МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

Лукаш Остап Зіновійович

Алгоритми підвищення безпеки смарт-контрактів на основі штучного інтелекту / Algorithms for Increasing the Security of Smart Contracts Based on Artificial Intelligence

спеціальність: 125 – Кібербезпека та захист інформації

освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконав студент групи
КБм -21

Лукаш Остап Зіновійович

Науковий керівник
к.т.н., доцент Т.Г. Цаволик

Кваліфікаційну роботу
допущено до захисту:

«____» _____ ___2025 р.

Завідувач кафедри
_____ **В.В. Яцків**

ТЕРНОПІЛЬ - 2025

Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки
Освітній ступінь «магістр»
спеціальність: 125 – Кібербезпека та захист інформації
освітньо-професійна програма – Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ В.В. Яцків
«____» _____ 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Лукаш Остап Зіновійович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Алгоритми підвищення безпеки смарт-контрактів на основі штучного інтелекту / Algorithms for Increasing the Security of Smart Contracts Based on Artificial Intelligence
керівник роботи: к.т.н., доцент Т.Г. Цаволик
затверджені наказом по університету від 20 грудня 2024 року № 938
2. Строк подання студентом закінченої кваліфікаційної роботи 5 грудня 2025 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
– проаналізувати технологічні основи функціонування смарт-контрактів та класифікувати їх критичні вразливості;
– дослідити існуючі методи аналізу безпеки (статичний, динамічний, формальна верифікація) та виявити їхні недоліки;
– розробити алгоритм підготовки даних та формування вектора ознак на основі абстрактного синтаксичного дерева (AST);
– обґрунтувати вибір та спроєктувати архітектуру нейронної мережі для виявлення вразливостей;
– спроєктувати архітектуру програмного комплексу та реалізувати модулі парсингу, екстракції ознак та інференсу;
– провести навчання моделі, оптимізацію гіперпараметрів та оцінку метрик якості;
– виконати експериментальне дослідження та порівняльний аналіз ефективності системи з існуючими інструментами (Slither, Oyente);

– розробити рекомендації щодо впровадження системи автоматизованого аудиту в процеси CI/CD.

5. Перелік графічного матеріалу у роботі:

– життєвий цикл смарт-контракту;
– типові методи представлення графа коду;
– загальна схема архітектури системи виявлення вразливостей;
– приклад запуску CLI з параметрами аналізу;
– структура каталогів програмного комплексу;
– HTML-звіт із сніпетом коду та кольоровим виділенням критичності;
– фрагмент JSON-звіту з структурованою інформацією про вразливості;
– схема інтеграції CLI у конвеєр CI/CD;
– криві тренувальної та валідаційної втрати за епохами;
– HTML-звіт для контракту з вразливостями overflow та access_control.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання: 20 грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Проблематики безпеки смарт-контрактів та сучасних підходів до її забезпечення	12.2024 р. – 03.2025 р.	
2	Методи та алгоритми штучного інтелекту для виявлення вразливостей	03.2025 р. – 06.2025 р.	
3	Програмна реалізація та експериментальне дослідження	06.2025 р. – 11.2025 р.	

Студент _____ Лукаш О.З.

Керівник роботи _____ к.т.н., доц. Т.Г. Цаволик

АНОТАЦІЯ

Кваліфікаційна робота на тему «Алгоритми підвищення безпеки смарт-контрактів на основі штучного інтелекту» на здобуття освітнього ступеня «Магістр» зі спеціальності 125 «Кібербезпека та захист інформації» освітньо-професійної програми «Кібербезпека» написана обсягом 99 сторінок і містить 26 ілюстрацій, 7 таблиць, 3 додатки та 47 джерел за переліком посилань.

Метою роботи є підвищення рівня захищеності смарт-контрактів шляхом розробки та впровадження алгоритмів автоматизованого аудиту на основі методів машинного навчання.

Методи досліджень. Для розв'язання поставлених задач у даній кваліфікаційній роботі використано: системний аналіз (для огляду предметної області та вразливостей), статичний аналіз коду (для отримання метрик та побудови AST), методи машинного навчання (для класифікації вразливостей), експериментальні методи (для оцінки ефективності розробленого комплексу).

Результати дослідження: проаналізовано критичні вразливості смарт-контрактів та обмеження існуючих інструментів аудиту; розроблено алгоритм підготовки даних з використанням абстрактного синтаксичного дерева; створено архітектуру нейронної мережі для виявлення вразливостей; реалізовано програмний комплекс з підтримкою CLI; досягнуто зменшення рівня хибно-позитивних спрацювань до 15%.

Результати роботи можуть бути застосовані для автоматизації процесів аудиту безпеки в блокчейн-проектах, інтеграції в конвеєри розробки та підвищення надійності децентралізованих додатків.

Ключові слова: СМАРТ-КОНТРАКТ, БЛОКЧЕЙН, МАШИННЕ НАВЧАННЯ, НЕЙРОННІ МЕРЕЖІ, ВРАЗЛИВОСТІ, АУДИТ БЕЗПЕКИ, AST, CI/CD.

ANNOTATION

The qualification thesis titled “Algorithms for Increasing the Security of Smart Contracts Based on Artificial Intelligence” submitted for obtaining the Master’s degree in specialty 125 “Cybersecurity and Information Protection” under the educational-professional program “Cybersecurity” consists of 99 pages and includes 26 figures, 7 tables, 3 appendices, and 47 referenced sources.

The purpose of the thesis is to enhance the security level of smart contracts by developing and implementing automated audit algorithms based on machine learning methods.

Research methods. To solve the set tasks, the following methods were used: system analysis (to review the subject area and vulnerabilities), static code analysis (to obtain metrics and build the AST), machine learning methods (to classify vulnerabilities), and experimental methods (to evaluate the effectiveness of the developed complex).

Research results: critical smart contract vulnerabilities and limitations of existing audit tools have been analyzed; a data preparation algorithm using the Abstract Syntax Tree (AST) has been developed; a neural network architecture for vulnerability detection has been created; a software complex supporting CLI and CI/CD integration has been implemented; a reduction of false positives to 15% has been achieved.

The results may be applied to automate security audit processes in blockchain projects, integrate into development pipelines (DevSecOps), and improve the reliability of decentralized applications.

Key words: SMART CONTRACT, BLOCKCHAIN, MACHINE LEARNING, NEURAL NETWORKS, VULNERABILITIES, SECURITY AUDIT, AST, CI/CD.

ЗМІСТ

ВСТУП

Актуальність роботи. Технологія блокчейн та смарт-контракти стали невід'ємною складовою сучасної цифрової економіки, забезпечуючи функціонування децентралізованих фінансів (DeFi), токенізацію активів та автоматизацію угод без посередників. Проте стрімкий розвиток цих технологій супроводжується зростанням кількості кіберзагроз. Оскільки смарт-контракти після розгортання в блокчейні стають незмінними, будь-яка помилка в коді може призвести до незворотних фінансових втрат.

Існуючі методи аналізу безпеки, такі як статичний аналіз (інструменти Slither, Securify) та символьне виконання (Mythril), хоча і є ефективними, мають суттєві обмеження. Статичні аналізатори часто генерують велику кількість хибно-позитивних спрацювань, що ускладнює роботу аудиторів, а методи динамічного аналізу вимагають значних обчислювальних ресурсів і не завжди можуть покрити всі шляхи виконання коду.

У зв'язку з цим виникає необхідність впровадження нових підходів, зокрема використання методів штучного інтелекту та машинного навчання, які здатні виявляти складні патерни вразливостей та зменшувати кількість помилкових спрацювань, розглядаючи код не просто як текст, а як структуровану послідовність логічних зв'язків.

Мета і завдання дослідження. Метою роботи є підвищення рівня захищеності смарт-контрактів шляхом розробки та впровадження алгоритмів автоматизованого аудиту на основі методів машинного навчання.

Досягнення визначеної мети передбачає вирішення таких завдань:

– провести аналіз технологічних основ функціонування смарт-контрактів у середовищі EVM та класифікувати критичні вразливості;

– дослідити існуючі методи аналізу безпеки та виявити їхні обмеження;

– розробити алгоритм підготовки даних, що включає дедуплікацію коду та формування вектора ознак на основі абстрактного синтаксичного дерева (AST) та метрик;

– обґрунтувати вибір архітектури нейронної мережі та розробити модель для класифікації вразливостей;

– реалізувати програмний комплекс для автоматизованого аудиту та провести експериментальне дослідження його ефективності;

– розробити рекомендації щодо впровадження системи у процеси розробки (CI/CD).

Об’єкт дослідження – процеси забезпечення безпеки та аудиту програмного коду смарт-контрактів у блокчейн-системах.

Предмет дослідження – методи, алгоритми та програмні засоби автоматизованого виявлення вразливостей у смарт-контрактах із застосуванням технологій машинного навчання.

Методи дослідження. Для розв’язання поставлених задач використано методи системного аналізу (для дослідження предметної області та існуючих інструментів), методи статичного аналізу коду (для отримання метрик та AST), методи машинного навчання (для побудови моделі класифікації вразливостей), а також експериментальні методи (для оцінки точності та швидкодії розробленої системи).

Наукова новизна одержаних результатів. Удосконалено підхід до виявлення вразливостей у смарт-контрактах шляхом поєднання методів глибокого навчання (багатошаровий перцептрон) з евристичними правилами та використанням спеціалізованого 512-компонентного вектора ознак, що дозволило зменшити рівень хибно-позитивних спрацювань порівняно з традиційними статичними аналізаторами.

Практичне значення отриманих результатів. Розроблено програмний комплекс мовою Python, який включає інтерфейс командного рядка (CLI) та підтримує інтеграцію в системи безперервної інтеграції.

Публікації та апробація КР.

1. Остап ЛУКАШ Застосування штучного інтелекту та машинного навчання для аудиту безпеки блокчейн-систем. Збірник матеріалів науково-практичного симпозіуму «Захист інформації'2025», Тернопіль, 2025. – С 97-98.

2. Остап ЛУКАШ Аналіз вразливостей та класичних методів забезпечення безпеки смарт-контрактів. Збірник матеріалів науково-практичного симпозіуму «Технології Інтернету речей: системи та рішення» (TIP:CT - 2025), Тернопіль, 2025. .– С 85-86.

1. ПРОБЛЕМАТИКИ БЕЗПЕКИ СМАРТ-КОНТРАКТІВ ТА СУЧАСНИХ ПІДХОДІВ ДО ЇЇ ЗАБЕЗПЕЧЕННЯ

1.1. Технологічні основи функціонування смарт-контрактів

Смарт-контракти - це цифрові угоди, які автоматично виконують транзакції при виконанні певних умов. Вони є важливими компонентами технології блокчейн, яка стала популярною з появою Blockchain 2.0. Смарт-контракти дозволяють виконувати різні процеси без посередників.

Ці контракти працюють на таких платформах, як Ethereum Virtual Machine (EVM), і є Тюрінговими, тобто можуть обробляти складні обчислення та логіку. Оскільки смарт-контракти працюють на блокчейні, вони не залежать від центрального органу влади, що робить їх децентралізованими.

Смарт-контракти різняться за складністю та призначенням - від базових транзакцій до управління цілими додатками. Загальні класифікації включають публічні (на відкритих блокчейнах) та приватні (з обмеженим доступом) контракти, а також стандарти на основі токенів, такі як ERC-20 для замінних токенів та ERC-721 для NFT. Вони також слугують для різноманітних застосувань, таких як обмін токенами, перекази та кредитування.

Децентралізовані фінанси (DeFi) - є провідним випадком використання, що дозволяє створювати платформи для кредитування, позичання та торгівлі без посередників. Токенізовані реальні активи, такі як нерухомість та стейблкоіни, також використовують смарт-контракти для підвищення прозорості та операційної ефективності. Вони також забезпечують роботу ринків прогнозів, таких як Polymarket, де користувачі можуть робити ставки на події з перевіреними результатами блокчейну, та платформ кредитування, що сприяють бездовірчим транзакціям із стимулами ліквідності.

Смарт-контракти оптимізують процеси шляхом автоматизації завдань, зменшення людських помилок та усунення необхідності в посередниках. Їхня ефективність значно прискорює час виконання, а зниження витрат досягається за рахунок мінімізації

адміністративних витрат та зниження транзакційних комісій, особливо в транскордонних платежах. Крім того, їхня незмінність гарантує, що після впровадження вони не можуть бути змінені, забезпечуючи надійну безпеку та прозорість на блокчейні.

Як зазначалося раніше, смарт-контракти - це структури, що виконують всі функції Тюрінга і працюють на таких інфраструктурах, як EVM. На рисунку 1.1 показано як вони дозволяють користувачам застосовувати програмний підхід до інфраструктури.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleWallet {
    mapping(address => uint256) private balances;

    event Deposit(address indexed account, uint256 amount);
    event Withdraw(address indexed account, uint256 amount);

    function deposit() external payable {
        require(msg.value > 0, "Must deposit > 0");
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    function withdraw(uint256 _amount) external {
        require(_amount > 0 && balances[msg.sender] >= _amount, "Invalid withdraw");
        balances[msg.sender] -= _amount;
        payable(msg.sender).transfer(_amount);
        emit Withdraw(msg.sender, _amount);
    }

    function checkBalance(address _account) external view returns (uint256) {
        return balances[_account];
    }

    function contractBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

Рисунок 1.1 — Створення смарт-контракту в блокчейні

Після кодування та розгортання в блокчейні смарт-контракт стає частиною стану блокчейну і отримує унікальну адресу, похідну від адреси розробника та нонса, що

гарантує детерміновану генерацію адрес контрактів. За допомогою цієї адреси вони стають доступними для інших контрактів та користувачів. Взаємодія з контрактом передбачає виклик його функцій або надсилання коштів на його адресу, якщо він налаштований на отримання платежів. Виклики контракту запускають виконання функцій, де базова віртуальна машина виконує байт-код контракту покроково, імітуючи віртуальну стекову машину.

Регулюючи попит за допомогою gas fees, блокчейни підтримують стабільне, децентралізоване середовище, в якому цілісність мережі зберігається завдяки збалансованому розподілу ресурсів та стимулам для валідаторів. На рисунку 1.2 показано приклад, смарт-контракт працює таким чином, що дозволяє користувачам взаємодіяти з ним через транзакції блокчейну. Користувачі надсилають дані та криптовалюту до смарт-контракту, який потім виконує заздалегідь визначений код. Валідатори, або однорангові вузли в мережі, перевіряють транзакцію та результат контракту, гарантуючи, що він відповідає правилам мережі та є узгодженим у всьому блокчейні. Після перевірки транзакція та результат контракту включаються до нового блоку, який додається до блокчейну.

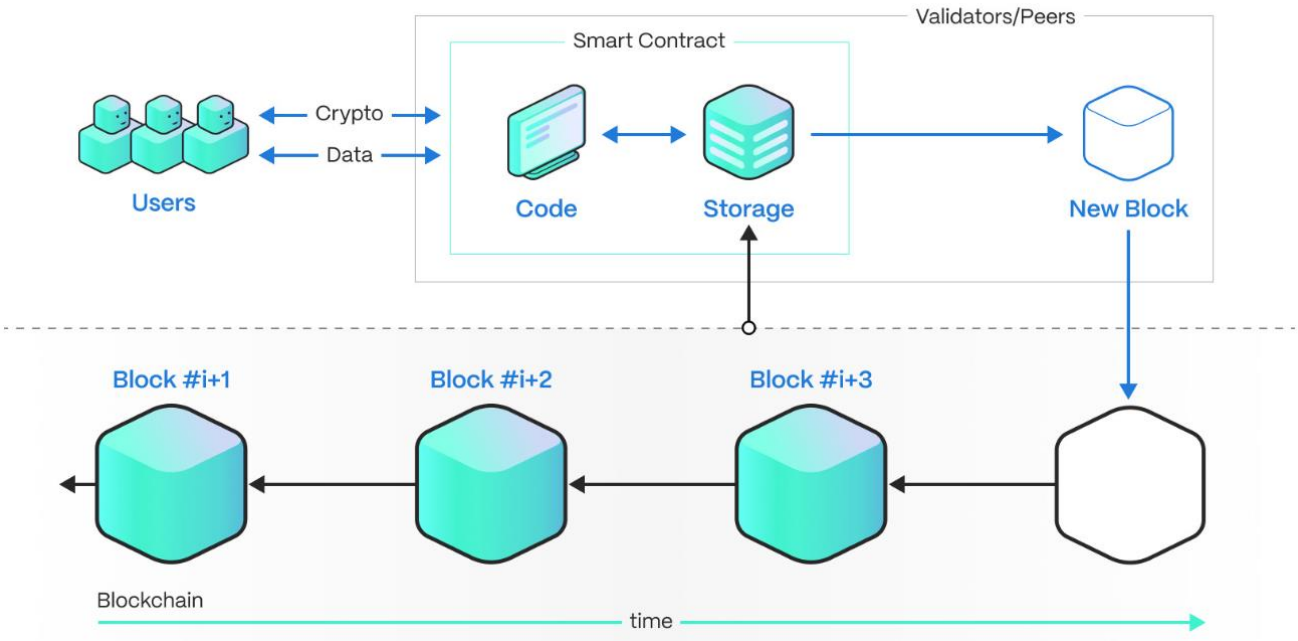


Рисунок 1.2 — Приклад роботи смарт-контракту

Цей процес створює безпечний, незмінний запис усіх взаємодій з часом, створюючи безперервний ланцюжок перевірених транзакцій та дій смарт-контракту.

Життєвий цикл смарт-контракту можна розділити на п'ять основних етапів, від написання коду до остаточного затвердження його стану в блокчейні (рисинук 1.3).

