності розробки децентралізованих додатків цілому і конкретно смарт-контрактів, компанії зацікавленні у використанні всіх доступних інструментів аналізу програмного коду смарт-контрактів, а особливо таких, що дозволяють перевіряти функціональні вимоги до смарт-контрактів.

Іншою зацікавленою стороною виступають окремі розробники, які бажають витрачати менше часу на перевірку відповідності коду функціональним вимогам та підвищити якість свого коду.

У табл. 5.1 зведено всі групи зацікавлених сторін, їх інтереси, та вплив (міра зацікавленості у вирішенні наявних проблем).

Таблиця 5.1

Зацікавлені сторони проєкту

Зацікавлена сторона	*Інтерес зацікавленої особи*	*Вплив зацікавленої особи*	*Стратегії приваблення зацікавлених сторін*
Компанії, що розробляють програмне забезпечення	Підвищити якість розроблюваного програмного коду і зменшити ймовірність виявлення проблем з невідповідністю коду смарт-контактів	Високий	Проведення презентації для представників зацікавлених осіб. Участь у спеціалізованих виставках, блокчейн подіях, конференціях, хакатонах.
Розробники, які бажають підвищити якість свого коду	Витрачати менше часу на перевірку відповідності коду функціональним вимогам.	Низький	

5.3. Комерційне рішення. Основні характеристики

На основі дерева проблем можна сформулювати такі вимоги до комерційного рішення:

- визначення відхилення від функціональних вимог на ранніх етапах розробки та підтримка перевірки вимог під час всього часу розробки програмного коду смарт-контрактів;
- аналіз та виділення місць відхилень від функціональних вимог у програмному коді;
- підтримка аналізу контрактів, розроблених мовою Solidity.

Розроблений продукт представляє собою програмний модуль, що представляє форк інструменту Slither з аналізатором, який реалізує модифікований метод автоматизованої верифікації програмного коду смарт-контракту, що описаний у попередніх розділах. Такий модуль може бути інтегровано з існуючими інструментами для розробників, наприклад середовищами програмування, інструментами автоматизованого тестування тощо.

5.4. Конкурентні переваги рішення

На даний час існує велика кількість різноманітних інструментів для розробників, які слугують для покращення, автоматизації чи спрощення процесу аналізу програмного коду смарт-контрактів. Деякі з цих інструментів, наприклад, середовища розробки, надають поверхневу інформацію про потенційні проблемні у коді.. Втім, дана інформація не є вичерпною, носить лише ознайомчий характер і не може повністю вирішити задачу аналізу бази програмного коду проєкту.

Найбільш популярним рішенням є статичні та динамічні інструменти аналізу ПЗ, які дозволяють перевіряти програмний код смарт-контракту на наявність вразливостей безпеки. При цьому такі перевірки перевіряють лише відомі і описані вразливості і не допомагають визначати

відповідність програмного коду смарт-контракту функціональним вимогам.

Отже, використання модифікації методу автоматизованої верифікації програмного коду смарт-контрактів, описаного в дослідженні, могло б дозволити перевіряти смарт-контракти на відповідність функціональним вимогам, що сприятиме ранньому визначенню функціональних вимог і перевірка відповідності програмного коду цим вимогам впродовж всього циклу розробки.

5.5. Клієнти. Сегменти ринку споживання

Розроблений продукт найбільше підходить для реалізації на ринку b2b. Основними потенційними клієнтами продукту є компанії, що займаються комерційною розробкою програмного забезпечення.

Продуктом можуть зацікавитись компанії, що займаються розробкою програмного забезпечення для безпосередньо аналізу програмного коду або для будь-яких інших інструментів для розроблення ПЗ в сфері децентралізованих додатків, наприклад, спеціалізованих середовищ розробки, програмних комплексів для автоматичного рефакторингу, автоматизації розробки, візуалізації виконання блокчейн-застосунків тощо.

Крім того, продукт може бути реалізовано і на ринку b2c. В такому разі клієнтами будуть окремі розробники, що хочуть отримувати більш повні дані від аналізу програмного коду розроблених ними смарт-контрактів Однак, даний сегмент не є перспективним, оскільки переважна кількість комерційних проєктів в галузі блокчейн-розробок та децентралізованих додатків розробляються спеціалізованими командами, а не окремими спеціалістами.

5.6. Унікальна ціннісна пропозиція

На основі дерева проблем було сформульовано завдання програмного продукту, а в описі зацікавлених сторін було визначено,

якими є потенційні користувачі та якими будуть їхні очікування від продукту.

Компанії з розробки програмного забезпечення, а особливо компанії, що займаються розробленням програмних систем для автоматизованого аналізу програмного коду та інструментами для розробників в сфері розробки децентралізованих додатків, потребують методів аналізу програмного коду смарт-контрактів на відповідність функціональним вимогам. Це дозволяє на ранніх етапах розробки формулювати функціональні вимоги, знаходити невідповідності на етапі їх виникнення, та під час всього циклу розроблення бути певним, що смарт-контракт відповідає заданим вимогам.

Запропонований програмний модуль, що представляє форк інструменту Slither з аналізатором, який реалізує модифікований метод автоматизованої верифікації програмного коду смарт-контракту, відповідає вищеописаним вимогам, а також вирішує проблеми користувачів та відповідає їхнім очікуванням.

Унікальною ціннісною пропозицією є програмний модуль, що дозволяє проводити аналіз програмного коду смарт-контракту на відповідність заданим функціональним вимогам.

5.7. Доходи та витрати

Дохід складається з продажів ліцензії на користування створеним програмним модулем.

Доцільним є створення декількох типів ліцензій, що покривають різні сценарії користування модулем аналізу

Наприклад, ліцензія для особистого використання дозволятиме окремим розробникам застосовувати аналіз на відповідність функціональних вимог для смарт-контрактів, що реалізовані для їхніх власних проєктів. В свою чергу, ліцензія для комерційного використання дозволяла б компаніям застосовувати аналіз на відповідність

функціональним вимогам за допомогою реалізованого модуля у проєктах, що призначені для комерційних цілей.

Основними статями витрат будуть:

- сплата податків;
- оплата юридичного консультування та оформлення патентних документів;
- заробітна плата команди розробників продукту;
- заробітна плата команди маркетингу;
- витрати на рекламу.

В таблиці 5.2. подано доходи та витрати на кожен місяць першого року роботи стартап-проєкту.

Таблиця 5.2

Доходи та витрати проєкту

	1-й місяць, т. $	*2-й місяць, т. $*	*3-й місяць, т. $*	*4-й місяць, т. $*	*5-й місяць, т. $*	*6-й місяць, т. $*
Витрати	15	15	15	15	15	15
Заплановані доходи	0	0	0	0	0	0
Результат без оподаткування	-15	-15	-15	-15	-15	-15
	7-й місяць, т. $	*8-й місяць, т. $*	*9-й місяць, т. $*	*10-й місяць, т. $*	*11-й місяць, т. $*	*12-й місяць,т. $*
Витрати	20	25	17	17	17	17
Заплановані доходи	20	30	45	50	62	75
Результат без оподаткування	0	5	28	33	35	38

Продовження табл. 5.2

Загальна сума витрат, т. $	*Загальна сума доходів, т. $*	*Загальний результат без оподаткування, т. $*
203	282	79

5.8. Бізнес-модель

Для того, щоб підбити підсумок описаного у розділі, опишемо бізнес модель у вигляді lean canvas.

Споживачі: компанії з розробки програмного забезпечення, що спеціалізуються на децентралізованих рішеннях

Проблема: за наявності в коді смарт-контракту дефектів їх неможливо виправити після публікації в блокчейн.

Рішення: програмний модуль, який реалізує модифіковану модифікований метод автоматизованої верифікації програмного коду смарт-контракту.

Унікальна ціннісна пропозиція: програмний модуль реалізує перевірку заданих розробниками формальних вимог до смарт-контракту, що робить можливим автоматизацію перевірки функціональних вимог.

Структура витрат:

- сплата податків;
- оплата юридичного консультування та оформлення патентних документів;
- заробітна плата команди розробників продукту;
- заробітна плата команди маркетингу;
- витрати на рекламу.

Канали: рекламна кампанія продукту, блокчейн-конференції та хакатони.

Ключові метрики: кількість проданих ліцензій на користування модулем.

Прихована перевага: програмний модуль постачається вже інтегрованим в інструмент розробки Slither, що дозволяє проводити перевірки на широковідомі та описані вразливості.

Таблиця 5.3

Канва бізнес-моделі в форматі lean canvas

Проблема За наявності в коді смарт-контракту дефектів їх неможливо виправити після публікації в блокчейн.	*Рішення* Програмний модуль, який реалізує метод автоматизованої верифікації програмного коду смарт-контрактів.	*Унікальна ціннісна пропозиція* Програмний модуль реалізує перевірку заданих розробниками формальних вимог до смарт-контракту, що робить можливим автоматизацію перевірки функціональних вимог.	*Прихована перевага* Програмний модуль постачається вже інтегрованим в інструмент розробки Slither, що дозволяє проводити перевірки на широковідомі та описані вразливості.
Споживачі Компанії з розробки програмного забезпечення, що спеціалізуються на децентралізованих рішеннях	*Ключові метрики* Кількість проданих ліцензій на користування модулем	*Потоки доходів* Доходи від продажу ліцензій на користування програмним модулем.	*Канали збуту* Рекламна кампанія продукту, блокчейн-конференції та хакатони.

Продовження табл. 5.3

Структура витрат - сплата податків; - оплата юридичного консультування та оформлення патентни документів; - заробітна плата команди розробників продукту; - заробітна плата команди маркетингу; - витрати на рекламу.

Зведену бізнес модель, подану в форматі lean canvas, представлено у таблиці 5.3.

5.9. Висновки до розділу 5

В розділі було проаналізовано поточну ситуацію у сфері розробки та аналізу програмного коду смарт-контрактів, а також побудовано дерево проблем. Було визначено сторони, потенційно зацікавлені у вирішенні описаних проблем, а також наведено ступені їхнього впливу.

