НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

«На правах рукопису»
УДК

«До захисту допущено»
В.о. завідувача кафедри
_____ Дмитро ЛАНДЕ
“___”_____2022 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Системи, технології та математичні методи кібербезпеки»

зі спеціальності 125 «Кібербезпека»

на тему: Автоматизація виявлення вразливостей у смарт-контрактах

Виконав здобувач ступеня магістра 2 курсу, групи ФБ-01 мн
(шифр групи)

Звінський Тарас Сергійович _____ _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник к.е.н., доцент Ткач В. М _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Здобувач ступеня магістра _____
(підпис)

Київ – 2022 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – другий (магістерський)
Спеціальність – 125 «Кібербезпека»
Освітньо-наукова програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Дмитро ЛАНДЕ
(підпис)
«___»_____2022 р.

ЗАВДАННЯ
на магістерську дисертацію здобувачу ступеня магістра

Звінському Тарасу Сергійовичу _____
(прізвище, ім'я, по батькові)

1. Тема дисертації Автоматизація виявлення вразливостей у смарт-контрактах

науковий керівник дисертації к.е.н., доцент Ткач Володимир Миколайович
_____ _____ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «____» _____ 2022 р. № _____

2. Термін подання здобувачем дисертації 06.06.2022 р.

3. Об'єкт дослідження _____
_____

4. Предмет дослідження_____
_____
_____
_____

5. Перелік завдань, які потрібно розробити _____
_____
_____
_____
_____

6. Орієнтовний перелік ілюстративного матеріалу _____
_____
_____

7. Орієнтовний перелік публікацій _____
_____
_____

8. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка

Здобувач ступеня магістра _____ _____
(підпис) (власне ім'я, ПРІЗВИЩЕ)

Науковий керівник _____ _____
(підпис) (власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Робота обсягом 77 сторінок містить 17 ілюстрацій, 3 таблиці та 26 літературних посилань.

Актуальність роботи. Технологія смарт-контрактів змінює звичайні промислові та бізнес-процеси. Будучи вбудованими у блокчейни, смарт-контракти дозволяють автоматично виконувати умови договору без втручання третьої сторони, якій довіряєш. В результаті це дозволяє скоротити адміністрування та заощадити витрати на послуги, підвищити ефективність бізнес-процесів та знизити ризики. Хоча смарт-контракти обіцяють стимулювати нову хвилю інновацій у бізнес-процесах, існує ряд проблем, які потрібно вирішити. Серед іншого, контракти не можуть бути змінені після публікації, широко застосовується концепція відкритого коду, одні контракти можуть вільно взаємодіяти з іншими. Усе це формує потребу точно виявити слабкі місця у смарт-контрактах до етапу публікації.

Мета і завдання.

Метою роботи є підвищення ефективності виявлення вразливостей у смарт-контрактах, шляхом розроблення модифікованого підходу до символьного виконання та формулюванні покращених алгоритмів, відповідно до типу вразливості.

Завданням роботи є дослідження й аналіз існуючих рішень для автоматизованого виявлення вразливостей, відбір таких, що передбачають перспективу масштабування. Наступне, модифікація модулів одного з них для покращення точності пошуку визначених вразливостей. Порівняння з існуючими засобами та аналіз результатів.

Об’єктом дослідження є процес автоматизованого аналізу смарт-контрактів.

Предметом дослідження є методи та програмні інструменти автоматизованого пошуку вразливостей у коді чи виконуваному файлі смарт-контрактів.

Методи дослідження. У роботі використовується теорія програмних систем, теорія алгоритмів, методи статичного та динамічного аналізу коду.

Наукова новизна.

У роботі удосконалено архітектуру виявлення вразливостей у смарт-контрактах на основі символьного виконання. На прикладі інструменту Mythril, ця модель застосована щоб покращити його ефективність. По-перше, це дозволяє виявляти потенційно вразливі області коду за допомогою статичного аналізу та визначати критичні шляхи, які можуть мати дефекти безпеки. Потім, маючи на меті проблему, що традиційні алгоритми пошуку не можуть активно знаходити та досліджувати критичні шляхи, ця робота представляє стратегію багатоцільового орієнтованого пошуку шляхів, засновану на пріоритетах.

Практичне значення одержаних результатів.

Отримана стратегія описує правила безпеки та пропонує відповідні логіки виявлення для різних категорій уразливості. Її застосування дозволяє отримати високу точість автоматичного виявлення. Це скорочує ручну оцінку результатів на етапі тестування смарт-контракту.

Апробація результатів роботи. Основні положення і результати роботи доповідалися та обговорювалися на III міжнародній науково-практичній конференції “Globalization of scientific knowledge”.

Ключові слова: смарт-контракт, символьне виконання, граф потоку керування, mythril, стан, SMT-вирішувач, критична інструкція.

ABSTRACT

Work in volume of 77 pages contains 17 illustrations, 3 tables and 26 literary references.

Research motivation. Smart contract technology is changing conventional industrial and business processes. Being built into the blockchain, smart contracts will automatically fulfill the terms of the contract without a third party you trust. As a result, it reduces administration and saves on service costs, increases business process efficiency and reduces risks. Although smart contracts promise to stimulate a new wave of innovation in business processes, there are a number of issues that need to be addressed. Among other things, contracts cannot be changed after publication, the concept of open source is widely used, some contracts can interact with others. All this creates the need to clarify the weaknesses in smart contracts before the publication stage.

Goals and objectives of research.

The aim of the work is to increase the efficiency of vulnerability detection in smart contracts by developing a modified approach to symbolic execution and formulating improved algorithms according to the type of vulnerability.

The objective of the work is to study and analyze solutions for automated detection of such vulnerabilities, selection that provide a perspective of scaling. Next, modify the modules of one of them to improve the precision of the search for certain vulnerabilities. Comparison with functional tools and analysis of results.

Object of research is the process of automated analysis of smart contracts.

Subject of research is methods and software tools for automated search of vulnerabilities in the code or executable file of smart contracts.

Research methods. The paper uses the theory of software systems, the theory of algorithms, methods of static and dynamic code analysis.

Scientific novelty.

The paper improves the architecture of vulnerability detection in smart contracts based on symbolic execution. On the example of the Mythril tool, this model is used to improve its effectiveness. First, it allows the detection of potentially vulnerable area

codes by static analysis and identifies critical pathways that may have security flaws. Then, exploring the goal of the problem that traditional search algorithms cannot actively find and explore critical paths, this paper presents a strategy of goal-oriented path finding based on the best.

The practical significance of the results obtained.

The resulting strategy describes the rules and proposed appropriate security logics for different categories of vulnerabilities. This application allows you to get high accuracy of automatic detection. This reduces the manual evaluation of results during the smart contract testing phase.

Approbation of work results. The main provisions and results of the work were reported and discussed at the III International Scientific and Practical Conference "Globalization of Scientific Knowledge".

Keywords: smart contract, symbolic execution, control flow graph, mythril, state, SMT-solver, critical instruction.

Зміст

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

EVM – Ethereum Virtual Machine

CFG, ГПК – граф потоку керування

АСД – абстрактне синтаксичне дерево

SMT (Satisfiability Modulo Theories) – це задача розв'язності для логічних формул з урахуванням теорій, які лежать в їх основі. Прикладами таких теорій для SMT формул є: теорії цілих та дійсних чисел, теорії списків, масивів, бітових векторів та ін.

CN – critical node

TOD (Transaction order dependency) – залежність від порядку транзакцій.

TP (true positive) – істинно правильне

TN (true negative) – істинно помилкове

FP (false positive) – хибно правильне (помилка 1-го роду)

FN (false negative) – помилково негативне (помилка 2-го роду)

ВСТУП

Програми на публічних блокчейнах часто обробляють цінні активи, що робить їх привабливою мішенню для атаки. У той же час розробити правильні блокчейн-застосунки досить складно. Перевірка коду на наявність потенційних вразливостей є життєздатним варіантом підвищення довіри. Тому було запропоновано безліч методів та інструментів, щоб підтримати розробників і аналітиків у виявленні вразливостей коду. Більше того, публікації постійно з'являються з різною спрямованістю, обсягом та якістю, що ускладнює завдання йти в ногу з цією сферою та визначати відповідні тенденції. Таким чином, регулярні огляди є необхідними, щоб не відставати від різноманітних подій у структурований спосіб.

Смарт-контракти можна впевнено назвати одним з найбільш популярних та доступних засобів для інтеграції децентралізованих технологій. Вони стають все більш привабливими у різних сферах: фінансовій, ігровій або сфері верифікації авторських прав. Платформа Ethereum найбільш широко використовується і найкраще задокументована. Більше того, застосункам на основі Ethereum довіряють мільярди доларів. Як і на подібних блокчейнах, вони зазнають численних атак і втрат через вразливості, які існують на всіх рівнях екосистеми. Великим попитом користуються засоби протидії.

Різноманітні підходи та інструменти, що проводять аналіз коду смарт-контракту до його публікації в блокчейн мережу є невід'ємним елементом процесу розробки таких застосунків. Багато популярних нині інструментів не враховують усіх підходів до того, які виправлення вразливостей могли бути додані в код, тому часто можна зустріти хибно позитивні виявлення.

Таким чином, розроблення методів та інструментів, які будуть прагматично підходити до особливостей кожної вразливості під час автоматичного пошуку є актуальною задачею.

Для розробки архітектури автоматизованого пошуку вразливостей необхідно виконати наступні завдання:

1. Дослідити існуючі підходи автоматичного аналізу смарт-контрактів. Підібрати практичний метод, що вже є досить ефективним, але не враховує всіх нюансів.
2. Проаналізувати масштабованість популярних інструментів (відкритість коду, його структурованість). Підібрати такий, що носить перспективу застосовуватись до великої кількості вразливостей.
3. Розробити модифікований метод пошуку помилок смарт-контрактів, що враховує особливості їх архітектури. Реалізувати покращення для існуючого інструменту.
4. Порівняти результати аналізу смарт-контрактів розробленим інструментом із аналогічними, надати оцінки точності.

Дане дослідження пропонує модифікований метод автоматичного пошуку вразливостей у смарт-контрактах на основі методів символьного виконання, який, зокрема, додатково проводить перевірки досяжності зміни станів акаунтів для відкидання хибно позитивних виявлень.

1 ОГЛЯД СМАРТ-КОНТРАКТІВ ТА ІНСТРУМЕНТІВ АВТОМАТИЧНОГО АНАЛІЗУ

1.1 Блокчейн системи

Блокчейн можна розглядати як публічну книгу, в якій не можна сфальсифікувати всі транзакції. Рис. 1.1 ілюструє приклад блокчейну. Блокчейн — це ланцюг блоків, що постійно зростає. Коли створюється новий блок, усі вузли мережі братимуть участь у перевірці блоку. Після перевірки блоку він буде доданий до блокчейна.

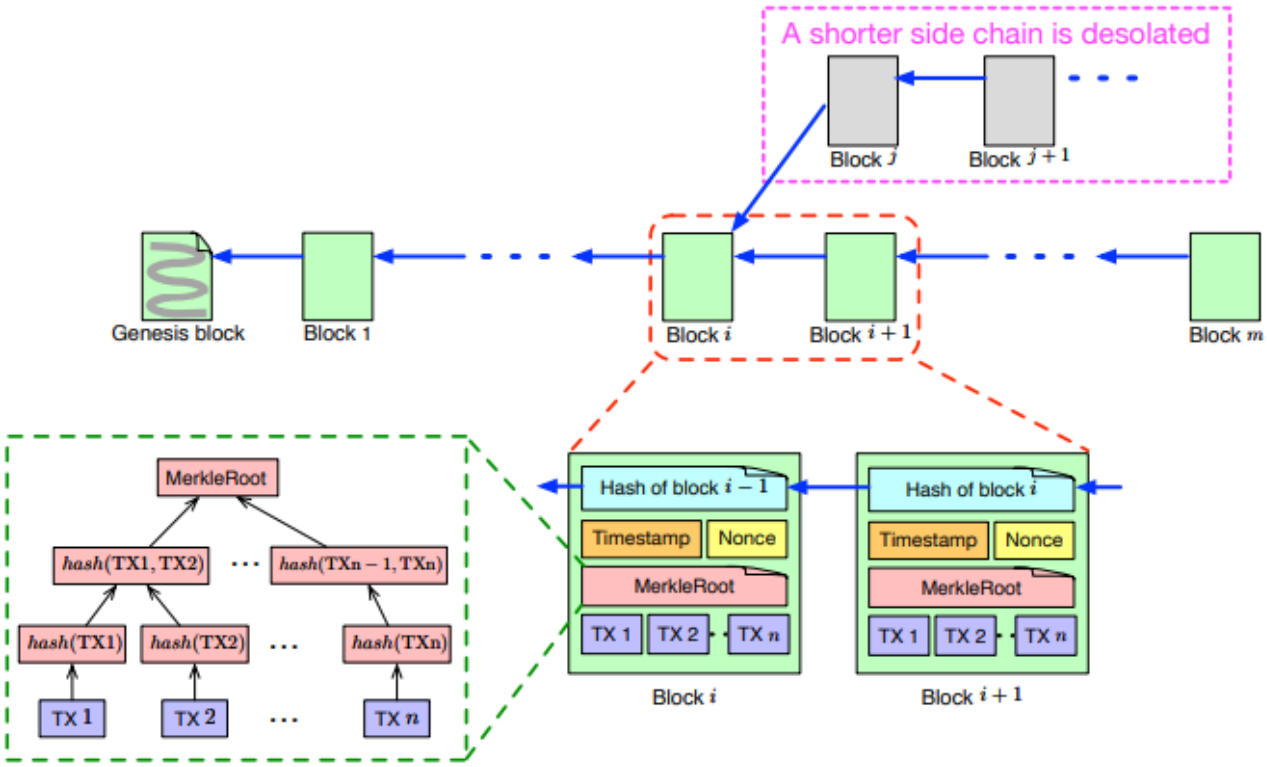

Рисунок 1.1 - Блокчейн складається з послідовності блоків, кожен з яких містить зворотний хеш, що вказує на його батьківський блок. Тим часом всередині блоку зберігається ряд транзакцій.

Для перевірки надійності блоків розроблені алгоритми консенсусу. Алгоритми консенсусу визначають, який вузол формуватиме наступний блок і як новий доданий блок буде перевірено іншими вузлами. Репрезентативні алгоритми консенсусу включають proof of work (PoW), proof of stake (PoS) і

практичну візантійську відмовостійкість (PBFT). Алгоритми консенсусу зазвичай виконують користувачі, які першими розв’язують головоломку (PoW або PoS). Цих користувачів називають майнерами. Кожен майнер зберігає повну копію блокчейну. На відміну від PoW і PoS, PBFT вимагає багатораундового процесу голосування для досягнення консенсусу. Алгоритми розподіленого консенсусу можуть гарантувати, що транзакції здійснюються без втручання третіх сторін, наприклад банків. В результаті можна заощадити транзакційні витрати. Крім того, користувачі здійснюють транзакції зі своїми віртуальними адресами, а не з реальними ідентифікаторами, тому конфіденційність користувачів також зберігається.

У системах блокчейну можливо, що кілька вузлів можуть успішно досягти консенсусу (тобто розв’язати головоломку) одночасно, отже, це може спричинити роздвоєння гілок. Щоб усунути невідповідність, короткочасний бічний ланцюг відокремлюється, як показано на рис. 1.1, а найдовший ланцюг вибирається як дійсний ланцюг. Цей механізм ефективний, оскільки довший ланцюжок більш толерантний до зловмисних атак у розподілених системах. Підсумовуючи, технологія блокчейн має ключові характеристики децентралізації, незмінності, стійкості та анонімності.

1.2 Смарт контракти

Смарт-контракти можна розглядати як великий прогрес у технології блокчейн [7]. У 1990-х роках смарт-контракт був запропонований як комп'ютеризований протокол транзакції, який виконує договірні умови угоди. Договірні положення, вбудовані в смарт-контракти, будуть автоматично застосовуватися, коли буде виконано певну умову (наприклад, одна сторона, яка порушить договір, буде автоматично покарана).

Блокчейни дозволяють це реалізувати. Смарт-контракти по суті реалізуються поверх блокчейнів. Затверджені договірні положення перетворюються на виконувані комп’ютерні програми. Логічні зв’язки між

договірними пунктами також були збережені у формі логічних потоків у програмах (наприклад, оператор if-else-if). Виконання кожного контракту записується як незмінна транзакція, що зберігається в блокчейні. Смарт контракти гарантують належний контроль доступу та виконання контрактів. Зокрема, розробники можуть призначити дозвіл доступу для кожної функції в контракті. Як тільки будь-яка умова в смарт-контракті буде задоволена, ініційований оператор автоматично передбачуваним чином виконає відповідну функцію. Наприклад, Аліса і Боб домовляються про покарання за порушення договору. Якщо Боб порушить договір, відповідний штраф (як зазначено в контракті) буде автоматично сплачено (утримано) з депозиту Боба.