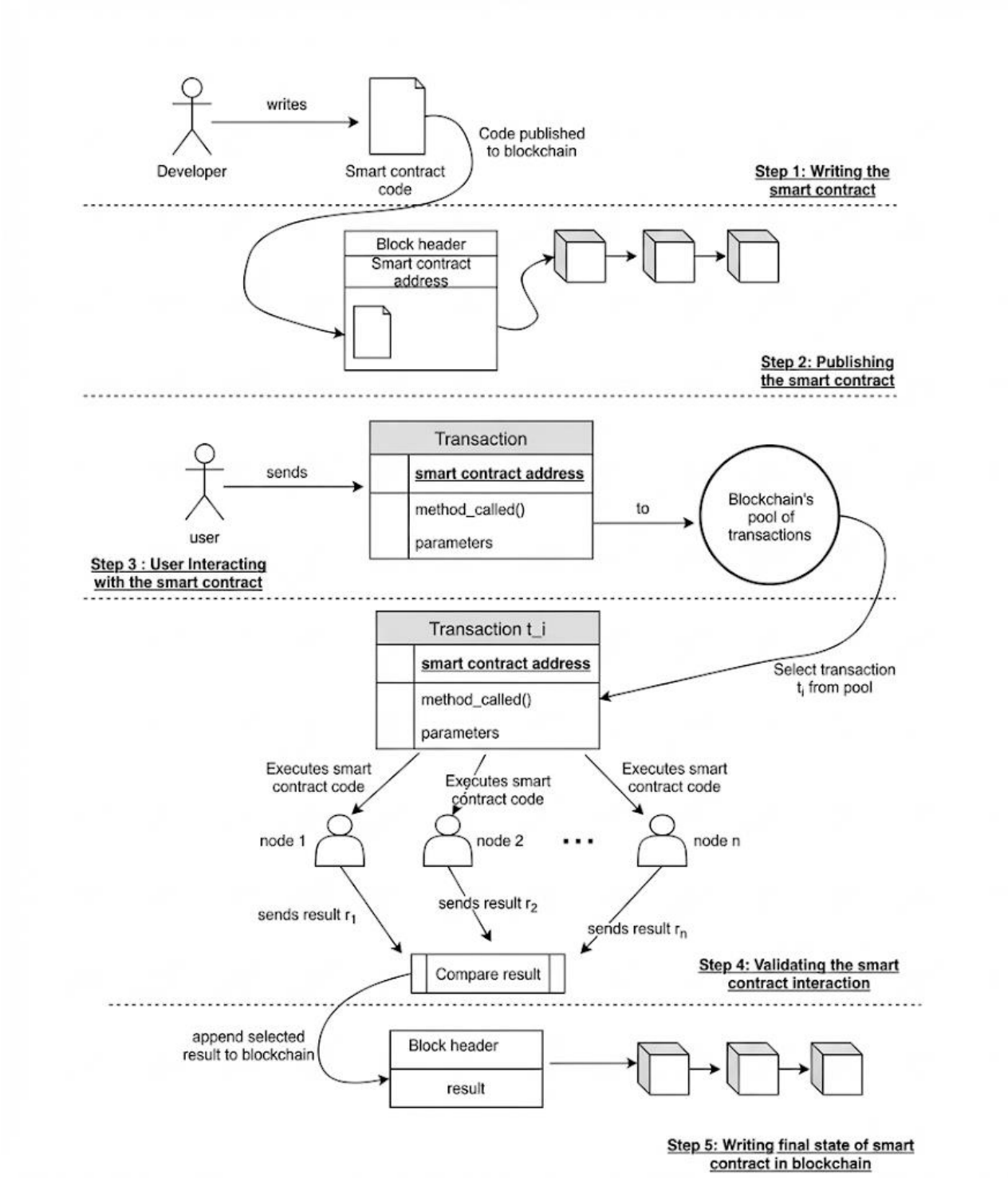


Рисунок 1.3 - Життєвий цикл смарт-контракту

П'ять основних етапів можна охарактеризувати так:

1) Створення - життєвий цикл починається з того, що розробник пише код смарт-контракту. Цей код визначає логіку, функції та структури даних контракту, які визначають, як контракт буде поводитися під час виконання. Після завершення роботи над кодом його компілюють у байт-код, який сумісний з віртуальною машиною цільового блокчейну (наприклад, віртуальною машиною Ethereum для мереж Ethereum).

2) Розгортання - після компіляції байт-код розгортається в блокчейні за допомогою транзакції. Ця транзакція розгортання присвоює смарт-контракту унікальну адресу, яка походить від адреси розгортача та нонса транзакції. Після розгортання код контракту стає незмінним і назавжди зберігається в стані блокчейну. Створюється заголовок блоку, який інкапсулює адресу контракту та пов'язаний з ним код, офіційно публікуючи контракт у мережі.

3) Взаємодія з користувачами - після розгортання користувачі можуть взаємодіяти зі смарт-контрактом, надсилаючи транзакції на його унікальну адресу. Кожна транзакція визначає функцію, яку потрібно викликати *method_called()*, та будь-які необхідні параметри. Ці транзакції потрапляють до пулу транзакцій блокчейну, де чекають на підтвердження мережевими вузлами. Корисне навантаження транзакції, яке включає адресу контракту, метод та параметри, є основою взаємодії та визначає конкретні дії, які буде виконувати контракт.

4) Перевірка - мережеві валідатори або майнери вибирають транзакції з пулу і виконують вказаний код контракту на своїх вузлах. Кожен вузол виконує ту саму логіку контракту в ізольованому середовищі, щоб забезпечити узгодженість результатів у всій мережі. Кілька вузлів виконують контракт незалежно один від одного, отримуючи результати (позначені як *f1, f2, ..., fn*). Потім ці результати порівнюються для перевірки консенсусу; якщо результати збігаються між вузлами, транзакція вважається дійсною. Цей механізм консенсусу гарантує, що виконання є детермінованим, підтримуючи надійність та цілісність мережі.

5) Остаточне оновлення стану - після перевірки остаточний стан, що є результатом транзакції, заноситься до блокчейну. Заголовок блоку оновлюється, щоб включити перевірені результати, що відображають новий стан смарт-контракту в незмінному реєстрі блокчейну. Це оновлення стану завершує життєвий цикл транзакції, і блокчейн тепер містить точний, захищений від підробки запис про стан контракту після взаємодії.

Найбільш критичною фазою життєвого циклу смарт-контракту є процес його створення. Дуже важливо переконатися, що він не містить помилок, які можуть призвести до втрати мільйонів або навіть більших сум коштів після його впровадження.

Безпека смарт-контрактів має першочергове значення через незворотний характер транзакцій у блокчейні. Після розгортання смарт-контракту його неможливо змінити, тому надзвичайно важливо переконатися, що код не містить вразливостей. Один-єдиний недолік може бути використаний, що призведе до несанкціонованого доступу, втрати коштів або навіть повного збою системи.

Вразливості смарт-контрактів зазвичай виникають через типові помилки в коді або логічні помилки. Такі проблеми, як неперевірені зовнішні виклики, неправильна валідація та арифметичні помилки, можуть призвести до зловживань.

В таблиці 1.1 показано типи вразливостей смарт-контрактів, кожен з яких становить значну загрозу для блокчейн-додатків. Розуміння цих вразливостей є необхідним для розробки надійних і безпечних смарт-контрактів.

Таблиця 1.1 Основні вразливості смарт-контрактів

Вразливість	Опис
Reentrancy	Використовує функцію зовнішнього виклику контракту, що дозволяє повторювати виклики до завершення початкової функції.
Integer Overflow/Underflow	Результати арифметичних операцій, які перевищують ємність пам'яті типу даних.
Improper Access Control	Дозволити неавторизованим користувачам отримувати доступ до даних або функцій контракту

	(наприклад, незахищене зняття коштів) або змінювати їх через недостатні обмеження доступу.
Front-Running	Використовує часовий проміжок між трансляцією транзакції та її включенням до блокчейну
Denial of Service (DoS)	Зробити контракт недоступним або нереагуючим, витративши весь доступний вміст або спричинивши постійні збої транзакцій.
Weak Randomness	Використання небезпечних методів, пов'язаних з блоками, для генерації випадкових чисел, якими можна маніпулювати.
Vulnerable External Calls	Ризики, пов'язані з здійсненням зовнішніх викликів без належної перевірки.
Logic Errors	Включає недоліки в логіці контракту, що призводять до несподіваної поведінки.
Oracle Manipulation	Спотворення цінних даних або інших позаланцюгових даних з метою викрадення активів.
Flashloan Attacks	Використання незабезпечених кредитів для маніпулювання ринками або експлуатації вразливостей контрактів.

Проекти Web3 повинні ставити безпеку в своїх блокчейн-додатках на перше місце. Дотримуючись рекомендацій та активно усуваючи вразливості, розробники можуть завоювати довіру користувачів, захистити активи та сприяти стабільності та зростанню екосистеми блокчейну. Пріоритет безпеки захищає окремі проекти та зміцнює всю децентралізовану фінансову сферу.

Вразливість смарт-контрактів призвела до значних фінансових втрат, що створює серйозні проблеми для організацій, які не мають надійних стратегій безпеки. Для малих і середніх підприємств незахищений смарт-контракт може призвести до непоправної шкоди, тому кібербезпека для малого бізнесу та кібербезпека для підприємств є нагальним пріоритетом. Було багато атак на смарт-

контракти, які коштували великих грошей. Однак найчастіше обговорюються атака на DAO та хакерські атаки на Parity Wallet.

У травні 2016 року кілька учасників спільноти Ethereum запустили DAO [26]. Спочатку він був відомий як genesis DAO. DAO був смарт-контрактом з відкритим кодом, який дозволяв будь-кому обмінювати токени DAO на ефір. Цей метод обміну допоміг зібрати близько 150 мільйонів доларів. надаючи DAO великий краудфандинг. Учасники з токенів DAO мали право голосувати за пропозиції і отримувати винагороду, якщо це приносило прибуток (рисунок1.4).

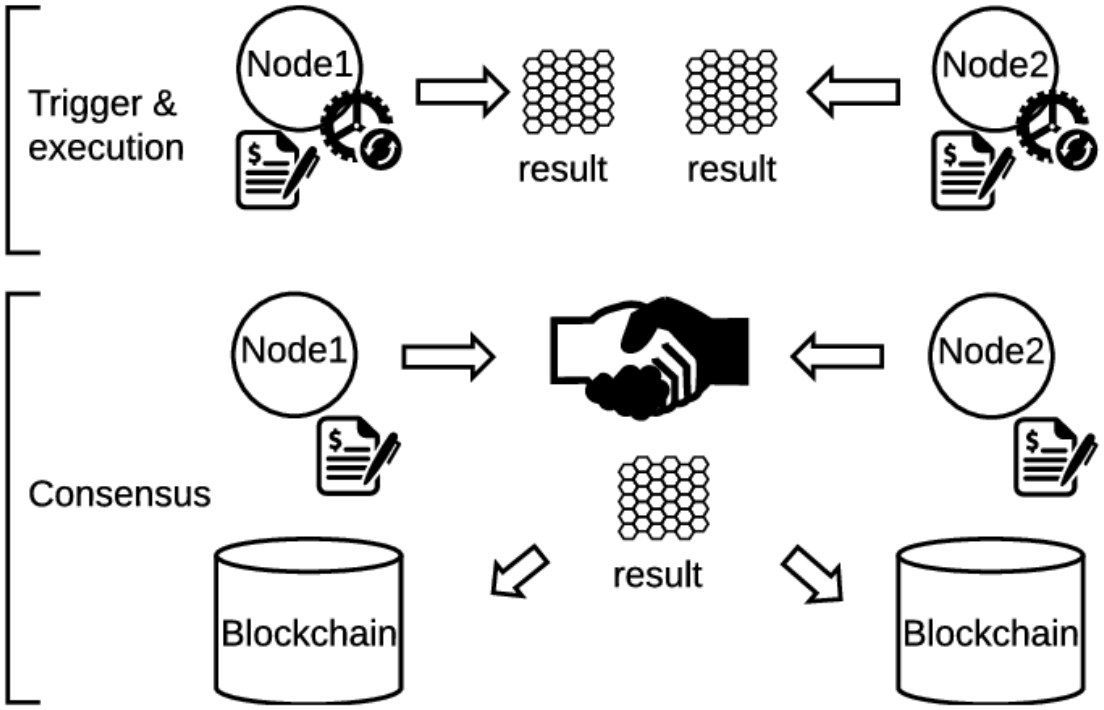


Рисунок 1.4 - Метод обміну у смарт-контракті

Однак контракт DAO містив серйозні недоліки, що дозволяли зловмисникам виводити кошти. Існувала лазівка, яка дозволяла зловмиснику неодноразово запитувати кошти зі смарт-контракту до оновлення балансу. Вразливість виникла через помилки в коді, де розробники не врахували можливість рекурсивного виклику. Таким чином, це дозволило зловмисникам викрасти ефір на мільйони

доларів протягом перших кількох годин. Сценарій атаки на DAO демонструє, наскільки руйнівною може бути проста вразливість смарт-контракту.

Аналогічно, злом Parity Wallet — це ще одна вразливість, яка була виявлена в Parity Multisig Wallet з версією 1.5+ [27], [28]. Ця вразливість дозволила зловмиснику викрасти понад 150 000 ETH (30 млн доларів США). Для здійснення атаки зловмисник передав дві транзакції з метою отримати право власності на Multisig, щоб викрасти всю валюту.

Після здійснення атаки був ініційований контракт Parity Multisig Wallet Library. Однак він містив помилку, яка дозволяла будь-кому запустити *initWallet* [29]. Атака була здійснена двічі, тому її називають Parity Wallet hack 1 і 2. Під час першої атаки зловмисник зміг змінити статус гаманця, ініціювавши виклик i*nitWallet*. В результаті зловмисник був визнаний власником і безперешкодно вивів кошти.

1.2. Огляд існуючих методів аналізу безпеки

Смарт-контракти, вважаються високоефективними з точки зору безпеки, проте також є багато відомих як і вразливих смарт-контрактів.

Для аудиту смарт-контрактів та аналізу результатів можна виділити три найпоширеніші та найактуальніші інструменти з відкритим кодом, а саме: Mythril, Securify, Slither.

Mythril - один з найпоширеніших інструментів для аудиту байт-коду віртуальних машин Ethereum. Смарт-контракти тестуються за допомогою методу concolic, який включає символьне виконання, обчислення SMT та аналіз забруднення. Інструмент Mythril можна використовувати для виявлення вразливостей, таких як числовий переповнення, перезапис власника, повторне виконання функції тощо. Нижче наведено приклад звіту Mythril (рисунок1.5).

```
==== Unprotected Selfdestruct ====
SWC ID: 106
Severity: High
Contract: KillBilly
Function name: commencekilling()
PC address: 354
Estimated Gas Usage: 574 - 999
The contract can be killed by anyone.
Anyone can kill this contract and withdraw its balance to an arbitrary
address.
--------------------
In file: killbilly.sol:22

selfdestruct(msg.sender)
```

Рисунок 1.5 - Функціонал Mythril

Окрім мережі Ethereum, інструмент здатний аналізувати смарт-контракти мереж, що підтримують EVM. До них належать: Hedera, Quorum, Vechain, Roostock, Tron та інші.

Securify - це інструмент для аудиту безпеки смарт-контрактів, розроблений Ethereum Foundation і ChainSecurity у 2017 році. З моменту запуску Securify було перевірено понад 22 000 смарт-контрактів Ethereum. Це допомогло їх розробникам виправити велику кількість вразливостей з різним рівнем ризику. Securify статично аналізує байт-код віртуальних машин Ethereum і отримує всю інформацію про алгоритм і потік даних у ньому. Цей процес повністю автоматизований за допомогою Souffle, мови програмування, створеної для статичного аналізу Oracle та іншого програмного забезпечення.

Потім результати перевіряються з метою виявлення вразливостей і надання рекомендацій щодо їх усунення (рисунок1.6).

Рисунок 1.6 - Функціонал Securify

Аналіз проводиться у три основні етапи:

1) Розбір байт-коду EVM та його декомпіляція у форму, придатну для аналізу;

2) визначення семантичних фактів про смарт-контракт;

3) об'єднання шаблонів та правил для виявлення вразливостей.

Slither — це інструмент для статичного аналізу коду смарт-контрактів. Окрім автоматичного виявлення вразливостей, Slither здатний знаходити можливості для оптимізації коду та візуалізувати алгоритм виконання, допомагаючи аудиторам краще та швидше зрозуміти структуру смарт-контракту. Інструмент може працювати з такими фреймворками, як Truffle, Hardhat, Embark та Dapp, без додаткової конфігурації.

Алгоритм роботи Slither виглядає наступним чином (рисунок1.7):

1) Представлення вихідного коду у вигляді абстрактного синтаксичного дерева за допомогою компілятора Solidity.

2) Побудова діаграми успадкування елементів, діаграми потоку управління та списку всіх виразів у смарт-контракті.

3) Ілюстрація вихідного коду внутрішньою мовою Slither, що спрощує аналіз.

4) Вся отримана інформація передається до модулів пошуку вразливостей.

5) Аудит та звітність.

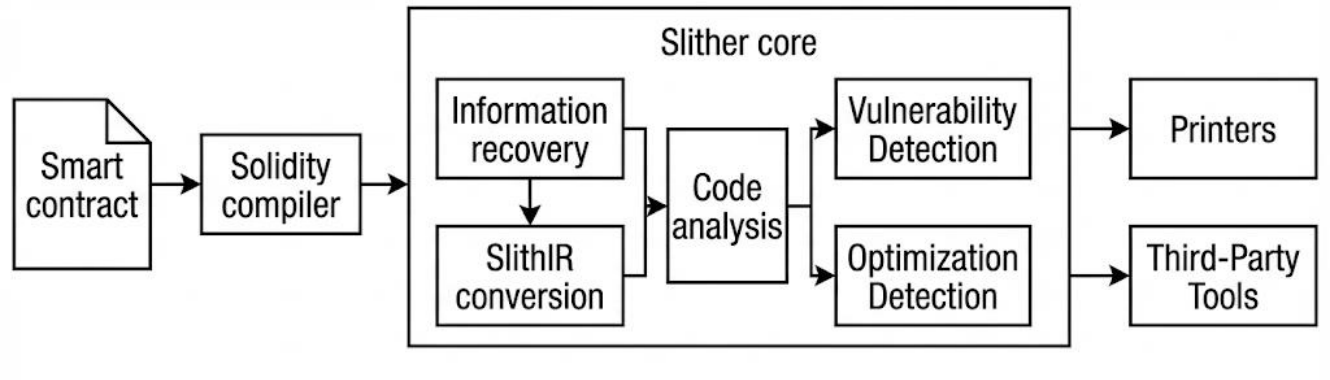


Рисунок 1.7 -Функціонал Slither

Таким чином, на даний момент не можна стверджувати, що будь-який із вищезазначених інструментів є значно кращим за інші для практичного аналізу безпеки смарт-контрактів. Для підвищення ефективності аналізу необхідно одночасно використовувати кілька універсальних та вузькоспеціалізованих інструментів.

Fuzzing - це економічно ефективний метод динамічного аналізу, що використовується для виявлення вразливостей у програмному та апаратному забезпеченні. Він може застосовуватися до будь-якого типу програмного забезпечення - від протоколів до окремого програмного забезпечення; від стандартного програмного забезпечення, такого як ERP, CRM і системи баз даних, включаючи індивідуальне програмне забезпечення для компаній, до веб-додатків, операційних систем, апаратного забезпечення та вбудованих систем, а також додатків для смартфонів.

Досі невідомі помилки та вразливості можна виявити за допомогою інструментального та економічно ефективного методу динамічного аналізу: фазінгу, який можна застосовувати без знання користувачем вихідного коду. Метод Fuzzing успішно використовується малими та середніми підприємствами і великими виробниками програмного забезпечення для своєчасного виявлення вразливостей і, таким чином, для зменшення витрат на всю процедуру виправлення. Навіть кінцеві споживачі використовують цю техніку для проведення тестів на затвердження поставок програмного забезпечення.

Fuzzing - це напівавтоматичний метод, що використовується для виявлення вразливостей в апаратному та програмному забезпеченні (які можуть бути використані для атак): за допомогою інструменту Fuzzing інтерфейс цільової програми отримує неправильно сформовані вхідні дані для виявлення несподіваних вхідних даних, які не були враховані в програмному коді. Неправильна або недостатня обробка цих даних призводить до несподіваної поведінки (збій, високе використання таких ресурсів, як час обчислення ЦП, ємність пам'яті) цільової програми. Аномальна поведінка програми реєструється, попередньо аналізується і відображається на моніторі. Помилкові спрацьовування можна виключити, проаналізувавши результати моніторингу, тоді як вразливості можна виявити, навмисно повторно викликавши аномалію і розробивши експлойт (рисунок1.8).

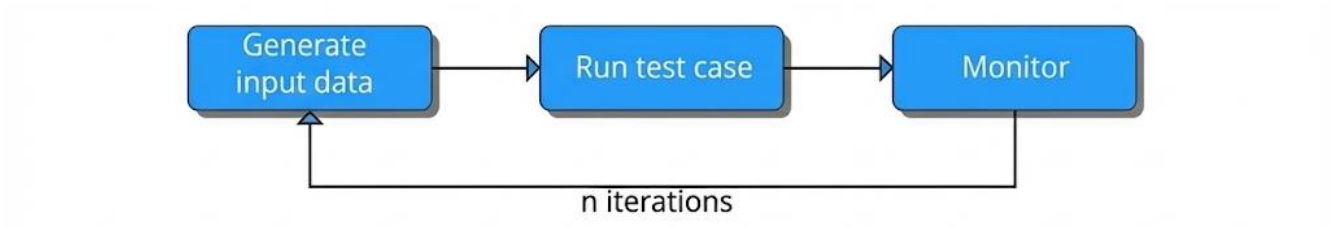


Рисунок 1.8 — Процес Fuzzing

Fuzzing вже успішно використовується великими та малими виробниками програмного забезпечення для стійкого зниження витрат на виправлення помилок та встановлення патчів. Цей метод також застосовується користувачами програмного забезпечення для проведення тестів на затвердження поставок програмного забезпечення.