В результаті аналізу сформульоване та описане комерційне рішення, із вказанням сегментів ринку споживання, унікальної ціннісної пропозиції, переваг та прихованих переваг, структури доходів та витрат. Всі ці елементи також подані в канві бізнес-моделі в форматі lean canvas, що дозволяє прогнозувати потенціальну прибутковість проєкту і доцільність його запуску.

ВИСНОВКИ

На основі дослідження, проведеного в даній магістерській дисертації, можна зробити наступні висновки:

1. Розглянуто основні властивості блокчейн-систем, а також децентралізовану мережу та протокол Ethereum разом із середовищем виконання смарт-контрактів. Обґрунтовано доцільність автоматизованого аналізу програмного коду смарт-контрактів. Описано можливі типи аналізу за такими характеристиками як середовище виконання або рівень абстракції. Наведено опис існуючих методів, а також інструментів з відкритим вихідним кодом, які реалізовують різноманітні підходи аналізу.
2. Запропоновано модифікований метод верифікації програмного коду смарт-контракту, за основу якого взято метод обмеженої перевірки моделі на графі станів смарт-контракту з використанням проміжного представлення SlithIR та який передбачає побудову графу станів смарт-контракту. Метод передбачає категоризацію змінних користувачем, а також опис формальних вимог з використанням заданої категоризації з наступною перевіркою вимог за критерієм досяжності відповідної вершини в графі станів смарт-контракту.
3. Проведено аналіз технологій розроблення, а також аналіз інструменту Slither, обраного як основи для реалізації модифікованого методу. Модифікований метод верифікації програмного коду смарт-контрактів вбудовано в інструмент як аналізатор.
4. Проведено аналіз переваг та недоліків запропонованого методу автоматизованої верифікації програмного коду смарт-контрактів. Розглянуто обмеження поточної реалізації, пов’язані з

конструкціями мови Solitity, обмеженням на вимоги на здійснювану категоризацію. Описано проблему вибуху стану, з якою стикається модифікований метод.

Подальшими напрямами роботи є введення спеціальної категоризації для змінних типу address, оптимізація вибору числа k для методу обмеженої перевірки моделі, а також автоматизована категоризація на основі вихідного коду смарт-контракту або заданих функціональних вимог.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Andreas M. Antonopoulos. 2014. Mastering Bitcoin: Unlocking Digital Crypto-Currencies (1st. ed.). O'Reilly Media, Inc.
2. Antonopoulos, A.M. and Wood, G. and Wood, G. 2018, Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media, Inc.
3. Кравченко П. Блокчейн і децентралізовані системи : навч. посібник у 3 ч. Ч. 1 / П. Кравченко, Б. Скрябін, О. Дубініна. – Харків : ПРОМАРТ, 2019. – 452 с.
4. Explaining the dao exploit for beginners in solidity. [Електронний ресурс] — Режим доступу до ресурсу: https://medium.com/@MyPaoG/explaining-the-dao-exploit-for-beginners-in-solidity-80ee84f0d470 , дата доступу 10.12.2021
5. Parity: The bug that put $169m of ethereum on ice? yeah, it was on the todo list for months [Електронний ресурс] — Режим доступу до ресурсу: https://www.theregister.com/2017/11/16/parity_flaw_not_fixed, дата доступу 10.12.2021
6. Tolmach, Palina & Li, Yi & Lin, Shang-Wei & Liu, Yang & Li, Zengxiang. (2021). A Survey of Smart Contract Formal Specification and Verification. ACM Computing Surveys. 54. 1-38. 10.1145/3464421.
7. S. Alqahtani, X. He, R. Gamble, and P. Mauricio. 2020. Formal verification of functional requirements for smart contract compositions in supply chain management systems. In Proc. of HICSS.
8. Clarke, Edmund & Klieber, William & Nováček, Miloš & Zuliani, Paolo. (2012). Model Checking and the State Explosion Problem. 10.1007/978-3-642-35746-6_1.
9. Clarke, E.; Biere, A.; Raimi, R.; Zhu, Y. (2001). "Bounded Model Checking Using Satisfiability Solving". Formal Methods in System Design. 19: 7–34. doi:10.1023/A:1011276507260.

10. A. Li, J. A. Choi, and F. Long. 2020. Securing smart contract with runtime validation. In Proc. of ACM PLDI. ACM, 438–453
11. ConsenSys: Mythril: a security analysis tool for ethereum smart contracts. months [Електронний ресурс] — Режим доступу до ресурсу: https://github.com/ConsenSys/mythril-classic, дата доступу 11.12.21
12.. Wang, S. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer. 2019. Formal specification and verification of smart contracts for Azure blockchain. [Електронний ресурс] — Режим доступу до ресурсу: https://www.microsoft.com/en-us/research/publication/formal-specification-and-verification-of-smart-contracts-for-azure-blockchain/, дата доступу 11.12.21
13. M. Mossberg et al., "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts," 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1186-1189, doi: 10.1109/ASE.2019.00133.
14. Alex Groce, Josselin Feist, and Gustavo Greico. “Slither: A Static Analysis Framework For Smart Contracts,” 2019. doi:10.1109/wetseb.2019.00008.
15. Bishop Y. M. M., Fienberg S. E., Holland P. W. Discrete Multivariate Analysis: Theory and Practice. — MIT Press, 1975. — ISBN 978-0-262-02113-5.
16. SlithIR [Електронний ресурс] — Режим доступу до ресурсу: https://github.com/crytic/slither/wiki/SlithIR, дата доступу 11.12.21
17. Python 3.8.3 documentation [Електронний ресурс]. – Режим доступу до ресурсу: https://docs.python.org/3/ , дата доступу 11.12.21
18. Slither, the Solidity source analyzer [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/crytic/slither, дата доступу 15.12.21
19. Clarke E.M., Klieber W., Nováček M., Zuliani P. (2012) Model Checking and the State Explosion Problem. In: Meyer B., Nordio M. (eds) Tools for

Practical Software Verification. LASER 2011. Lecture Notes in Computer Science, vol 7682. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35746-6_1

20. Biere, Armin & Cimatti, Alessandro & Clarke, Edmund & Strichman, Ofer & Zhu, Yunshan. (2003). Bounded Model Checking. Advances in Computers. 58. 117 - 148. 10.1016/S0065-2458(03)58003-2.