Весь життєвий цикл смарт-контрактів складається з чотирьох послідовних фаз, як показано на рис. 1.2:

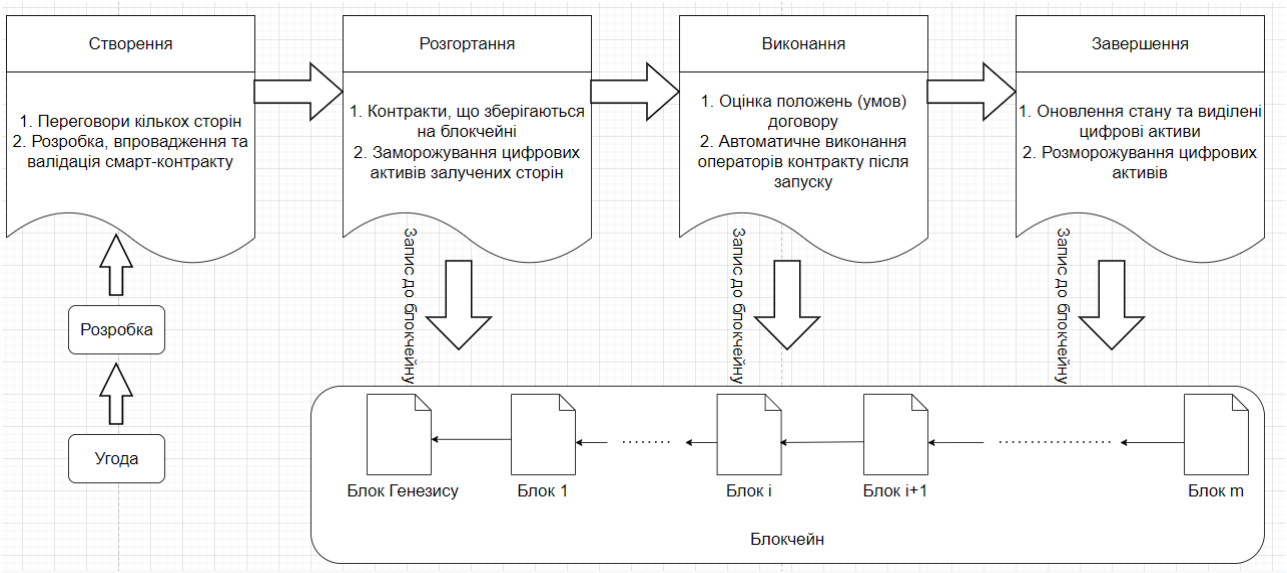

Рисунок 1.2 - Життєвий цикл смарт-контракту складається з чотирьох основних фаз: створення, розгортання, виконання та завершення

1) Створення смарт-контрактів. Кілька залучених сторін спочатку домовляються про зобов'язання, права та заборони за контрактами. Після кількох раундів обговорень можна досягти згоди. Юристи або консультанти допоможуть сторонам скласти початкову договірну угоду. Потім розробники ПЗ перетворюють цю угоду, написану природними мовами, у смарт контракт, написаний комп’ютерними мовами, включаючи декларативні мови та мови

правил на основі логіки [8]. Подібно до розробки комп'ютерного ПЗ, процедура конвертації смарт-контракту складається з проектування, впровадження та перевірки (тестування). Варто зазначити, що створення смарт-контрактів — це ітераційний процес, що включає кілька раундів переговорів та ітерацій.

2) Розгортання смарт-контрактів. Перевірені смарт-контракти потім можуть бути розгорнуті на платформах поверх блокчейнів. Контракти, що зберігаються в блокчейнах, не можуть бути змінені через незмінність блок-ланцюгів. Будь-яка зміна вимагає укладання нового договору. Після того, як смарт-контракти будуть розгорнуті на блокчейнах, усі сторони зможуть отримати доступ до контрактів через блокчейни. Крім того, цифрові активи обох сторін смарт-контракту блокуються шляхом заморожування відповідних цифрових гаманців. Наприклад, перекази монет (вхідних або вихідних) на гаманці, що стосуються контракту, блокуються. При цьому сторони можуть бути ідентифіковані за їхніми цифровими гаманцями.

3) Виконання смарт-контрактів. Як тільки умови контракту досягнуті (наприклад, отримання продукту), договірні процедури будуть автоматично виконані. Варто зазначити, що смарт-контракт складається з ряду декларативних операторів з логічними зв'язками. Коли умова спрацьовує, відповідний оператор буде автоматично виконано, отже, транзакція буде виконана та перевірена майнерами в блокчейнах [9]. Після цього здійснені транзакції та оновлені стани зберігаються в ланцюгах блоків.

4) Завершення смарт-контрактів. Після виконання смарт-контракту нові стани всіх залучених сторін оновлюються. Відповідно, транзакції під час виконання смарт-контрактів, а також оновлені стани зберігаються в блокчейнах. Тим часом цифрові активи були передані від однієї сторони до іншої (наприклад, переказ грошей від покупця до постачальника). Таким чином, цифрові активи залучених сторін були розблоковані. Тоді смарт контракт завершив весь життєвий цикл.

Такий опис фаз дозволяє категоризувати завдання, пов'язані з тестуванням контрактів [2]. Варто зазначити, що під час розгортання, виконання та

завершення смарт-контракту виконується послідовність транзакцій (кожна відповідає оператору в смарт-контракті) і зберігається в блокчейні. Таким чином, ці три фази записують дані в блокчейн.

1.3 Уразливості смарт-контрактів

1.3.1 Існуючі таксономії вразливостей.

Одна з перших академічних таксономій [3] класифікує вразливості відповідно до того, де вони з'являються: у вихідному коді (зазвичай Solidity), на машинному рівні (у байт-коді або пов'язаних із семантикою інструкцій) або на рівні блокчейну.

Таксономії, створені завдяки спільноті. Проект DASP [4], визначає десять груп уразливостей смарт-контрактів, проте не пояснює, як вони були відібрані та ранжовані. Деякі дослідження [5] використовують DASP Top 10 як основу, але наразі десять категорій недостатньо. SWC Registry [6] пов'язує вразливості смарт-контрактів з типологією Common Weakness Enumeration (CWE) і збирає тестові випадки. Наразі реєстр містить 36 вразливих місць з описами, посиланнями, пропозиціями щодо виправлення та зразками контрактів Solidity.

1.3.2 Перелік актуальних вразливостей

Ethereum тривалий час вважається найпопулярнішою платформою DApp [10]. Більшість DApps використовують Ethereum, а платформа має найбільшу кількість щоденних активних користувачів і смарт-контрактів. Після Ethereum найбільш широко використовуються EOSIO і BSC. Популярність Ethereum, швидше за все, пов'язана з тим, що це був перший блокчейн, який запропонував повні за Тьюрінгом (за винятком обмеження газу) смарт-контракти з достатньо низькою затримкою, прийнятною для більшості децентралізованих додатків. Смарт контракти Ethereum зазвичай написані мовою високого рівня, наприклад

Solidity або Vyper. Обидві мови компілюються в байт-код EVM. Нині існує безліч інших блокчейнів, які підтримують смарт-контракти. Решта цього розділу перелічує відомі вразливості смарт-контрактів, які можуть з'явитися на будь-якій із вищезгаданих платформ смарт-контрактів, але для прикладів використовується Ethereum.

1.3.2.1 Повторний вхід.

Уразливість повторного входу [11] може бути використана, коли функція контракту F, метою якої є вилучення коштів, синхронно викликає функцію D іншого ненадійного контракту. D може викликати F ще раз, перш ніж F оновить свій стан, таким чином знімаючи більше коштів, ніж слід було б. Існують клька способів виправити це. 1) F має оновити свій стан перед виконанням зовнішнього виклику до D – на (рис. 1.3) рядки 3 і 4 слід поміняти місцями 2)застосувати мьютекс, 3) використовувати оновлений метод call() і вказати обмеження на газ або замінити call() на transfer().

```
1 function withdraw(uint x) {
2   require(balances[msg.sender] >= x);
3   msg.sender.call.value(x)();
4   balances[msg.sender] -= x;
5 }
```

Рисунок 1.3 - Функція withdraw(), уразлива до повторного входу.

У 2016 році частина смарт-контракту DAO [11] була вразливою до повторного входу та піддалася атаці, яка призвела до втрати 3,6 мільйонів ETH або близько 50 мільйонів доларів США на той час. Саме ця подія спричинила хардфорк Ethereum. Початковий ланцюг Ethereum був перейменований в Ethereum Classic, а новий форк отримав оригінальну назву Ethereum.

1.3.2.2 Арифметичні проблеми.

Мова Solidity за замовчуванням не вловлює переповнення цілих чисел. Якщо не перехопити, то переповнення може призвести до несподіваної поведінки (рис. 1.4). Цілі числа без знака представлені 256 бітами в Solidity.

Отже, арифметична операція над цілим числом без знака, яка призводить до того, що результат буде більшим за $2^{256} - 1$ або менше 0 можуть бути використані.

```
1  function withdraw(uint x) {
2    require(balances[msg.sender] - x > 0);
3    msg.sender.transfer(x);
4    balances[msg.sender] -= x;
5  }
```

Рисунок 1.4 - Уразлива функція withdraw() для переповнення цілого числа. Що станеться, якщо x велике?

Одним із рішень цієї проблеми є використання безпечних функцій, наданих зовнішньою бібліотекою смарт-контрактів, таких як функції SafeMath OpenZeppelin, або використання мови, яка має вбудований захист від переповнень, наприклад Vyper.

1.3.2.3 Незахищена функція selfdestruct().

Інструкція SELFDESTRUCT EVM робить смарт-контракт недійсним та надсилає залишок коштів на адресу резервного гаманця. Неправильний модифікатор доступу або погана верифікація автора виклику може дозволити комусь видалити контракт. А якщо йому вдастся при цьому маніпулювати резервною адресою (напр. вона передається як параметр), то він присвоїть увесь баланс контракту.

Як правило, по можливості слід уникати використання самознищення. Проте, згадана вище атака на DAO тривала кілька днів, і організація навіть помітила, що в цей час їхній контракт зазнав нападу. Однак вони не змогли зупинити атаку або передати ефіри через особливість незмінності смарт-контрактів. Якби контракт містив функцію самознищення, організація DAO могла б легко передати всі ефіри та зменшити фінансові втрати. Інші переваги та недоліки використання selfdestruct() можна знайти в [1].

1.3.2.4 Проблеми видимості.

Рекомендується явно позначати видимість функції для всіх функцій і змінних. Видимість за замовчуванням є загальнодоступною для функцій у Solidity. Аналогічно, будь-які дані, які записуються в область зберігання EVM, можуть бачити будь-хто, оскільки вони зберігаються в блокчейні. Будь-який виклик до іншого контракту, включаючи аргументи функції, також є загальнодоступним, оскільки він створює транзакцію. Секрети не повинні зберігатися відкритим текстом у сховищі EVM.

1.3.2.5 Слабка випадковість.

Генерація випадкових чисел у смарт-контрактах є важкою проблемою. Розробники розумних контрактів, яким потрібні випадкові числа, можуть піддатися спокусі використовувати передбачувані дані ланцюга як джерело випадковості. Такі дані ланцюга включають номер блоку, хеш блоку та мітку часу блоку. Усіми цими значеннями можуть маніпулювати майнери блоків, і їх не слід використовувати для генерування випадкових чисел. Контракт SmartBillions є відомим прикладом поганого використання хешу блоків, що дозволило зловмисникам використати контракт і вивести всі гроші з лотереї. Одним з можливих рішень є використання безпечного генератора випадкових чисел для смарт-контрактів, таких як RANDAO або RBGC.

1.3.2.6 Залежність від порядку транзакцій.

Залежність від порядку транзакцій може призвести до несправедливої винагороди у випадку смарт-контрактів. Уявіть, що смарт-контракт реалізує вікторину, яка просить гравців вирішити проблему. Перший гравець, який надсилає правильне рішення може претендувати на приз. Припустимо, що Аліса довго працювала і нарешті знайшла рішення проблеми. Вона здійснює транзакцію, щоб його надіслати. Зловмисник бачить рішення Аліси в пулі транзакцій і негайно надсилає те саме рішення, але з більшою платою за транзакцію. Транзакція зловмисника в пріоритеті майнерів через вищі комісійні, які він сплачує, і ця транзакція вставляється в блок перед транзакцією Аліси.

Щоб запобігти цій атаці, можна використовувати схему зобов’язань commit-reveal [12], яка дозволяє гравцям безпечно розкривати рішення проблеми.

1.4 Методи, що використовуються в автоматизованому аналізі

У цьому розділі проводиться огляд методів, які використовуються для виявлення вразливостей у смарт-контрактах. Їх умовно можна умовно поділити чотири групи: статичний аналіз коду, динамічний аналіз коду, формальну специфікацію й перевірку та різне. Різниця між статичним аналізом і формальними методами є певною мірою довільною, оскільки останні здебільшого використовуються в статичному контексті. Більше того, такі методи, як символьне виконання, регулярно використовують формальні методи як чорний ящик. Ключовою відмінністю є прагнення формальних методів бути строгими, вимога коректності та прагнення до повноти. У цьому сенсі абстрактну інтерпретацію слід вважати скоріше формальним методом, але вона нагадує символьне виконання і тому представлена там.

1.4.1 Статичний аналіз коду

Статичний аналіз коду перевіряє код, не виконуючи його в звичайному середовищі. Аналіз починається або з вихідного, або з машинного коду контракту. У більшості випадків метою є виявлення шаблонів коду, які вказують на вразливості. Деякі інструменти також обчислюють вхідні дані, щоб викликати підозрювану вразливість і перевіряти, чи була атака ефективною, тим самим усуваючи помилкові спрацьовування.

Щоб розглянути різні методи, ми уважніше розглянемо процес компіляції програми з мови високого рівня, як-от Solidity, до машинного коду. Послідовність символів спочатку стає потоком лексем (що містять, наприклад, літери ідентифікатора). Синтаксичний аналізатор перетворює лінійний потік лексем в абстрактне синтаксичне дерево (АСД) і виконує семантичні перевірки.

Наступні фази отримують АСД у вигляді проміжного представлення (ПП). Після цього може відбутися кілька раундів аналізу коду, оптимізації коду та інструментування коду, з результатом кожного раунду знову в ПП. На заключній фазі ПП перетворюється в код для цільової машини, як EVM у випадку Ethereum. Цей останній крок лінеаризує будь-які ієрархічні структури, що залишилися, шляхом упорядкування фрагментів коду в послідовність і шляхом перетворення залежностей потоку керування в інструкції переходу.

1.4.1.1 Графи потоків керування

Для аналізу коду кращим є графічне представлення коду, оскільки воно забезпечує доступ до структури програми та потоку керування, а також усуває невідповідні деталі, такі як імена змінних або розподіл регістра. Такі уявлення легко доступні, якщо починати з вихідного коду, оскільки АСД і ПП є побічними продуктами компіляції. Наприклад, деякі інструменти шукають в АСД синтаксичні шаблони, характерні для вразливих контрактів. Цей підхід швидкий, але йому не вистачає точності, якщо вразливість не може бути належним чином охарактеризована такими моделями.

Відновлення графу потоку керування (ГПК) з машинного коду за своєю суттю є більш складним. Його вузли відповідають основним блокам програми. Базовий блок - це послідовність інструкцій, які виконуються лінійно одна за одною, і закінчується першою інструкцією, яка потенційно змінює потік керування, має, зокрема, умовний і безумовний стрибок. Вузли з’єднуються орієнтованим ребром, якщо відповідні базові блоки можуть виконуватися один за одним. Досяжність коду важко визначити, оскільки непрямі стрибки витягують цільову адресу з регістру або стеку, де вона була збережена за допомогою попередніх обчислень. Нарізка станів вирішує багато ситуацій, відстежуючи походження цілей стрибка. Якщо це не вдається, аналіз має вибір між надмірною або заниженою апроксимацією, або розглядаючи всі блоки як потенційних наступників, або ігноруючи наступників, які не можна виявити.

Деякі інструменти продовжують трансформувати ГПК (і специфікацію вразливості) в обмежену форму Horn Logic, яка називається DataLog, яка не є універсальною з точки зору обчислень, але допускає ефективні алгоритми міркування.

Починаючи з ГПК, декомпіляція намагається повернути назад також інші фази процесу компіляції з метою отримання вихідного коду з машинного коду. Результат призначений для ручної перевірки людьми, оскільки зазвичай він не повністю функціональний і не компілюється.

1.4.1.2 Символьне виконання

Символьне виконання — це метод, який виконує байт-код, як звичайна машина, але із символами як заповнювачами для довільних вхідних даних та даних середовища. Будь-яка операція над такими символами призводить до символьного виразу, який передається наступній операції. У випадку розвилки досліджуються всі гілки, але вони анотовані додатковими символьними умовами, які обмежують символи тими значеннями, які призведуть до виконання конкретної гілки. Через певні проміжки часу запускається розв’язувач SMT (Satisfiability Modulo Theories, задача розв’язності формул), щоб перевірити, чи є обмеження на поточному шляху одночасно задовольняються. Якщо вони суперечливі, шлях не відповідає реальним слідам виконання і може бути пропущений. В іншому випадку дослідження продовжуються. Коли символьне виконання досягає коду, який відповідає шаблону вразливості, повідомляється про потенційну вразливість. Якщо, крім того, SMT-розв’язувачу вдасться обчислити задовільне призначення для обмежень на шляху, його можна використовувати для розробки експлойту, який перевіряє існування вразливості.

Ефективність символьного виконання обмежується кількома факторами. По-перше, кількість шляхів експоненціально зростає з глибиною, тому аналіз не масштабується до великих програм. По-друге, деякі аспекти машини важко точно змоделювати, як-от зв’язок між сховищем і осередками пам’яті, або складні операції, напр. хеш-функції. По-третє, SMT-вирішувачі обмежені

певними типами обмежень, і навіть для них оцінка може закінчитися, а не виявити (не)розв’язність.

Concolic виконання перемежовує символьне виконання з фазами, де програма запускається з конкретним введенням (concolic = конкретне + символьне). Символьне виконання того самого шляху дає формальні обмеження, що характеризують шлях. Після скасування певного обмеження SMT-розв’язувач шукає відповідне значення. Використання його як вхідних даних для наступного циклу веде, за побудовою, до дослідження нового шляху. Таким чином, concolic виконання досягає кращого охоплення коду.

Аналіз плям (taint analysis) позначає значення з вхідних даних, середовища або зі сховища за допомогою тегів («плям»). Правила поширення визначають, як теги перетворюються інструкціями. Деякі вразливі місця можна визначити, перевіривши теги, що надходять у певні місця коду. Аналіз плям часто використовується в поєднанні з іншими методами, такими як символьне виконання.