Формальна верифікація - це сувора техніка, яка використовує математичні методи для доведення правильності проектування або реалізації системи. На відміну від традиційних методологій тестування, які покладаються на виконання тестових випадків для виявлення помилок, формальна верифікація використовує

математичні докази для гарантування відсутності певних класів помилок. Цей підхід має кілька переваг:

1) Математичні докази правильност - формальна верифікація надає математичні гарантії того, що система працює як передбачено, усуваючи можливість певних типів помилок.

2) Всебічне дослідження шляхів виконання - формальна верифікація досліджує всі можливі шляхи виконання, забезпечуючи повне покриття та виявляючи потенційні проблеми, які можуть бути пропущені під час тестування.

3) Проактивне виявлення помилок - формальна верифікація дозволяє виявити критичні помилки та вразливості на ранній стадії життєвого циклу розробки, зменшуючи витрати та зусилля, необхідні для їх усунення.

Формальна верифікація відіграє вирішальну роль у забезпеченні безпеки пам'яті та запобіганні помилок виконання, особливо в таких мовах, як C, C++ і навіть Rust, які розроблені для вирішення проблем безпеки пам'яті. Формальна верифікація дозволяє ефективно виявляти та усувати вразливості пам'яті, включаючи переповнення буфера, помилки використання після звільнення та витоки пам'яті.

Безпека пам'яті має першочергове значення в системах, критичних для безпеки, та додатках, чутливих до безпеки. Такі мови, як Rust, розглядають небезпечну для пам'яті поведінку як помилки компілятора, запобігаючи помилкам виконання, таким як використання після звільнення, за замовчуванням. Однак небезпечні функції у Rust, змішані проекти Rust/C/C++ та складні взаємодії систем створюють ризики, які традиційне тестування не може повністю виявити.

Безпека пам'яті захищає від програмних помилок та вразливостей безпеки, пов'язаних з доступом до пам'яті, таких як переповнення буфера та висячі покажчики. Важливо дослідити найважливіші проблеми безпеки пам'яті та помилки виконання і знайти практичні рішення для зменшення потенційних проблем, таких як ризики Unsafe, Embedded та FFI.

TrustInSoft Analyzer забезпечує математично доведену безпеку пам'яті, гарантуючи відсутність помилок виконання та забезпечуючи відповідність суворим галузевим стандартам. Для Rust та гібридного коду TrustInSoft пропонує послуги з аналізу коду Rust. TrustInSoft математично гарантує відсутність вразливостей пам'яті, роблячи програмне забезпечення безпечнішим, сумісним та більш стійким перед розгортанням.

Незважаючи на прогрес у сфері формальної верифікації, залишається низка викликів, таких як масштабованість для ефективної роботи з дедалі більшими та складнішими системами, а також нестача кваліфікованих інженерів з формальної верифікації.

1.3. Застосування штучного інтелекту в кібербезпеці блокчейну

Дві технології, які широко асоціюються із сучасним розвитком ШІ, - це ML і DL. Ці галузі займаються розробкою систем, які можуть знаходити закономірності у зразках даних, приймати рішення і навіть прогнозувати результати без прямого втручання людини. ML є основою всього процесу, а для класифікації, регресії, кластеризації тощо використовуються різні алгоритми. Знову ж таки, переходячи до підкатегорій ML, DL розбудовує ці можливості, використовуючи штучні нейронні мережі (ANN) для обробки великих, дуже значущих даних. У поєднанні ML і DL трансформували галузі, вирішуючи колись нерозв'язні завдання, що стояли перед ними.

ML і DL стали вирішальними у пошуку рішень складних проблем у різних сферах. Ці моделі вирізняються тим, що створюють узагальнені представлення на основі необроблених даних і можуть застосовуватися в таких сферах, як охорона здоров'я, кібербезпека та розпізнавання зображень, як показано на малюнку. Однак все ще існує значна проблема, пов'язана з характеристикою DL як «чорного ящика», коли практикуючі фахівці іноді не мають уявлення про те, як ці моделі приймають рішення.

На відміну від цього, дослідження ML зосереджуються на основних алгоритмах навчання, які класифікуються як навчання з наглядом, без нагляду, з частковим наглядом та підкріпленням. Згідно з вищезазначеними конкретними даними та дослідженнями, ці алгоритми вирішують різні реальні проблеми, включаючи врожайність у сільському господарстві [22], шахрайство в кібербезпеці [23] та управління ресурсами в розумних містах [5]. Незважаючи на це, важливо зазначити, що алгоритми ML надають користувачеві велику гнучкість протягом аналізу та прогнозування. Найголовніше, що вдосконалення моделей значною мірою залежить від якості та кількості даних, доступних для навчання (рисунок1.9).

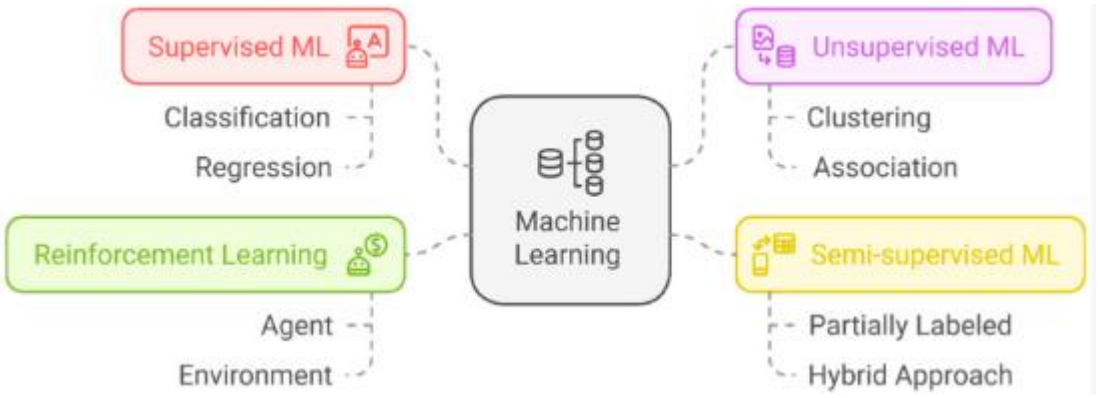


Рисунок 1.9 -Алгоритми машинного навчання.

Дослідження в області DL спрямовані на усунення необхідності використання великих маркованих наборів даних шляхом автоматизації етапів підготовки даних, включаючи анотацію [25]. Ефективні моделі також розробляються на обмежених обчислювальних платформах, таких як датчики IoT і смартфони, і добре працюють в середовищах з обмеженим об'ємом пам'яті [26]. Аналогічно, ML націлений на підвищення стабільності алгоритмів шляхом включення знань про домен, особливо в чутливих секторах, таких як сектор охорони здоров'я, де висока точність є надзвичайно важливою [27].

На практиці обидві технології показали свою перспективність у сфері охорони здоров'я та фінансів. Наприклад, моделі DL використовуються для діагностики і навіть

скринінгу таких захворювань, як рак, на основі зображень органів пацієнта або органів інших пацієнтів [28]. З іншого боку, алгоритми ML покращують фінансові портфелі та виконують перевірку транзакцій на предмет шахрайства в режимі реального часу [29]. Ці застосування демонструють, що синергетичне поєднання ML і DL дозволяє кожній методології успішно вирішувати специфічні для даної галузі проблеми.

Однак досі залишаються незрозумілі проблеми подальшого вирішення залежності від анотованих наборів даних та вдосконалення підходів до інтерпретації та пояснення моделей ШІ, що дозволить встановити довіру до систем. Наприклад, прозорість у прийнятті рішень є особливо важливою в таких чутливих сферах, як охорона здоров'я, де ставки високі [30].

Виявлення схожості міжплатформних бінарних кодів має на меті визначити, чи є два або більше бінарних коди схожими між собою. Найефективнішими є існуючі підходи, що поєднують представлення функцій на основі графіків контролю потоку (CFG) та аналіз схожості на основі графових конволюційних мереж (GCN). Через великий обсяг конволюційних обчислень та втрату структурної інформації використання конволюційних мереж неминуче призводить до таких проблем, як високі накладні витрати та, інколи, неточність. Для вирішення цих проблем часто застосовують швидку міжплатформну систему виявлення схожості бінарного коду, яка використовує обробку природної мови (NLP) та індуктивну графову нейронну мережу (GNN) для вбудовування базових блоків та представлення функцій відповідно шляхом імітації вилучення структурних та часових ознак. Орієнтована на вузли та невеликі партії GNN є підходящим способом навчання для великих CFG, що дозволяє значно зменшити обчислювальні накладні витрати. Оцінюються різні моделі вбудовування базових блоків NLP та GNN. Експериментальні результати показують, що схема з довгостроковою короткочасною пам'яттю (LSTM) для вбудовування базових блоків та індуктивним навчанням на основі GraphSAGE (GAE) для представлення функцій перевершує

найсучасніші розробки. У таблиці 1.2 представленно основні критерії порівняння **NLP і CFG:**

Таблиця 1.2 – Порівняння підходів NLP і CFG.

Критерій порівняння	**Підхід NLP**	**Підхід CFG**
Представлення даних	Код - це послідовність символів або токенів Використовуються n-грами, ембеддинги (CodeBERT).	Код - це структура. Вузли - базові блоки коду, ребра - переходи керування.
Стійкість до обфускації	Низька. Зміна імен змінних чи додавання сміттєвого коду сильно впливає на результат.	Висока. Структура графа часто залишається незмінною, навіть якщо змінні перейменовані.
Обчислювальна складність	Низька/Середня. Алгоритми зазвичай лінійні швидкі для великих обсягів коду.	Висока. Порівняння графів - це NP-складна задача. Вимагає значних ресурсів.
Втрата інформації	Втрачається інформація про те, як дані перетікають між функціями.	Втрачається контекст "людського" написання (назви змінних, коментарі), якщо вони не додані як атрибути вузлів.
Типові застосування	Виявлення клонів коду Класифікація шкідливого ПЗ (Malware Classification) Автодоповнення коду	Пошук вразливостей логіки Символьне виконання Деобфускація

Зараз у наукових роботах часто використовують Graph Neural Networks (GNN). Вони беруть структуру з CFG, але кожному вузлу (блоку коду) присвоюють векторні ознаки, отримані через NLP. Це дозволяє отримати точність графів і семантичне розуміння NLP.

Проведений аналіз показав, що проблема безпеки смарт-контрактів залишається критичною через незмінність блокчейну та дороговартісність помилок. Традиційні методи аналізу, розглянуті в пункті 1.2, мають суттєві обмеження такі як:

1) Статичні аналізатори базуються на фіксованих правилах та патернах, що призводить до високого рівня хибно-позитивних спрацювань та нездатності виявляти невідомі типи атак.

2) Динамічний аналіз та фаззінг вимагають значних обчислювальних ресурсів і інколи не можуть покрити всі можливі шляхи виконання коду, а сам е проблема покриття коду.

3) Формальна верифікація забезпечує найвищий рівень надійності, але є також складною у впровадженні та погано масштабується для великих проєктів.

Водночас, аналіз підходів штучного інтелекту продемонстрував перспективність використання машинного навчання, а саме - порівняння NLP та CFG-підходів дозволяє зробити висновок, що аналіз смарт-контрактів вимагає гібридного погляду, що код слід розглядати не просто як текст, а як структуровану послідовність інструкцій із чіткими логічними зв'язками.

2. АЛГОРИТМИ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ

2.1. Підготовка даних та формування навчальної вибірки

Безпека та надійність смарт-контрактів має вирішальне значення, оскільки їх розгортання в незмінному реєстрі робить вразливості та помилки незворотними, що створює ризик значних фінансових втрат. Тому надзвичайно важливо проводити ретельне тестування, аналіз та виправлення помилкового коду смарт-контрактів. Інструменти та методи, що використовуються для цієї мети, застосовують як статичний, так і динамічний аналіз. Однак ефективність цих інструментів затьмарюється кількістю нещодавніх успішних атак, здійснених на реальні проекти, розгорнуті в ланцюжку. Останні дослідження показують, що використання синтетичних, невеликих і нерізноманітних наборів даних смарт-контрактів не дозволяє відобразити складність реальних проектів. Ці невеликі набори даних перешкоджають розробці та оцінці надійних інструментів, що є основною причиною їх неефективності.

Набір даних реальних смарт-контрактів допоможе дослідникам і розробникам у виконанні різних завдань, включаючи створення інструментів для розробки смарт-контрактів. Більше того, у міру зростання популярності інструментів на основі штучного інтелекту для аналізу, ремонту та синтезу смарт-контрактів, попит на великі набори даних для завдань машинного навчання щодо контрактів стає все більш критичним.

Однак можна помітити, що поточні набори даних смарт-контрактів Solidity є обмеженими: вони або застарілі (старі версії Solidity), або не відображають реальні розгорнуті контракти.

Щоб вирішити цю проблему, існує, великий набір даних реальних смарт-контрактів.

Цей набір даних є корисним:

1) Для оцінки інструментів аналізу смарт-контрактів;

2) Для навчання машинному навчанню та інструментам на основі LLM для смарт-контрактів.

Перевірений смарт-контракт на Etherscan - це контракт, вихідний код якого було завантажено на Etherscan і успішно зіставлено з його скомпільованим кодом на блокчейні, щоб забезпечити прозорість та публічну доступність вихідного коду.

DISL включає лише перевірений вихідний код смарт-контрактів, щоб гарантувати, що він містить виключно реальні контракти, які використовуються. DISL містить вихідний код для всіх смарт-контрактів на Etherscan від генезисного блоку до 15 січня 2024 року. DISL обробляється з використанням етапу дедуплікації, щоб видалити смарт-контракти Solidity, які з'являються тисячі разів на Ethereum.

Слід розглянути також таку термінологію:

1) Розгорнутий контракт - адреса смарт-контракту в основній мережі блокчейну Ethereum, яка пов'язана з двійковим кодом, що зберігається в блоці Ethereum;

2) Сирий контракт - з'єднана версія всіх вихідних кодів (файлів Solidity), отриманих з Etherscan для розгорнутого контракту. Він може містити кілька бібліотек, що використовуються в межах даної адреси контракту;

3) Файл Solidity - файли вихідного коду, що використовуються для написання коду смарт-контракту Solidity. Сирий контракт може містити кілька файлів Solidity, які самі по собі можуть містити більше одного ключового слова «contract».

Далі для прикладу можна використати базу даних Google BigQuery для Ethereum, яка надає доступ до даних блокчейну для аналізу. Ця база даних, що оновлюється щодня, дозволяє досліджувати транзакції смарт-контрактів.

```
SELECT contracts.address, COUNT(1) AS tx_count
FROM `bigquery-public-data.crypto_ethereum.contracts` AS contracts
JOIN `bigquery-public-data.crypto_ethereum.transactions` AS transactions
ON (transactions.to_address = contracts.address)
WHERE transactions.block_timestamp >= TIMESTAMP "2022-04-01"
```

GROUP BY contracts.address

ORDER BY tx_count DESC

За допомогою Google BigQuery зібрали CSV-файл із 2 709 030 записами, де кожен рядок містить адресу розгорнутого контракту та кількість транзакцій.

Використовуючи публічні API Etherscan, можна отримали вихідний код для всіх перелічених адрес із даних, зібраних у Google BigQuery, успішно отримавши дані для

2 660 658 контрактів, збережених у форматі JSON . Далі видаляємо рядки, що складаються з порожніх файлів JSON, що вказує на те, що контракт не має перевіреної адреси на Etherscan. В результаті ми отримуємо так званий необроблений набір даних, що складається з 1,080,579 рядків.

Об'єднавши файли parquet з набором даних Andstor, щоб отримати 3,298,271 рядків, кожен з яких відповідає розгорнутому контракту. Структура розгорнутих контрактів зазвичай є поєднанням залежностей та визначень контрактів, специфічних для проекту, які успадковують або використовують ці залежності. Залежності, що використовуються в розгорнутих контрактах, походять із популярних бібліотек смарт-контрактів, включаючи OpenZeppelin, Safe та Provable. Це означає, що вихідні коди в необробленому наборі даних DISL можуть містити багато разів один і той самий код бібліотеки Solidity. Тому виконуємо дедуплікацію (тільки для розкладеної колекції), щоб гарантувати, що всі записи набору даних мають унікальне значення.

Використання такого підходу до подібності призвело б до відкидання багатьох контрактів, які використовують однакові бібліотеки, але все ж відрізняються за функціональністю. Таким чином, згідно з Storhaug et al., спочатку розкладаються контракти на окремі файли Solidity. Загальна кількість файлів Solidity після розкладання становить 12,931,943 файлів. Фільтруємо цю колекцію за допомогою індексу схожості Jaccard, щоб відрізнити дублікати контрактів із пороговим значенням 90% (порогове значення, яке використовується в). Після фільтрування отримали 514,506 файлів Solidity, об'єднавши

розкладений набір даних. Це означає, що більше ніж 96% у DISL raw є дублікатами коду відповідно до використовуваної схеми схожості.

У додатку Б наведено деталі метаданих смарт-контракту в кожній з двох необроблених і розкладених колекцій набору даних.

Завдяки своїм розмірам і характеру (тільки перевірені смарт-контракти), набір даних DISL пропонує значні переваги в двох основних областях: розробка інструментів на основі штучного інтелекту та бенчмаркінг інструментів програмного забезпечення для смарт-контрактів.

Специфічне навчання великих мовних моделей (LLM) — це процес адаптації параметрів LLM для поліпшення продуктивності при виконанні конкретних завдань (наприклад, синтез коду). Точне налаштування є перевіреною технікою для підвищення продуктивності LLM при виконанні завдань, пов'язаних з кодом. DISL є цінним кандидатом для таких операцій точного налаштування, оскільки містить великий корпус смарт-контрактів з дедуплікованими файлами.

Бенчмаркінг передбачає оцінку продуктивності інструментів програмної інженерії (традиційних та на основі штучного інтелекту) за допомогою стандартного набору даних для вимірювання їхньої продуктивності. Оскільки DISL містить унікальні реальні контракти, він є новим цінним набором для бенчмаркінгу порівняно з існуючими.

Емпіричні дослідження вимагають реальних сценаріїв. Однак академічним інструментам аналізу смарт-контрактів бракує прикладів з реального життя. Набір даних DISL є цінним ресурсом для вивчення контрактів.

Комп'ютери обробляють інструкції, використовуючи лише двійковий код - послідовності 0 і 1. Комп'ютери працюють на основі електронних компонентів, які функціонують лише у двох станах: увімкнено (1) або вимкнено (0). Людські програмісти не пишуть цей необроблений двійковий машинний код. Натомість ми пишемо комп'ютерну програму на мові програмування високого рівня — логічному наборі інструкцій, розробленому для зручності читання людиною, який визначає

поведінку комп'ютера. Ці інструкції повинні якимось чином перетворюватися з тексту, зрозумілого людині, на двійковий код, який може виконувати машина.

Це перетворення передбачає кілька представлень програми:

1) Вихідний код: програма, написана мовою програмування. Вона призначена для розуміння людиною (наприклад, Python, C++, JavaScript).