ДОДАТКИ

Додаток 1

Копії графічних матеріалів

Діаграма потоку даних модифікованого методу

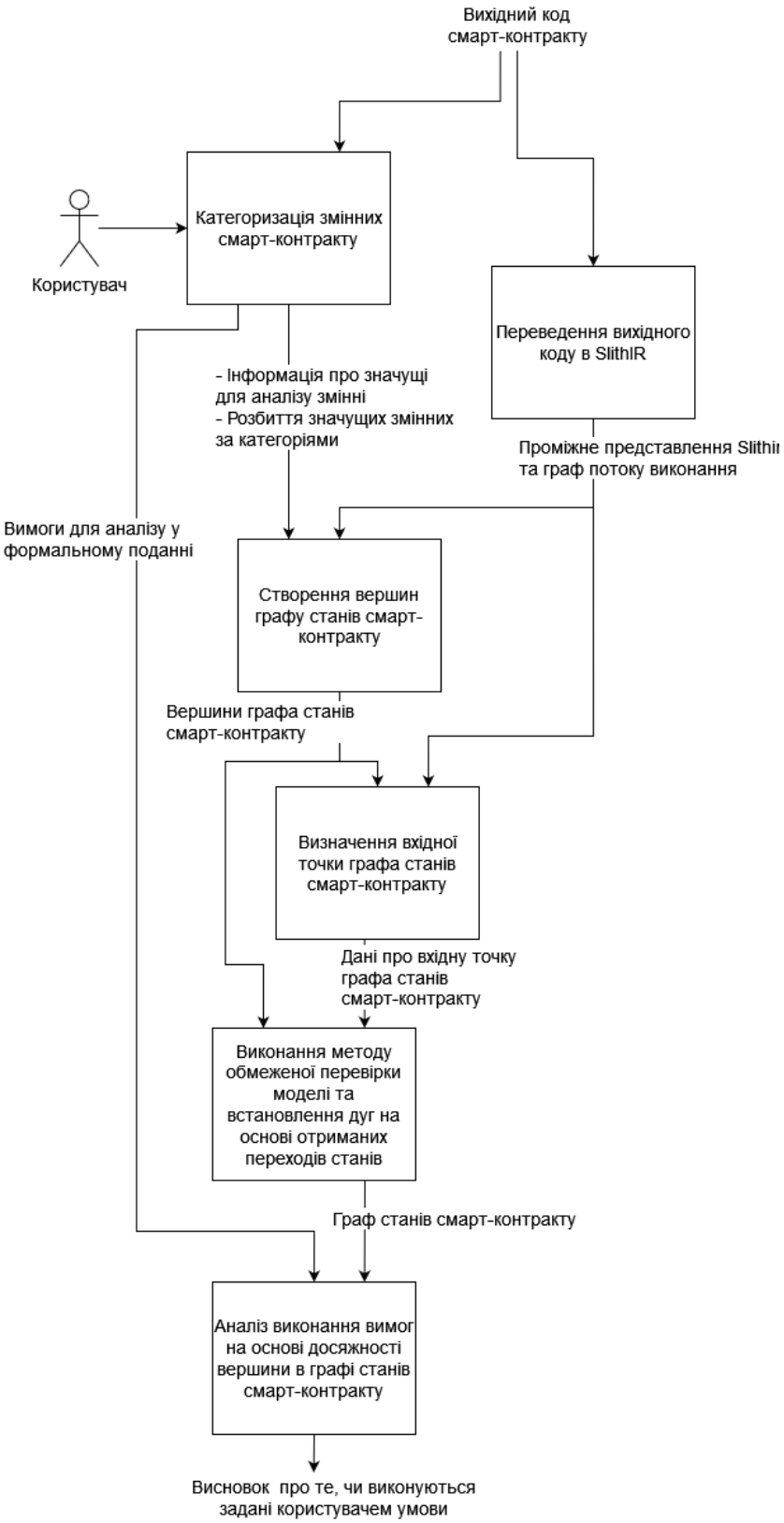
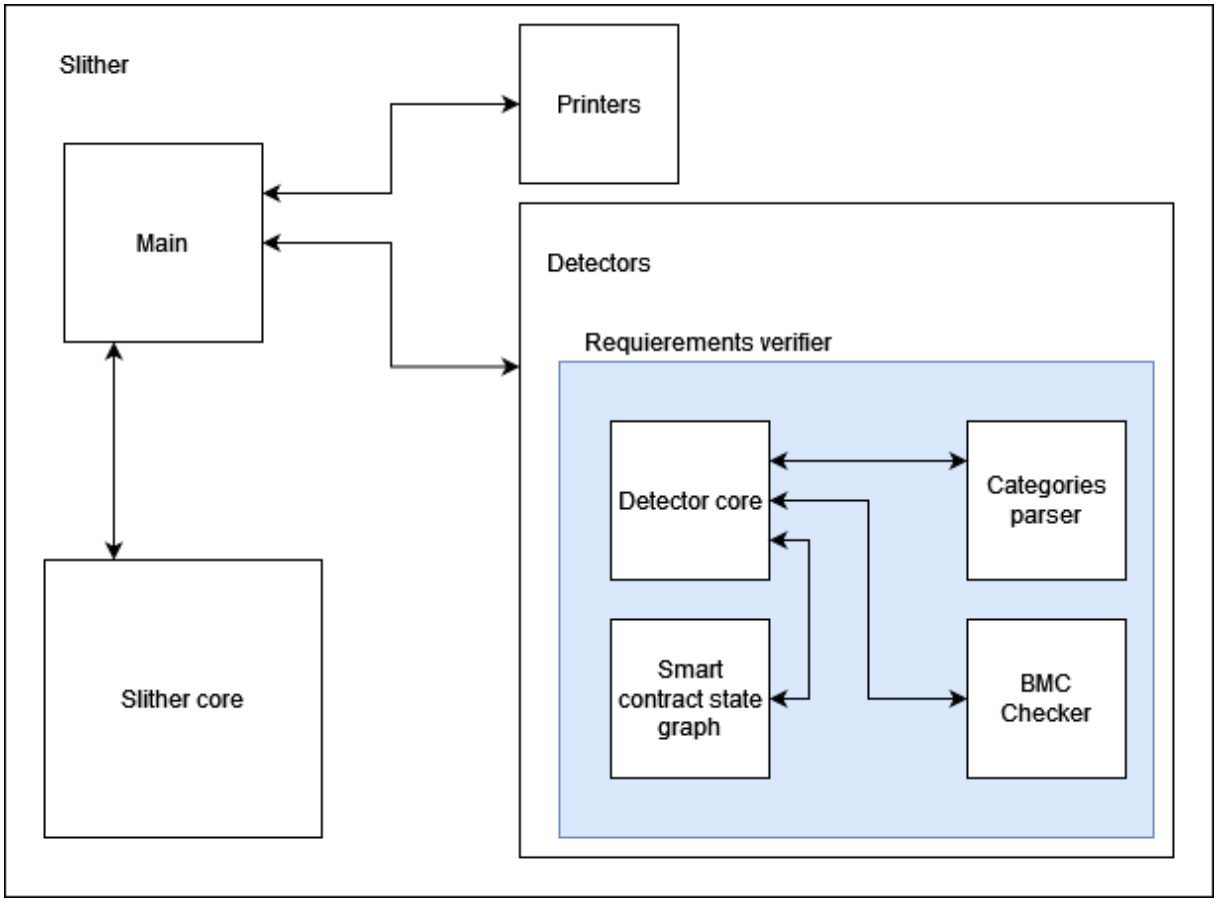
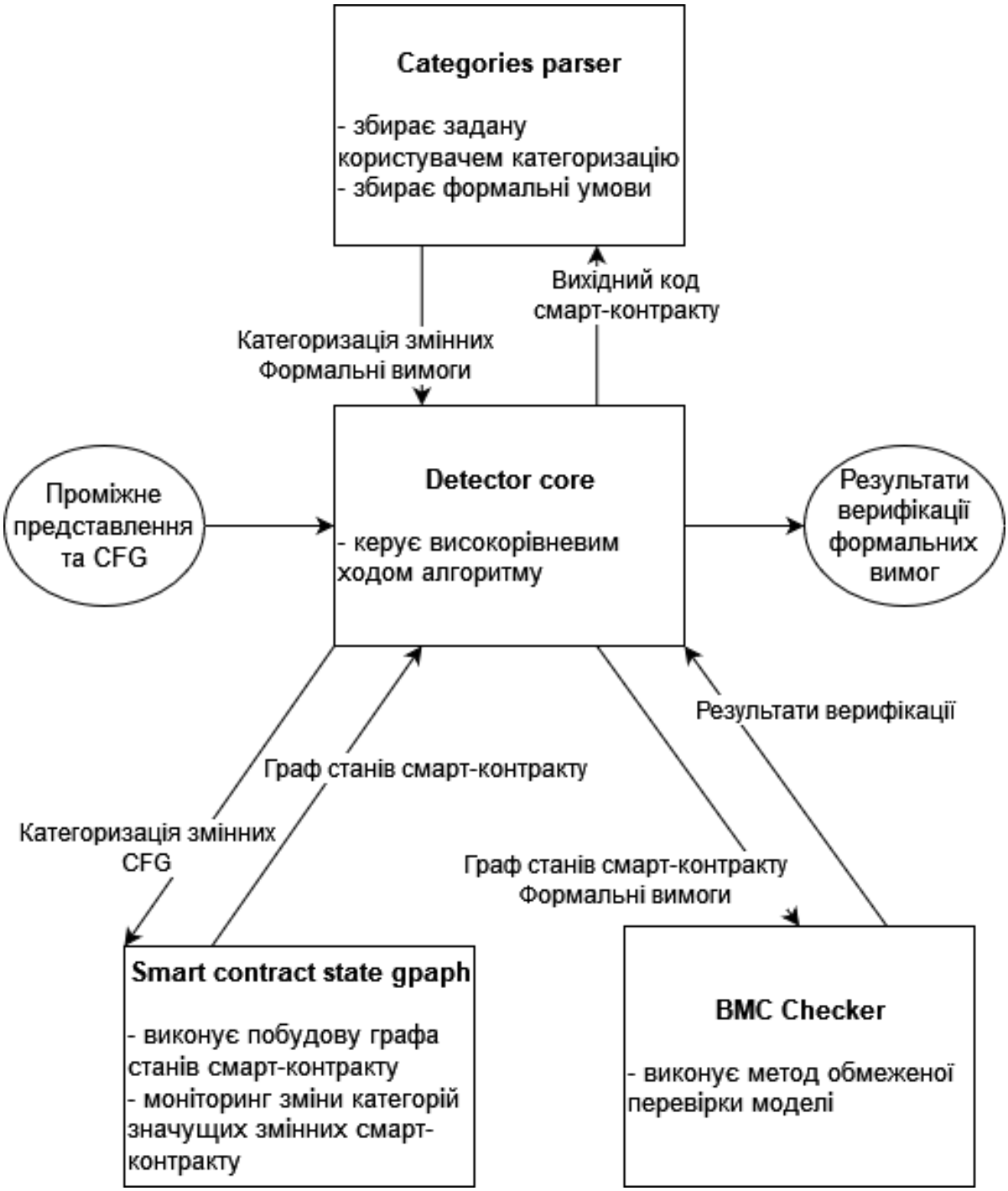


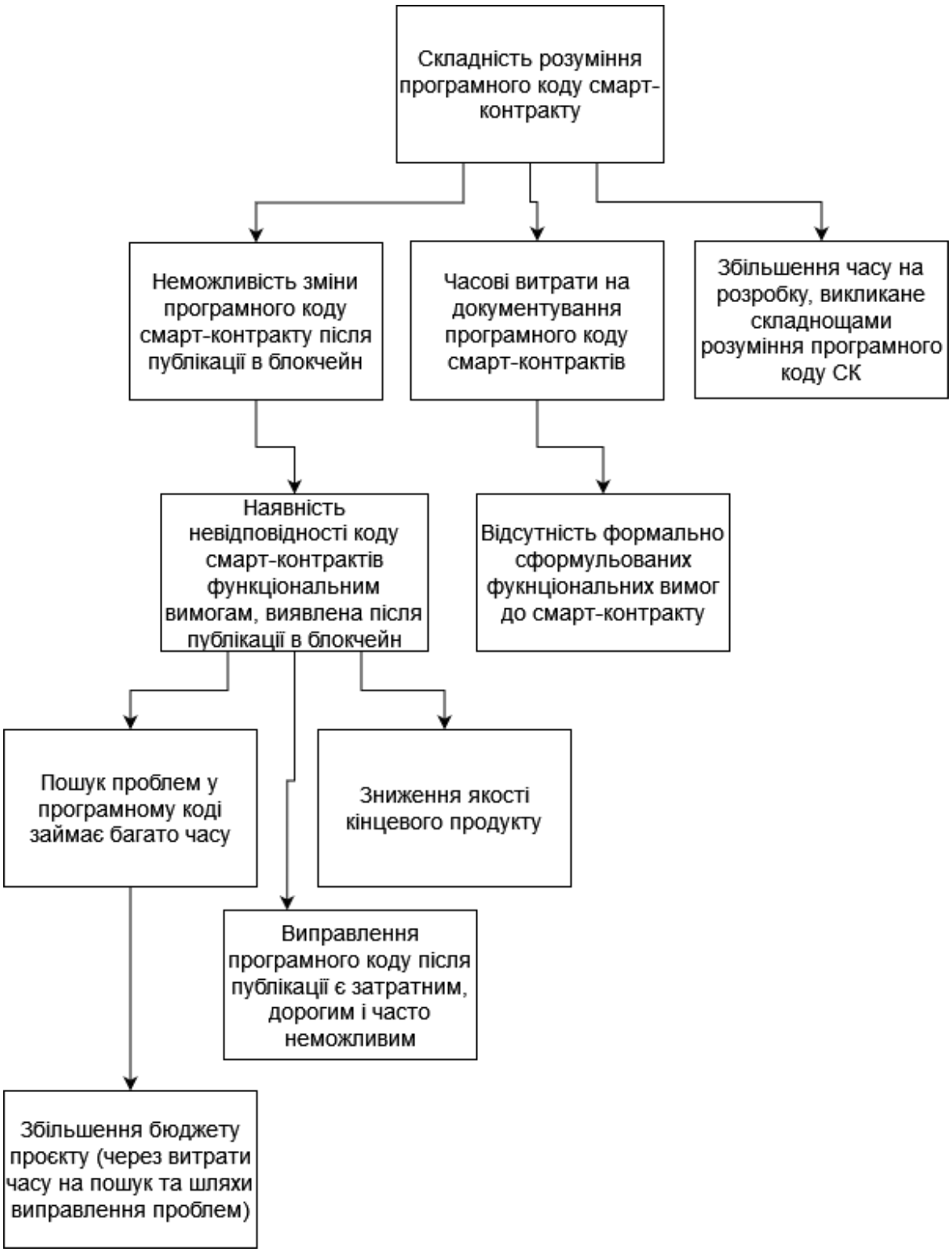
Схема інтеграції модуля модифікованого методу в інструмент Slither