1.4.1.3 Абстрактна інтерпретація

Більшість статичних методів виявлення вразливостей не є ні надійними, ні повними. Вони можуть повідомляти про вразливості там, де їх немає (хибно позитивні), і можуть не виявити вразливості, наявні в коді (хибно негативні). Перше обмеження виникає через неможливість вказати необхідні умови для наявності вразливостей, які можна ефективно перевірити. Друге є наслідком нездійсненно великої кількості обчислювальних шляхів, які необхідно досліджувати, і складності знайти достатні умови, які можна перевірити.

Абстрактна інтерпретація спрямована на повноту, зосереджуючи увагу на властивостях, які можна оцінити для всіх слідів виконання. Як приклад, абстрактна інтерпретація може розділити діапазон цілих чисел на три групи: нульових, додатних і негативних значень. Замість використання символьних виразів для отримання точного результату інструкцій, абстрактна інтерпретація пояснює, як властивість належності до однієї з трьох груп поширюється з

кожною інструкцією. Таким чином можна показати, що дільники в коді завжди належать до позитивної групи, виключаючи поділ на нуль для будь-якого входу. Завдання полягає в тому, щоб знайти властивість, яка є достатньо сильною, щоб спричинити відсутність певної вразливості, але достатньо слабкою, щоб дозволити досліджувати простір пошуку. На відміну від символьного виконання та більшості інших методів, цей підхід не вказує на наявність уразливості, але доводить, що контракт безперечно вільний від певної вразливості (гарантії безпеки).

1.4.2 Динамічний аналіз коду

Динамічний аналіз коду перевіряє поведінку коду, в той час, коли обробляє дані в його «природному» середовищі. Найпоширенішим методом є тестування, коли код запускається з вибраними входами, а його вихід порівнюється з очікуваним результатом.

Фазінг — це техніка, яка запускає програму з великою кількістю рандомізованих введених даних, щоб спровокувати збої або іншу несподівану поведінку.

Інструментарій коду доповнює програму додатковими інструкціями, які перевіряють ненормальну поведінку або контролюють продуктивність під час виконання. Тоді спроба використати уразливість може викликати виняток і припинити виконання. Як приклад, програма може бути систематично розширена за допомогою тверджень, гарантуючи, що арифметичні операції не викликають переповнення.

Машинне інструментування подібне до інструментування коду, але додає додаткові перевірки на рівні машини, дотримуючись їх для всіх контрактів. Наприклад, розширений EVM може перевіряти переповнення під час кожної арифметичної операції. Деякі автори йдуть ще далі, пропонуючи зміни до семантики транзакцій або протоколу Ethereum, щоб запобігти вразливостям. Хоча такі пропозиції цікаві з концептуальної точки зору, їх важко реалізувати,

оскільки вони вимагають хардфорка, який також впливає на вже розгорнуті контракти.

Тестування на мутації – це методика, яка оцінює якість наборів тестів. Вихідний код програми піддається невеликим синтаксичним змінам, відомим як мутації, які імітують поширені помилки при розробці програмного забезпечення. Наприклад, мутація може змінити математичний оператор або заперечити логічну умову. Якщо набір тестів здатний виявити такі штучні помилки, більш імовірно, що він також знайде реальні помилки програмування.

1.4.3 Формальна специфікація та перевірка

Програмісти віддають перевагу мовам програмування високого рівня перед ассемблером, оскільки це дозволяє їм виражати логіку програми більш абстрактним способом, знижуючи частоту помилок і прискорюючи розробку. Моделювання смарт-контрактів на ще більш високому рівні абстракції пропонує додаткові переваги, як-от формальні підтвердження властивостей контракту. Основну логіку багатьох блокчейн-додатків можна моделювати як кінцеві автомати (FSM) з обмеженнями, що відмічають переходи. Оскільки складові FSM є простими формальними об’єктами, такі методи, як перевірка моделі, можна використовувати для перевірки властивостей, зазначених у варіантах логіки дерева обчислень. Після того, як модель закінчена, інструменти переводять FSM у звичайний вихідний код, де можна додати додаткові функції.

Висока вартість помилок і малий розмір блокчейн-програм робить їх перспективною ціллю для формальних підходів до перевірки. На відміну від тестування, яке виявляє наявність помилок, формальна перевірка спрямована на підтвердження відсутності помилок і вразливостей. Як необхідна передумова, середовище виконання та семантика мови програмування чи машини мають бути формалізовані. Потім можна додати функціональні властивості та властивості безпеки, виражені певною мовою специфікації. Нарешті, автоматизовані

довідники теорем або напівавтоматичні помічники доведення можна використовувати, щоб показати, що дана програма задовольняє властивості.

F* — це функціональна мова програмування з помічником підтвердження для перевірки програми. Бхаргаван та ін. (2016) розробили фреймворк F*, який перекладає як вихідний код Solidity, так і байт-код EVM, на F*, щоб перевірити властивості високого та низького рівня. Грищенко та ін. (2018) використовують F*, щоб указати семантику малого кроку EVM.

KEVM — це формальна специфікація EVM на мові специфікації K (Hildenbrandt та ін., 2018). Відповідно до специфікації, фреймворк K може генерувати такі інструменти, як інтерпретатори та перевірки моделей, а також дедуктивні верифікатори програм.

Логіка Хорна є обмеженою формою логіки першого порядку, але все ще універсальна з точки зору обчислень. Вона становить основу програмування, орієнтованого на логіку, і є привабливою як мова специфікації, оскільки формули Хорна можна читати як правила «якщо-то».

1.4.4 Різне

Як і в багатьох інших областях, методи машинного навчання набувають популярності також в аналізі смарт-контрактів. Такі методи, як моделювання довгострокової пам’яті (LSTM), згорткові нейронні мережі або моделі мови N-грам, можуть досягти високої точності тесту. Поширеною проблемою є отримання маркованого навчального набору, який є достатньо великим і достатньої якості.

1.5 Огляд інструментів аналізу безпеки смарт-контрактів

Розглянемо деякі широко розповсюджені інструменти аналізу смарт-контрактів. Вони використовують статичний або гібридний (concolic execution) підходи, згадані в попередньому розділі.

1.5.1 Slither

Slither — це інструмент статичного аналізу, написаний на Python3. Slither використовує власну систему виявлення вразливостей, де виконуються три основні частини аналізу коду - читання та запис змінних, контрольований доступ («керовані функції») та залежність змінних від даних [13]. Потім до всієї системи підключаються детектори помилок, щоб виявити необхідні вразливості. Рис. 1.5 зображує внутрішню структуру інструмента статичного аналізу Slither.

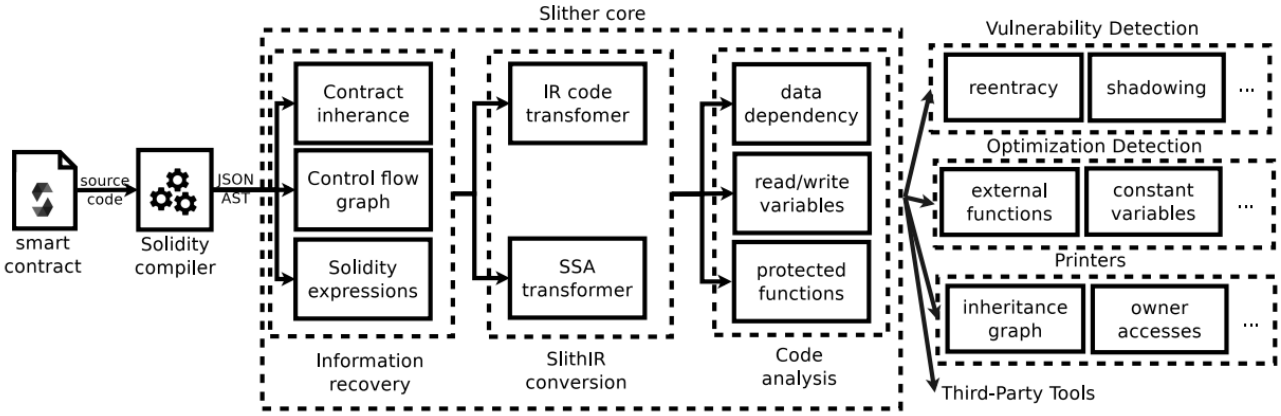

Рисунок 1.5 - Операційна схема Slither

У документації по детектору помилок Slither є 75 різних детекторів уразливостей для смарт-контрактів Solidity, починаючи від класичного повторного входу або tx.origin до мертвого коду, використання подвійного конструктора або існування типів тавтологій уразливостей. Slither добре задокументований і позначений кольорами, щоб визначити, які вразливості є критичними, а які лише рекомендаційними. Інструмент достатньо гнучкий, щоб розширювати його функціонал новими модулями аналізу та візуалізації.

1.5.2 SmartCheck

SmartCheck — написаний на Java у 2018. Він використовує генератор синтаксичних аналізаторів ANTLR і спеціальну граматику Solidity для побудови

АСД, які разом генерують дерево аналізу XML, що діє як проміжне представлення. Це цікавий підхід, якого немає в інших інструментах статичного аналізу, але XML, як правило, зручний, коли справа доходить до створення різних структур деревоподібного типу. У описі також згадується, що вразливості виявляються за допомогою запитів XPath, що фактично означає, що в аналізаторі є список уразливостей, жорстко закодований, і він намагається порівняти список із заданим шаблоном XPath. Це подібний підхід пошуку та звітування, який використовується плагіном статичного аналізу Remix IDE. Однак цей підхід простий, хоча він може бути неефективним щодо фрагментів коду, які мають той самий тип уразливості, який повинен знайти інструмент, але вихідний код не точно збігається з тими, які містяться в аналізаторах. Автори SmartCheck визнають, що більш складні шаблони можуть давати помилкові результати [14].

1.5.3 Oyente

Oyente, один з найпопулярніших інструментів автоматичного аналізу безпеки для смарт-контрактів EVM, використовує символьне виконання за допомогою розв’язувача обмежень Z3 [18], щоб знайти п’ять поширених помилок (а саме, залежність від порядку транзакцій, залежність від часових міток, неправильно оброблені винятки, арифметичні переповнення і повторний вхід). На рис. 1.6 показано архітектуру Oyente, яка приймає байт-код і поточний глобальний стан Ethereum як вхідні дані. Чотири компоненти, а саме CFG (Control Flow Graph) Builder, Explorer, Core Analysis і Validator, утворюють ядро Oyente. CFG Builder створює граф потоку керування смарт-контрактами, який живить провідник і візуалізатор графів. Провідник символьно виконує контракт, а результат потім передається в Core Analysis, де Oyente націлює на поширені помилки. Нарешті, Validator відфільтровує помилкові результати перед остаточним звітом. Серед 19 366 існуючих смарт-контрактів Oyente позначає 8 833 як вразливі, включаючи помилку DAO. Однак Oyente виявляє лише вразливі частки за допомогою помилкових результатів. Крім того, він реалізує лише

полегшену семантику байт-коду EVM і, таким чином, пропускає кілька важливих команд, що стосуються викликів і створення контрактів.

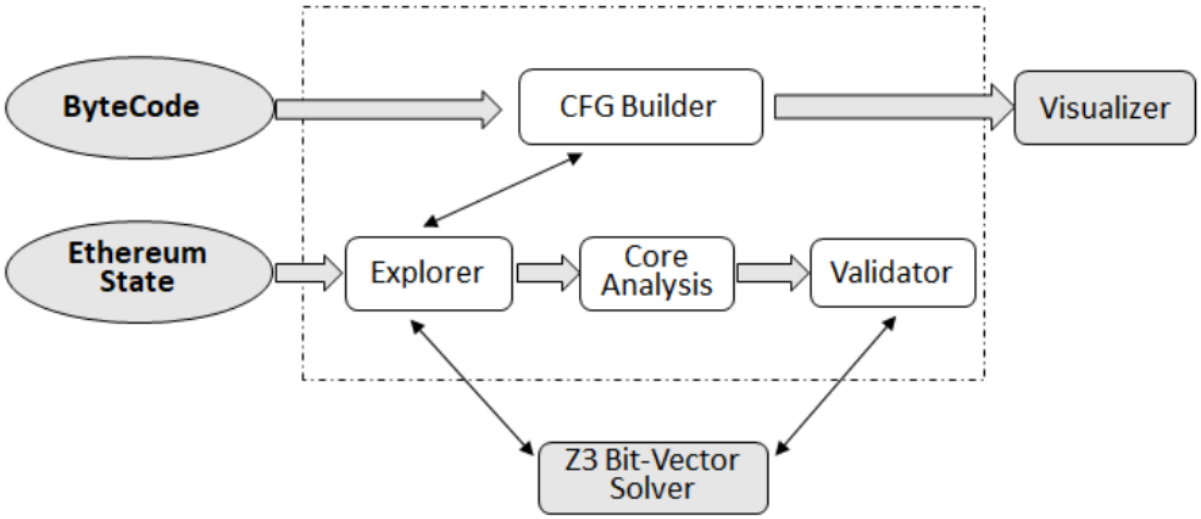

Рисунок 1.6 – Архітектура Oyente

1.5.4 Manticore

Manticore [15] використовує символьне виконання для пошуку унікальних шляхів обчислень у двійкових файлах EVM (і ELF). За допомогою SMT-вирішувача Z3 він знаходить вхідні дані, які запускатимуть ці шляхи обчислень. Він записує відповідні сліди виконання. Що стосується EVM, Manticore компілює код Solidity в байт-код для його аналізу, перевіряє сліди на наявність уразливостей, таких як повторний вхід і доступні операції самознищення, і повідомляє про них у контексті вихідного коду. Інформації про методи та їх обмеження мало. Інструмент можна використовувати з командного рядка або через API Python.

1.5.5 Securify

Securify є абстрактним інтерпретатором, який досліджує всі можливі шляхи. Крім того, Securify, є доступним для аналізу коду смарт-контракту за допомогою

веб-сторінки. Securify аналізує графік залежностей контрактів у стратифікованому журналі даних, щоб отримати точну семантичну інформацію з байт-коду EVM. Потім він перевіряє відповідність і шаблони порушень, щоб визначити, чи зберігається властивість безпеки. У порівнянні з Oyente, аналіз Securify повністю автоматизований.

1.5.6 Mythril

Mythril [16, 17] – символьно виконує байт-код EVM, при цьому використовує ті ж концепції, які згадувалися в Oyente і Slither, – бере стани програми та поміщає їх у ГПК, при цьому вузли містять дизасембльований код, а ребра позначаються формулами шляху. Тут також використовується SMT-вирішувач Z3 для скорочення простору пошуку та для обчислення конкретних значень для використання однієї потенційної вразливості. Перевагою mythril є використання concolic-виконання, що дозволяє спрощувати обчислення. Список вразливостей відомих Mythril пов’язаний з реєстром SWC.

Висновки до розділу 1

У розділі 1 було розглянуто роботу блокчейн систем, життєвий цикл смарт-контрактів, коротко описано поширені вразливості. Це дозволяє покращувати методики тестування, зокрема для виявлення слабких місць. Описано види автоматизованого пошуку помилок смарт-контрактів, зокрема статичні, динамічні, формальної специфікації та інші.

Перевагами статичних інструментів є, зокрема:

- можливість виявляти баги на ранньому етапі розробки,
- висока точність при перевірках граничних ситуацій та виключень,
- велика швидкість і простота у використанні.

Переваги динамічних засобів:

- перевірки результатів операцій, типів даних, підрахунки викликів,
- виявлення помилок, пов’язаних з доступом до пам’яті, ресурсів,
- врахування особливостей віртуального середовища,
- можливість провести аналіз бінарного файлу без вихідного коду.

Проведено огляд деяких популярних інструментів, що використовують статичний або гібридний(concolic execution) підходи. Ці програми мають відкритий вихідний код, але не завжди схильні до масштабування, це завдання потребує детальнішого аналізу в наступних розділах.

2 МЕТОД ВЕРИФІКАЦІЇ ПРОГРАМНОГО КОДУ СМАРТ-КОНТРАКТІВ

У цьому розділі проводиться аналіз методу символьного виконання на прикладі mythril. Оглядається архітектура цього інструменту і пропонуються модифікації для покращення точності.

Символьний аналіз байт-коду Ethereum

Під час символьного виконання інтерпретатор відстежує стани програми, з якими він стикається, і збирає обмеження на вхідні дані від предикатів, які зустрічаються в інструкціях розгалуження. Кожен виявлений шлях виконання може бути виражений у вигляді пропозиційної формули. Отримане представлення станів програми та потоку керування можна використовувати для підтвердження певних властивостей програми, визначення доступності станів помилки та виконання різних типів аналізу безпеки.

Хоча символьне виконання є дуже потужним теоретично, воно має деякі недоліки в реальних робочих програмах. По-перше, виявлення всіх можливих шляхів виконання в досить складній програмі вимагає багато пам’яті та часу. Функції операційної системи (такі як файлові системи, сокети та багатопотокове керування) також важко моделювати. Однак віртуальна машина Ethereum проста в порівнянні з операційною системою для настільних комп’ютерів або мобільних пристроїв, і це дозволяє досягти високого покриття шляхів за допомогою типових смарт-контрактів.

2.1 Моделювання виконання транзакції

Ethereum найкраще описувати як розподілений скінченний автомат на основі транзакцій: у будь-який момент часу вузли Ethereum погоджуються про world state, який з часом змінюється транзакціями, прийнятими в блокчейн.

World state – це відображення між адресами і станами облікових записів. У будь-який момент часу є результатом усіх модифікацій стану, починаючи зі

стану генезису. Облікові записи Ethereum можуть містити код («смарт контракти»), який виконується у віртуальній машині Ethereum (EVM) щоразу, коли надходить транзакція.