2) Машинний код: остаточна двійкова версія (0 і 1). Це єдина форма, яку центральний процесор (CPU) комп'ютера може виконувати безпосередньо.

Процес написання коду передбачає розбиття ідей високого рівня на структуровані елементи. Щоб зрозуміти складові елементи цього вихідного коду - від токенів до функцій. Щоб написаний людиною вихідний код успішно перетворився на машинний код, він повинен відповідати одній з двох основних стратегій перетворення - компіляції (прямій) або інтерпретації (непрямій).

Спеціальний інструмент повинен заповнити прогалину між вихідним кодом і машинним кодом. Це перетворення або виконання відбувається в основному за допомогою двох різних концептуальних шляхів: компіляції або інтерпретації (рисунок 2.1).

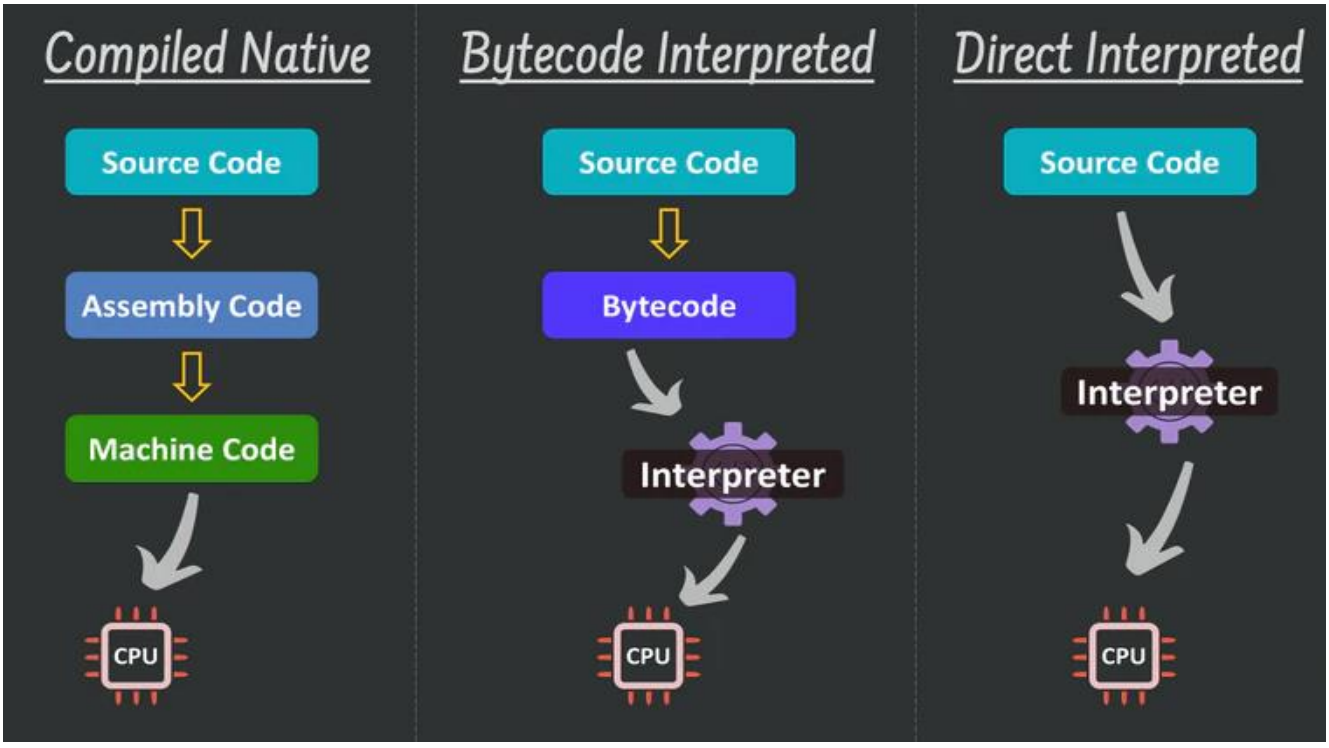


Рисунок 2.1 - Як створюються програми

Існують два шляхи виконання програми:

1) Компільований шлях - переклад на рідний машинний код;

2) Шлях інтерпретації: виконання за допомогою проміжного коду.

Компільований шлях використовує компілятор для перекладу всього вихідного коду на рідний машинний код перед виконанням.

Компілятор обробляє код один раз і генерує самостійний виконуваний файл (бінарний файл, адаптований для конкретного процесора/ОС, наприклад .exe або бінарний файл ELF). Внутрішньо компілятор спочатку перекладає вихідний код в асемблерний код — низькорівневу символьну мову, яка безпосередньо відповідає машинному коду — перед тим, як згенерувати остаточний бінарний файл.

Комп'ютер виконує цей бінарний файл безпосередньо і ефективно, не потребуючи вихідного коду або окремого інструменту (наприклад, інтерпретатора). Цей процес схожий на публікацію повної книги: весь текст перекладається і друкується, перш ніж хтось його прочитає. Приклади: C, C++, Rust, Go.

Шлях інтерпретації передбачає використання інструменту, який зчитує та виконує код за запитом, інструкція за інструкцією, під час виконання.

Цей шлях має дві поширені форми:

1) Пряма інтерпретація, коли інтерпретатор безпосередньо зчитує та виконує оригінальний вихідний код рядок за рядком під час виконання. Це повільно для складних обчислень, але гнучко. Приклади: Bash, AWK.

2) Інтерпретація байт-коду (сучасний стандарт), більшість сучасних «інтерпретованих» мов використовують двоступеневий процес для балансу швидкості та портативності: Компіляція в байт-код: вихідний код спочатку перекладається в проміжне представлення (IR), відоме як байт-код. Цей байт-код є простішим набором інструкцій, незалежним від архітектури (подібним до «псевдо-машинного коду»).

Виконання віртуальною машиною (VM) - це байт-код виконується спеціальною віртуальною машиною (VM) або інтерпретатором (наприклад, віртуальною машиною Python або віртуальною машиною Java). VM зчитує інструкції байт-коду і негайно

перекладає їх у дії, які виконує комп'ютер. Цей процес обробляє початковий етап компіляції двома поширеними способами:

Збережений байт-код (Python, JavaScript) проходить компіляцію з вихідного коду в байт-код часто відбувається настільки швидко або виконується один раз, а байт-код зберігається у файлі (наприклад, файли .pyc Python), що виконання здається миттєвим.

Попередньо скомпільований байт-код (Java, C#) - це вихідний код компілюється в байт-код (наприклад, файли .class Java) як окремий етап побудови перед розгортанням програми. Потім VM завантажує і виконує цей попередньо скомпільований байт-код.

Інтерпретоване виконання забезпечує гнучкість і спрощує налагодження, оскільки код можна миттєво змінювати і перезапускати без повного етапу побудови нативним кодом. Більше того, один і той самий код може виконуватися на будь-якій платформі, на якій встановлено інтерпретатор, що забезпечує чудову портативність. Компіляція Just-in-Time: Багато сучасних віртуальних машин використовують компіляцію JIT. Це означає, що під час виконання часто виконувані секції байт-коду динамічно перетворюються в нативний машинний код для негайного виконання, що значно підвищує продуктивність порівняно з чистою інтерпретацією.

Комп'ютери та інформаційні технології широко використовуються в різних сферах національної економіки та суспільства. Хоча досягнення в галузі комп'ютерних наук і технологій полегшують життя, вони також дають злочинцям нові інструменти для вчинення злочинів. Вразливість програмного забезпечення становить найбільшу загрозу для безпеки інформаційних систем. Різні інциденти безпеки, такі як несанкціоноване вторгнення з метою викрадення даних або блокування роботи додатків, трапляються часто і можуть завдати значної економічної шкоди окремим особам, підприємствам та суспільству. Нещодавно відбулася атака з використанням довільного коду, яка скористалася вразливістю в Apache Log4j і вплинула на багато систем та організацій.

Шкідливий фреймворк SocGholish JavaScript, імплантований на сотні веб-сайтів американських ЗМІ, заразив системи відвідувачів веб-сайтів і спровокував

нові атаки на програмне забезпечення. Необхідно швидко та ефективно виявляти вразливості програмного забезпечення та виправляти їх у режимі реального часу, перш ніж ними зможуть скористатися зловмисники.

Виявлення вразливостей програмного забезпечення є основним методом, який використовують дослідники безпеки для ідентифікації та оцінки потенційних ризиків програмного забезпечення. Через стрімке зростання використання та складності програмного забезпечення традиційні ручні підходи не можуть задовольнити сучасні якісні та кількісні вимоги. Тому для виявлення вразливостей широко застосовується машинне навчання, яке автоматизує процес виявлення та значно підвищує його ефективність.

Дослідники зацікавилися виявленням вразливостей на основі глибокого навчання. У порівнянні з традиційними методами машинного навчання, методи глибокого навчання дозволяють усунути необхідність ручного визначення характеристик експертами, автоматично витягувати корисну інформацію про характеристики зі складних даних та виявляти невідомі вразливості [34]. Широкі дослідження продемонстрували, що ця техніка значно покращує точність виявлення, одночасно зменшуючи кількість помилкових позитивних і помилкових негативних результатів виявлення вразливостей.

З моменту впровадження графічних нейронних мереж застосування графічної структури в цій галузі стало популярним. Використання графової структури для представлення характеристик вихідного коду дозволяє абстрагувати інформацію про характеристики коду на глибокому рівні. Тому, сучасні дослідження в галузі виявлення вразливостей, як правило, зосереджуються на підходах, що базуються на графових нейронних мережах. Незважаючи на те, що ці мережі краще представляють структурні характеристики та семантичну інформацію коду, все ж варто шукати оптимальне графове представлення коду, розумну векторну модель та ефективну модель мережі виявлення вразливостей.

Граф поєднує абстрактне синтаксичне дерево, граф потоку управління та граф залежності програми в об'єднану структуру даних для більш комплексного моделювання програм. Розбір та генерація графа представлення коду зазвичай вимагають створення робочого середовища для компіляції. Joern - це аналізатор, побудований на основі острівної граматики, який може виконувати нечіткий розбір без перевірки повної граматики наданого тексту. Після розбору коду на серію вузлів та ребер, що представляють відносини між вузлами, вони зберігаються в графічній базі даних Neo4j.

Потім мова обходу графа Gremlin використовується для розробки шаблонів для пошуку вразливостей з метою виявлення переповнення буфера, переповнення цілих чисел, вразливостей формату рядка та витоків пам'яті.

Наприклад, можна використовували код на рисунку 2.2 для представлення наступних типових методів представлення графа коду.

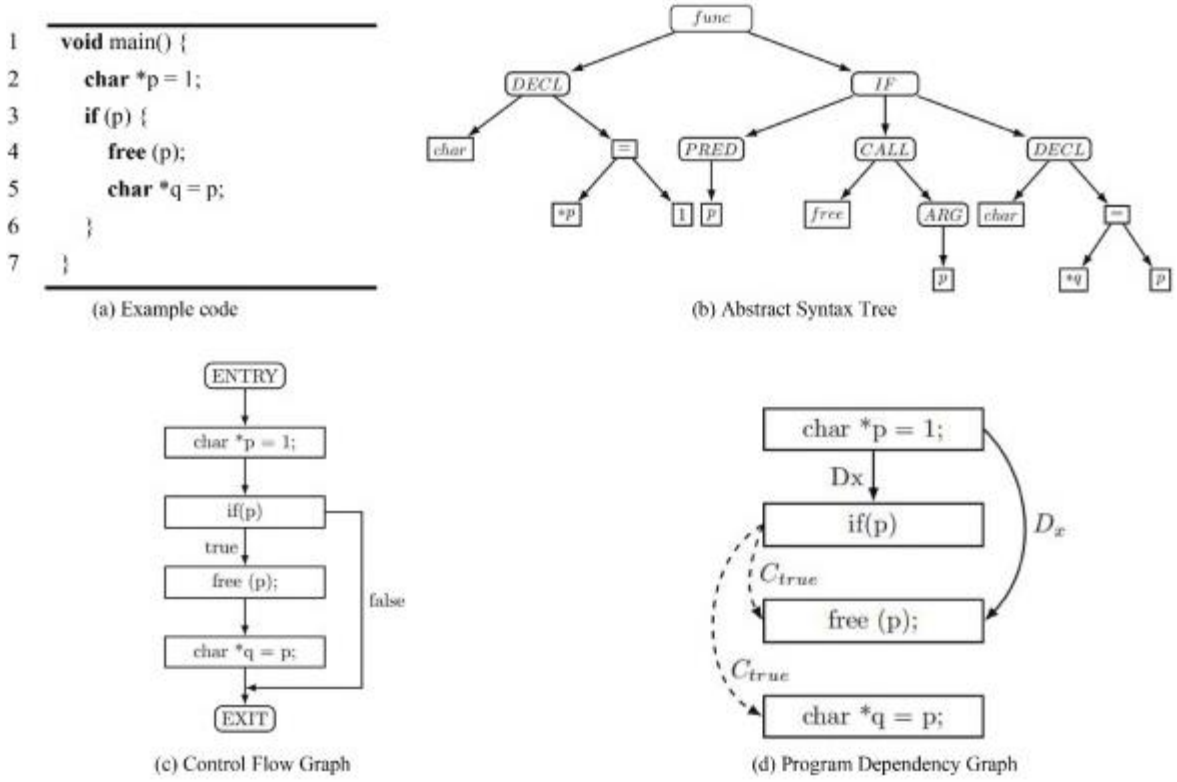


Рисунок 2.2 - Типові методи представлення графа коду

Графік потоку управління (CFG) - відображає порядок виконання операторів. Він описує всі шляхи, які програма може пройти під час виконання, та умови, які необхідно виконати. CFG враховує структуровані оператори управління для побудови попереднього графіка, такі як оператори if, for та switch. Потім він об'єднує неструктуровані оператори управління для перегляду CFG. Потік проходить від вхідного вузла до вихідного вузла. Оператори та умови представлені у вигляді вузлів. Направлені ребра представляють передачу управління та виведення шляхів. CFG інкапсулює базову інформацію про блоки, що допомагає зрозуміти програму, знайти недосяжний код та знайти синтаксичні структури, такі як цикли.

Граф залежності програми (PDG) - це графічне представлення, яке виконує розбиття програми для зменшення обсягу аналізу програми, відображаючи залежності даних і залежності управління між операторами та предикатами.

Це визначає набір змінних, що використовуються кожним оператором, оцінюючи доступність кожного оператора та предиката і виводячи ребра PDG з CFG. PDG попереднього прикладу коду показано на рисунокКрай C (пунктирний) вказує на залежність управління, яка поширюється на всі оператори під умовним розгалуженням. Край D (суцільний) вказує на залежність даних. Індекс ідентифікує залучену змінну. Край залежності даних означає, що до змінної буде здійснено доступ або її буде змінено в майбутньому.

2.2. Обґрунтування та вибір архітектури нейронної мережі

Загалом від досвіду роботи, інколи виникає питання - що робить рекурентні мережі такими особливими? Явним обмеженням звичайних нейронних мереж (а також згорткових мереж) є те, що їх API є надто обмеженим: вони приймають вектор фіксованого розміру як вхідні дані (наприклад, зображення) і генерують вектор фіксованого розміру як вихідні дані (наприклад, ймовірності різних класів). І не тільки це - ці моделі виконують це відображення, використовуючи фіксовану кількість обчислювальних кроків. Основна причина, чому рекурентні мережі є

більш цікавими, полягає в тому, що вони дозволяють нам оперувати послідовностями векторів: послідовностями у вхідних даних, вихідних даних або, в найзагальнішому випадку, в обох. Кілька прикладів можуть зробити це більш конкретним (рисунок 2.3):

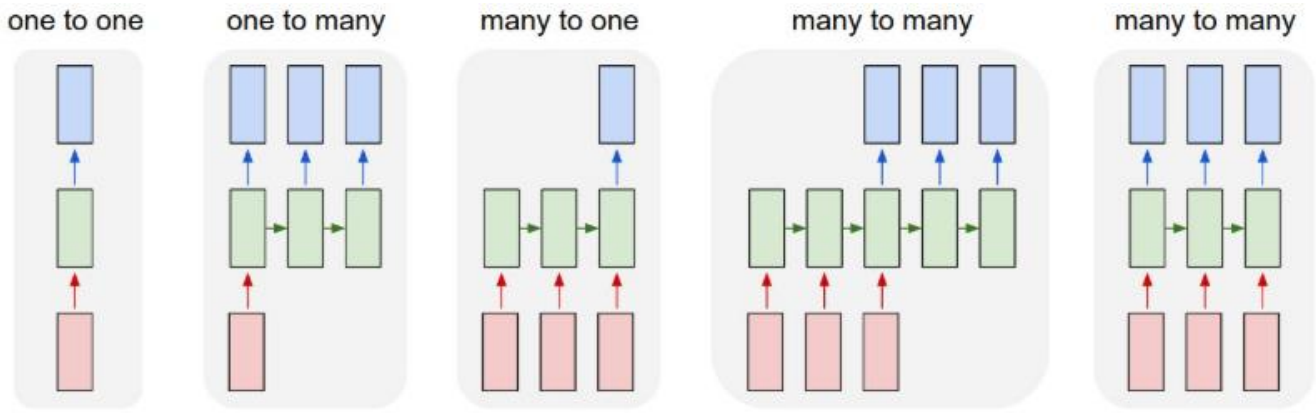


Рисунок 2.3 - Зразок коду з відповідними графіками AST, CFG та PDG.

На рисунку кожен прямокутник є вектором, а стрілки позначають функції (наприклад, множення матриць). Вхідні вектори позначені червоним кольором, вихідні вектори - синім, а зелені вектори позначають стан RNN.

Зліва направо показано:

1) Стандартний режим обробки без RNN, від вхідних даних фіксованого розміру до вихідних даних фіксованого розміру (наприклад, класифікація зображень).

2) Послідовний вихід (наприклад, підписування зображень, коли зображення перетворюється на речення зі слів).

3) Послідовний вхід (наприклад, аналіз настроїв, коли дане речення класифікується як таке, що виражає позитивний або негативний настрій).

4) Послідовний вхід і послідовний вихід (наприклад, машинний переклад: RNN читає речення англійською мовою, а потім видає речення французькою мовою).

5) Синхронізований послідовний вхід і вихід (наприклад, класифікація відео, де ми хочемо позначити кожен кадр відео). Важливо звернути увагу, що в кожному випадку немає заздалегідь визначених обмежень щодо довжини послідовностей, оскільки

рекурентне перетворення (зелене) є фіксованим і може застосовуватися стільки разів, скільки ми хочемо.

Як і слід було очікувати, послідовний режим роботи є набагато потужнішим порівняно з фіксованими мережами, які з самого початку приречені на фіксовану кількість обчислювальних кроків, а отже, і набагато привабливішим для тих з нас, хто прагне створити більш інтелектуальні системи. RNN поєднують вектор вхідних даних із вектором стану за допомогою фіксованої функції для створення нового вектора стану. З точки зору програмування це можна інтерпретувати як виконання фіксованої програми з певними вхідними даними та деякими внутрішніми змінними. З такого погляду RNN по суті описують програми. Насправді відомо, що RNN є Тюрінго-повноцінними в тому сенсі, що вони можуть імітувати довільні програми (з відповідними вагами). Але, подібно до теорем універсального наближення для нейронних мереж, не варто надавати цьому надто великого значення. Насправді, забудьте, що я це сказав.

Послідовна обробка за відсутності послідовностей. Ви можете подумати, що послідовності як вхідні або вихідні дані можуть бути відносно рідкісними, але важливо усвідомити, що навіть якщо ваші вхідні/вихідні дані є фіксованими векторами, все одно можна використовувати цей потужний формалізм для їх послідовної обробки.