Діаграма потоку даних модуля модифікованого методу верифікації програмного коду смарт-контрактів

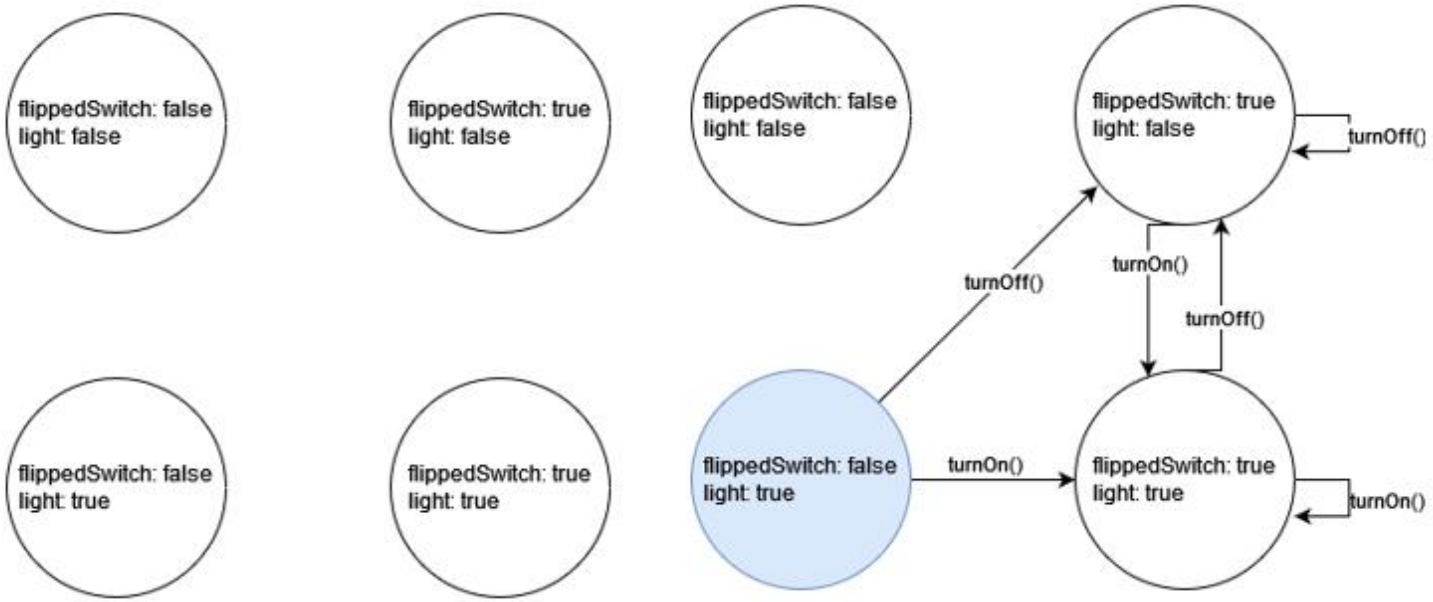


Дерево проблем



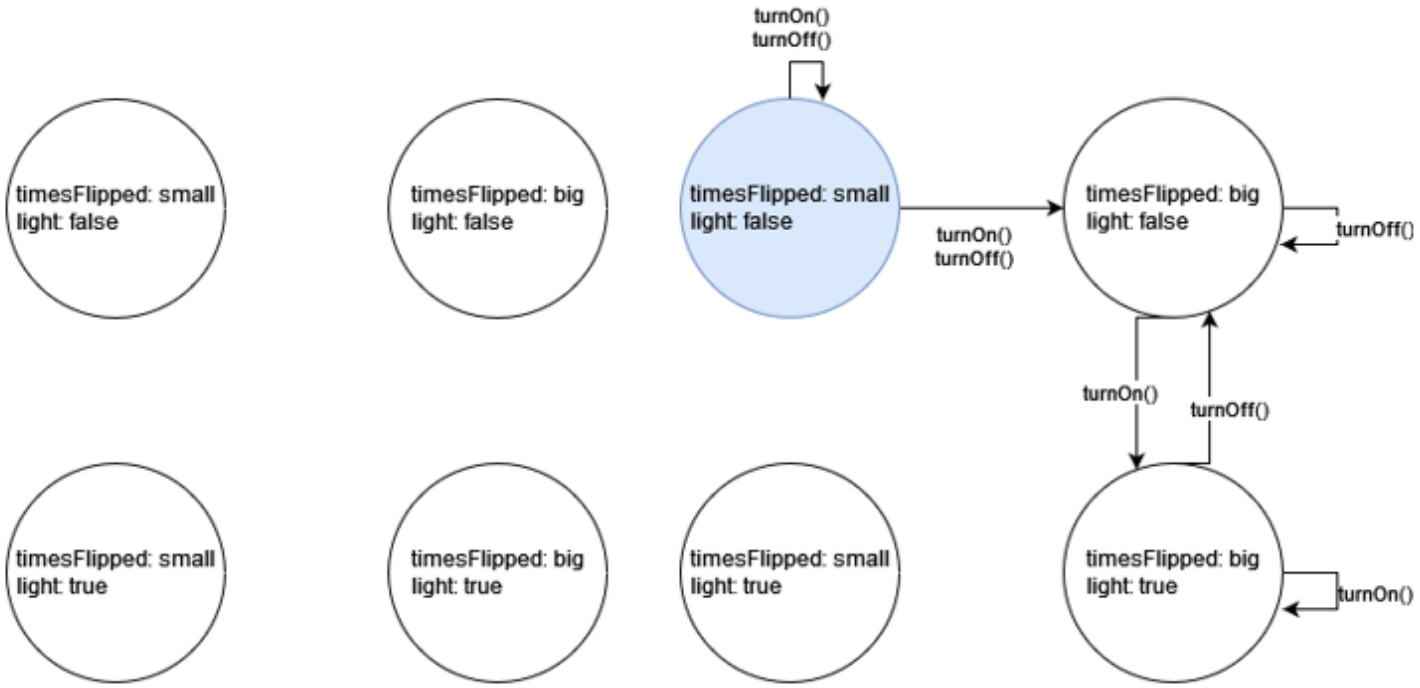
Приклад застосування модифікованого методу без використання категоризації

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract Light {
    bool flippedSwitch;
    bool light;

    constructor() {
        flippedSwitch = false;
        light = true;
    }

    function turnOn() public {
        flippedSwitch = true;
        light = true;
    }

    function turnOff() public {
        flippedSwitch = true;
        light = false;
    }
}
```



Приклад застосування модифікованого методу з використанням категоризації

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract Light2 {
    uint timesFlipped;
    bool light;

    constructor() {
        timesFlipped = 1;
        light = false;
    }

    function turnOn() public {
        if (timesFlipped >= 3) {
            light = true;
        }
        timesFlipped += 1;
    }

    function turnOff() public {
        light = false;
        timesFlipped += 1;
    }
}
```