LASER-Ethereum [19] – символьний інтерпретатор байт-коду Ethereum. Враховуючи один або кілька облікових записів смарт-контрактів як вхідні дані, він повертає набір абстрактних станів програми. Стан складається з набору значень, які присвоєно змінним віртуальної машини (наприклад, програмний лічильник, стек віртуальної машини та залишки на рахунках) у певний момент під час виконання.

Жовтий документ [20,21] формально визначає скінченний автомат Ethereum і віртуальну машину, там описано три набори змінних стану:

- World State (σ): зіставлення адрес Ethereum з обліковими записами, які включають дані та баланс облікових записів.

- Стан машини (μ): програмний лічильник, пам'ять і стек віртуальної машини.

- Середовище виконання (I): змінні, що мають відношення до транзакції, яка в даний момент виконується (адреса абонента, вартість транзакції тощо).

У LASER загальний стан представлено об’єктом Python GlobalState, який містить стан машини, середовище та world state (рис. 2.1).

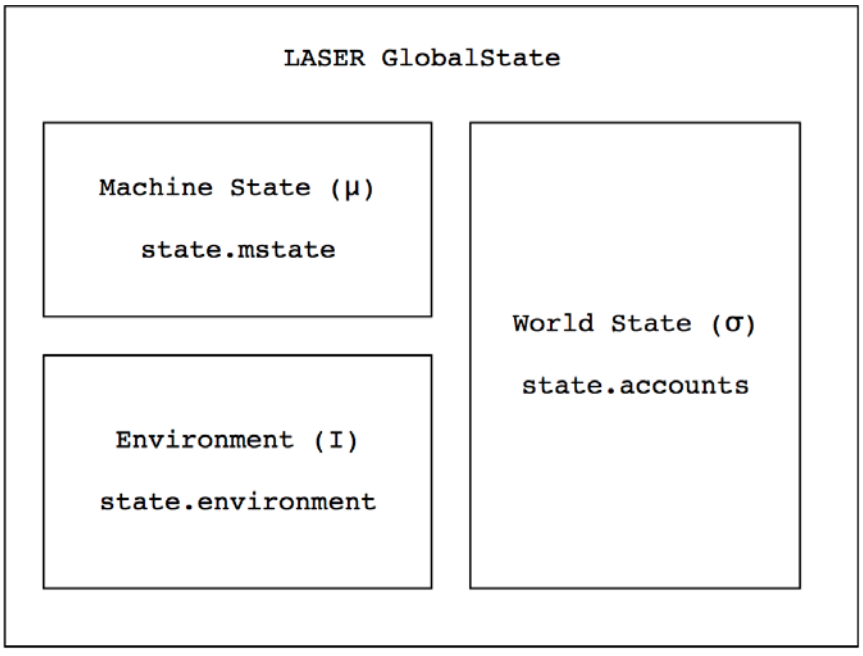

Рисунок 2.1 – об’єкт глобального стану

Отримання простору станів програми за допомогою LASER так само просто, як ініціалізація облікового запису контракту та виклик методу sym_exec():

```
>>> account = svm.Account(address, disassembly, "MyContract")
>>> accounts = {address: account}
>>>
>>> laser = svm.LaserEVM(accounts)
>>> statespace = laser.sym_exec(address)
>>>
>>> statespace.nodes[0].states [<laser.ethereum.svm.GlobalState
object at 0x10ae1e978>,
<laser.ethereum.svm.GlobalState object at 0x10ae1ea58>,...]
```

Об’єкт LASER GlobalState містить три члени: world state, стан машини та середовище. Розглянемо складові стану більш детально.

2.1.1 World state (σ)

Відповідно до жовтого документу [20], world state складається з відображення адрес Ethereum до облікових записів, кожен з яких має такі чотири поля:

- nonce: скалярне значення, що дорівнює кількості транзакцій, надісланих з облікового запису, або, якщо облікові записи мають пов’язаний код, кількості створених контрактів, створених обліковим записом. nonce позначається $\sigma[a]_n$.

- balance: скалярне значення, що дорівнює числу Wei, яким володіє зіставлена адреса. Позначається $\sigma[a]_b$.

- StorageRoot: 256-бітовий хеш кореневого вузла дерева Merkle Patricia, який кодує вміст сховища облікового запису (відображення між 256-бітними цілочисельними значеннями), що позначається $\sigma[a]_s$.

- codeHash: хеш коду EVM зіставленого облікового запису. Це код, який виконується, якщо зіставлена адреса отримує виклик; він незмінний і тому, на

відміну від усіх інших полів, не може бути змінений після побудови. Усі такі фрагменти коду містяться в базі даних стану під відповідними хешами для подальшого пошуку. Цей хеш формально позначається $\sigma[a]_c$, і, таким чином, код може бути позначений як b, якщо $KEC(b) = \sigma[a]_c$.

У LASER world state представлений словником, який відображає шістнадцяткове кодування адрес на об'єкти облікового запису:

```
>>>
state.accounts['0x0000000000000000000000000000000000000000'].as_dict()
{'nonce': 0,
'code': <mythril.disassembler.disassembly.Disassembly object at
0x106413940>,
'balance': balance, 'storage': {}
}
```

Реалізація відповідає жовтому паперу, за винятком поля codeHash. Замість хеша об'єкт облікового запису містить сам код у вигляді об'єкта Mythril Disassembly. Такий об'єкт може бути згенерований з байт-коду або вихідного коду за допомогою кількох рядків коду Python:

```
from mythril.ether.soliditycontract import SolidityContract
contract = SolidityContract("solidity_examples/underflow.sol",
"Under") disassembly = contract.get_disassembly()
```

2.1.2 Стан машини (μ)

Цитуючи жовтий документ [20], «стан машини μ визначається як кортеж (g, pc, m, i, s), елементами якого є наявний газ, програмний лічильник pc ∈ P256, вміст пам'яті, активна кількість слів у пам'яті (враховуючи безперервно від позиції 0) і вміст стека».

У LASER стан машини представлено GlobalState.mstate:

```
>>> state.mstate.as_dict()
{'pc': 0,
'stack': [],
'memory': [], 'memsize': 0,
```

```
'gas': 10000000
}
```

2.1.3 Середовище виконання (I)

Середовище виконання складається з наступних змінних:

- I_a, адреса облікового запису, якому належить виконуючий код.

- I_o, адреса відправника транзакції, яка ініціювала виконання.

- I_p, ціна газу в транзакції, яка виконується.

- I_d, вхідний масив байтів для виконання. Якщо агентом-виконавцем є транзакція, I_d — це дані транзакції.

- I_s, адреса облікового запису, який спричинив виконання коду.

- I_v, значення в Wei, передане цьому обліковому запису як частина процедури, якій належить виконання. Якщо агентом-виконавцем є транзакція, I_v є значенням транзакції.

- I_b, масив байтів, представлений машинним кодом, який буде виконано.

- I_H, заголовок поточного блоку.

- I_e, глибина поточного виклику повідомлення або створення контракту (тобто кількість викликів або CREATE, що виконуються).

У LASER середовище представлено GlobalState.environment:

```
>>> state.environment.as_dict()
{'active_account': <laser.ethereum.svm.Account object at 0x1064a4780>, 'sender': caller,
'calldata': [], 'gasprice': gasprice,
'callvalue': callvalue, 'origin': origin,
'calldata_type': <CalldataType.SYMBOLIC: 2>}
```

2.1.4 Потік керування

LASER організовує стани програми за допомогою графу потоку керування (ГПК) (рис 2.2). Кожен вузол графу представляє базовий блок коду, який виконується. Кожен вузол має набір умов шляху. Також надається список ребер між вузлами та обмеження для кожного ребра. Граф потоку керування не є

суворо потрібним для символьного аналізу, але він дає можливість додаткових корисних типів аналізу та відтворення приголомшливих візуальних зображень.

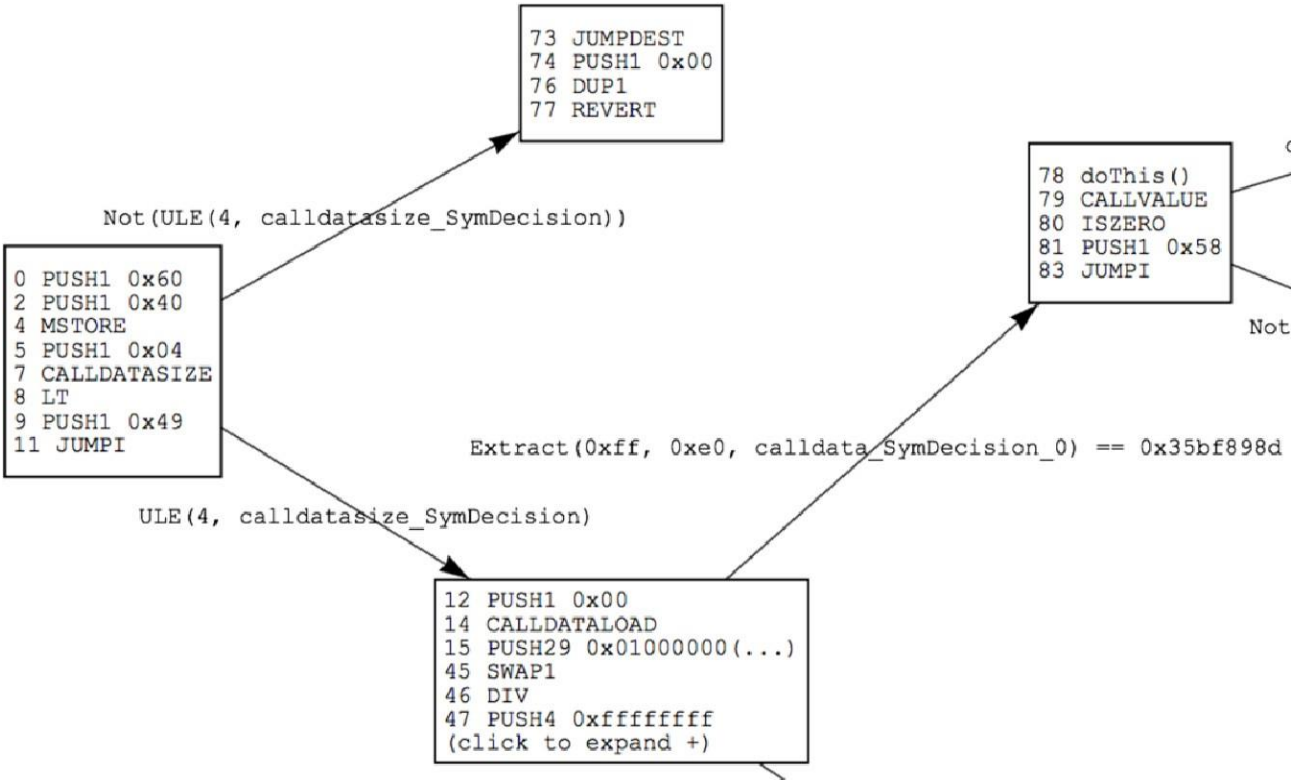

Рисунок 2.2 - Стани програми, відображені вздовж графу потоку керування. Вузли представляють основні блоки коду; Кожен рядок у вузлі представляє стан програми. На ребрах анотовані обмеження шляхів.

Ключовою концепцією символьного аналізу є «формула шляху», логічна формула, що складається з обмежень на заданому шляху. Формула шляху даного вузла є результатом логічного І всіх ребер на шляху до цього вузла. Формула шляху для досягнення функції doThis() на рис. 2.2:

```
calldatasize < 4 ^ calldata[0:4] = 0x35bf8089d
```

0x35bf8089d відповідає хешу підпису функції doThis().

2.1.5 Розв’язання SMT та формальні докази

Використовуючи LASER, можна представити виконання розумного контракту як простір станів і формул шляхів у пропозиційній логіці. Але яка користь для аналізу безпеки?

Можна створювати твердження («теореми»), використовуючи пропозиційну логіку, і намагатися довести або спростувати ці твердження в просторі виявлених абстрактних станів. Ми могли б, наприклад, запитати, чи існує можливий шлях до певної програми. Запитання, подібні до цього, можна виразити у вигляді задач на булеве виконання (SAT), наприклад: Чи можна задовольнити формулу a ∧ (b ∨ c) ∧ ¬ c ∧ ¬ (b ∧ d)?

Щоб вирішити ці проблеми, ми використовуємо автоматизовані інструменти міркування, які називаються SMT-вирішувачі. Розв’язувач може перевірити виконання логічних формул над однією або кількома «теоріями». EVM обчислює за допомогою 256-бітових біт-векторів, тому ми можемо використовувати теорію біт-векторів, щоб міркувати про ці обчислення.

2.2 Запропонований метод покращення виявлень для символьного виконання

2.2.1 Загальна архітектура

Оптимізація реалізована на основі фреймворку Mythril. На рис. 2.3 представлена загальна архітектура.

Існує п’ять основних компонентів, серед яких оптимізовані компоненти сірого кольору: (1) до LASER-Ethereum додано модуль статичного аналізатора; (2) модуль Symbolic Execution Engine змінено; (3) модуль детектора оптимізовано. Інші компоненти, включаючи Дизасемблер, Веб-переглядач та конструктор ГПК, були реалізовані Mythril та іншими існуючими інструментами. Деталі кожного модуля описані нижче.

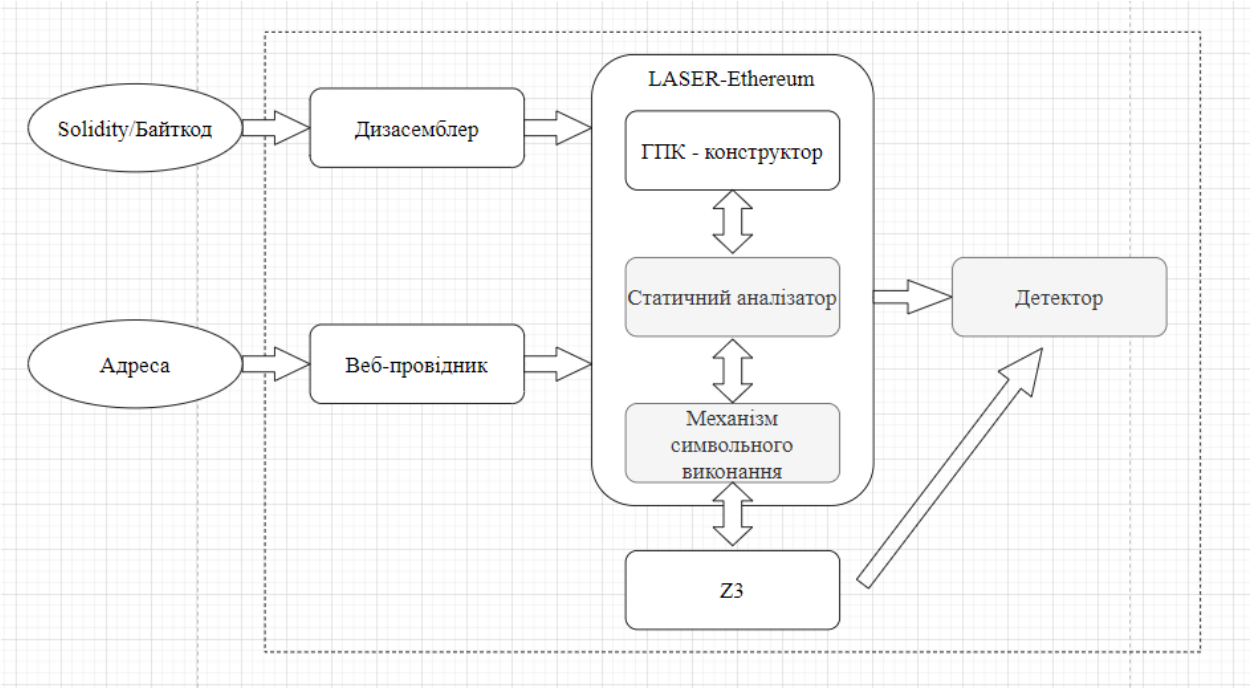

Рисунок 2.3 – Загальна архітектура

Модуль Дизасемблер використовується для компіляції вихідного коду та перетворення байт-коду в код операції. Веб-провідник отримує адресу, надану користувачами, і шукає відповідний договір в Інтернеті. LASER-Ethereum є основним компонентом Mythril, який виконує символьне виконання. Ця робота додає новий модуль, статичний аналізатор, до цього компонента. Цей модуль визначає конфіденційні інструкції та позначає критичні вузли. У LASER-Ethereum змінюється стратегія пошуку шляху її оригінального Механізму символьного виконання. SMT-розв'язувач Z3 [18] використовується для виконання завдань розв'язування обмежень. Детектор – це компонент, який здійснює виявлення вразливостей. Тут можна покращити логіку виявлення, яка детально описана в розділі 2.3.

2.2.2 Граф потоку керування

ГПК може бути представлений четвіркою G = (N, E, En, Ex), де N – набір вузлів, кожен з яких означає базовий блок. E – набір ребер, що вказують на структуру розгалуження та структуру циклу програми. Враховуючи смарт-контракт, створення ГПК включає наступні кроки:

(1) Перетворення вихідного коду або байт-код у послідовність читабельних інструкцій коду операції за допомогою компілятора.

(2) Розкладення інструкції коду операції на основні блоки. Межі входу та виходу базових блоків визначаються згідно з деякими спеціальними інструкціями коду операції.

(3) Визначення зв'язків базових блоків і побудова ребер ГПК.

(4) Розв’язати висячі блоки за допомогою моделювання стека.