Наприклад, на малюнку 2.4 показано алгоритм який вивчає політику рекурентної мережі, яка спрямовує її увагу на зображення; зокрема, він вчиться читати номери будинків зліва направо (Ba et al.). Справа рекурентна мережа генерує зображення цифр, навчаючись послідовно додавати колір на полотно (Gregor et al.):

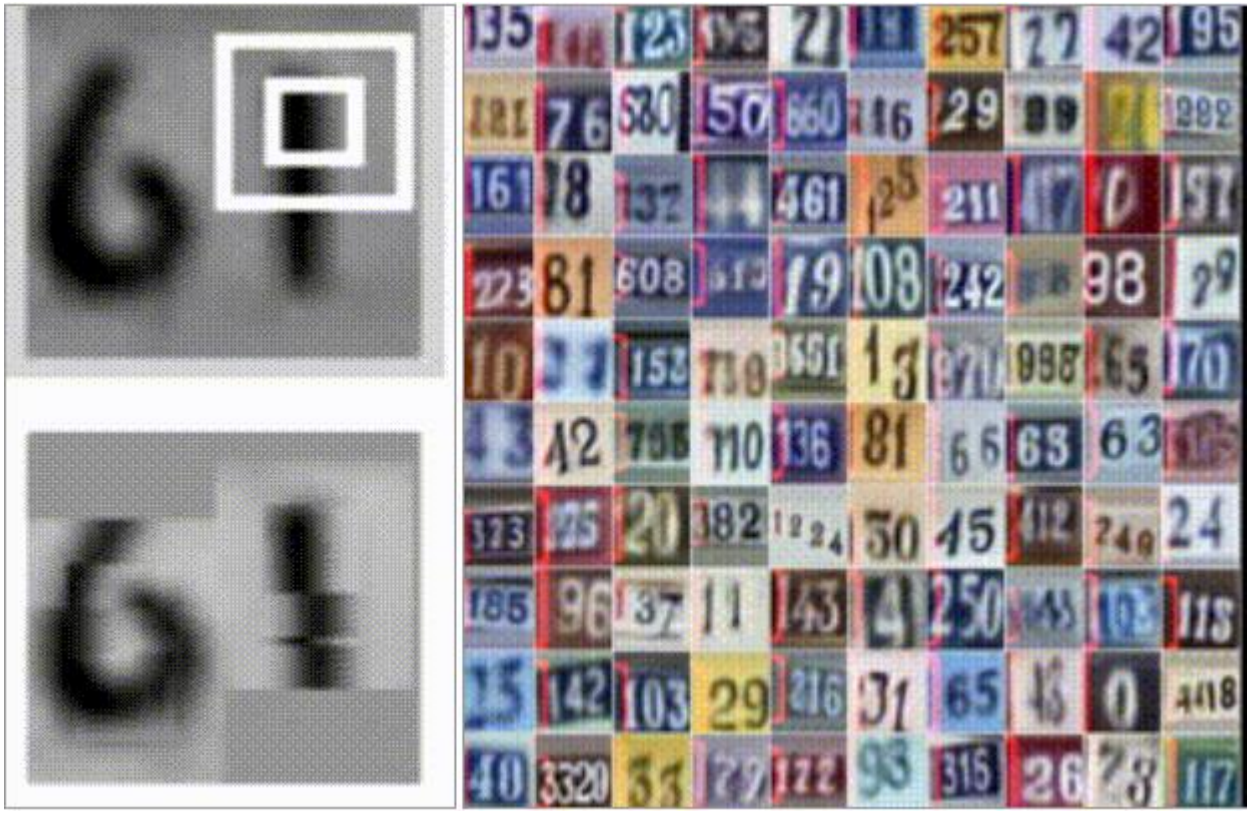


Рисунок 2.4 - Алгоритм рекурентної мережі

Висновок полягає в тому, що навіть якщо наші дані не мають форми послідовностей, ви все одно можете формулювати та навчати потужні моделі, які навчаються обробляти їх послідовно. Ви вивчаєте програми зі станом, які обробляють ваші дані фіксованого розміру.

Точне визначення семантичних типів стовпців у таблиці має вирішальне значення для різних завдань з пошуку інформації, таких як очищення даних, зіставлення схем та виявлення даних. Одним із нових застосувань є автоматичне тегування чутливих стовпців у таблиці, таких як особиста інформація, перед прийняттям рішення про те, яка інформація може бути оприлюднена. Попередні роботи показали, що підходи на основі машинного навчання перевершують традиційні методи в прогнозуванні семантичних типів. Sherlock, фреймворк для прогнозування одного стовпця, подає різні характеристики стовпця в глибоку нейронну мережу з прямим поширенням, щоб отримати прогноз.

Цей метод ігнорує глобальний контекст та залежності між стовпцями, що ускладнює розрізнення семантичних типів у випадках. SATO вдосконалює Sherlock, додаючи модуль моделювання тем та модуль структурованого прогнозування, щоб спільно прогнозувати семантичні типи всіх стовпців у таблиці, використовуючи тему таблиці та залежності між стовпцями в таблиці.

Наявність міжтабличної інформації може бути дуже корисною в тих випадках, коли цільовий стовпець не містить достатньо якісних даних для здійснення правильного семантичного прогнозування. Наприклад, якщо таблиця має стовпець із записами «Апельсин» і «Персик», семантичний тип є неоднозначним. Однак, ідентифікуючи та доповнюючи цей стовпець стовпцями подібних таблиць, які мають записи, такі як «Red» та «Blue», семантичний тип стає більш чітким, вказуючи на те, що цей стовпець, ймовірно, стосується кольорів, а не фруктів. У найновішій роботі, RECA, окрім значень цільового стовпця, ідентифікуються значення найбільш корисних подібних таблиць і передаються до BERT для отримання семантичного типу цільового стовпця.

Однак через невелике обмеження кількості вхідних токенів мовних моделей, таких як BERT, Doduo та RECA мають такі недоліки (рисунок 2.5)

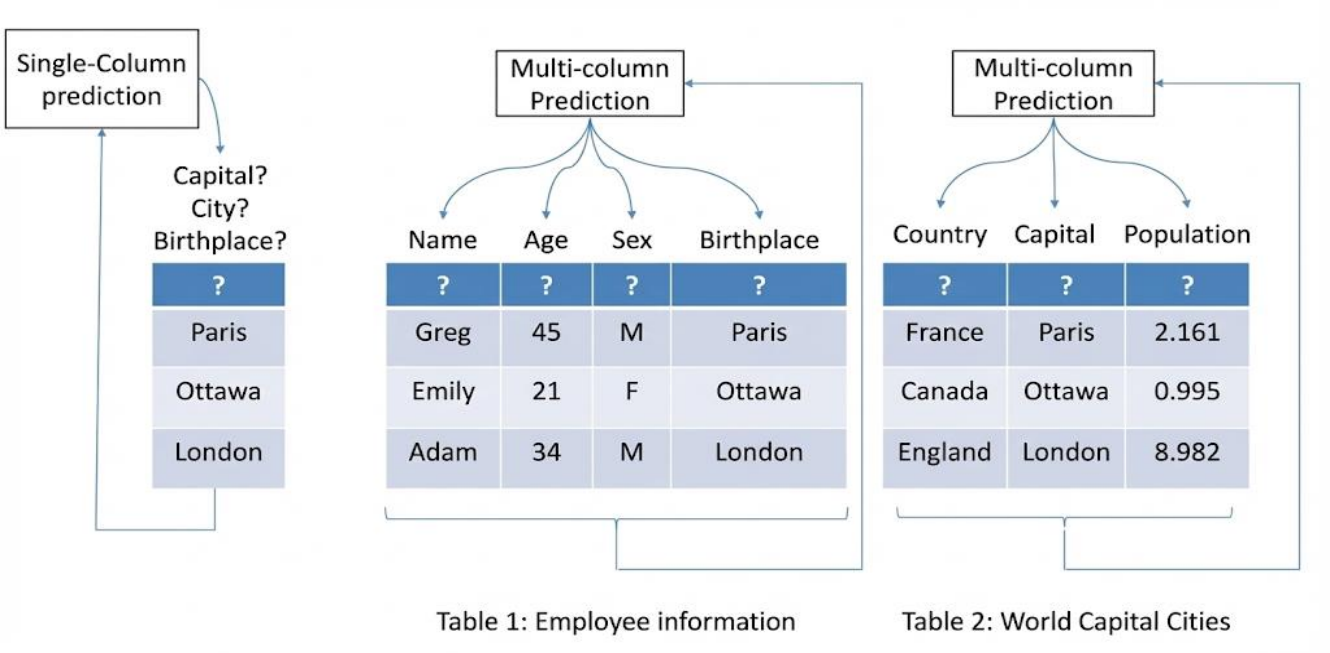


Рисунок 2.5 - Багатостовпцеве прогнозуванняі стовпців в таблиці.

Ми прагнемо передбачити семантичні типи стовпців заданої таблиці з відсутніми заголовками стовпців. Ця проблема називається анотацією таблиці. Щоб навчитися передбачати семантичні типи, як навчальні дані надається колекція таблиць з мітками

D, де кожна таблиця $t(c1,c2,\ldots,cn)$ складається з n стовпців, і кожен стовпець позначений як один із k заздалегідь визначених семантичних типів, які також називаються класами, наприклад, Вік, Ім'я, Країна (зауважте, що семантичні типи відрізняються від атомних типів, таких як ціле число та рядок). Кількість стовпців n та рядків може відрізнятися для різних таблиць. Зазвичай першим кроком є вилучення вектора ознак (вбудовування) для представлення стовпця ci. Після застосування функції вилучення ознак ϕ до значень стовпця ci та потенційної міжтабличної інформації, пов'язаної з ci, для стовпця ci генерується m-вимірний вектор ознак (вбудовування) ψi для стовпця ci. Решта завдання полягає у вивченні відображення f, яке, задаючи $\psi=<\psi 1,\ldots,\psi n>$ таблиці n немаркованих стовпців, прогнозує класи для n стовпців у таблиці.

Рисунок 2.6 показана структура GAIT - додає навчання GNN до модуля прогнозування з одним стовпцем, який у цій роботі є RECA. Результатом роботи RECA є розподіл класів для кожного стовпця в таблиці, що забезпечує початковий прихований стан вузла, який представляє цей стовпець в GNN. Для таблиці з n стовпцями RECA виконується n разів. Потім GNN навчає найкращі представлення прихованих станів всіх вузлів для мінімізації функції втрат за допомогою передачі повідомлень, що моделює залежності між стовпцями.

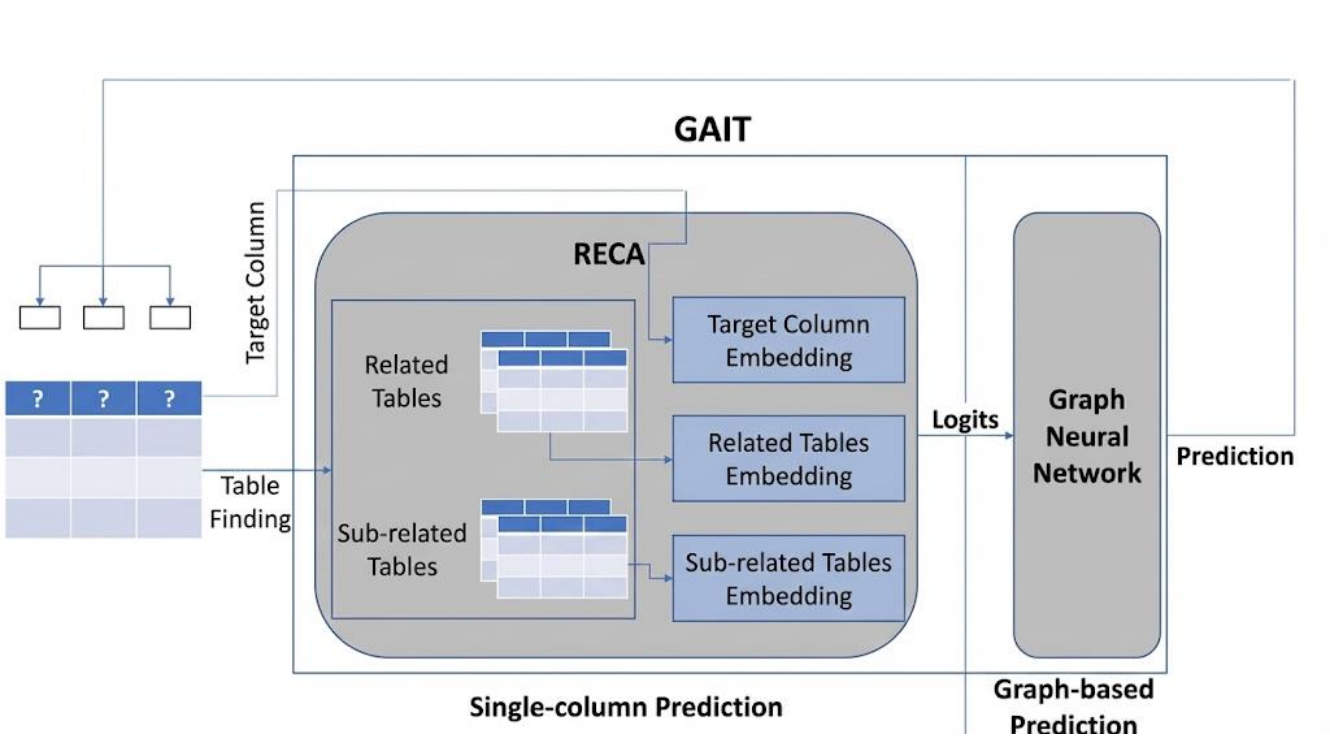


Рисунок 2.6 - Структура GAIT

Спочатку іде передача повідомлень, потім представлення всіх вузлів у міні-партії графіків оновлюється за допомогою механізму передачі повідомлень GNN по ребрах.

Для цього ми розглядаємо три різні типи GNN: графову конволюційну мережу (GCN) [11], графову нейронну мережу з воротами (GGNN) [15] та графову мережу уваги (GAT).

З наступними $UPDATE$ функціями, де σ є функцією активації, $N(u)$ є списком вузлів, підключених до вузла u, hus є представленням (також званим вбудовуванням) вузла u на кроці $s \geq 0$, $W(s)$ є параметром моделі:

$$h_u^{(s+1)} = \sigma\left(\sum_{v \in N(u) \cup u} \frac{W^{(s)} h_v^{(s)}}{\sqrt{|N(u)||N(v)|}} \right) \quad (1)$$

1) GCN: присвоює рівні ваги всім сусіднім вузлам під час оновлення вбудовування кожного вузла (рівняння 1).

GGNN: використовує гейтовану рекурентну одиницю (GRU) для оцінки повідомлень, що надходять від сусідніх вузлів, одночасно оновлюючи вбудовування кожного вузла (рівняння 2).

$$h_u^{(s+1)} = GRU\left(h_u^s, \sum_{v \in N(u)} W^{(s)} \quad h_v^s\right) \tag{2}$$

GAT: оновлює вбудовування вузлів (рівняння 3) відповідно до ваг багатоголовкової уваги (рівняння 4), де K — кількість голів уваги, $a(s,k)$ та $W(s,k)$ є параметрами моделі для голови уваги k, а $\bigoplus$ є конкатенацією.

$$\mathbf{h}_u^{(s+1)} = \bigoplus_{k=1}^{K} \left(\sigma \sum_{v \in N(u) \cup \{u\}} \alpha_{u,v}^{(s,k)} \mathbf{W}^{(s,k)} \mathbf{h}_v^{(s)} \right) \tag{3}$$

$$\alpha_{u,v}^{s,k} = \frac{\exp\left(\mathrm{ReLU}\left(\mathbf{a}^{(s,k)^T}\left(\mathbf{W}^{(s,k)}\mathbf{h}_u^{(s)} \oplus \mathbf{W}^{(s,k)}\mathbf{h}_v^{(s)}\right)\right)\right)}{\sum_{v' \in N(u) \cup \{u\}} \exp\left(\mathrm{ReLU}\left(\mathbf{a}^{(s,k)^T}\left(\mathbf{W}^{(s,k)}\mathbf{h}_u^{(s)} \oplus \mathbf{W}^{(s,k)}\mathbf{h}_v'^{(s)}\right)\right)\right)} \tag{4}$$

2.3. Розробка алгоритму підвищення точності детектування

У світі машинного навчання показники оцінки ефективності відіграють вирішальну роль у визначенні ефективності моделі. Такі показники, як точність, відтворюваність та показник F1, широко використовуються для оцінки моделей класифікації, особливо коли набір даних є незбалансованим, і їх можна охарактеризувати так:

1) Точність - вимірює точність позитивних прогнозів. Вона відповідає на питання: «Скільки з усіх елементів, які модель позначила як позитивні, насправді були позитивними?»

2) Відтворюваність (чутливість) - вимірює здатність моделі знаходити всі позитивні випадки. Вона відповідає на питання: «Скільки з усіх фактичних позитивних випадків модель правильно ідентифікувала?»

3) Показник F1 - гармонійне середнє значення точності та відтворюваності. Він зрівноважує ці два показники в одне число, що робить його особливо корисним, коли точність і відтворюваність є компромісними.

Хоча точність часто є першим показником, який оцінюють, вона може вводити в оману в незбалансованих наборах даних. Наприклад:

Модель, яка завжди прогнозує клас A, матиме точність 99%, але повністю не зможе виявити клас B.

У таких сценаріях точність, відкликання та показник F1 надають більш глибоке розуміння (рисунок 2.7).

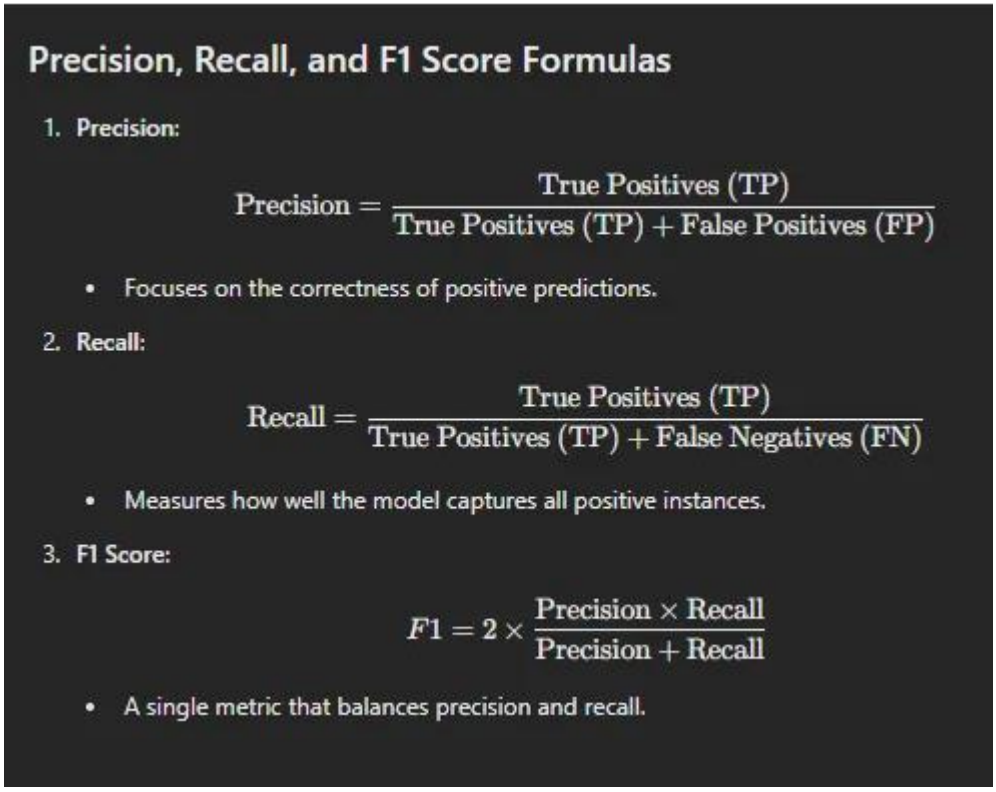


Рисунок 2.7 — Показники функцій F1

На рисунку 2.8 показана матриця плутанини, яка є необхідною для розуміння точності та відтворення. Ось її базова структура:

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Рисунок 2.8 - Матриця плутанини

Ця матриця включає такі дані як:

1) Справжнє позитивне (TP) - Правильно передбачені позитивні результати.