Додаток 2

Фрагменти тексту програми

```
"""
    Main module
"""
import json
import logging
import os
import re
from typing import Optional, Dict, List, Set, Union

from crytic_compile import CryticCompile

from slither.core.compilation_unit import SlitherCompilationUnit
from slither.core.context.context import Context
from slither.core.declarations import Contract
from slither.slithir.variables import Constant
from slither.utils.colors import red

logger = logging.getLogger("Slither")
logging.basicConfig()


def _relative_path_format(path: str) -> str:
    """
    Strip relative paths of "." and ".."
    """
    return path.split("..")[-1].strip(".").strip("/")


# pylint: disable=too-many-instance-attributes,too-many-public-methods
class SlitherCore(Context):
    """
    Slither static analyzer
    """

    def __init__(self):
        super().__init__()
```

```
        self._filename: Optional[str] = None
        self._raw_source_code: Dict[str, str] = {}
        self._source_code_to_line: Optional[Dict[str, List[str]]] = None

        self._previous_results_filename: str = "slither.db.json"
        self._results_to_hide: List = []
        self._previous_results: List = []
        # From triaged result
        self._previous_results_ids: Set[str] = set()
        # Every slither object has a list of result from detector
        # Because of the multiple compilation support, we might analyze
        # Multiple time the same result, so we remove dupplicate
        self._currently_seen_resuts: Set[str] = set()
        self._paths_to_filter: Set[str] = set()

        self._crytic_compile: Optional[CryticCompile] = None

        self._generate_patches = False
        self._exclude_dependencies = False

        self._markdown_root = ""

        # If set to true, slither will not catch errors during parsing
        self._disallow_partial: bool = False
        self._skip_assembly: bool = False

        self._show_ignored_findings = False

        self._compilation_units: List[SlitherCompilationUnit] = []

        self._contracts: List[Contract] = []
        self._contracts_derived: List[Contract] = []

    @property
    def compilation_units(self) -> List[SlitherCompilationUnit]:
```

```
        return list(self._compilation_units)


    def add_compilation_unit(self, compilation_unit:
SlitherCompilationUnit):
        self._compilation_units.append(compilation_unit)


    # endregion

###################################################################################
########

###################################################################################
########
    # region Contracts

###################################################################################
########

###################################################################################
########


    @property
    def contracts(self) -> List[Contract]:
        if not self._contracts:
            all_contracts = [
                compilation_unit.contracts for compilation_unit in
self._compilation_units
            ]
            self._contracts = [item for sublist in all_contracts for item
in sublist]
        return self._contracts


    @property
    def contracts_derived(self) -> List[Contract]:
        if not self._contracts_derived:
            all_contracts = [
                compilation_unit.contracts_derived for compilation_unit in
self._compilation_units
            ]
            self._contracts_derived = [item for sublist in all_contracts
for item in sublist]
```

```
        return self._contracts_derived


    def get_contract_from_name(self, contract_name: Union[str, Constant]) -
> List[Contract]:
        """
            Return a contract from a name
        Args:
            contract_name (str): name of the contract
        Returns:
            Contract
        """
        contracts = []
        for compilation_unit in self._compilation_units:
            contracts +=
compilation_unit.get_contract_from_name(contract_name)
        return contracts



###############################################################################
########


###############################################################################
########
    # region Source code


###############################################################################
########


###############################################################################
########


    @property
    def source_code(self) -> Dict[str, str]:
        """{filename: source_code (str)}: source code"""
        return self._raw_source_code


    @property
    def filename(self) -> Optional[str]:
        """str: Filename."""
        return self._filename
```

```
    @filename.setter
    def filename(self, filename: str):
        self._filename = filename

    def add_source_code(self, path):
        """
        :param path:
        :return:
        """
        if self.crytic_compile and path in self.crytic_compile.src_content:
            self.source_code[path] = self.crytic_compile.src_content[path]
        else:
            with open(path, encoding="utf8", newline="") as f:
                self.source_code[path] = f.read()

    @property
    def markdown_root(self) -> str:
        return self._markdown_root

    def print_functions(self, d: str):
        """
        Export all the functions to dot files
        """
        for compilation_unit in self._compilation_units:
            for c in compilation_unit.contracts:
                for f in c.functions:
                    f.cfg_to_dot(os.path.join(d, "{}.{}.dot".format(c.name,
f.name)))

    # endregion

###################################################################################
########

###################################################################################
########
    # region Filtering results
```

```
################################################################################
########

################################################################################
########


    def has_ignore_comment(self, r: Dict) -> bool:
        """
        Check if the result has an ignore comment on the proceeding line,
in which case, it is not valid
        """
        if not self.crytic_compile:
            return False
        mapping_elements_with_lines = (
            (

os.path.normpath(elem["source_mapping"]["filename_absolute"]),
                elem["source_mapping"]["lines"],
            )
            for elem in r["elements"]
            if "source_mapping" in elem
            and "filename_absolute" in elem["source_mapping"]
            and "lines" in elem["source_mapping"]
            and len(elem["source_mapping"]["lines"]) > 0
        )


        for file, lines in mapping_elements_with_lines:
            ignore_line_index = min(lines) - 1
            ignore_line_text = self.crytic_compile.get_code_from_line(file,
ignore_line_index)
            if ignore_line_text:
                match = re.findall(
                    r"^\s*//\s*slither-disable-next-line\s*([a-zA-Z0-9_,-
]*)",
                    ignore_line_text.decode("utf8"),
                )
                if match:
                    ignored = match[0].split(",")
```

```
                    if ignored and ("all" in ignored or any(r["check"] == c
for c in ignored)):
                        return True


        return False


    def valid_result(self, r: Dict) -> bool:
        """
        Check if the result is valid
        A result is invalid if:
            - All its source paths belong to the source path filtered
            - Or a similar result was reported and saved during a previous
run
            - The --exclude-dependencies flag is set and results are only
related to dependencies
            - There is an ignore comment on the preceding line
        """

        # Remove dupplicate due to the multiple compilation support
        if r["id"] in self._currently_seen_resuts:
            return False
        self._currently_seen_resuts.add(r["id"])

        source_mapping_elements = [
            elem["source_mapping"].get("filename_absolute", "unknown")
            for elem in r["elements"]
            if "source_mapping" in elem
        ]
        source_mapping_elements = map(
            lambda x: os.path.normpath(x) if x else x,
source_mapping_elements
        )
        matching = False

        for path in self._paths_to_filter:
            try:
                if any(
```

```
                    bool(re.search(_relative_path_format(path),
src_mapping))
                    for src_mapping in source_mapping_elements
                ):
                    matching = True