2.2.3 Визначення критичних шляхів

Хакери використовують уразливості смарт-контрактів з метою отримання економічних вигод, наприклад, крадіжки токенів або заморожування коштів. Смарт контракти зазвичай дозволяють лише авторизованим обліковим записам Ethereum приймати токени. Якщо контракт дозволяє надсилати Ether на адресу, контрольовану зловмисником, вона вважається вразливою. Відповідно, шляхи, які беруть участь у передачі ефіру, визначаються як критичні.

1) Чутливі інструкції

Критичний шлях повинен містити чутливу інструкцію, яка може реалізувати або вплинути на передачу ефіру. Хоча всі інструкції вимагають газу, що технічно робить їх пов’язаними з передачею ефіру, вони не вважаються чутливими інструкціями, оскільки споживання газу в більшості випадків не є вразливим місцем. Посилаючись на набір інструкцій смарт-контрактів [22], розглянемо ті, що визначаються як конфіденційні.

Сховище зберігає змінні стану смарт-контрактів назавжди, і його може змінювати лише SSTORE. Зловмисник може писати в певний індекс сховища через SSTORE, щоб здійснити незаконну передачу.

Функції Solidity, такі як send(), transfer() і call().value(), компілюються в байт-коди серії CALL для передачі Ether. Інструкції серії CALL включають CALL, STATICCALL, CALLCODE і DELEGATECALL. Якщо їх параметром

«адреса» можна керувати, контракт абонента може відправити ефір на адресу, зазначену зловмисником, або бути введеним у будь-який шкідливий код.

SELFDESTRUCT знищує поточний контракт і надсилає його баланс на вказаний рахунок, наданий параметром «адреса».

2) Критичні шляхи

Шляхи, що стосуються передачі ефіру, вважаються критичними. Під час побудови ГПК для кожного вузла встановлюється прапор _is_Critical_Node, щоб вказати, чи містить базовий блок конфіденційні інструкції. Якщо вузол містить будь-який з SSTORE, CALL, CALLCODE, DELEGATECALL, STATICCALL і SELFDESTRUCT, прапор _is_Critical_Node встановлюється на 1, а вузол називається критичним вузлом (CN). Шлях програми, який проходить через CN, є критичним.

2.2.4 Автоматизоване тестування смарт-контракту, відповідно до цілей

Щоб якомога швидше охопити всі критичні шляхи в автоматизованому тестуванні, розроблено алгоритм автоматизованого тестування, як показано в Алгоритмі 2.1. По-перше, набір шляхів-кандидатів β та вхідний вектор програми λ (випадково ініціалізований на λ_0) ініціалізуються. Коли шлях ω виконується під вхідним вектором λ, стан цільового вузла CovCN перезаписується. Потім механізм виконання символів збирає всі шляхи-кандидати (тобто префікси шляхів). Відповідно до цільового набору критичних вузлів, який не було охоплено, непов’язані шляхи-кандидати усуваються, а досяжні шляхи-кандидати додаються до набору шляхів. Після цього, відповідно до стратегії пошуку шляху, зазначеної заздалегідь, шлях-кандидат P вибирається як наступний для дослідження. Вирішивши обмеження шляху P, можна отримати новий тестовий вхід λ. Попередні кроки повторюються до тих пір, поки не будуть пройдені всі шляхи-кандидати або доки покриття коду не досягне очікуваного значення.

Кінцевий результат включає тестові випадки, які відповідають цільовому охопленню.

Алгоритм 2.1 Цільовий алгоритм автоматизованого тестування

```
Input: β: candidate path set, G: CFG, λ: input vector,
ψ: path search strategy, CovCN: already covered CN,
UncovCN: uncovered CN
Output: T : test cases
1: β ← Ø,CovCN ← Ø, UncovCN ← CN, λ ← λ0
/*initialization*/
2: repeat
3: ω ← DSE-EXEC(λ) /*dynamic symbolic execution*/
4: CovCN ← Update(CovCN, ω) /*update*/
5: UncovCN ← CN – CovCN
6: path← SelectPath (ω) /*collect candidate paths*/
7: β ← FilterPath (UncovCN, ω, path)
/*unrelated path pruning*/
8: P ← NextPath(β, ψ) /*select the next path based
on ψ */
9: λ ← SolveConstraints(P) /*new testcase generation */
10: T ← T + {λ}
11: until β = Ø ∨ UncovCN = Ø ∨ CoveragePercent ≥ θexpect
```

Ключовими модулями в автоматизованому процесі тестування є обрізання неров'язаних шляхів і стратегія пошуку шляху.

1) Стратегія пошуку багатоцільно-орієнтованого шляху

Пошук шляху є основною частиною динамічного виконання символів. Його функція полягає у виборі стану програми, який потрібно виконати першим до досягнення цілі. Стратегією пошуку шляху за замовчуванням для більшості існуючих інструментів виконання символів, таких як Mythril, є пошук у глибину (DFS). DFS проходить шляхи програми від кореня ГПК. Щоразу, коли досягається вузол розгалуження, символьний механізм виконання клонує поточний стан програми та переходить до однієї з гілок. Процес пошуку припиняється, якщо вивчені всі шляхи або наближено до цільового місця.

Алгоритм DFS є вичерпним у заданому просторі станів і підходить для повного тестування охоплення шляхів програми. Йому бракує активного позиціонування та дослідження критичних шляхів, схильних до дефектів, що призводить до низької ефективності.

Евристичний пошук більш адаптивний до наближення до чутливих інструкцій та швидкого охоплення критичних шляхів. Такі алгоритми подібні до пошуку в ширину (BFS), за винятком того, що вони оцінюють кожне місце розташування та здійснюють пошук із найкращого. Іншими словами, алгоритми евристичного пошуку переважно досліджують уздовж вузлів. Ці вузли можуть бути найкращим способом досягти цілі. Ключем до евристичного алгоритму пошуку є функція оцінки, яка зазвичай розробляється для конкретної задачі. Функція оцінки оцінює вартість переходу від конкретного вузла до вузла призначення. Використання евристичного пошуку може уникнути багатьох непотрібних шляхів і підвищити ефективність дослідження.

Враховуючи ситуацію, коли один шлях виконання проходить повз кілька цілей, або найкоротший шлях із кількох цілей проходить через ту саму гілку поточного шляху виконання, тобто існує область перекриття. Якщо є кілька шляхів-кандидатів програми, вибір тих шляхів, які можуть досягти більшої кількості цілей, може прискорити тестування покриття. Тому в цій роботі використовується евристична стратегія пошуку шляху, заснована на пріоритеті шляху, яка називається багатоцільовий орієнтований пошук шляху.

(1) Пріоритетна оцінка шляху кандидата

Для кожного вузла розгалуження вже виконаного шляху оцінюється щільність решти цільових вузлів уздовж його локальної області, і щільність означає пріоритет відповідного шляху-кандидата. Пріоритет насправді є евристичною оцінкою, яка визначається як кількість цілей, які можуть бути досягнуті після виконання шляху-кандидата, виражена як

$$priority(P) = UncoveredCN(Br, SearchCN(Br, \gamma))$$

У формулі Br є кінцевим вузлом шляху-кандидата. Функція SearchCN(Br, γ) служить для початку з Br і кроку вперед для рівнів γ, підраховуючи кількість

невідновлених CN, які можуть бути досягнуті. Якщо l – висота Br у ГПК, область пошуку буде обмежена шаром [l, l + γ]. Діапазон шарів можна налаштувати, встановивши значення γ. Оскільки прогноз локальної області інтуїтивно менш дорогий, ніж глобальний прогноз, зазвичай вибирається менше значення.

(2) Вибір шляху кандидата

Щоб досягти якомога більшої кількості цілей покриття під час одного виконання, спочатку вибирається шлях-кандидат з найбільшим значенням пріоритету. Тобто шляхи ранжуються на основі їхнього внеску в охоплення цільових вузлів. Якщо два шляхи-кандидати мають однакове значення пріоритету, перевага надається коротшому. Після ітерації програми значення пріоритету всіх інших шляхів-кандидатів оновлюються.

Як показано на рис. 2.4, вузли, позначені тінню, є CN. Якщо виконується шлях 1-2-3-5-8, уздовж гілки шляху додаються два шляхи-кандидати, а саме 1-2-3-6 і 1-2-4. Відповідно до політики багатоцільового орієнтованого пошуку шляху, якщо γ встановлено на 3, шлях 1-2-4 має вищий пріоритет виконання, оскільки він може досягати 2 неохоплених CN.

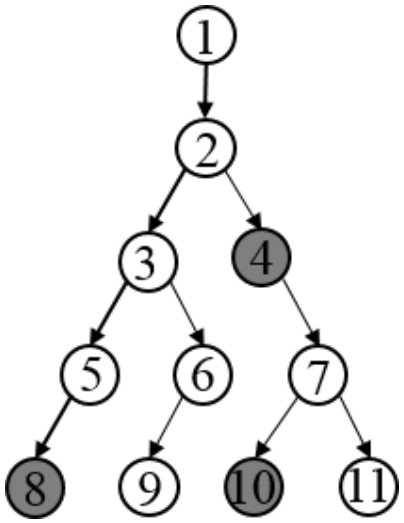

Рисунок 2.4 - Вибір шляху кандидата

2) Обрізка непов’язаних шляхів

Щоб розрізнити шляхи-кандидати, дається таке визначення: якщо шлях-кандидат, імовірно, досягне критичного вузла, який не був охоплений

перевіреним контрактом, то це відповідний шлях, інакше це непов’язаний шлях. Алгоритм 2.2 показує метод обрізання непов'язаних шляхів.

Алгоритм 2.2 Алгоритм обрізання непов'язаного шляху

```
Input: β: candidate path set, UncovCN: uncovered CN,
path: newly collected candidate path set
Output: β: candidate path set after path pruning
1: for all P ∈ path do
2:    /*P is path prefix, rather than a complete path*/
3:    if isRelevant( )==True then
4:         β.push(P)
5:    end if
6: end for
7: function isRelevant(Path P):
8: if Const.solve()== Ø then
9:    return False
10: end if
11: if P.SuccsnUncovCN == Ø then
12:   return False
13: end if
14: return True
```

2.2.5 Аналіз плям

Для допомоги модулю виявлення використовується техніка аналізу плям. Для певних результатів введення програми або операції, які можуть генерувати вразливості, до нього додається символьна змінна зі спеціальною назвою як тег плями. Ця символьна змінна приймає значення 0 і називається «TAINTVAR_PC», в якій «PC» є значенням програмного лічильника.

Як показано на рис. 2.5, мітка плям поширюється на наступні гілки разом з потенційно небезпечними даними, беручи участь у обчисленні без зміни результатів.

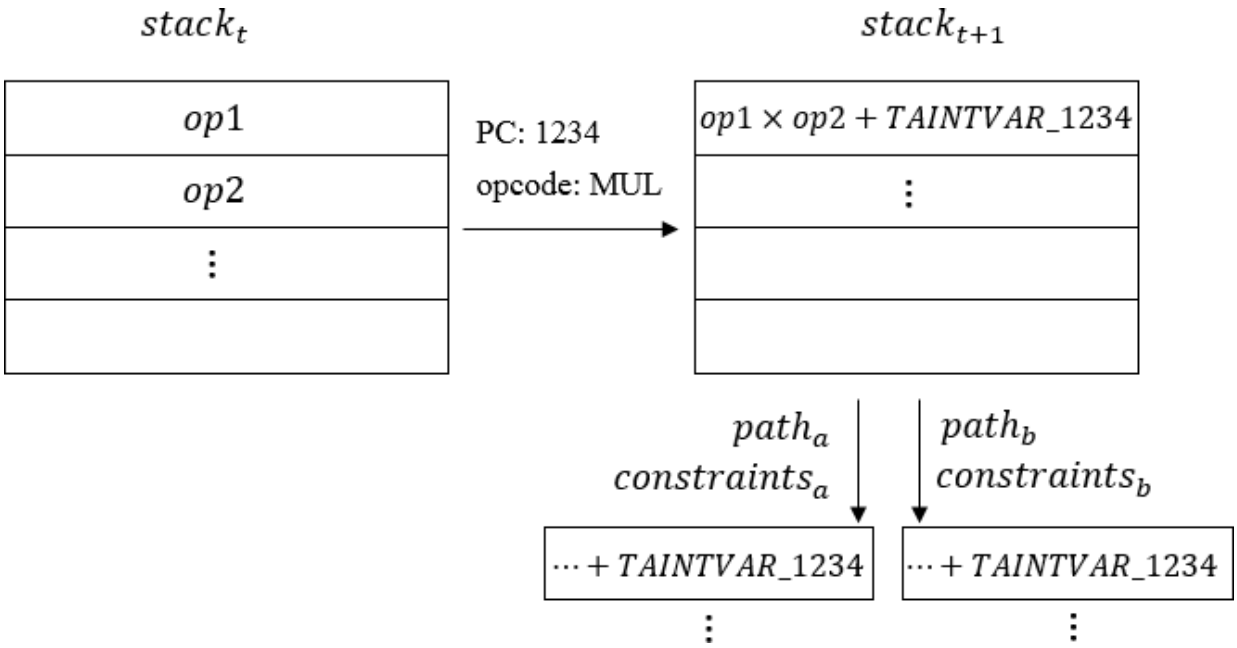

Рисунок 2.5 - Відстеження плям.

Процес відстеження плям полягає у виконанні аналізу потоку даних відповідно до правила поширення. Правило поширення визначає список операцій, які можуть розповсюджувати повідомлення про порушення або можуть призвести до нових повідомлень під час виконання. У певних пунктах програми критичні для безпеки параметри або змінні стану оцінюються як заплямовані чи ні за запитом, залежно від того, несуть вони теги плям чи ні. Якщо критичні для безпеки дані небезпечної точки забруднені, її джерело забруднення можна визначити відповідно до мітки забруднення.

2.2.6 Формування обмежень

SMT-вирішувач Z3 використовується для вирішення обмежень і допомоги механізму символьного виконання. Елементи фіксованої довжини, такі як значення виклику або адреса абонента, представлені бітвектором фіксованої довжини. Елементи змінної довжини, такі як дані виклику, пам'ять і сховище, моделюються за допомогою виразів масиву Z3. Коли виконання досягає чутливої інструкції, параметри в стеку отримуються з поточного простору станів і будуть

використані для подальшого аналізу. Для даного шляху P вираз обмеження є логічним I всіх умов розгалуження на шляху. Кінцевим результатом модуля генерації обмежень є формула для змінних у стані машини (μ) і середовищі виконання (I).

2.3 Уразливості та логіка виявлення

У попередньому підрозділі обговорювалася базова структура виявлення вразливостей смарт-контрактів на основі покриття критичного шляху. Обмеження для критичних шляхів та інформація про поширення плям були отримані за допомогою символьного виконання. На цій основі модуль виявлення вразливостей виконує перевірку безпеки. У поєднанні з досвідом аудиту смарт-контрактів і інцидентами безпеки, у цій роботі вибрано декілька типів уразливостей, описані властивості їх виконання та розроблена логіка виявлення відповідно до цього розділу.

2.3.1 Повторний вхід

1) Опис.

Коли контракт A викликає інший контракт B, A чекає завершення виконання виклику, перш ніж перейти до наступної інструкції за звичайних умов. Але якщо контракт, що викликається, B перериває цей виклик, викликаючи резервну функцію A, змушуючи контракт A працювати у невідповідному внутрішньому стані, що, очевидно, порушує наміри розробника і може призвести до несподіваної поведінки.

На основі численних випадків уразливості та відповідної літератури підсумовуються три режими повторного входу.

(1) Повторний вхід з тією ж функцією

Повторний вхід з однаковою функцією означає повторне введення тієї ж функції контракту. Це найпоширеніший метод повторного входу.

(2) Міжфункціональний повторний вхід

Цей режим означає, що один і той же контракт повторно вводиться в різні функції. Смарт-контракти, як правило, забезпечують кілька інтерфейсів, які можуть читати або записувати ті самі внутрішні змінні стану, що робить міжфункціональні атаки повторного входу такими ж небезпечними, як і атаки повторного входу з однаковими функціями.

(3) Повторне створення контракту

У Solidity новий контракт можна створити з ключовим словом «new» і реалізувати за допомогою команди CREATE на рівні EVM. Після створення нового контракту конструктор контракту виконується негайно, під час якого він може здійснювати виклики інших шкідливих контрактів. Можливий спосіб атаки: контракт-жертва має намір створити новий контракт, а потім оновить свій внутрішній стан. Однак конструктор нового контракту здійснює зовнішній виклик на адресу, контрольовану зловмисником, повторно входячи в контракт жертви та експлуатуючи неузгоджений внутрішній стан.

Незалежно від режиму повторного входу, ключовим моментом є те, що контракт виконується у непостійному внутрішньому стані після зловмисного повторного входу. Таким чином, три умови становлять вразливість повторного входу: (1) зовнішній виклик до іншого контракту; (2) змінна пам'яті, яка викликає невідповідність, використовується для управління рішенням потоку під час зовнішнього виклику; (3) змінна оновлюється після повернення із зовнішнього виклику.

2) Логіка виявлення

Серед існуючих інструментів для пошуку вразливостей смарт-контрактів Oyente підтримує лише виявлення повторного входження з однаковою функцією, Securify і Mythril можуть виявляти часткове міжфункціональне повторне входження, в той час як ці інструменти не можуть виявити вразливість повторного входу при створенні контракту.

Згідно з описом уразливості вище, перевірка оновлень стану є ключем до виявлення вразливостей повторного входу. Mythril використовує таку стратегію:

виявити всі виклики повідомлень, які надсилаються на вказану користувачем адресу, і доставляють газ. Функції send() і transfer() Solidity встановлюють газ на 2300, і це налаштування може запобігти атакам повторного входу. Якщо виявлено зовнішній виклик на ненадійну адресу, аналізується ГПК, щоб визначити, чи може відбутися зміна стану після повернення з виклику. Якщо виявлено оновлення стану, видається попередження. Однак ця стратегія занадто консервативна. Він позначає будь-яке оновлення стану після зовнішнього виклику як уразливість, незалежно від того, чи насправді оновлення стану викликає небезпеку. Тому генерується багато помилкових спрацьовувань. Наприклад, кожна критична функція захищена мьютексом, повторний виклик не дасть досягнути якого-небудь виконання, але аналізатор бачить лише, що виклик функції доступний.