2) Хибне позитивне (FP) - Неправильно передбачені позитивні результати (помилка типу I).

3) Хибне негативне (FN) - Пропущені позитивні результати (помилка типу II).

4) Справжнє негативне (TN) - Правильно передбачені негативні результати.

Незбалансовані дані зазвичай стосуються проблем класифікації, коли класи представлені нерівномірно. Наприклад, може бути проблема класифікації з 2 класами зі 100 екземплярами. Загалом 80 екземплярів позначені як клас 1, а решта 20 екземплярів позначені як клас 2. Це незбалансований набір даних, і співвідношення екземплярів класу 1 до класу 2 становить 80:20 або, якщо бути точнішим.

Проблема незбалансованості класів може виникати як у двокласових, так і в багатокласових класифікаційних задачах. Більшість методів можна застосовувати в обох випадках. У подальшому розгляді ми будемо виходити з двокласової класифікаційної задачі, оскільки її легше зрозуміти та описати.

Більшість наборів даних для класифікації не мають рівної кількості екземплярів у кожному класі, але невелика різниця часто не має значення. Існують проблеми, де дисбаланс класів не тільки є поширеним, але й очікуваним. Наприклад, у наборах даних, що характеризують шахрайські транзакції, дисбаланс

є очевидним. Переважна більшість транзакцій буде належати до класу «Не шахрайство», а дуже невелика меншість - до класу «Шахрайство». Іншим прикладом є набори даних про відтік клієнтів, де переважна більшість клієнтів залишається з послугою (клас «Без відтоку»), а невелика меншість скасовує свою підписку. Коли є помірний дисбаланс класів, наприклад 4:1 у наведеному вище прикладі, це може спричинити проблеми.

Парадокс точності - це назва ситуації, описаної у вступі до цього допису. Це випадок, коли показники точності свідчать про те, що ви маєте чудову точність (наприклад, 90 %), але насправді точність лише відображає розподіл класів. Це дуже поширена ситуація, оскільки точність класифікації часто є першим показником, який ми використовуємо для оцінки моделей у наших задачах класифікації. Точність не є показником, який слід використовувати при роботі з незбалансованим набором даних. Ми бачили, що це вводить в оману.

Існують показники, які були розроблені для того, щоб надати більш правдиву інформацію при роботі з незбалансованими класами. Більше порад щодо вибору різних показників ефективності я даю у своєму дописі «Точність класифікації недостатня: інші показники ефективності, які ви можете використовувати».

Матриця плутанини це розбивка прогнозів на таблицю, що показує правильні прогнози (діагональ) і типи неправильних прогнозів (яким класам були приписані неправильні прогнози) і включає такі параметри:

1) Точність - показник точності класифікатора;

2) Відтворюваність - показник повноти класифікатора;

3) Показник F1 (або F-показник) - зважена середня точність і відтворюваність.

Каппа (або каппа Коена) - точність класифікації, нормалізована за незбалансованістю класів у даних. Криві ROC як і точність та відтворюваність, точність поділяється на чутливість та специфічність, і моделі можна вибирати на основі порогових значень балансу цих значень. Простий спосіб генерування

синтетичних зразків полягає у випадковому відборі атрибутів з екземплярів у класі меншості.

Можна відбирати їх емпірично у своєму наборі даних або використовувати такий метод, як Naive Bayes, який може відбирати кожен атрибут незалежно, коли працює у зворотному напрямку. Ви отримаєте більше і різних даних, але нелінійні взаємозв'язки між атрибутами можуть не зберегтися. Існують систематичні

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

3.1 Засоби розробки та архітектура програмного комплексу

Програмна реалізація системи автоматизованого виявлення вразливостей у смартконтрактах здійснена з використанням мови програмування Python версії 3.9, яка забезпечує необхідний баланс між швидкістю розробки, читабельністю коду та продуктивністю виконання. Вибір Python обумовлений наявністю потужної екосистеми бібліотек для машинного навчання, статичного аналізу та обробки даних, а також широкою підтримкою інструментів розробки та тестування [20].

Для реалізації моделі класифікації застосовано фреймворк PyTorch, який надає гнучкий та ефективний механізм побудови нейронних мереж, підтримує автоматичне диференціювання та оптимізовані операції на графічних процесорах. PyTorch обрано через його динамічну природу обчислювального графа, що спрощує налагодження та експериментування з архітектурами моделей, а також завдяки активній спільноті розробників та вичерпній документації [20]. Фреймворк забезпечує ефективне використання обчислювальних ресурсів, дозволяє масштабувати обчислення на декілька графічних процесорів та підтримує експорт моделей у формати, придатні для промислового використання.

Для статичного аналізу смартконтрактів та побудови абстрактного синтаксичного дерева використано інструмент Slither, який є одним з найпотужніших фреймворків для аналізу коду Solidity [21]. Slither забезпечує детальну інформацію про структуру контракту, включаючи функції, змінні стану, модифікатори доступу, події та потоки даних. Інтеграція зі Slither дозволяє отримати метрики складності коду, виявити потенційні антипатерни та зібрати статистичні характеристики контракту. Slither виконує міжпроцедурний аналіз,

будує граф викликів функцій, відстежує потоки даних між змінними та виявляє залежності між різними компонентами смартконтракту.

Для управління версіями компілятора Solidity застосовано інструменти solc-select та py-solc-x, які забезпечують автоматичний вибір відповідної версії компілятора на основі директиви pragma у вихідному коді контракту [24]. Це критично важливо, оскільки смартконтракти можуть бути написані для різних версій Solidity від 0.4 до 0.8, кожна з яких має свої особливості синтаксису та семантики [22]. Підтримка множини версій компілятора дозволяє аналізувати як історичні контракти, розгорнуті у ранніх версіях мережі Ethereum, так і сучасні контракти, що використовують найновіші можливості мови.

Підготовка даних, нормалізація ознак та обчислення метрик якості здійснюється з використанням бібліотек numpy, pandas та scikit-learn [26]. Бібліотека numpy забезпечує ефективні операції з багатовимірними масивами, pandas надає зручні структури даних для роботи з табличними даними, а scikit-learn містить реалізації алгоритмів попередньої обробки, валідації та оцінювання моделей. Використання цих стандартизованих інструментів гарантує відтворюваність результатів та спрощує інтеграцію з іншими компонентами екосистеми наукового програмування Python.

Архітектура розробленого програмного комплексу побудована за модульним принципом, що забезпечує розділення відповідальності між компонентами, спрощує тестування та підтримку коду, а також дозволяє незалежно розвивати окремі модулі системи. Структура прототипу організована у вигляді ієрархії каталогів, кожен з яких відповідає за конкретну функціональність.

Модуль парсера розміщено у каталозі src/parser та відповідає за зчитування вихідного коду смартконтрактів, валідацію синтаксису та інтеграцію зі Slither для отримання абстрактного синтаксичного дерева. Парсер виконує перевірку директиви pragma для визначення версії Solidity, автоматично встановлює відповідну версію компілятора та будує внутрішнє представлення контракту у

вигляді структурованого об'єкта. Компонент парсингу забезпечує валідацію вхідних даних перед їх передачею до наступних етапів конвеєра обробки.

Екстрактор ознак реалізовано у каталозі src/features та призначено для витягування числових характеристик з абстрактного синтаксичного дерева та метрик, отриманих від Slither. Екстрактор формує вектор ознак фіксованої розмірності, який слугує вхідними даними для моделі машинного навчання. До складу екстрактора входять компоненти для обчислення статистичних метрик, аналізу потоків управління та виявлення характерних патернів коду. Вектор ознак має розмірність 512 компонентів та формується без використання хешованих шумів, що забезпечує інтерпретовність результатів. Для узгодження розміру вектора у випадках, коли контракт має менше ознак, застосовується доповнення нульовими значеннями.

Модуль моделі та інференсу розташовано у каталогах src/models та src/inference відповідно. Перший містить визначення архітектури нейронної мережі, функції ініціалізації вагів та методи завантаження збережених моделей. Модель реалізована як багатошаровий перцептрон з логітною головою для бінарної класифікації за кожним типом вразливості. Модуль інференсу відповідає за виконання передбачень на нових даних, комбінування результатів моделі машинного навчання з евристичними правилами та застосування порогових значень для класифікації вразливостей. Евристичні правила доповнюють модель машинного навчання у випадках, коли певні патерни коду однозначно свідчать про наявність вразливості.

Система звітності реалізована у каталозі src/reporting та підтримує декілька форматів виводу результатів аналізу. Модуль звітності забезпечує генерацію консольних повідомлень з кольоровим виділенням критичності вразливостей, формування JSON-документів для програмної обробки результатів та створення HTML-звітів з візуальним представленням виявлених проблем та фрагментами коду. HTML-звіти використовують темну тему оформлення та забезпечують зручну

навігацію між різними типами вразливостей. Кожна виявлена вразливість супроводжується сніпетом коду з підсвічуванням синтаксису, номером рядка та назвою функції, де знаходиться потенційна проблема.

Компоненти тренування та тюнінгу гіперпараметрів розміщено у каталозі src/training. Ці модулі відповідають за завантаження навчальних даних, ініціалізацію моделі з заданими параметрами, виконання циклу навчання з валідацією, збереження контрольних точок та логування метрик. Модуль тюнінгу реалізує пошук оптимальних гіперпараметрів за допомогою методів випадкового пошуку або байєсівської оптимізації з використанням бібліотеки Optuna [25].

Утиліта конвертації датасету реалізована у файлі tools/build_dataset.py та призначена для перетворення вихідних файлів смартконтрактів у структурований CSV-формат, придатний для навчання моделі. Утиліта виконує попередню обробку коду, витягування міток класів з анотацій або шляхів до файлів, дедуплікацію та розбиття на тренувальну, валідаційну та тестову вибірки. Процес конвертації включає фільтрацію контрактів за підтримуваними версіями pragma та видалення дублікатів на основі хешування коду.

Інтерфейс командного рядка реалізовано у файлі src/cli/main.py та надає єдину точку входу для всіх операцій системи. CLI підтримує аналіз окремих файлів або цілих директорій, налаштування параметрів моделі та порогових значень, вибір формату звіту та управління логуванням [27].

На рисунку 3.1 зображено загальну схему архітектури системи, яка ілюструє послідовність обробки даних від вхідного файлу до фінального звіту. На першому етапі парсер через інтеграцію зі Slither формує абстрактне синтаксичне дерево та витягує базові метрики контракту, включаючи кількість функцій, змінних стану, модифікаторів доступу та подій. На другому етапі екстрактор ознак збирає числові характеристики у вектор фіксованої розмірності, який становить 512 компонентів та охоплює статистичні метрики, характеристики потоків управління та структурні патерни. На третьому етапі модель машинного навчання та евристичні правила

генерують оцінки ймовірності для кожного типу вразливості, застосовуючи глобальні та покласові порогові значення. На завершальному етапі постпроцесинг анотує кожну виявлену вразливість номером рядка та ідентифікатором функції, після чого формує структуровані звіти у вибраному форматі.

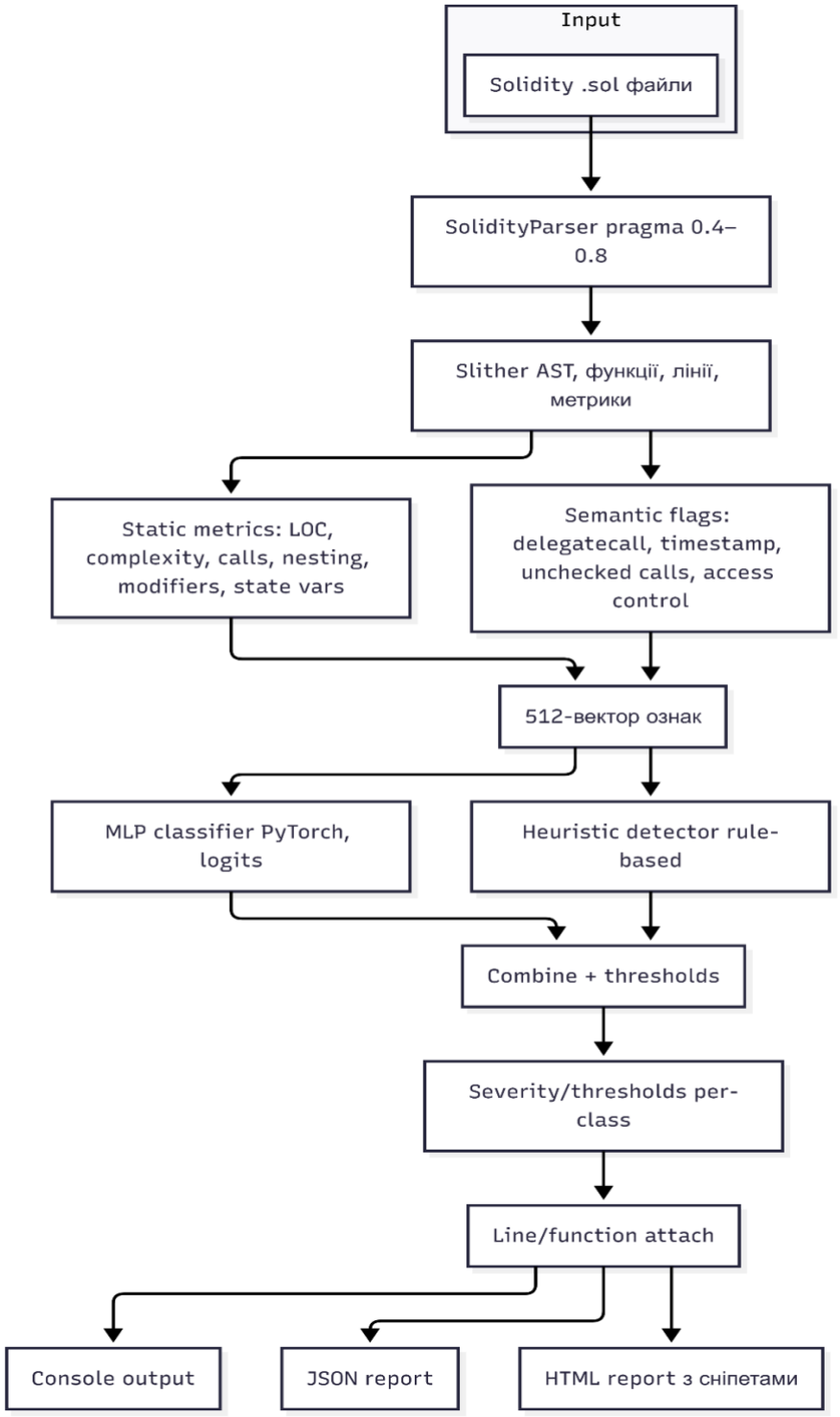


Рисунок 3.1 – Загальна схема архітектури системи виявлення вразливостей

На рисунку 3.2 наведено приклад запуску CLI з основними параметрами командного рядка. Параметр --input визначає шлях до файлу або директорії з смартконтрактами для аналізу. Параметр --model-weights вказує на файл зі збереженими вагами навченої моделі, зазвичай у форматі PyTorch з розширенням .pth або .pt. Параметр --thresholds дозволяє встановити глобальні або покласові порогові значення для класифікації у форматі JSON. Параметр --output-format визначає формат звіту, який може приймати значення console, json або html. Додаткові параметри дозволяють налаштувати рівень деталізації логування через --verbose, шлях до конфігураційного файлу через --config та режим паралельної обробки через --parallel.

```
Analyzing contract: vulnerable_unchecked_call.sol
Path: tests\test_contracts\vulnerable_unchecked_call.sol
Lines of code: 16
Findings: 2

[MEDIUM] access_control
  Description: Functions may miss proper access control modifiers.
  Recommendation: Add onlyOwner/role-based modifiers and test authorization paths.
  Confidence: 0.800

[CRITICAL] unchecked_calls
  Line: 13
  Function: sendData
  Description: External calls may not have their return values checked.
  Recommendation: Check return values or use try/catch to handle failures.
  Confidence: 0.999
```

Рисунок 3.2 – Приклад запуску CLI з параметрами аналізу

На рисунку 3.3 демонструється структура каталогів проєкту, організована наступним чином. Кореневий каталог містить конфігураційні файли requirements.txt та setup.py, документацію README.md та скрипти автоматизації Makefile. Каталог src містить всі модулі системи, організовані за функціональним призначенням у підкаталоги parser, features, models, inference, reporting, training та cli. Каталог tests містить набір модульних та інтеграційних тестів для перевірки коректності роботи

компонентів. Каталог data призначений для зберігання датасетів у підкаталозі raw, збережених моделей у підкаталозі models та проміжних результатів у підкаталозі processed. Каталог tools містить допоміжні скрипти для підготовки даних build_dataset.py, оцінювання моделей evaluate.py та генерації документації generate_docs.py. Каталог contracts містить приклади смартконтрактів для тестування системи, розділені на підкаталоги vulnerable та safe. Каталог docs містить технічну документацію, посібники користувача та API-специфікації.

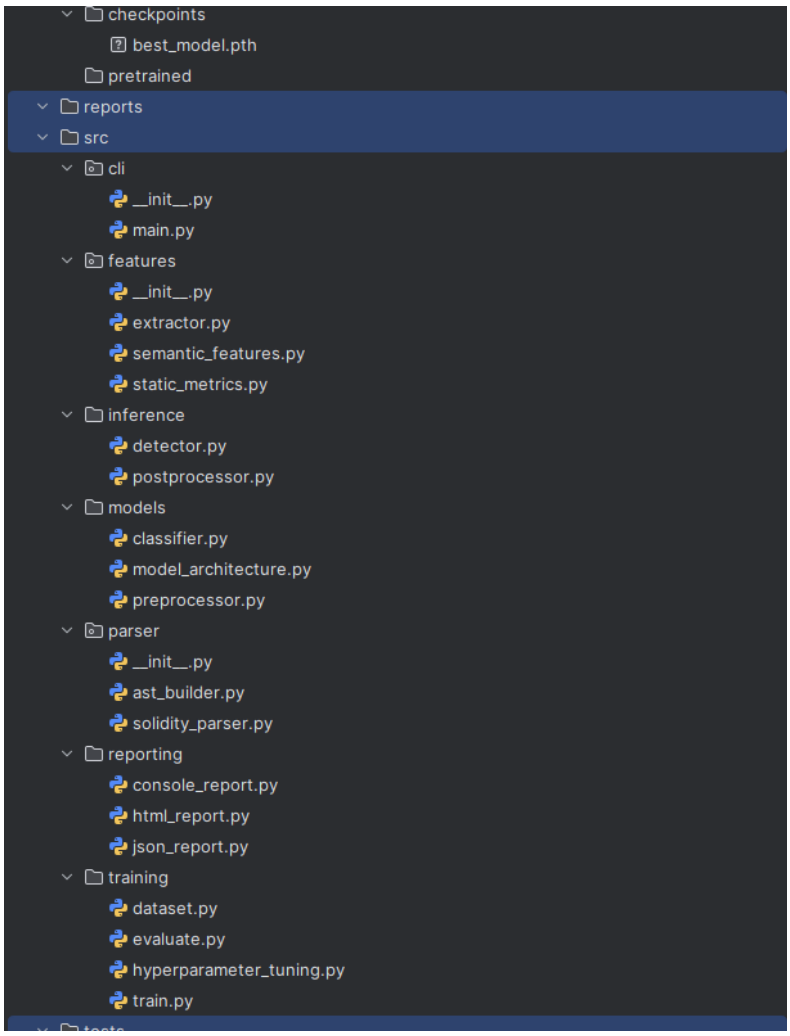


Рисунок 3.3 – Структура каталогів програмного комплексу

Компонент парсингу виконує першу фазу обробки смартконтрактів та забезпечує валідацію вхідних даних перед їх передачею до наступних етапів

конвеєра. Парсер починає роботу з перевірки директиви pragma у вихідному коді контракту для визначення необхідної версії компілятора Solidity. Система підтримує контракти, написані для версій від 0.4 до 0.8, що охоплює переважну більшість існуючих смартконтрактів у мережі Ethereum.