                    break
            except re.error:
                logger.error(
                    f"Incorrect regular expression for --filter-paths
{path}."
                    "\nSlither supports the Python re format"
                    ": https://docs.python.org/3/library/re.html"
                )


        if r["elements"] and matching:
            return False
        if r["elements"] and self._exclude_dependencies:
            return not all(element["source_mapping"]["is_dependency"] for
element in r["elements"])
        if self._show_ignored_findings:
            return True
        if r["id"] in self._previous_results_ids:
            return False
        if self.has_ignore_comment(r):
            return False
        # Conserve previous result filtering. This is conserved for
compatibility, but is meant to be removed
        return not r["description"] in [pr["description"] for pr in
self._previous_results]


    def load_previous_results(self):
        filename = self._previous_results_filename
        try:
            if os.path.isfile(filename):
                with open(filename) as f:
                    self._previous_results = json.load(f)
                    if self._previous_results:
                        for r in self._previous_results:
```

```
                            if "id" in r:
                                self._previous_results_ids.add(r["id"])
        except json.decoder.JSONDecodeError:
            logger.error(
                red("Impossible to decode {}. Consider removing the
file".format(filename))
            )

    def write_results_to_hide(self):
        if not self._results_to_hide:
            return
        filename = self._previous_results_filename
        with open(filename, "w", encoding="utf8") as f:
            results = self._results_to_hide + self._previous_results
            json.dump(results, f)

    def save_results_to_hide(self, results: List[Dict]):
        self._results_to_hide += results

    def add_path_to_filter(self, path: str):
        """
        Add path to filter
        Path are used through direct comparison (no regex)
        """
              self._paths_to_filter.add(path)
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

МЕТОД АВТОМАТИЗОВАНОЇ ВЕРИФІКАЦІЇ ПРОГРАМНОГО КОДУ СМАРТ-КОНТРАКТІВ

Виконала: Корунська Анна Михайлівна

Науковий керівник: доцент кафедри ПЗКС, к.т.н., доцент,
Заболотня Тетяна Миколаївна

Київ – 2021

АКТУАЛЬНІСТЬ РОБОТИ

- широке застосування смарт-контрактів у різних сферах, насамперед фінансовій;
- за наявності в коді смарт-контракту дефектів їх неможливо виправити після публікації в блокчейн;
- перевірка програмного коду смарт-контрактів вручну людьми не гарантує відповідності фукнціональним вимогам та вимогам безпеки.

Розроблення автоматизованих інструментів контролю якості і відповідності програмного коду контрактів бізнес-вимогам та вимогам безпеки є актуальною задачею.

ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Об’єкт дослідження: процес автоматизованої верифікації програмного коду.

Предмет дослідження: методи та програмні засоби автоматизованої верифікації програмного коду смарт-контрактів.

ОКРЕМІ ЗАВДАННЯ ТА МЕТА ДОСЛІДЖЕННЯ

1. Провести дослідження підходів до аналізу програмного коду смарт-контрактів.
2. Розробити модифікований метод аналізу програмного коду смарт-контрактів, який дозволяє проводити верифікацію вимог, заданих користувачем.
3. Проаналізувати переваги та недоліки запропонованого методу аналізу програмного коду смарт-контрактів.
4. Розробити програмне забезпечення, що реалізує модифікований метод аналізу програмного коду смарт-контрактів.

Мета дослідження: підвищення ефективності верифікації функціональних вимог до роботи смарт-контракту, заданих користувачем, шляхом розроблення модифікованого методу автоматизованої верифікації програмного коду смарт-контрактів.

ІСНУЮЧІ ПІДХОДИ АНАЛІЗУ ПРОГРАМНОГО КОДУ СМАРТ-КОНТРАКТІВ

- методи перевірки моделі, model checking
- методи символьної верифікації, symbolic execution
- методи верифікації в реальному часі, runtime verification
- фазинг

ІСНУЮЧІ ІНСТРУМЕНТИ АНАЛІЗУ ПРОГРАМНОГО КОДУ СМАРТ-КОНТРАКТІВ

Manticore

SLITHER

Mythril

microsoft/verisol

A formal verifier and analysis tool for Solidity Smart Contracts

☆ 187 C# Updated on 8 Jun

МЕТОД ОБМЕЖЕНОЇ ПЕРЕВІРКИ МОДЕЛІ

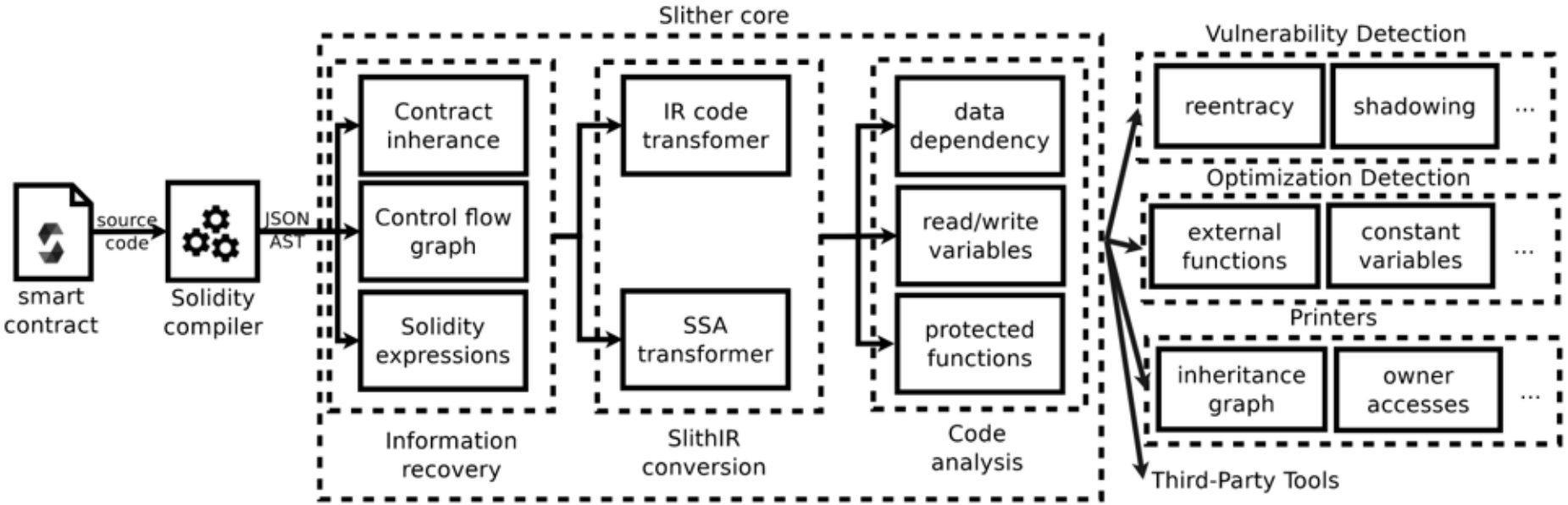
Метод обмеженої перевірки моделі, bounded model checking – це підвид методів символьної верифікації, основна ідея якого полягає в обході графа станів програми за k кроків таким чином, щоб виявити неприпустимий стан, який вказував би на порушення виконання формальної вимоги, заданої користувачем.

ІНСТРУМЕНТ АНАЛІЗУ ПРОГРАМНОГО КОДУ SLITHER ТА ПРОМІЖНЕ ПРЕДСТАВЛЕННЯ SLITHIR

- використовує методи статичного аналізу;

- дозволяє проводити перевірки за допомогою вже реалізованих модулів аналізу, які можуть надати інформацію про розповсюджені помилки або можливі проблемні місця коду, що можуть містити вразливості.

ІНСТРУМЕНТ АНАЛІЗУ ПРОГРАМНОГО КОДУ SLITHER ТА ПРОМІЖНЕ ПРЕДСТАВЛЕННЯ SLITHIR



ІНСТРУМЕНТ АНАЛІЗУ ПРОГРАМНОГО КОДУ SLITHER ТА ПРОМІЖНЕ ПРЕДСТАВЛЕННЯ SLITHIR

1. Slither відновлює зі скомпільованих файлів важливу для аналізу інформацію, таку як граф наслідування контракту та повний перелік команд.

2. Весь код смарт-контракту конвертується у SlithIR, внутрішню мову представлення. SlithIR використовує єдину статичну оцінку (SSA) для полегшення обчислення різноманітних метрик коду.

3. Slither проводить набір визначених перевірок, які виконуються заздалегідь описаними модулями аналізу. Вони перевіряють такі сценарії небажаної поведінки, як переприсвоєння константних значень, потенційно небезпечні рекурсивні виклики тощо.

ЗАПРОПОНОВАНИЙ МЕТОД АВТОМАТИЗОВАНОЇ ВЕРИФІКАЦІЇ ПРОГРАМНОГО КОДУ СМАРТ-КОНТРАКТУ

1. Категоризація змінних смарт-контракту і запис вимог для перевірки з використанням цих категорійних змінних.
2. Переведення вихідного коду смарт-контракту в проміжне представлння SlithIR і побудова графу потоку виконання.
3. Створення графу станів смарт-контракту:
    1. Створення вершин на основі усіх комбінаторно можливих значень категорійних змінних.
    2. Визначення стану смарт-контракту після ініціалізації і позначення відповідної вершини як точки входу.
    3. Виконання методу обмеженої перевірки моделі на графі потоку виконання контракту.
    4. Встановлення дуг на основі отриманих результатів переходів станів.
4. Аналіз виконання вимог на основі досяжності відповідної вершини у графі станів смарт-контракту.

ПРИКЛАД ЗАСТОСУВАННЯ МОДИФІКОВАНОГО МЕТОДУ

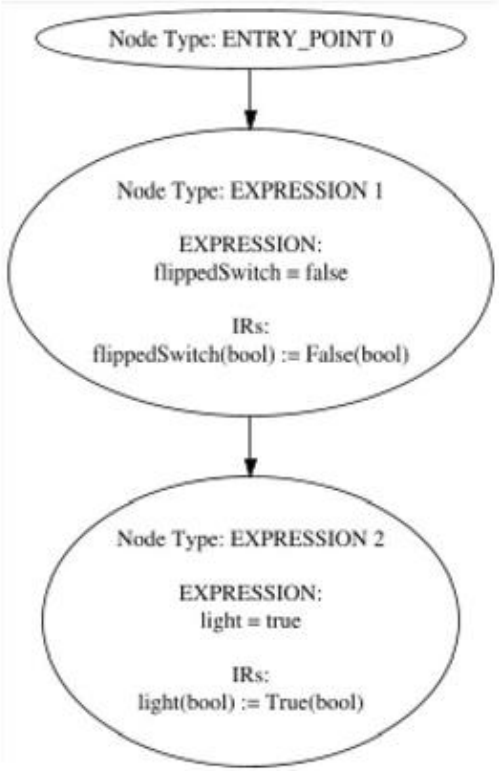
Побудова проміжного представлення:

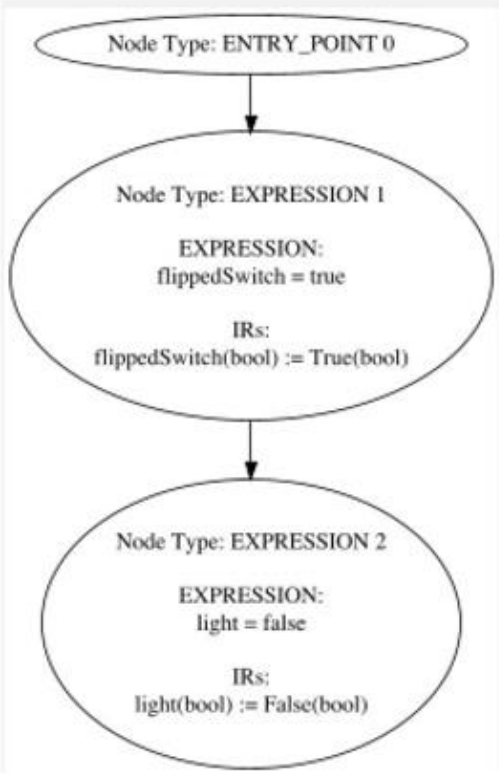
Вихідний код	Проміжне представлення SlithIR

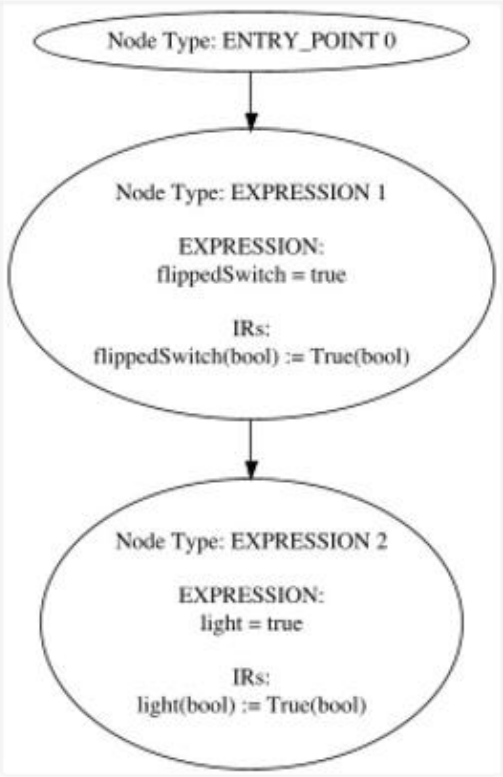
`// SPDX-License-Identifier: MIT`
`pragma solidity ^0.8.10;`
 	
`contract Light {`
`  bool flippedSwitch;`
`  bool light;`
 	
`  constructor() {`
`  flippedSwitch = false;`
`  light = true;`
`  }`
 	
`  function turnOn() public {`
`  flippedSwitch = true;`
`  light = true;`
`  }`
 	
`  function turnOff() public {`
`  flippedSwitch = true;`
`  light = false;`
`  }`
`}`
	`Contract Light`
	`  Function Light.constructor() (*)`
	`  Expression: flippedSwitch = false`
	`  IRs:`
	`  flippedSwitch(bool) := False(bool)`
	`  Expression: light = true`
	`  IRs:`
	`  light(bool) := True(bool)`
	`  Function Light.turnOn() (*)`
	`  Expression: flippedSwitch = true`
	`  IRs:`
	`  flippedSwitch(bool) := True(bool)`
	`  Expression: light = true`
	`  IRs:`
	`  light(bool) := True(bool)`
	`  Function Light.turnOff() (*)`
	`  Expression: flippedSwitch = true`
	`  IRs:`
	`  flippedSwitch(bool) := True(bool)`
	`  Expression: light = false`
	`  IRs:`
	`  light(bool) := False(bool)`

ПРИКЛАД ЗАСТОСУВАННЯ МОДИФІКОВАНОГО МЕТОДУ

Візуалізація графа потоку виконання (CFG):

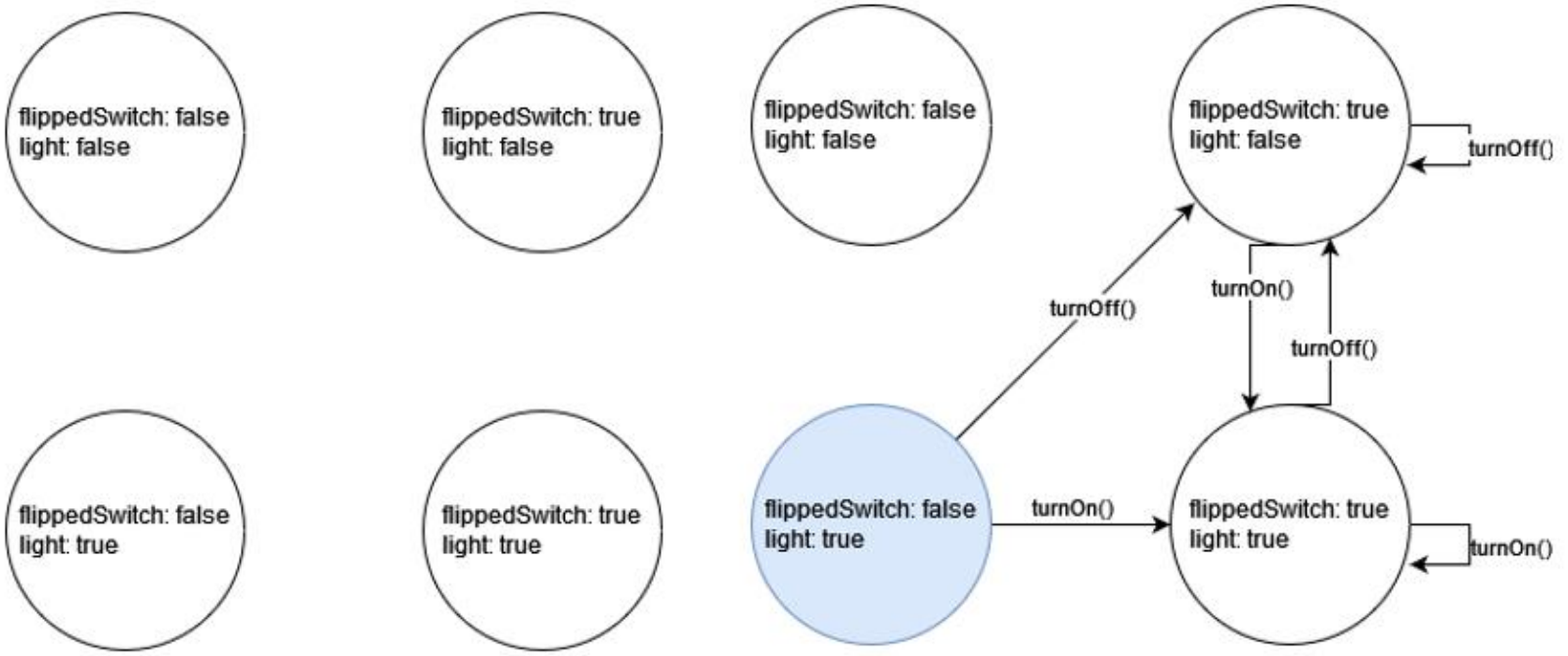






ПРИКЛАД ЗАСТОСУВАННЯ МОДИФІКОВАНОГО МЕТОДУ

Побудова графа станів смарт-контракту:



ПРИКЛАД ЗАСТОСУВАННЯ МОДИФІКОВАНОГО МЕТОДУ З ВИКОРИСТАННЯМ КАТЕГОРИЗАЦІЇ

Побудова проміжного представлення:

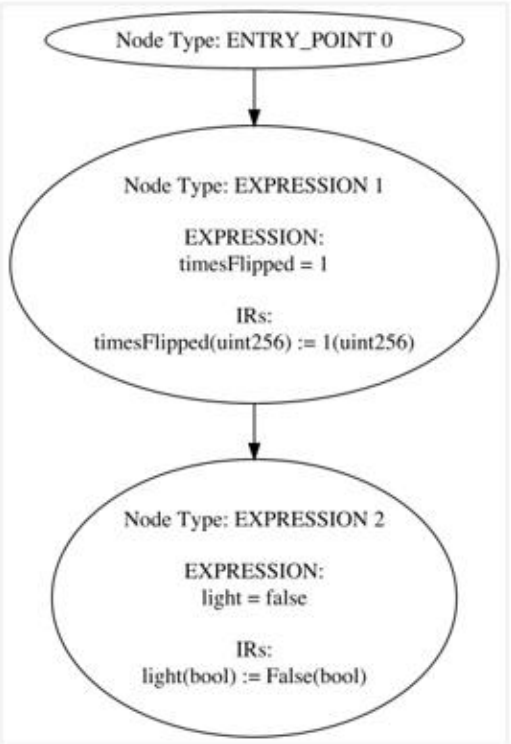
Вихідний код	Проміжне представлення SlithIR

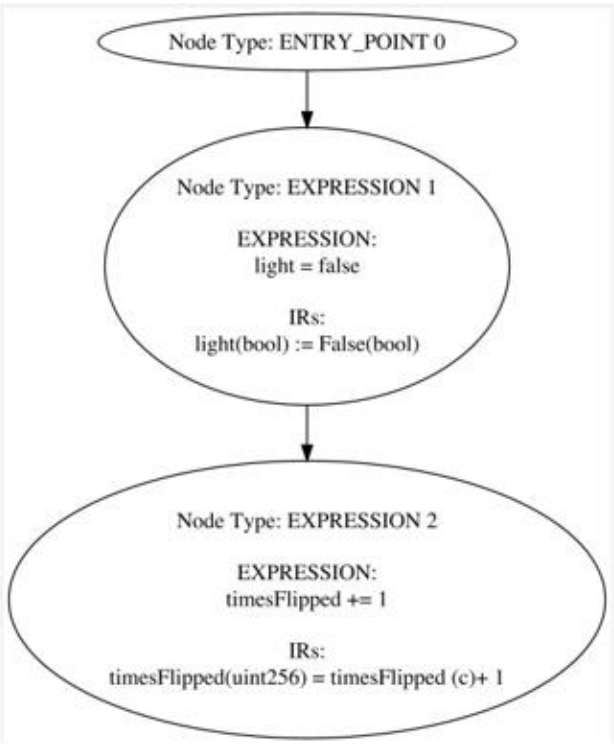
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract Light2 {
    uint timesFlipped;
    bool light;

    constructor() {
        timesFlipped = 1;
        light = false;
    }

    function turnOn() public {
        if (timesFlipped >= 3) {
            light = true;
        }
        timesFlipped += 1;
    }

    function turnOff() public {
        light = false;
        timesFlipped += 1;
    }
}
```