Для покращення більш розумною логікою виявлення варто перевірити, чи може оновлений стан вплинути на рішення потоку керування, це представлено на рис. 2.6. Оскільки спільні змінні між контрактами завжди зберігаються в сховищі, а змінна зберігання є єдиною внутрішньою змінною стану, яка може вплинути на потік керування при повторному введенні контракту, для виявлення потрібно лише звернути увагу на змінні в сховищі. Тому використовується методика аналізу плям. Змінні пам’яті позначаються як потенційно вразливі і відстежуються в програмі. Якщо рішення потоку керування в інструкції умовного переходу JMPI залежить від деяких змінних пам’яті, то зловмисник може маніпулювати напрямком розгалуження, повторно ввівши контракт, таким чином маніпулюючи поведінкою контракту. Потім записується набір змінних пам’яті, що використовуються для керування рішенням потоку, і призначається тег VAR_FLAG=1. Якщо попередній виклик контракту намагається оновити змінну зі значенням тегу 1, видається попередження про вразливість повторного входу.

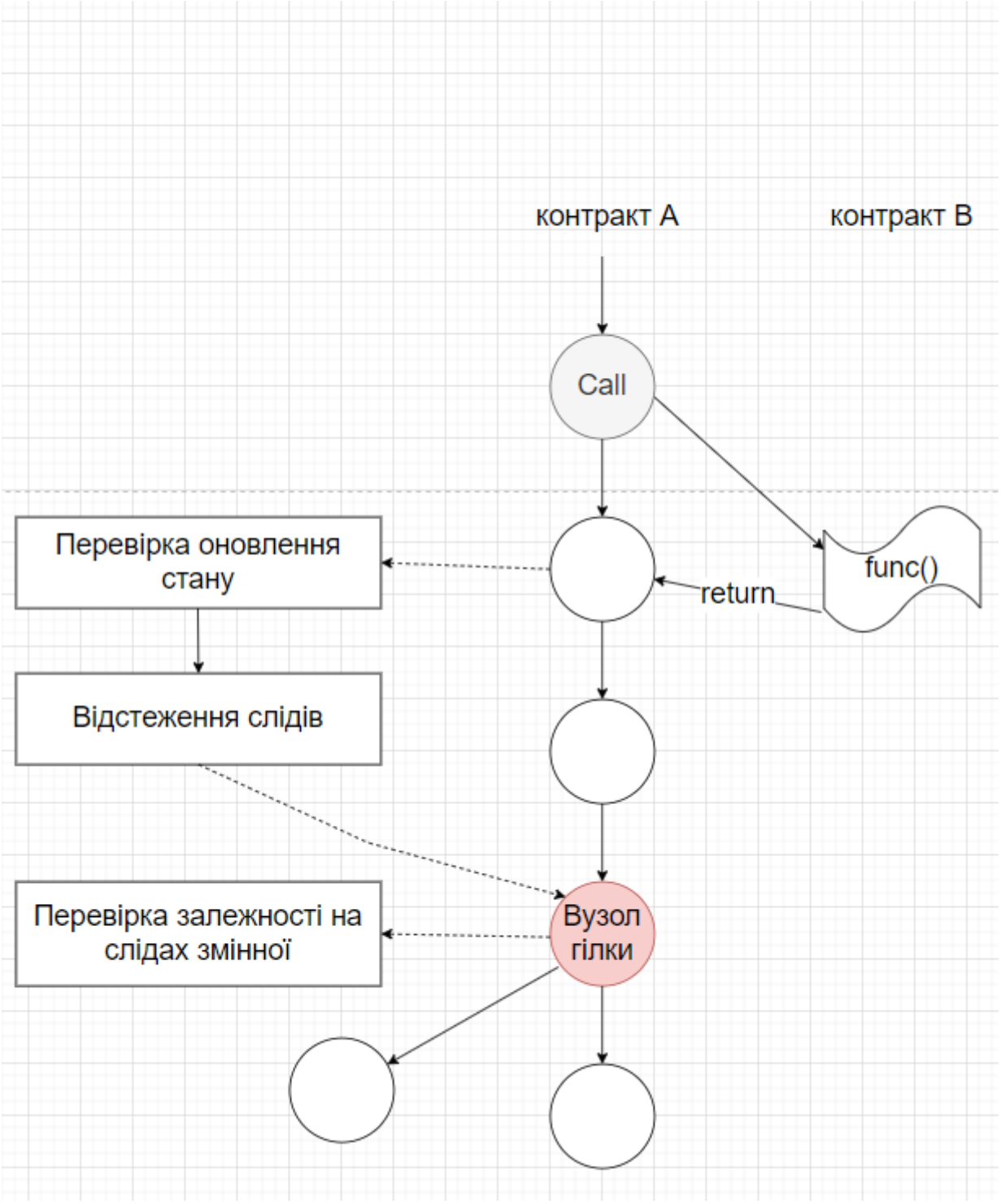

Рисунок 2.6 – логіка виявлення повторного входу

2.3.2 Переповнення цілочисельних типів

1) Опис.

Цілочисельне переповнення включає overflow і underflow. Відповідні коди операцій: ADD, MUL і SUB. Для SUB, якщо op1>op0, може виникнути недостатній поток. Для інструкцій ADD або MUL, якщо op1 + op0>2^{32} − 1 або op0 × op1>2^{32} −1, може статися переповнення. Чи можуть ці операції справді викликати уразливість, залежить від двох аспектів. По-перше, чи виконується перевірка переповнення або обмеження введення в контексті програми, тобто чи може операція переповнення бути успішно виконана і, таким чином, створювати точку переповнення. По-друге, як використовується це значення переповнення, тобто чи може це значення завдати шкоди. Якщо значення переповнення застосовується до критичного місця, це може призвести до серйозних наслідків і навіть викликати інші вразливості. таке критичне місце визначається як небезпечна точка. Точки небезпеки переповнення цілих чисел зазвичай поділяються на наступні три категорії.

(1) Запис у пам'ять: позначені плямою дані постійно зберігаються контрактом як глобальна змінна стану, наприклад, баланс рахунку.

(2) Оператор розгалуження: в умовному операторі гілки значення переповнення контролює напрямок гілки, змушуючи дані обійти перевірку безпеки або спричинити виконання програмою незаконної операції. Пов’язані випадки включають відомі випадки вразливості SMT (SmartMesh Token) і BEC (Beauty Chain).

(3) Передача даних: позначені плямою дані передаються до зовнішньої функції для невідомих операцій.

2) Логіка виявлення.

Рис. 2.7 показує логіку виявлення переповнення цілого числа, яка виконується в три кроки.

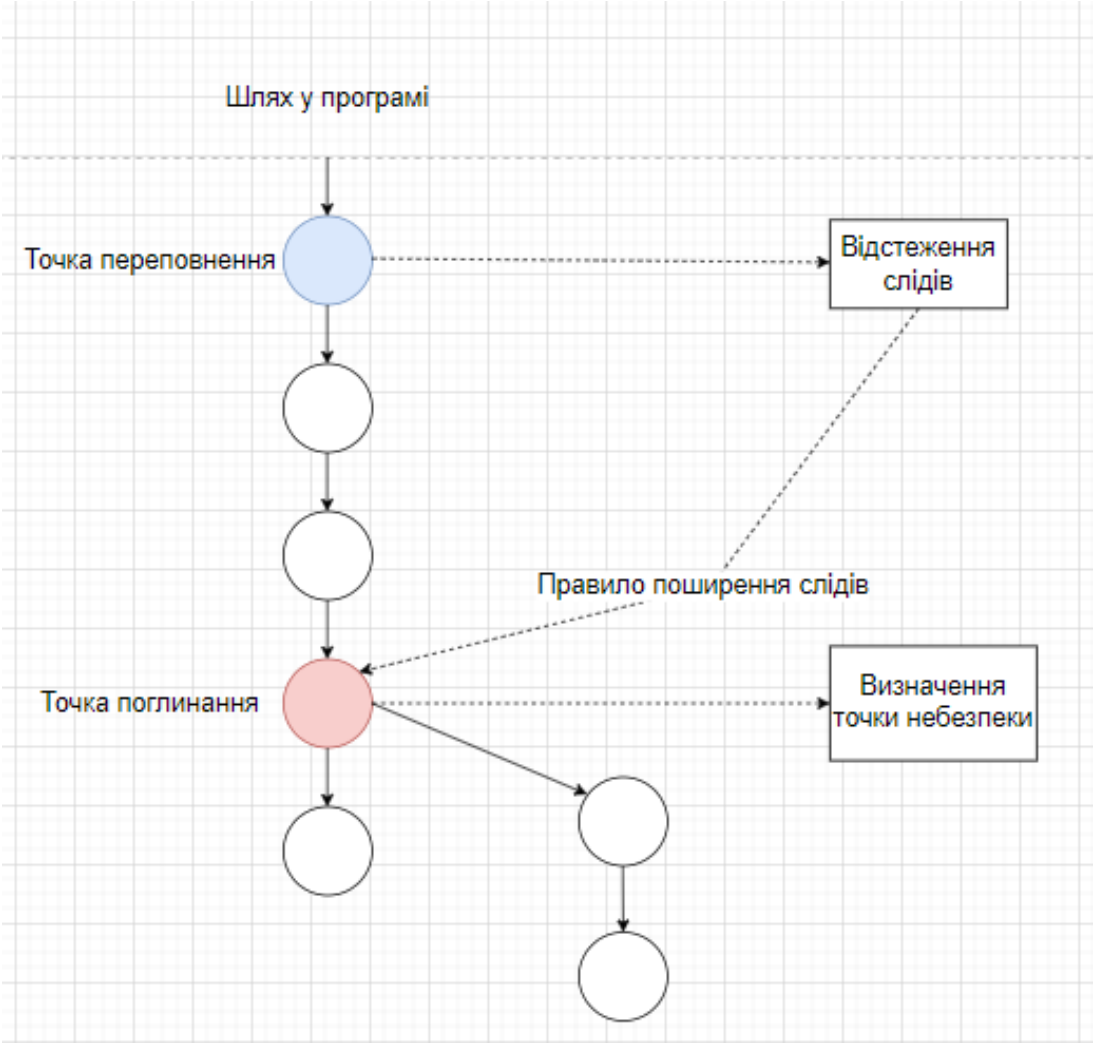

Рисунок 2.7 – логіка виявлення для переповнення цілих типів

(1) Виявлення точки переповнення

Визначення точки переповнення здійснюється на основі інструкцій ADD, MUL і SUB. Щоб визначити, чи пов’язані з ними операції призведуть до переповнення, два операнди, op0 і op1, отримують з вершини стека, а потім цілі операнди перетворюються в 256-бітовий бітвектор. Після цього, згідно з інструкцією операції, будується вираз судження про переповнення. Нарешті, розв’язувач Z3 використовується для обчислення того, чи має вираз розв’язок. Нижче наведено вирази переповнення, що відповідають синтаксису Z3.

Overflow: Or(And(ULT (expr, op0), op1! = 0), And(ULT (expr, op1), op0! = 0))

Where, for ADD instruction, expr = op0 + op1; for MUL instruction, expr = op1 × op0.

Underflow: UGT (op1, op0)

Якщо обмеження шляху поточного стану та виразу переповнення можуть бути задоволені одночасно, це точка переповнення.

(2) Відстеження плям.

Результат роботи точки переповнення не обов'язково завдає шкоди. Тому, коли точка переповнення знайдена, вона позначається як вихідна точка для аналізу слідів та відстежується відповідно до правила поширення слідів. На цьому кроці буде отримана вся інформація про «забруднення» програми.

(3) Визначення точки небезпеки

Ідентифікація небезпечних точок – це виявлення критичних операцій у критичних точках програми (точках поглинання), на які впливає інформація про забруднення. Точка небезпеки – це місце, де фактично вступає в дію значення переповнення. Відповідно до аналізу точки небезпеки, наведеного вище, точки програми, що працюють за інструкцією SSTORE та інструкцією JMPI, вважаються точкою поглинання. Якщо інструкція SSTORE записує в сховище позначені плямою дані, або умова переходу JMPI не залежить від таких даних, вважається, що існує вразливість переповнення цілого числа.

2.3.3 Зловживання delegateCall()

1) Опис.

Прототип функції delegatecall може бути представлений як address.delegatecall(. . .) returns(bool). Він викликає функцію адреси відповідно до ідентичності контракту абонента і повертає false, якщо виконання не вдається. За замовчуванням весь доступний газ передається, а кількість газу регулюється. Значення вбудованої змінної msg.sender після виклику не змінюється для абонента, але середовище викликаного контракту є середовищем виконання виклику. Неправильне використання delegatecall дозволить контракту використовувати код інших контрактів без передачі власного стану (наприклад, балансу, даних акаунту), що призведе до виконання неочікуваного коду.

Наприклад, якщо користувач передає цільову адресу, яку потрібно викликати, і послідовність символів, тоді можна викликати функцію будь-якої адреси. Крім того, якщо викликаюча і та, що викликається мають однакову змінну, і викликана функція змінює значення цієї змінної, то модифікується перша, якщо середовище виконання delegatecall не змінювалось.

```
    Contract Wallet{
     function () payable {
     if(msg.value>0)
          Deposit(msg.sender, msg.value);
     else if(msg.data.length>0)
          _walletLibrary.delegatecall(msg.data);
     }
    }
    Contract WalletLibrary{
     function initWallet (address[] _owners, uint _required,
uint _daylimit){
          initDaylimit (_daylimit);
          initMultiowned(_owners, required);
     }
    }
```

У контракті Wallet вище, рядок 6 виконує делегуючий виклик і передає параметр msg.data, щоб можна було викликати будь-яку публічну функцію в walletLibrary. Таким чином, зловмисник може викликати функцію initWallet() у рядку 11 і стати власником контракту. Після цього він може відправити ефір з гаманця на свою адресу.

2) Логіка виявлення.

Алгоритм 2.3 показує логіку виявлення зловживання делегуючим викликом. Для всіх зовнішніх викликів у формі DELEGATECALL, якщо викликана функція є резервною функцією, третій елемент $\mu_s[-3]$ можна отримати у верхній частині стека. Цей елемент відноситься до початкової адреси необхідних параметрів, що зберігаються в пам'яті, яка може бути представлена як μ_m [$\mu_s[-3]$] відповідно до моделі транзакції. Якщо змінна адреси є конкретним значенням, перевіряється рядок з місця в пам’яті. Якщо рядок містить дані виклику, контракт передає дані виклику через DELEGATECALL у функції fallback, що означає, що будь-яка

функція у викликаному контракті може бути виконана, а викликаний контракт може змінити дані у пам’яті того, що викликає. Якщо викликаний контракт є символьним значенням, це вказує на те, що цільова адреса делегуючого виклику надається користувачем. Наступним кроком є визначення того, чи отримана адреса з даних виклику чи зі сховища, в обох випадках викликаний контракт може отримати доступ до змінної стану контракту абонента без обмежень.

Алгоритм 2.3 Логіка виявлення зловживання делегуючим викликом

```
Input: statespace, issues[ ]
Output: issues[ ]: security defects
1: issues[ ] ← ∅
2: for all delegatecall.func_name == fallback do
3:    if is(_Concrete(μₘ [μₛ[-3]])==True) and
(Search(calldata, μm [μₛ[-3]])==True) then
4:         issues.Push(Dele_issue1)
5:    else
6:         if ‘‘calldata’’ in str(call.to) then
7:              issues.Push(Dele_issue2)
8:         end if
9:         if Storage_Write(str(call.to),storage_idx) then
10:             issues.Push(Dele_issue3)
11:        end if
12:   end if
13: end for
14: report(issues)
```

2.3.4 Залежність від порядку транзакцій

1) Опис.

Кожен блок містить набір кількох транзакцій. Для користувачів порядок транзакцій у межах одного блоку невидимий (доступ до нього мають лише майнери). Тому стан блоку також невизначений. Припустимо, що блок зараз знаходиться в стані σ і містить дві транзакції, T1 і T2. T1 і T2 викликають один і той же договір одночасно. За таких обставин користувач не може знати стан контракту, оскільки він залежить від порядку виконання T1 і T2.

Уразливість TOD характеризується тим, що користувач може змінити стан поточного контракту (змінити змінні в сховищі) шляхом оформлення транзакції, а змінна може впливати на адресу призначення або значення токена, передане зовнішнім викликом.

2) Логіка виявлення.

Алгоритм 2.4 показує логіку виявлення TOD. Для всіх зовнішніх викликів спочатку знаходяться усі змінні пам'яті, які можуть вплинути на значення виклику або адресу виклику, а потім визначається, чи можуть вони змінитися за поточних обмежень вузла. Якщо можливо, додається змінну до відповідного списку даних Instor. Потім для кожного елемента в Instor отримується операція SSTORE, яка може його змінити, і записується відповідний (стан, вузол) кортеж. Якщо така операція SSTORE існує, є вразливість TOD.

Алгоритм 2.4 Логіка виявлення для TOD

```
Input: statespace, issues[ ]
Output: issues[ ]: security defects
1: issues[ ] ← ∅, Instor[ ] ← ∅
2: for all statespace.calls do
3:    Instor ← Relevant_storage(call.val, call.to)
4:    sstore_tupple ← Write_to(Instor)
5:    if sstore_tupple then
6:         issues.Push(TOD_issue)
7:    end if
8: end for
9: report(issues)
```

2.3.5 Незахищена функція selfdestruct()

1) Опис.