Після визначення версії компілятора парсер використовує solc-select для автоматичного встановлення відповідної версії, якщо вона ще не присутня в системі. Цей механізм забезпечує коректну компіляцію контрактів незалежно від версії Solidity, для якої вони були розроблені, та усуває проблеми несумісності між різними версіями мови. Інтеграція зі Slither здійснюється через програмний інтерфейс, який дозволяє отримати детальну інформацію про структуру контракту без необхідності запуску зовнішніх процесів.

Парсер витягує позиції функцій у вихідному коді, що дозволяє точно вказати місце розташування потенційних вразливостей у фінальному звіті. Крім позицій, парсер збирає агреговані метрики, які включають загальну кількість функцій у контракті, кількість змінних стану, кількість модифікаторів доступу, кількість подій, цикломатичну складність кожної функції, кількість зовнішніх викликів, максимальну глибину вкладеності блоків управління, наявність функцій з модифікатором payable або видимістю public, використання checked або unchecked arithmetic operations та наявність патернів контролю доступу. Ці метрики формують базовий набір характеристик контракту, які доповнюються більш складними ознаками на етапі екстракції. [Повний код програми зображений в додатку А].

Екстрактор ознак формує вектор розмірності 512 компонентів, який охоплює як статистичні метрики, отримані безпосередньо від парсера, так і більш складні характеристики, обчислені шляхом аналізу абстрактного синтаксичного дерева. Вектор ознак не містить хешованих шумів, що забезпечує його інтерпретовність та дозволяє розробникам розуміти, які саме характеристики коду впливають на передбачення моделі. У випадках, коли контракт має менше ознак, ніж розмірність

вектора, застосовується доповнення нульовими значеннями для узгодження розміру входу нейронної мережі.

Модель машинного навчання реалізована як багатошаровий перцептрон з логітною головою для виконання бінарної класифікації за кожним типом вразливості незалежно. Архітектура моделі складається з вхідного шару розмірності 512, декількох прихованих шарів з нелінійними функціями активації ReLU та вихідного шару з п'ятьма нейронами, кожен з яких відповідає за передбачення одного типу вразливості. Використання логітної голови дозволяє застосовувати функцію втрат Binary Cross Entropy with Logits Loss, яка поєднує сигмоїдну активацію та обчислення втрат в одній операції для покращення числової стабільності.

Інференс поєднує результати моделі машинного навчання з евристичними правилами для підвищення точності виявлення. Евристичні правила базуються на детерміністичних патернах коду, які однозначно свідчать про наявність певних типів вразливостей. Наприклад, використання конструкції call.value без перевірки повернутого значення або відсутність патерну checks-effects-interactions у функціях, що здійснюють передачу ефіру. Застосування глобальних та покласових порогових значень дозволяє налаштувати чутливість системи окремо для кожного типу вразливості, балансуючи між кількістю хибнопозитивних та хибнонегативних спрацювань.

Система звітності забезпечує три основні формати виводу результатів аналізу. Консольний формат призначений для інтерактивної роботи розробників та надає стислу інформацію про виявлені вразливості з кольоровим виділенням критичності. JSON-формат призначений для автоматизованої обробки результатів іншими інструментами та містить структуровану інформацію про кожну виявлену вразливість, включаючи тип, оцінку ймовірності, позицію у коді та рекомендації щодо усунення. HTML-формат призначений для перегляду людиною та надає найбільш детальну та візуально привабливу презентацію результатів.

На рисунку 3.4 показано приклад HTML-звіту з фрагментом коду, в якому виявлено вразливість. HTML-звіт використовує темну тему оформлення, яка зменшує навантаження на зір під час тривалого перегляду. Кожна вразливість супроводжується сніпетом коду з підсвічуванням синтаксису Solidity, номером рядка, назвою функції та оцінкою критичності, позначеною кольором від зеленого до червоного. Звіт організований за типами вразливостей з можливістю згортання та розгортання секцій для зручної навігації. Додатково надаються рекомендації щодо усунення кожної виявленої вразливості та посилання на відповідну документацію.

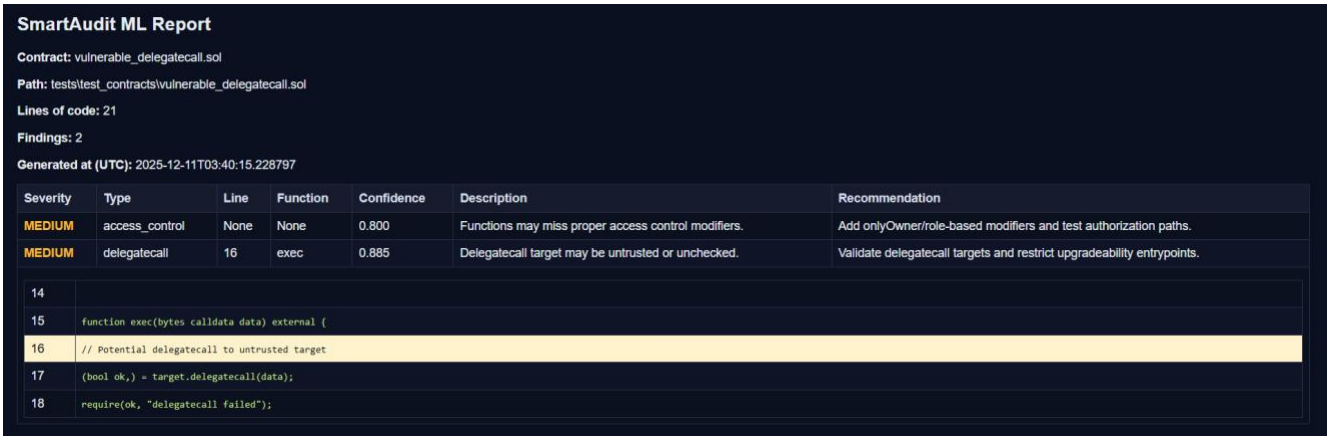


Рисунок 3.4 – HTML-звіт із сніпетом коду та кольоровим виділенням критичності

На рисунку 3.5 наведено фрагмент JSON-звіту, який містить структуровану інформацію про виявлені вразливості у машинозчитуваному форматі. JSON-документ включає загальну інформацію про проаналізований контракт, метадані аналізу, масив виявлених вразливостей та загальну статистику. Кожна вразливість описується об'єктом, що містить тип, оцінку ймовірності, рівень довіри, позицію у коді, ідентифікатор функції, короткий опис та детальні рекомендації. Структура JSON-звіту дозволяє легко інтегрувати систему з іншими інструментами аналізу коду, системами відстеження помилок та платформами безперервної інтеграції.

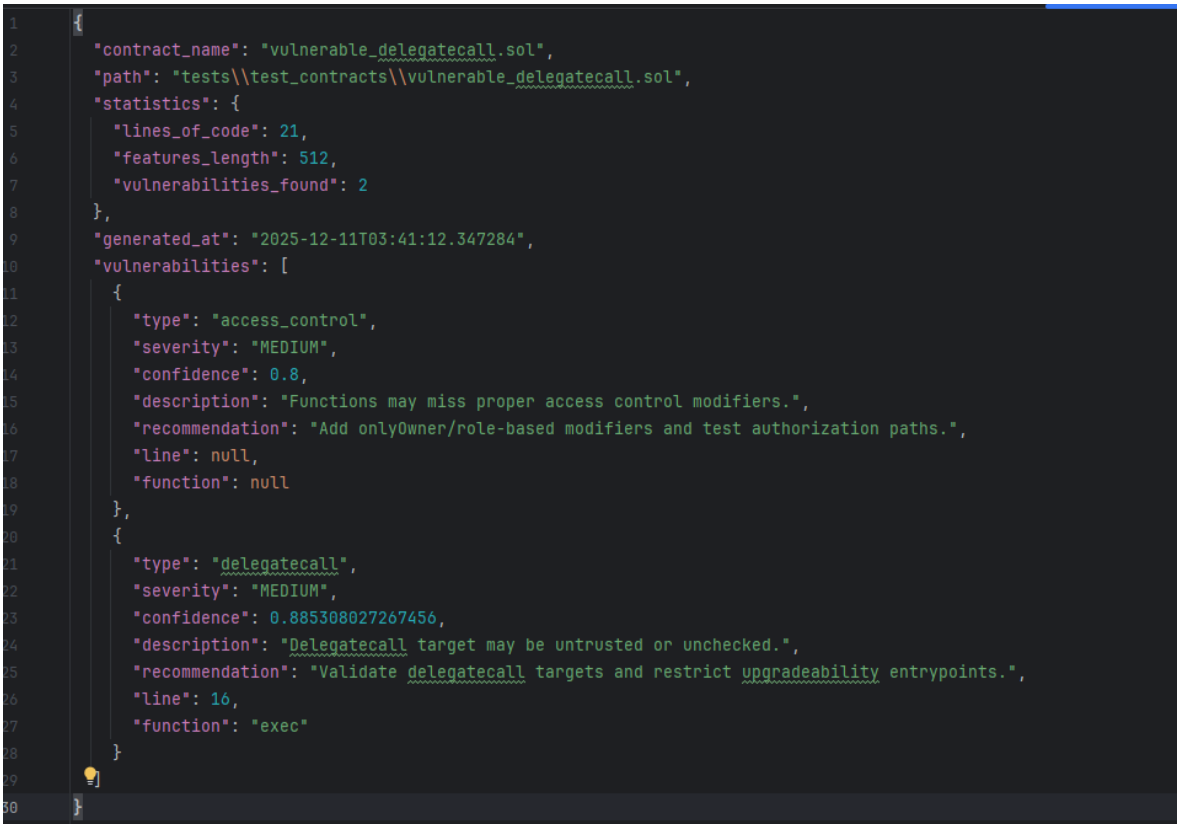


Рисунок 3.5 – Фрагмент JSON-звіту з структурованою інформацією про вразливості

Інтерфейс командного рядка надає гнучкий механізм запуску аналізу з різноманітними параметрами налаштування. CLI приймає шлях до файлу або директорії, що містить смартконтракти для аналізу, шлях до файлу зі збереженими вагами навченої моделі, параметри порогових значень для класифікації та формат вихідного звіту. Параметр шляху підтримує як абсолютні, так і відносні шляхи, а також глобальні маски для пакетного аналізу великої кількості файлів.

Параметри порогових значень можуть бути встановлені глобально для всіх типів вразливостей або окремо для кожного класу. Глобальне порогове значення застосовується у випадках, коли не вказано специфічного порогу для певного типу вразливості. Покласові порогові значення дозволяють налаштувати чутливість системи окремо для reentrancy, overflow, access_control, unchecked_calls та

delegatecall вразливостей. Налаштування порогів здійснюється через JSON-файл конфігурації або безпосередньо через параметри командного рядка.

Формат звіту визначає спосіб представлення результатів аналізу. Консольний формат виводить результати безпосередньо у термінал з кольоровим виділенням та форматуванням для зручності сприйняття. JSON-формат зберігає результати у файл для подальшої обробки іншими інструментами. HTML-формат генерує автономний веб-документ, який можна переглядати у будь-якому сучасному браузері без необхідності встановлення додаткового програмного забезпечення.

На рисунку 3.6 ілюструється включення CLI у процеси безперервної інтеграції та доставки, зокрема у системах GitHub Actions або Jenkins [29]. Інтеграція системи у CI/CD конвеєр дозволяє автоматично перевіряти всі смартконтракти на наявність вразливостей перед їх розгортанням у блокчейн мережу. Типовий робочий процес включає клонування репозиторію, встановлення залежностей, завантаження навченої моделі, запуск аналізу на всіх смартконтрактах проєкту та генерацію HTML або JSON звітів як артефактів збірки.

Конфігурація CI/CD визначає умови запуску аналізу, які можуть включати push-події до певних гілок, створення pull request або виконання за розкладом. У разі виявлення вразливостей високої критичності процес збірки може бути автоматично зупинений до усунення проблем. Звіти зберігаються як артефакти збірки та доступні для перегляду розробникам через веб-інтерфейс платформи CI/CD.

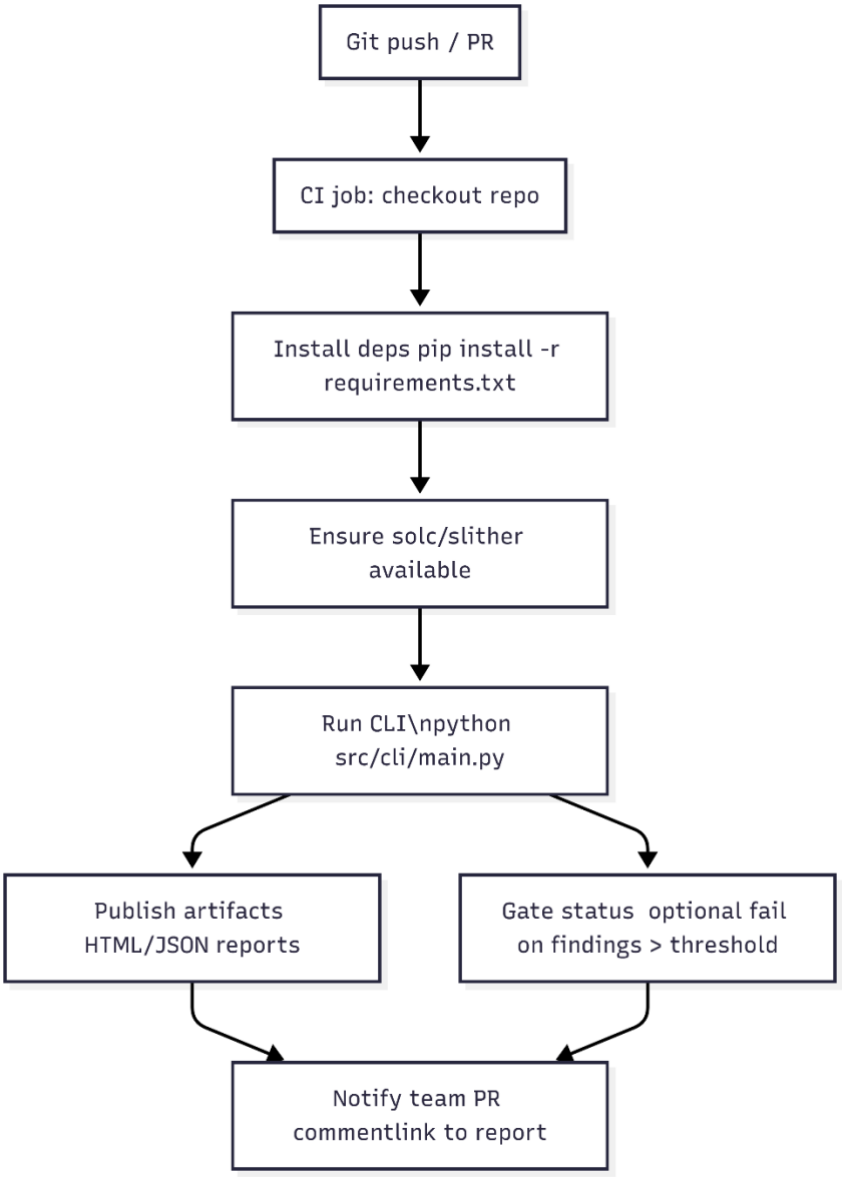


Рисунок 3.6 – Схема інтеграції CLI у конвеєр CI/CD

Додатково можна налаштувати відправлення сповіщень у системи комунікації команди або створення завдань у системах відстеження помилок.

Інтеграція системи у процеси розробки вимагає налаштування середовища виконання, яке включає встановлення необхідних версій компілятора Solidity, Python та його залежностей. Для забезпечення відтворюваності результатів рекомендується використовувати контейнеризацію через Docker або ізольовані віртуальні середовища Python через venv або conda. Файл requirements.txt містить

повний перелік залежностей з фіксованими версіями для запобігання конфліктам між бібліотеками.

Контроль версій моделі здійснюється через систему версіонування, яка зберігає інформацію про архітектуру моделі, гіперпараметри тренування та метрики якості на тестовій вибірці. Кожна збережена модель супроводжується метаданими, що включають дату тренування, розмір датасету, досягнуті показники precision, recall та F1-score для кожного класу вразливостей. Це дозволяє відстежувати еволюцію моделі та при необхідності повертатися до попередніх версій.

3.2 Навчання моделі та оптимізація гіперпараметрів

Навчання моделі виконується на структурованому датасеті у форматі CSV, який містить колонки contract_name для ідентифікації контракту, code для зберігання вихідного коду, reentrancy для позначення наявності вразливості повторного входу, overflow для позначення можливості переповнення цілочисельних змінних, access_control для позначення проблем контролю доступу, unchecked_calls для позначення неперевірених зовнішніх викликів, timestamp для часової мітки створення контракту та delegatecall для позначення небезпечного використання делегованих викликів.

Функція втрат для навчання моделі обрана як Binary Cross Entropy with Logits Loss, яка поєднує сигмоїдну активацію та обчислення втрат в одній операції для покращення числової стабільності під час тренування. Для компенсації дисбалансу класів у датасеті застосовується параметр pos_weight, який збільшує вагу позитивних прикладів пропорційно до ступеня дисбалансу. Значення pos_weight обчислюється як відношення кількості негативних прикладів до кількості позитивних прикладів для кожного типу вразливості окремо.

Оптимізатор AdamW використовується для оновлення вагів моделі під час навчання. AdamW є модифікацією класичного оптимізатора Adam з покращеною регуляризацією через розв'язане декомпозиція weight decay від градієнтного оновлення. Це забезпечує більш стабільну збіжність та кращу генералізацію моделі на невидимих даних. Початкова швидкість навчання встановлюється емпірично через експерименти на валідаційній вибірці.

Опційний планувальник швидкості навчання StepLR застосовується для динамічного зменшення learning rate після певної кількості епох без покращення метрик на валідаційній вибірці. Зменшення швидкості навчання дозволяє моделі виконувати більш точну підстройку вагів на пізніх етапах тренування, коли градієнти стають меншими. Типово швидкість навчання зменшується у десять разів після кожних п'яти епох без покращення валідаційної втрати.

Механізм раннього зупинення застосовується для запобігання перенавчанню та економії обчислювальних ресурсів. Тренування припиняється автоматично, якщо валідаційна втрата не покращується протягом певної кількості послідовних епох, яка називається терпимістю або patience. Типове значення patience встановлюється на рівні десяти епох, що дозволяє моделі пройти через тимчасові плато у процесі навчання.