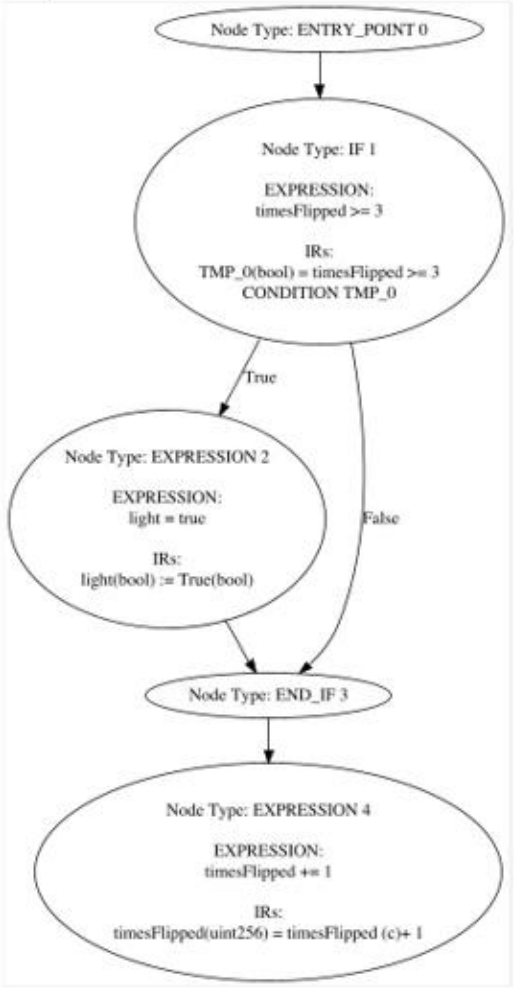
```
Contract Light2
        Function Light2.constructor() (*)
                Expression: timesFlipped = 1
                IRs:
                        timesFlipped(uint256) := 1(uint256)
                Expression: light = false
                IRs:
                        light(bool) := False(bool)
        Function Light2.turnOn() (*)
                Expression: timesFlipped >= 3
                IRs:
                        TMP_0(bool) = timesFlipped >= 3
                        CONDITION TMP_0
                Expression: light = true
                IRs:
                        light(bool) := True(bool)
                Expression: timesFlipped += 1
                IRs:
                        timesFlipped(uint256) = timesFlipped (c)+ 1
        Function Light2.turnOff() (*)
                Expression: light = false
                IRs:
                        light(bool) := False(bool)
                Expression: timesFlipped += 1
                IRs:
                        timesFlipped(uint256) = timesFlipped (c)+ 1
```

ПРИКЛАД ЗАСТОСУВАННЯ МОДИФІКОВАНОГО МЕТОДУ З ВИКОРИСТАННЯМ КАТЕГОРИЗАЦІЇ

Візуалізація графа потоку виконання (CFG):