Якщо контракт відповідає будь-якій з наведених нижче характеристик, він вважається договором самознищення. (1) До інструкцій SELFDESTRUCT може отримати доступ будь-хто. (2) Індекс зберігання, де зберігається адреса параметра інструкції SELFDESTRUCT, не обмежується msg.sender.

2) Логіку виявлення вразливості самознищення можна виразити формулою нижче.

$$\texttt{Instruction}(\mu_{pc}) == \texttt{SELFDESTRUCT} \wedge \mu_s[-1]! = I_C \wedge \texttt{Solve}(\texttt{node}(\mu_{pc}).\texttt{constraints})! = \emptyset$$

Для інструкції SELFDESTRUCT отримується верхній елемент стека $\mu_s[-1]$ і визначається, куди надсилається баланс після виконання команди самознищення (включаючи адресу абонента, адресу, збережену в сховищі, у параметрі функції, переданому через дані виклику, вказана адреса тощо). Потім отримується обмеження, що абонент не має I_C творця контракту. Якщо автор договору обмежує абонента, попередження не видаватиметься. І навпаки, якщо абонент не обмежений, повідомляється про вразливість самознищення, а рівень небезпеки визначається на основі адреси призначення $\mu_s[-1]$ передачі маркера.

2.3.6 Залежність від передбачуваних змінних

1) Опис.

Щоб написати функцію генерації випадкових чисел, розробники контрактів часто використовують параметри, пов'язані із заголовками блоків, такі як block.gaslimit, block.number, block.timestamp, block.difficulty і block.coinbase (адреса майнера поточного блоку), або іншу інформацію про блоки, наприклад дані, що зберігаються в контрактах, для генерування випадкових чисел. Проте властивості параметрів заголовка блоку можуть дізнатися майнери. Крім того, дані про ланцюжок часто прозорі. Таким чином, випадкові числа, що залежать від цих передбачуваних змінних, насправді є передбачуваними, що дає зловмиснику можливість скористатися цим. Якщо контракт надсилає Ether з кількістю більше 0 на основі обчислення передбачуваних змінних, існує вразливість залежностей передбачуваної змінної.

2) Логіка виявлення.

Алгоритм 2.5 показує логіку виявлення залежності від передбачуваної змінної.

Алгоритм 2.5 Логіка виявлення залежності від передбачуваної змінної

```
Input: statespace, issues[ ]
Output: issues[ ]: security defects
1: issues[ ] ← ∅, Predvar ←[‘‘coinbase’’, ‘‘gaslimit’’,
‘‘timestamp’’, ‘‘number’’, ‘‘difficulty’’]
2: for all statespace.calls do
3:    if is_Concrete(call.value) or call.value == 0 then
4:        continue
5:    end if
6:    Var_List ←Search(Prevar, call.to+node.constraints)
7:    for Var in Var_List do
8:        Reso ← Solve(node.constraints)
9:        if Reso then
10:           issues.Push(PVD_issue)
11:       end if
12:   end for
13:   Bh ← Search(‘‘blockhash’’, call.to+node.constraints)
14:   for Var in Bh do
15:       Reso ← Solve(node.constraints)
16:       if Reso then
17:           Find_Blocknum(‘‘blockhash’’,str(node.
constraints))
18:           issues.Push(PVD_issue)
19:       end if
20:   end for
21: end for
22: report(issues)
```

Для всіх інструкцій серії CALL у просторі станів фільтрується функція повернення коштів, а також інструкції з call.value, що вказують або дорівнюють 0 (не надсилають ефір). Для кожної з інших інструкцій проводиться двоетапний тест.

Спочатку знаходяться передбачувані змінні стану, згадані вище, в обмеженні шляху поточної інструкції або обмеженні об’єкта, що викликається. Якщо така змінна існує, додається відповідний вузол до списку. Для кожного елемента в списку викликається розв’язувач обмежень для рішення і, нарешті,

повідомляється про вразливість і вказується, від якої змінної середовища він залежить.

Далі знаходиться одержувач Ether, який залежить від хешу блоку. У рядку обмежень поточної інструкції або викликаного об’єкта, якщо рядок ‘‘blockhash’’ можна знайти, додатково оцінюється хеш, який блок використовується, і видається попередження. Зазвичай пов’язаний блок на кілька блоків попереду поточного блоку, який може бути обчислений за допомогою block.number мінус вказане ціле число або число, що зберігається в індексі зберігання.

2.3.7 Необроблені винятки

1) Опис.

Solidity надає дві функції: assert() та require(), щоб перевірити умови та створити виняток, якщо умова не виконується. Функцію assert слід використовувати для виявлення внутрішніх помилок і перевірки інваріантів. Функцію require слід використовувати для забезпечення виконання таких умов, як введення користувача та змінні стану контракту, або для перевірки значень, що повертаються від викликів зовнішніх контрактів. Solidity виконує операцію повернення (код операції 0xfd) для винятку в стилі вимоги і виконує недійсну операцію (код операції 0xfe), щоб викликати виняток у стилі assert. В обох випадках EVM скасовує всі зміни, внесені в стан поточного виклику та його субвикликів, а також видасть помилку абоненту. Винятки в стилі Assert зазвичай викликаються помилками типу, поділом на нуль, доступом до масиву поза межами, викликом ініціалізованої нулем змінної внутрішнього типу функції тощо. Правильно функціонуючий контракт ніколи не наблизиться до невдалого оператора assert. Інакше невдале твердження може зробити його нездатним досягти початкової мети.

2) Логіка виявлення.

Щоб запобігти подібним уразливостям, інструменти безпеки повинні перевірити, що контракт ніколи не виконуватиме недійсні коди операцій. Таким чином, метод виявлення полягає у розв'язанні обмежень шляху для кожного коду операції 0xfe. Якщо є рішення, умова невдалого підтвердження може бути виконана, і має бути повідомлено попередження. Логіку виявлення можна виразити у наведеній нижче формулі.

$$\texttt{Instruction}(\mu_{pc}) == \texttt{0xfe} \wedge \texttt{Solve}(\texttt{node}(\mu_{pc}).\texttt{constraints}) != \emptyset$$

2.3.8 Неперевірені значення, повернені функціями

1) Опис.

Виклики зовнішніх функцій будуть невдалими, якщо вони перевищують максимальний стек викликів у 1024 або закінчиться газ. У таких випадках Solidity створює виняток, який зазвичай «вибухає» автоматично під час додаткового виклику. Але це не стосується функції send() і функцій низького рівня, включаючи call(), delegatecall(), callcode() і staticcall(). Ці функції не генерують виняток, а повертають логічне значення «False», і контракт продовжить виконання так, ніби виклик був успішним. Таким чином, про те, чи успішно виконано договір, не можна судити лише за наявністю винятків.

2) Логіка виявлення.

Як показано в Алгоритмі 2.6, основна ідея цього модуля виявлення полягає в тому, щоб шукати, чи ігнорується повернуте значення виклику зовнішньої функції. Для всіх вузлів повернення виклику перевіряється наявність інструкції «ISZERO» у відповідному блоці. Для кожної інструкції «ISZERO» отримується елемент у верхній частині стека $\mu_s[-1]$, а потім визначається, чи містить елемент retval. Якщо вищезазначена умова виконана, це означає, що виконується IZSERO(retval). Коли retval дорівнює 0, стан повертається. В іншому випадку перевірка повернення значення пропущена, і має бути видано попередження.

Алгоритм 2.6 Логіка виявлення неперевірених значень функцій

```
Input: statespace, issues[ ]
Output: issues[ ]: security defects
1: issues[ ] ← ∅,
2: for node in CallRet_node do
3:    if ‘‘ISZERO’’ in node.opcode and Search(‘‘retval’’,
      μs[-1]) then
4:         retcheck←True
5:    end if
6:    retcheck←False
7:    issues.Push(RET_issue)
8: end for
9: report(issues)
```

Висновки до розділу 2

У даному розділі розглянуто механізм аналізу смарт-контрактів, на прикладі mythril, виявлено недоліки його архітектури. Сформульовано модифікований метод пошуку вразливостей на основі символьного виконання, а також описано його основні кроки. Крім цього описані деталі роботи алгоритмів виявлення для кількох поширених типів вразливостей.

3 АНАЛІЗ РОЗРОБЛЕНОГО МЕТОДУ НА ПРАКТИЦІ

3.1 Технологія mythril на прикладі

Традиційним варіантом використання символьного виконання є двійковий злом. Для прикладу опис вирішення завдання CTF із використанням LASER, що було показане на презентації mythril. На практиці такі задачі не виникають (як часто потрібно зламати DRM для смарт-контрактів?), але це чудовий спосіб зрозуміти можливості символьного виконання та Z3 Solver. Представлене завдання Ethereum CTF було розроблено Корантіном Огюстом для Тулузської конвенції хакерів [26].

У описі представлений байт-код смарт-контракту і сказано, що контракт дуже простий: він містить метод win(), який може викликати будь-хто. Цей метод приймає рядок як параметр. Якщо рядок є прапорцем, він викличе "selfdestruct(msg.sender)", щоб відправити всі свої гроші користувачу і деактивуватись. Прототип функції win: function win(bytes23 flag).

Аналіз починається зі створення графу викликів і дизасемблювання за допомогою Mythril:

```
    $ myth --max-depth 64 -g ./ctf.html -c
"6060604052[...]8260f"
    $ myth -dc "6060604052[...]8260f"
    (...)
    269 PUSH32
    0x0631194a95069d7e012c19795d0c5c4ccd4af1984e45570000000000
00000000
    302 DUP4
    303 PUSH9 0xffffffffffffffffff
    313 NOT
    314 AND
    315 EQ
    316 ISZERO
    317 PUSH2 0x0159
    320 JUMPI
```

```
321 CALLER
322 PUSH20 0xffffffffffffffffffffffffffffffffffffffff
343 AND
344 SUICIDE
(...)
```

У даному фрагменті є інструкція SUICIDE, яку потрібно виконати за адресою програмного лічильника 344. За адресою 320 є умовний стрибок (JUMPI), який залежить від порівняння вхідних даних із жорстко закодованим рядком. На графі потоку керування (рис 3.1) є цикл декодера XOR, представлений двома повторюваними базовими блоками. Ці блоки виконуються загалом 23 рази (цикли завжди розгортаються в просторі станів, тому для кожної ітерації створюється окремий вузол).

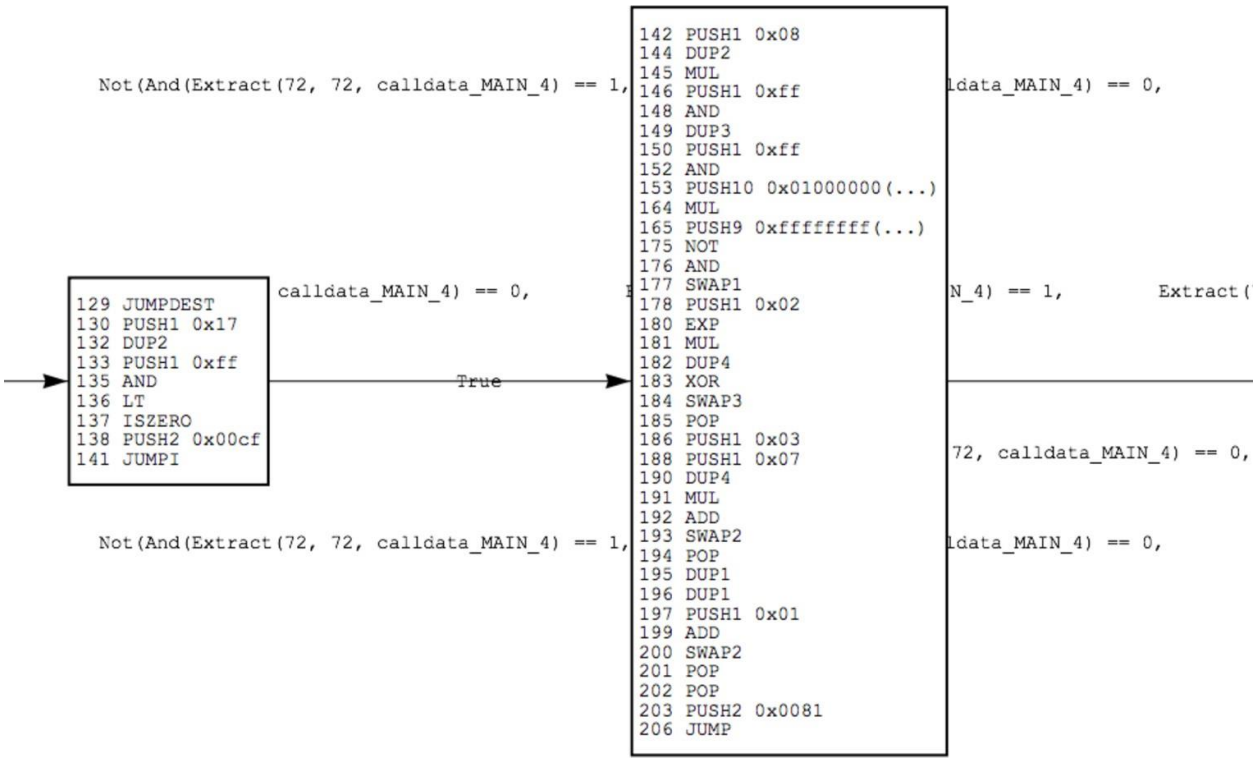

Рисунок 3.1 - Цикл декодера XOR (два блоки повторюються 23 рази).

Завдяки символьному виконанню тут не потрібно турбуватися про те, що саме робить декодер або що містить жорстко закодований рядок. Все, що потрібно зробити, це знайти стан, в якому виконується інструкція SUICIDE, і запитати у Z3 рішення формули шляху цього стану. Тут навіть не потрібно писати зайвий код, тому що Mythril за замовчуванням шукає незахищені

інструкції SUICIDE. Прапорець --verbose-reports означає повідомити про рішення.

```
    $ myth -m suicide --max-depth 64 --verbose-report -xc
"606060(…)0029"
    ==== Unchecked SUICIDE ==== Type: Warning Contract: MAIN
Function name: _function_0x50f753bd PC address: 344 The
function _function_0x50f753bd $ myth -m suicide --max-depth 64
--verbose-report -xc "606060(…)0029"

    ==== Unchecked SUICIDE ====
    Type: Warning
    Contract: MAIN
    Function name: _function_0x50f753bd
    PC address: 344
    The function _function_0x50f753bd executes the SUICIDE
instruction. The
    remaining Ether is sent to the caller's address.executes
the SUICIDE instruction. The remaining Ether is sent to the
caller's address.
    It seems that this function can be called without
restrictions.
    --------------------
    DEBUGGING INFORMATION:
    SOLVER OUTPUT:
    calldata_MAIN_4:
    0x5448437b776f775f737563685f45564d5f736b696c6c7d0000000000
00000000
    calldata_MAIN_0:
    0x50f753bd0000000000000000000000000000000000000000000000
00000000
    calldatasize_MAIN: 0x4
    callvalue: 0x0
```

У розділі «DEBUGGING INFORMATION» calldata_MAIN_4 показує вхідні дані, необхідні для досягнення інструкції SUICIDE за адресою програмного лічильника 344: це байти ASCII прапора, який треба знайти.

```
    >>>
    bytes.fromhex("5448437b776f775f737563685f45564d5f736b696c6
c7d").decode("UTF
```

```
8")
'THC{wow_such_EVM_skill}'
```

3.2 Особливості інструментів, виявлені в ході експерименту

Mythril і Oyente є найбільш репрезентативними і широко використовуваними інструментами символьного виконання для безпеки смарт-контрактів. З огляду на обмеження цих подібних інструментів, для великих контрактів із великою кількістю інструкцій виявлення зазвичай неефективне або навіть закінчується збоєм. Для уникнення цього, mythril дозволяє використовувати опції "--max-depth" та "--call-depth-limit", щоб обмежити глибину рекурсії та викликів відповідно. Також дослідники пропонують зосереджуватись лише на області коду, яка бере участь у передачі ефіру. У перевірці ефективності також було використано slither, що використовує модель АСД.

Mythril зазвичай виводить одну і ту ж уразливість кілька разів, наприклад, він іноді позначав уразливий рядок, а також позначав функцію, частиною якої є цей рядок, повторюючи вразливість без введення нових даних. Також двічі знаходив одну вразливу конструкцію, подаючи різні дані на вхід. Дані помилки показано на рис. 3.2 – 3.4. Такі виявлення у результатах рахувались як одне.