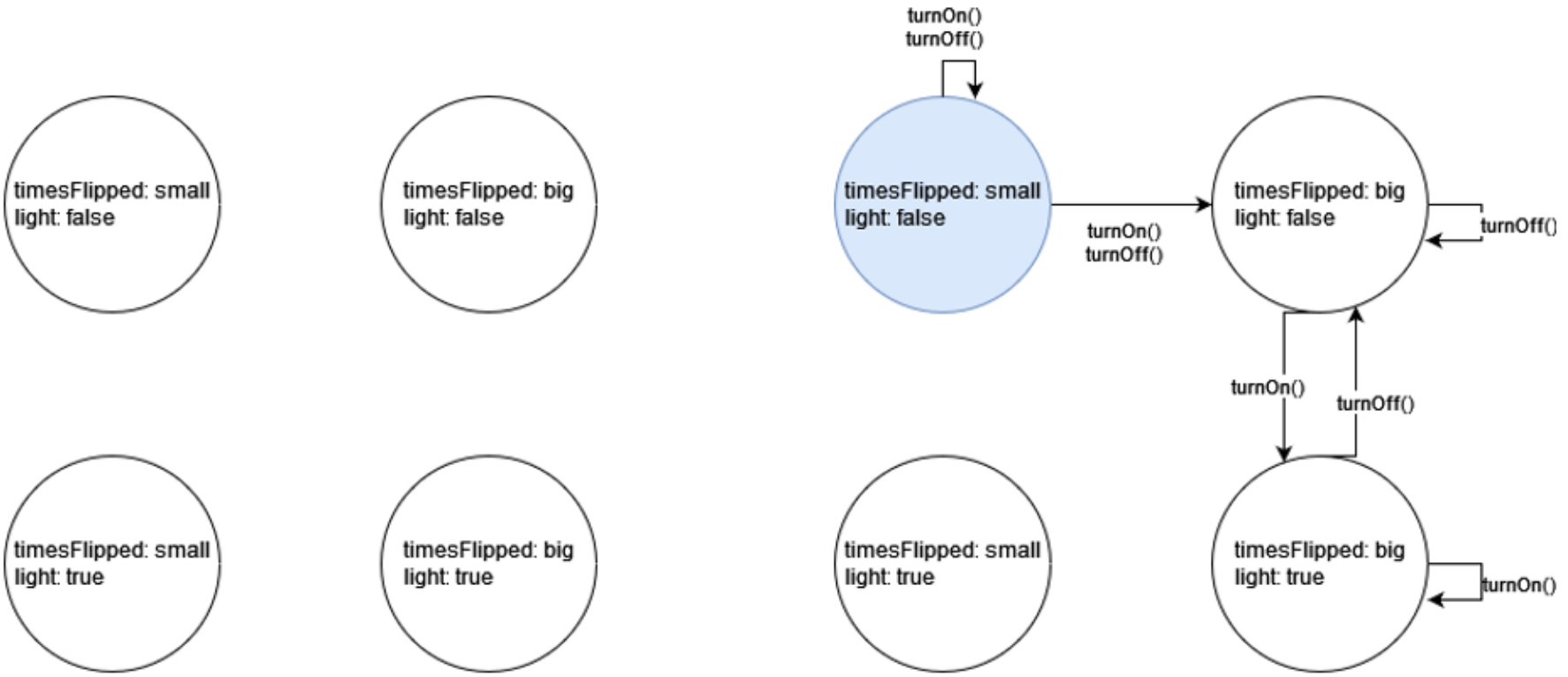




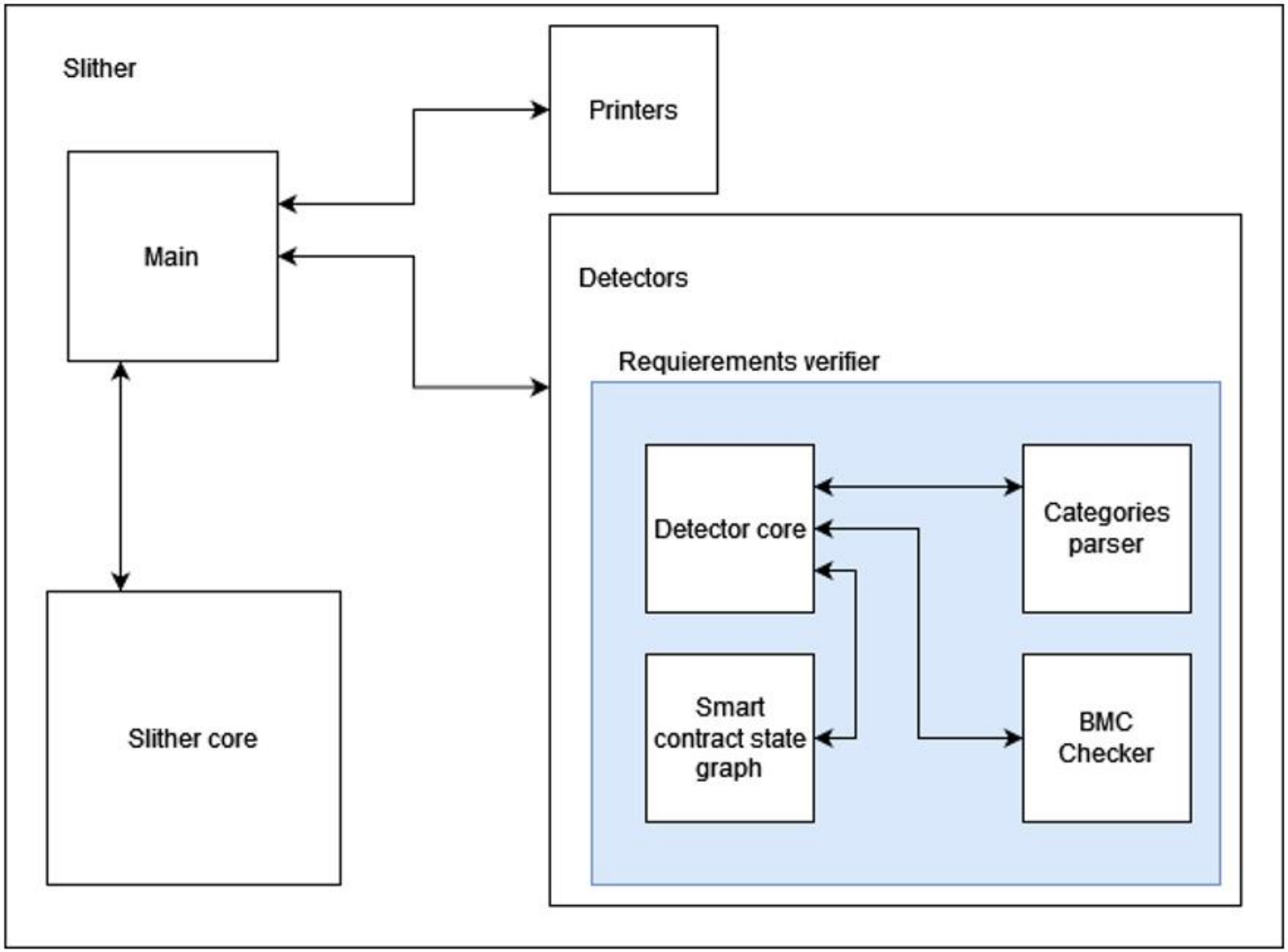


ПРИКЛАД ЗАСТОСУВАННЯ МОДИФІКОВАНОГО МЕТОДУ З ВИКОРИСТАННЯМ КАТЕГОРИЗАЦІЇ

Побудова графа станів смарт-контракту:



ПРОГРАМНА РЕАЛІЗАЦІЯ МОДИФІКОВАНОГО МЕТОДУ ЯК МОДУЛЯ АНАЛІЗУ SLITHER



ПЕРЕВАГИ ЗАПРОПОНОВАНОЇ МОДИФІКАЦІЇ

1. Запропонована модифікація дозволяє перевіряти вимоги, сформульовані користувачем, а не лише аналізувати розповсюджені вразливості.

2. Створення та візуалізація графу станів смарт-контракту надає можливість наглядно побачити стан контракту у точці входу і під час виконання.

3. Категоризація змінних сприятиме ранньому та детальному документуванню смарт-контрактів.

НЕДОЛІКИ ЗАПРОПОНОВАНОЇ МОДИФІКАЦІЇ

1. Категоризація значущих змінних і формулювання вимог має виконуватись розробником вручну.

2. State explosion problem.

3. Вибір числа k для оптимальної роботи методу обмеженої перевірки моделі.

БІЗНЕС-МОДЕЛЬ

Проблема. За наявності в коді смарт-контракту дефектів їх неможливо виправити після публікації в блокчейн	**Рішення.** Програмний модуль, який реалізує метод автоматизованої верифікації програмного коду смарт-контрактів.	**Унікальна ціннісна пропозиція.** Програмний модуль реалізує перевірку заданих розробниками формальних вимог до смарт-контракту, що робить можливим автоматизацію перевірки функціональних вимог.	**Прихована перевага.** Програмний модуль постачається вже інтегрованим в інструмент розробки Slither, що дозволяє проводити перевірки на широковідомі та описані вразливості.
Споживачі. Компанії з розробки програмного забезпечення, що спеціалізуються на децентралізованих рішеннях	**Ключові метрики.** Кількість проданих ліцензій на користування модулем	**Потоки доходів.** Доходи від продажу ліцензій на користування програмним модулем.	**Канали збуту.** Рекламна кампанія продукту, блокчейн-конференції та хакатони.
Структура витрат - сплата податків; - оплата юридичного консультування та оформлення патентних документів; - заробітна плата команди розробників продукту; - заробітна плата команди маркетингу; - витрати на рекламу.			

НАУКОВА НОВИЗНА

Вперше запропоновано модифікований метод аналізу програмного коду смарт-контрактів, заснований на використанні проміжного представлення SlithIR, який відрізняється від існуючих методів верифікації додатковим етапом побудови графа станів смарт-контракту із врахуванням категоризації значущих змінних, що дозволяє проводити верифікацію відповідності програмного коду смарт-контрактів заданим користувачем функціональним вимогам.

АПРОБАЦІЯ

Заболотня Т.М., Корунська А.М., Метод автоматизованої верифікації програмного коду смарт-контрактів, Збірник XIV наукової конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2021, с. 9-14

УНІКАЛЬНІСТЬ РОБОТИ

UNICHECK

Ім'я користувача:
Заболотня Тетяна Миколаївна

ID перевірки:
1009690637

Дата перевірки:
16.12.2021 00:59:57 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
16.12.2021 01:04:35 EET

ID користувача:
83364

Назва документа: МД - unicheck

Кількість сторінок: 46 **Кількість слів:** 9821 **Кількість символів:** 76502 **Розмір файлу:** 85.67 KB **ID файлу:** 1009689806

1.22%
Схожість

Найбільша схожість: 0.4% з джерелом з Бібліотеки (ID файлу: 1009689805)

0.16% Джерела з Інтернету	5	Сторінка 48
1.22% Джерела з Бібліотеки	122	Сторінка 48

НАПРЯМКИ ПОДАЛЬШОЇ РОБОТИ

1. Створення підходу автоматизованої категоризації на основі аналізу вихідного коду.

2. Введення додаткової категоризації для змінних типу *address* – «акторів», оскільки дані про те, хто саме є автором транзакції має неймовірно велике значення в смарт-контрактах.

3. Оптимізація вибору числа k для роботи методу обмеженої верифікації моделі при побудові графу станів смарт-контракту.

ВИСНОВКИ

1. Проведено дослідження підходів та інструментів аналізу програмного коду смарт-контрактів.

2. Розроблено модифікований метод аналізу програмного коду смарт-контрактів, який дозволяє проводити верифікацію вимог, заданих користувачем.

3. Проаналізовано переваги та недоліки запропонованого методу аналізу програмного коду смарт-контрактів.

4. Розроблено програмне забезпечення, що реалізує модифікований метод аналізу програмного коду смарт-контрактів.

5. Визначено напрямки подальшого дослідження.

Дякую за увагу