Рисунок 3.2 – Виявлення повторного входу інструментом mythril

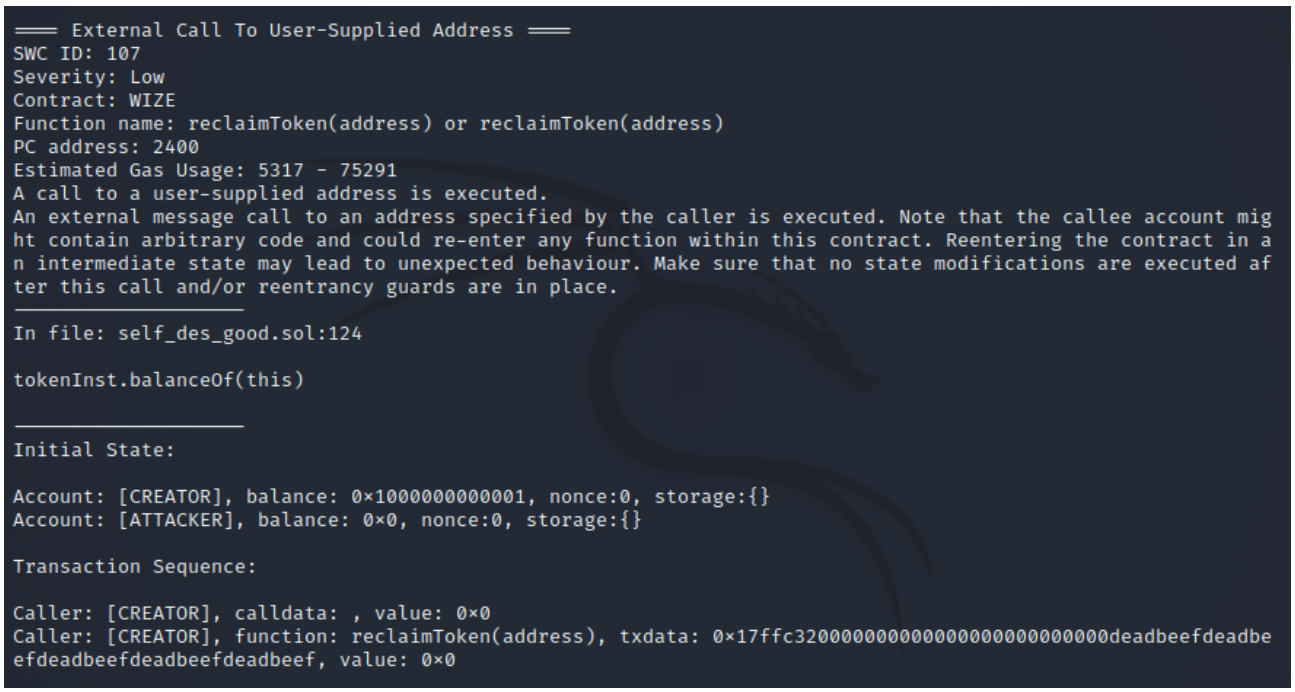

Рисунок 3.3 – Дублювання виявлення повторного входу з новими даними

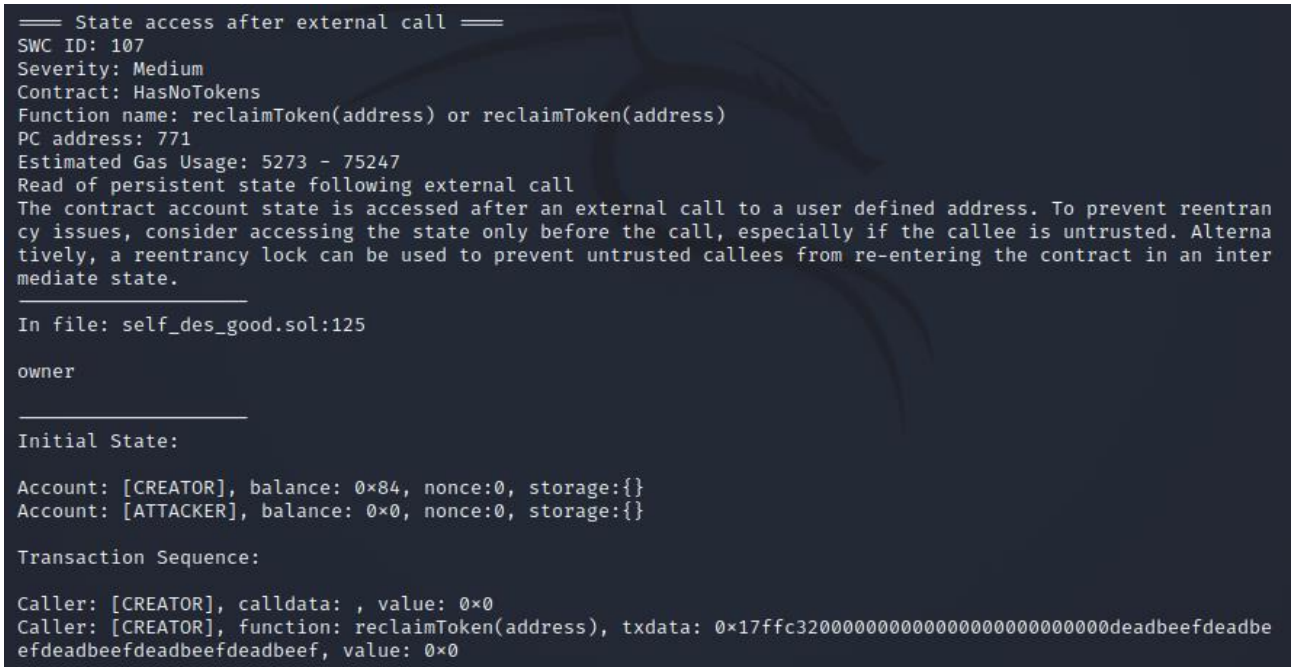

Рисунок 3.4 – Похідна від повторного входу вразливість: доступ до змінної стану

Є інструменти, які повідомляють про помилку декомпіляції, тоді як Mythril не створює жодного винятку, але може генерувати помилки або зациклюватися на етапі символьного виконання. При аналізі певних контрактів з Oyente була помічена велика кількість невідомих інструкцій. Однак схоже, що Oyente може давати результати аналізу навіть з такими повідомленнями про помилки (він ніколи не став аварійно). Тому перевагою Oyente є відсутність збоїв. Проте, Oyente може виявляти тільки 6 типів вразливостей, тому дає суттєву кількість FN.

Виявлені (істинно та хибно) вразливості, що не розглянуті у даному експерименті не було враховано.

3.3 Порівняння існуючих інструментів із модифікованим mythril

Оптимізовану стратегію виявлення для mythril було реалізовано програмно (далі – «mythril+») і проведено порівняння з іншими інструментами виявлення. Для цього було вибрано oyente, slither і, власне mythril. Попередньо було також розглянуто існуючі порівняння [23, 24]. Для дослідження було відібрано

контракти із 46 вразливостями, частину даних взято в датасеті [25]. Крім цього було додано виправлені контракти для ефективного пошуку FP. Експеримент проводився в 64-розрядній системі kali8 (ядро 5.10.0), що працює на віртуальній машині VMware Workstation, яка налаштована на 3 Гб пам’яті та чотири ядра процесора.

У таблиці 3.1 описано кількість виявлених вразливостей у кожній категорії, відповідно, кожним інструментом (формат – «виявлено/всього»). Таблиця 3.2 демонструє хибно позитивні виявлення.

Таблиця 3.1 – Виявлені вразливості за категоріями

	slither	oyente	mythril	mythril+
Арифметичні переповнення	5/7	4/7	5/7	6/7
Повторний вхід	4/11	6/11	10/11	10/11
Неперевірені значення функцій	1/5	0/5	4/5	5/5
Залежність від порядку транзакцій	0/4	2/4	1/4	2/4
Зловживання делегуючим викликом	4/5	0/5	3/5	4/5
Залежність від передбачуваних змінних	8/10	4/10	7/10	8/10
Незахищена selfdestruct()	3/4	0/4	3/4	3/4

Таблиця 3.2 – Хибно позитивні виявлення

	slither	oyente	mythril	mythril+
Всього	9	23	34	11
Арифметичні переповнення	-	6	6	3
Повторний вхід	6	10	16	2
Неперевірені значення функцій	-	-	3	-
Залежність від порядку транзакцій	-	-	-	1
Зловживання делегуючим викликом	-	-	-	-
Залежність від передбачуваних змінних	3	7	9	5
Незахищена selfdestruct()	-	-	-	-

Переповнення цілочисельних типів – у всіх інструментів були FN, пов’язані зі складною конвертацією типів, напр. операцією експоненти або перетворенням часових міток поєднані з урізанням типів uint256=>uint128.

Повторний вхід – всі інструменти погано помічають виправлення м’ютексом, що дає багато FP. У mythril+ це вдалося виправити. Крім цього slither погано виявляє зовнішній виклик на вказану користувачем адресу, а oyente не бачить міжфункціональний повторний вхід, що дало відповідні FN.

Неперевірені значення функцій – slither виявляє лише для send(). Mythril дав кілька FP у контрактах, які використовують значення, що повертається, send() як значення, що повертається функцією, і перевіряють повернуте значення вже там, де вона викликається. Наприклад, addr.send(), як показано коді нижче, є значенням, що повертається функції ForwardSend, і це значення перевіряється там де її викликають.

```
function ForwardSend(Address addr) returns (bool){
 return addr.send();}
```

FN пов’язані тут з тим, що дефект виникає у функції конструктора, тоді як байт-код функції конструктора не міститься в байт-коді часу виконання. Тому mythril це пропускає. Однак дефекти контракту у функції конструктора не зашкодять розгорнутим контрактам, оскільки функція конструктора буде виконуватися лише один раз під час розгортання контрактів у блокчейні.

TOD – slither може знаходити невизначений порядок операцій для конструкцій solidity, але не front-running. Оскільки це помилка, яка вже пов’язана із семантикою контракту, її пошук варто доповнити ручним, всі засоби були не точними. Спроба виправити це, зменшила FN на 1, але додала 1 FP.

Зловживання delegatecall() – якщо він стоїть у циклі, найкраще це виявляє slither. Загалом у інструментів, які використовують символьне виконання проблеми з циклами, адже вони можуть спричинити вибух станів. До речі, Oyente все ще повідомляє про вразливість атаки на глибину стека викликів, яка більше неможлива з хардфорком EIP 150.

Залежність від передбачуваних змінних – засоби символьного виконання, можуть не отримати правильний стан блоку під час роботи з пов’язаними кодами операцій, такими як «BLOCKHASH», «NUMBER» тощо. Це дає багато FP для такого типу вразливості. Однак це може не вплинути на процес виконання, оскільки всі результати цих кодів операції представлені у вигляді символу без обмежень і збережені в стеку.

selfdestruct() – єдиний FN, який пропустили всі засоби, був у складі комплексного багу, де некоректний модифікатор доступу, дозволяє виконати SSTORE, тобто змінити власника контракту внаслідок довільного запису.

3.4 Аналіз точності інструментів

Для отриманих результатів будуть суттєвими наступні оцінки:

- Коефіцієнт хибних негативів (FNR): FNR = FN / (FN + TP)
- Коефіцієнт хибних виявленнь (FDR): FDR = FP / (TP + FP)
- Чутливість = 1 - FNR
- Точність = 1 - FDR

Дані оцінки та час у роботі інструментів представлено в таблиці 3.3. Засоби символьного виконання потребують суттєво більше часу, затратними є як фаза обходу ГПК (збір шляхів), так і фаза розв’язування обмежень.

Таблиця 3.3 – Оцінки результатів аналізу

	slither	oyente	mythril	mythril+
Час у роботі (сек)	27	2433	3276	4129
TP	25	16	33	38
FP	9	23	34	11
FN	21	30	13	8
FDR	0.264706	0.589744	0.507463	0.22449
FNR	0.456522	0.652174	0.282609	0.173913
Точність	0.735294	0.410256	0.492537	0.77551
Чутливість	0.543478	0.347826	0.717391	0.826087

Результати оцінки показують, що mythril, який генерує високе значення TP, пропускає набагато більшу кількість хибнопозитивних результатів, що

призводить до дуже низької точності. Це було суттєво поліпшено в даній роботі. До того ж, його вже висока чутливість також зросла. Oyente дає багато FN, через можливість виявляти лише 6 типів вразливостей. Slither можна рекомендувати для виявлення залежності від передбачуваних змінних та зловживання делегуючим викликом. mythril+ показав суттєве покращення у виявленні повторного входу, арифметичних переповнень та необроблених результатів функцій.

Висновки до розділу 3

Було протестовано популярні інструменти статичного та гібридного пошуку вразливостей. Усі вони мають відкритий вихідний код, що дає користувачам можливість долучитись до масштабування. На прикладі покращення mythril це було показано. Помітно зменшена кількість помилок FP і FN для окремих типів вразливостей. Проте, це коштувало додаткового часу в роботі програми. Для вразливостей пов’язаних із семантикою контракту не вдалося досягти покращення, їх варто виявляти вручну. Також інструмент все ще не застрахований від вибуху станів, попри оновлений підхід, до обходу циклів. Це запобігалось використанням опцій, що обмежують «глибину» викликів та рекурсії.

ВИСНОВКИ

Смарт-контракти в Ethereum стають все більш застосовними як цифровий агент у розподілених додатках. Необхідно забезпечити безпеку смарт-контрактів, щоб уникнути непотрібних втрат і зловмисних атак.

У даній магістерській роботі було розглянуто кілька механізмів аналізу, реалізованих для перевірки та забезпечення коректності та невразливості шаблонів у смарт-контрактах. Розробники та користувачі повинні знати про точність і продуктивність цих методів аналізу. Методи статичного та динамічного аналізу виявляють лише свої конкретно визначені вразливі моделі, але зручні та ефективні для, відповідно, «своїх» типів вразливостей. Формальні методи перевірки використовують докази теорем для перевірки властивостей коректності в смарт-контрактах за допомогою їх інтерпретованих доказів, вони ефективні при перевірці відсутності вразливості.

Було досліджено механізм символьного виконання на прикладі mythril. Запропоновано оновлення для його модулів, зокрема механізм символьного виконання тепер робить евристичний пошук: відкидає недосяжні шляхи, а критичні знаходить за схемою пріоритетів; модуль детектор робить додатковий виклик розв’язувача для відкидання FP. Застосовано алгоритми для покращення роботи mythril для конкретних типів вразливостей. Для оцінки їх ефективності проведено порівняння з аналогічними інструментами. Досягнуто зниження кількості помилок, ціною більшого часу виконання. Не вдалося покращити виявлення вразливостей, які залежать від семантики контрактів.

Існує ряд відкритих питань. Це дослідження (як і багато інших) смарт-контрактів зосереджене на мові програмування, особливостях віртуального середовища, в той час як безпека самого блокчейну та застосунків для смарт-контрактів також створюють нові проблеми.

Список використаних джерел

1. J Chen, X Xia, D Lo, J Grundy - Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. ACM Transactions on Software Engineering and MethodologyVolume 31Issue 2 April 2022 Article No.: 30pp 1–37 https://dl.acm.org/doi/abs/10.1145/3488245
2. Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, ‘‘An overview on smart contracts: Challenges, advances and platforms,’’ Future Gener. Comput. Syst., vol. 105, pp. 475–491, Apr. 2020.
3. Atzei, N., Bartoletti, M., and Cimoli, T. (2017). “A Survey of Attacks on Ethereum Smart Contracts (Sok),” in International Conference on Principles of Security and Trus (Berlin Heidelberg: Springer), 164–186. vol. 10204 of POST 2017, Lecture Notes in Computer Science (LNCS). doi:10.1007/978-3-662-54455-6_8t
4. [Dataset] NCC Group (2018). Decentralized application security project (DASP) top 10. Available at: https://dasp.co
5. Durieux, T., Ferreira, J. F., Abreu, R., and Cruz, P. (2020). “Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts,” in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. (New York, NY, USA: Association for Computing Machinery), 530–541. ICSE 20. doi:10.1145/3377811.3380364
6. SWC Registry (2018). Smart Contract Weakness Classification and Test Cases. Available at: https://swcregistry.io/ .
7. John Ream, Yang Chu, and David Schatsky. Upgrading blockchains: Smart contract use cases in industry. Deloitte Press, 2016.
8. Florian Idelberger, Guido Governatori, Regis Riveret, and Giovanni ´ Sartor. Evaluation of logic-based smart contracts for blockchain systems. In International Symposium on Rules and Rule Markup Languages for the Semantic Web (RuleML), pages 167–183. Springer, 2016.

9. Riikka Koulu. Blockchains and online dispute resolution: smart contracts as an alternative to enforcement, 2016.
10. DApp Statistics. 2014. DApps Statistics. Retrieved from https://www.stateofthedapps.com/stats/
11. Vitalik Buterin. 2016. Critical Update Re: DAO Vulnerability. Retrieved from https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/.
12. Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2019. SoK: Transparent Dishonesty: front-running attacks on blockchain. https://arxiv.org/abs/1902.05164.
13. Josselin Feist, Gustavo Grieco and Alex Groce. "Slither: A Static Analysis Framework for Smart Contracts". In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). 2019, pp. 8–15. doi: 10.1109/WETSEB.2019.00008. url: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=8823898f .
14. Sergei Tikhomirov et al. "SmartCheck: Static Analysis of Ethereum Smart Contracts". https://s-tikhomirov.github.io/assets/papers/smartcheck.pdf
15. Manticore https://github.com/trailofbits/manticore
16. ConsenSys, "Mythril," Oct 2018, https://github.com/ConsenSys/mythril .
17. B. Mueller, "Smashing smart contracts," in 9th HITB Security Conference, 2018, https://tinyurl.com/y827tk72 .
18. Microsoft Corporation. The Z3 Theorem Prover. https://github.com/Z3Prover/z3
19. B. Mueller, "LASER-Ethereum," https://github.com/b-mueller/laser-ethereum
20. G. Wood, "Ethereum Yellow Paper, EIP-150 Revision," http://gavwood.com/paper.pdf
21. M. Dameron, "Ethereum Beige Paper," https://github.com/chronaeon/beigepaper
22. L. Jay, B. H. Omar, and G. Dan. (Feb. 11, 2018). *Crytic/EVM-Opcodes*. https://github.com/crytic/evm-opcodes

23. K. Byung Kim and J. Lee, “Automated generation of test cases for smart contract security analyzers,” IEEE Access, vol. 8, pp. 209 377–209 392, Nov. 24, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3039990. Available: https://ieeexplore.ieee.org/document/9268135 .
24. Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2019. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. arXiv preprint arXiv:1910.10601 (2019).
25. SmartBugs: A Framework to Analyze Solidity Smart Contracts https://github.com/smartbugs/smartbugs
26. K. Auguste, "THC CTF 2018 - Reverse of an ethereum smart contract on Github," https://github.com/ToulouseHackingConvention/reverse-palkeo-ethereum