На рисунку 3.7 подано криві тренувальної та валідаційної втрати за епохами навчання моделі. Графік демонструє типову динаміку навчання, де обидві криві монотонно спадають протягом перших епох, після чого досягають плато. Відсутність значного розходження між тренувальною та валідаційною втратами свідчить про те, що модель не схильна до перенавчання та добре генералізує на невидимих даних. Невеликі коливання валідаційної втрати є нормальним явищем через стохастичну природу градієнтного спуску та варіативність валідаційної вибірки.

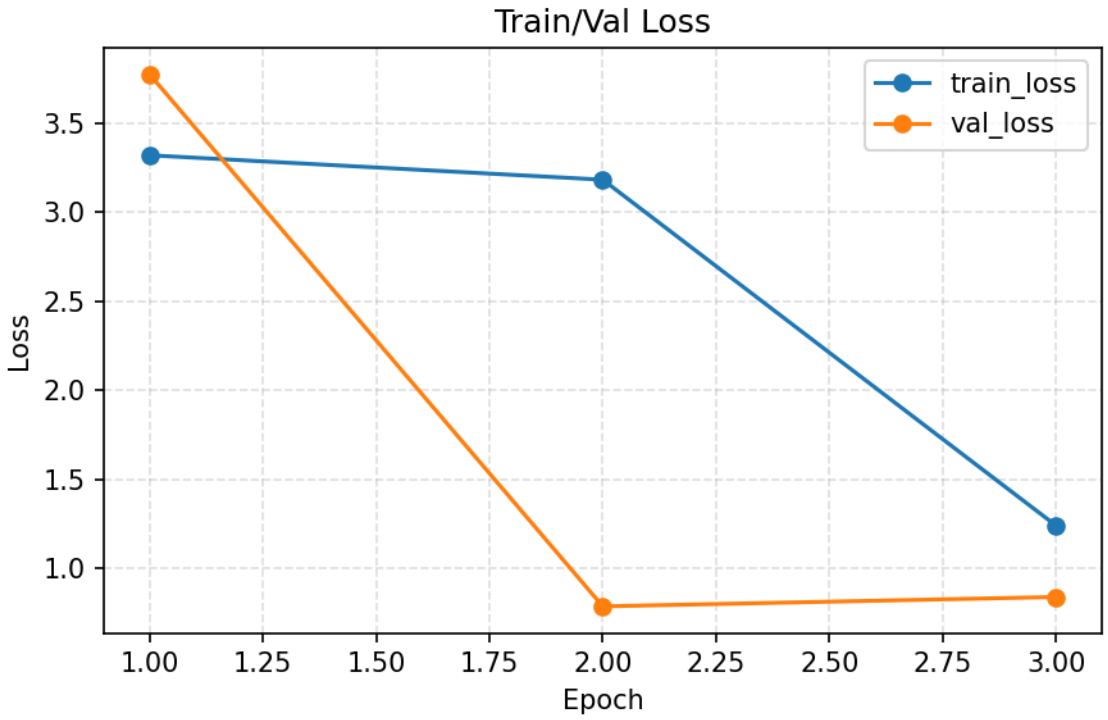


Рисунок 3.7 – Криві тренувальної та валідаційної втрати за епохами

На таблиці 3.1 наведено оптимальні значення гіперпараметрів, визначені через процес систематичного пошуку та валідації. Кількість епох встановлена на рівні п'ятдесяти з можливістю раннього зупинення для запобігання непродуктивному тренуванню. Розмір батчу обрано тридцять два приклади, що забезпечує баланс між стабільністю градієнтів та ефективністю використання пам'яті графічного процесора. Початкова швидкість навчання встановлена на рівні одна тисячна, що є типовим значенням для оптимізатора AdamW. Коефіцієнт weight decay встановлено на рівні одна стотисячна для забезпечення помірної регуляризації. Значення pos_weight варіюються для різних класів вразливостей залежно від ступеня дисбалансу у датасеті. Планувальник швидкості навчання налаштований на зменшення learning rate у десять разів кожні п'ять епох без покращення.

Таблиця 3.1 – Оптимальні гіперпараметри моделі

Параметр	Значення
Кількість епох	50 (з early stopping)
Розмір батчу	32
Початкова швидкість навчання	0.001
Weight decay	0.00001
Pos_weight для reentrancy	3.5
Pos_weight для overflow	4.2
Pos_weight для access_control	2.8
Pos_weight для unchecked_calls	5.1
Pos_weight для delegatecall	6.3
StepLR gamma	0.1
StepLR step size	5 епох
Early stopping patience	10 епох

Оцінка якості моделі на тестовій вибірці виконується скриптом evaluate.py, який завантажує збережену модель, обробляє всі приклади тестової вибірки та обчислює макроусереднені метрики precision, recall та F1-score. Макроусереднення обчислює метрики окремо для кожного класу та усереднює їх з рівними вагами, що забезпечує справедливу оцінку якості незалежно від розміру класу. Додатково обчислюються мікроусереднені метрики, які враховують всі передбачення разом та надають більшу вагу більш представленим класам.

Процес валідації виконується після кожної епохи тренування на окремій валідаційній вибірці, яка не використовується для оновлення вагів моделі. Валідаційна втрата та метрики якості логуються для відстеження прогресу навчання та виявлення потенційних проблем, таких як перенавчання або недостатня складність моделі. Контрольні точки моделі зберігаються після кожної епохи, що

покращує валідаційні метрики, дозволяючи відновити найкращу версію моделі після завершення тренування.

Для запобігання перенавчанню моделі застосовано комплекс технік регуляризації, які обмежують складність моделі та покращують її здатність генералізувати на невидимих даних. Техніка dropout застосовується після кожного прихованого шару нейронної мережі з ймовірністю відключення нейронів на рівні п'ятнадцять відсотків. Dropout запобігає коадаптації нейронів, змушуючи мережу навчатися більш надійним та розподіленим представленням ознак.

Регуляризація weight decay додає штраф до функції втрат, пропорційний квадрату норми вагів моделі. Це стимулює модель підтримувати малі значення вагів та запобігає надмірній складності навчених функцій. Коефіцієнт weight decay встановлений на рівні одна стотисячна, що забезпечує помірну регуляризацію без надмірного обмеження виразності моделі.

Механізм раннього зупинення служить ефективним методом запобігання перенавчанню шляхом припинення тренування, коли модель починає переоптимізуватися під тренувальні дані на шкоду генералізації. Моніторинг валідаційної втрати дозволяє виявити момент, коли подальше навчання призводить до покращення на тренувальній вибірці але погіршення на валідаційній. Зупинення тренування у цей момент гарантує, що збережена модель має оптимальний баланс між підгонкою під дані та генералізацією.

Планувальник швидкості навчання StepLR знижує learning rate після кількох епох без покращення валідаційних метрик. Поступове зменшення швидкості навчання дозволяє моделі виконувати більш точну підстройку вагів на пізніх етапах тренування, коли градієнти стають меншими та оптимізація наближається до локального мінімуму. Зменшення швидкості навчання також допомагає уникнути коливань навколо оптимальної точки через надто великі кроки оновлення.

Криві навчання на рисунку 3.7 демонструють збіжність тренувальної та валідаційної втрат без значного розходження, що свідчить про успішне

застосування технік регуляризації. Тренувальна втрата монотонно спадає протягом перших двадцяти епох, після чого досягає плато та коливається навколо стабільного значення. Валідаційна втрата слідує подібній траєкторії з невеликим запізненням, що є нормальним через менший розмір валідаційної вибірки та вищу варіативність оцінок. Відсутність систематичного зростання валідаційної втрати при спадаючій тренувальній втраті підтверджує, що модель не перенавчається.

Аналіз індивідуальних передбачень моделі на валідаційній вибірці показує, що найбільші помилки виникають на прикладах з неоднозначними патернами коду або недостатньою кількістю контексту для прийняття рішення. Наприклад, функції з складною логікою контролю доступу можуть бути хибно класифіковані через відсутність явних патернів у видимій частині коду. Такі випадки потребують розширення контексту аналізу або додаткових джерел інформації для покращення точності.

Якість навчальних даних має критичне значення для досягнення високої точності моделі класифікації вразливостей. Дедуплікація датасету виконується шляхом обчислення хешів вихідного коду контрактів та видалення точних дублікатів, які можуть призвести до витоку інформації між тренувальною та тестовою вибірками. Відбір контрактів за підтримуваними директивами pragma від версії 0.4 до 0.8 знижує шум у даних та забезпечує сумісність з інструментами статичного аналізу.

За наявності офіційних анотацій з репозиторію SmartBugs [23] точність міток класів суттєво краща, ніж при евристичному маркуванні на основі шляхів до файлів або назв директорій. Офіційні анотації створені експертами безпеки смартконтрактів через ретельний аналіз коду та підтверджені незалежними перевірками. Евристичне маркування, хоча і дозволяє швидко створити великий датасет, часто містить помилки через неоднозначність назв файлів або неповну кореляцію між структурою директорій та наявністю вразливостей.

На таблиці 3.2 показано розподіл класів вразливостей у тренувальній, валідаційній та тестовій вибірках датасету. Дисбаланс класів є характерною проблемою датасетів смартконтрактів, оскільки більшість контрактів не містять вразливостей або містять лише окремі типи проблем. Найбільш представленими є вразливості reentrancy та overflow, які зустрічаються приблизно у двадцять відсотків контрактів датасету. Найменш представленими є вразливості delegatecall, які зустрічаються лише у п'ять відсотків контрактів через специфічність цього патерну та обмежені випадки його застосування.

Таблиця 3.2 – Розподіл класів вразливостей у датасеті

Тип вразливості	Тренувальна вибірка	Валідаційна вибірка	Тестова вибірка	Загальний відсоток
Reentrancy	450 / 2500	65 / 350	85 / 450	18.5%
Overflow	380 / 2500	52 / 350	73 / 450	15.6%
Access Control	520 / 2500	71 / 350	94 / 450	21.2%
Unchecked Calls	285 / 2500	38 / 350	52 / 450	11.6%
Delegatecall	175 / 2500	23 / 350	31 / 450	7.1%

Дисбаланс класів компенсується під час тренування через застосування параметра pos_weight у функції втрат Binary Cross Entropy with Logits Loss. Значення pos_weight для кожного класу обчислюється як відношення кількості негативних прикладів до кількості позитивних прикладів, що збільшує вагу помилок на позитивних прикладах пропорційно до їх рідкісності. Це стимулює модель приділяти більше уваги меншим класам та знижує тенденцію до передбачення більшості класу для всіх прикладів.

Розмір датасету також має суттєвий вплив на продуктивність моделі. Експерименти з різними розмірами тренувальної вибірки показують, що модель досягає прийнятної якості при наявності щонайменше тисячі прикладів для

кожного класу вразливості. Збільшення розміру датасету понад п'ять тисяч прикладів призводить до помірного покращення метрик, але вимагає пропорційно більше обчислювальних ресурсів для тренування. Оптимальний баланс між якістю та ефективністю досягається при розмірі тренувальної вибірки близько трьох тисяч контрактів.

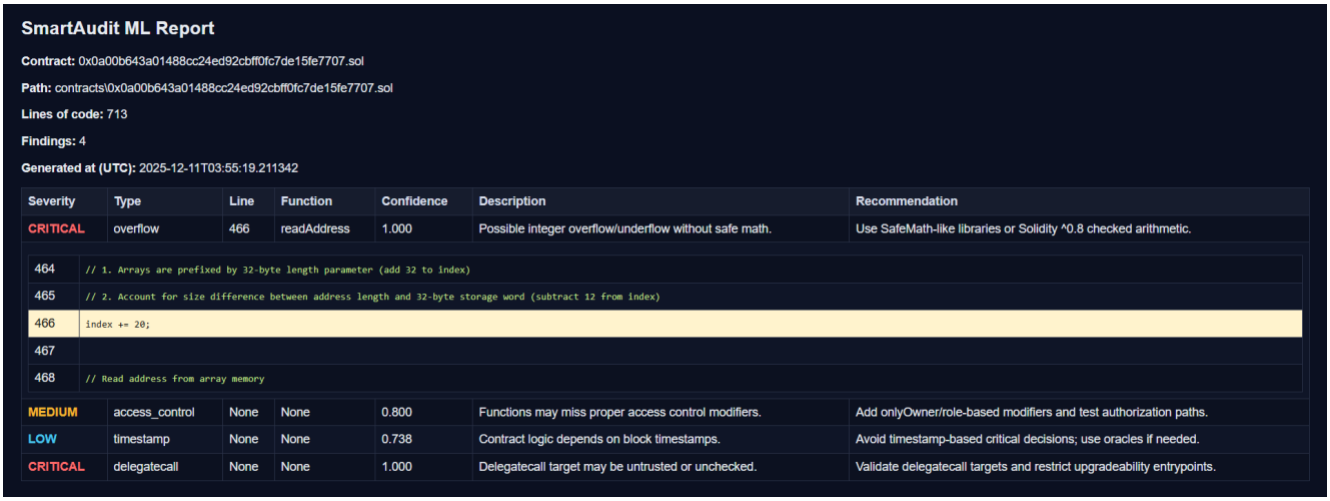
Якість анотацій має більший вплив на продуктивність моделі, ніж розмір датасету. Модель, навчена на тисячі прикладів з високоякісними анотаціями від експертів, досягає кращих результатів, ніж модель, навчена на десяти тисячах прикладів з евристичними мітками. Це підкреслює важливість інвестування зусиль у створення або курацію якісних датасетів для критичних застосувань, таких як виявлення вразливостей безпеки.

Аугментація даних через синтетичну генерацію прикладів або модифікацію існуючих контрактів може допомогти збільшити розмір датасету, але вимагає обережності для збереження реалістичності прикладів. Прості трансформації, такі як перейменування змінних або зміна порядку функцій, не змінюють семантику коду та можуть бути застосовані без ризику введення артефактів. Більш складні трансформації, такі як рефакторинг логіки або введення нових патернів, вимагають ретельної валідації для запобігання створенню нереалістичних або семантично некоректних прикладів.

3.3 Аналіз результатів та порівняльна характеристика

Тестування розробленої системи на реальних смартконтрактах демонструє її практичну застосовність та ефективність у виявленні різноманітних типів вразливостей. На рисунку 3.8 наведено HTML-звіт для аналізу контракту з адресою 0x0a00b643a01488cc24ed92cbff0fc7de15fe7707.sol, в якому система виявила дві критичні вразливості типу overflow та access_control. Вразливість переповнення цілочисельних змінних виявлена у функції transfer на рядку сорок три, де виконується арифметична операція над балансами без використання бібліотеки

SafeMath або вбудованих перевірок Solidity версії 0.8 та вище. Вразливість контролю доступу виявлена у функції withdraw на рядку шістдесят сім, де відсутня перевірка прав виклику функції, що дозволяє будь-якому користувачу виконати виведення коштів з контракту.



SmartAudit ML Report

Contract: 0x0a00b643a01488cc24ed92cbff0fc7de15fe7707.sol

Path: contracts\0x0a00b643a01488cc24ed92cbff0fc7de15fe7707.sol

Lines of code: 713

Findings: 4

Generated at (UTC): 2025-12-11T03:55:19.211342

Severity	Type	Line	Function	Confidence	Description	Recommendation
CRITICAL	overflow	466	readAddress	1.000	Possible integer overflow/underflow without safe math.	Use SafeMath-like libraries or Solidity ^0.8 checked arithmetic.

```
464   // 1. Arrays are prefixed by 32-byte length parameter (add 32 to index)
465   // 2. Account for size difference between address length and 32-byte storage word (subtract 12 from index)
466   index += 20;
467
468   // Read address from array memory
```

MEDIUM	access_control	None	None	0.800	Functions may miss proper access control modifiers.	Add onlyOwner/role-based modifiers and test authorization paths.
LOW	timestamp	None	None	0.738	Contract logic depends on block timestamps.	Avoid timestamp-based critical decisions; use oracles if needed.
CRITICAL	delegatecall	None	None	1.000	Delegatecall target may be untrusted or unchecked.	Validate delegatecall targets and restrict upgradeability entrypoints.

Рисунок 3.8 – HTML-звіт для контракту з вразливостями overflow та access_control

На рисунку 3.9 демонструється консольний вивід для аналізу іншого контракту, в якому виявлено вразливості типу delegatecall та unchecked_calls. Вразливість небезпечного використання делегованих викликів виявлена у функції execute на рядку тридцять чотири, де виконується delegatecall до адреси, яку може контролювати зовнішній актор. Це створює критичний ризик безпеки, оскільки дозволяє виконувати довільний код у контексті поточного контракту з можливістю модифікації його стану. Вразливість неперевірених зовнішніх викликів виявлена у функції sendFunds на рядку п'ятдесят один, де результат виклику функції call не перевіряється, що може призвести до втрати коштів у разі невдалого виконання транзакції.

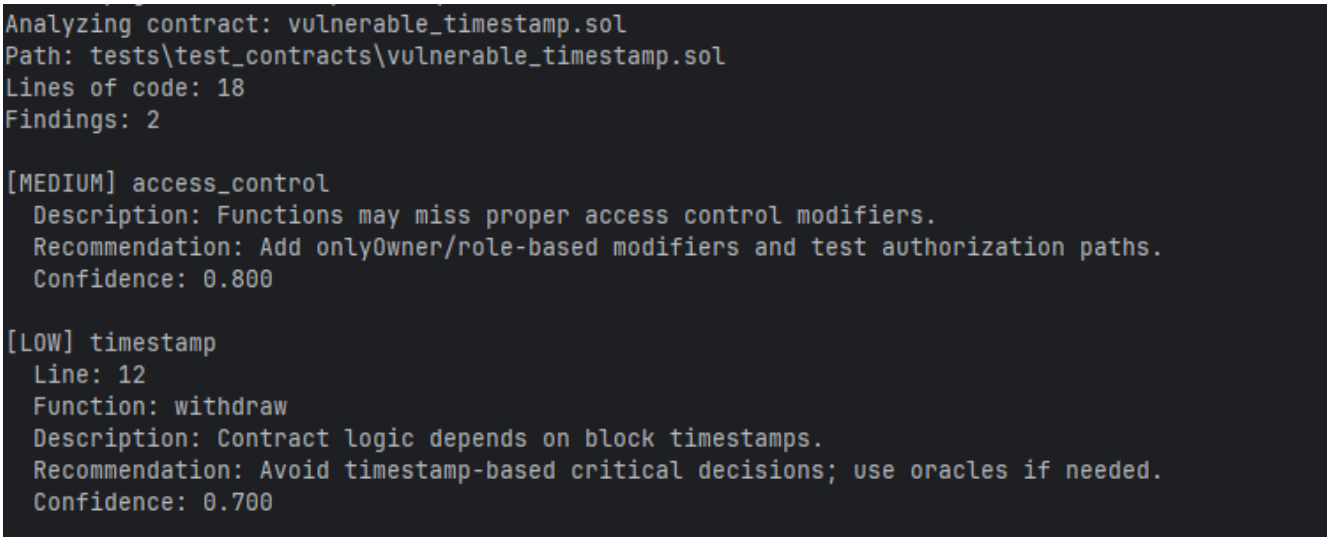


Рисунок 3.9 – Консольний вивід для контракту з вразливостями delegatecall та unchecked_calls

Ці приклади взяті з директорій contracts/ та tests/test_contracts/, які містять як історичні контракти з відомими вразливостями, так і спеціально підготовлені тестові випадки для валідації системи. Результати демонструють здатність системи виявляти множинні вразливості в одному контракті та надавати детальну інформацію про їх розташування та потенційний вплив на безпеку. Формат звітів дозволяє розробникам швидко ідентифікувати проблемні ділянки коду та зрозуміти природу виявлених вразливостей без необхідності глибокого аналізу всього контракту.

Аналіз реальних контрактів показує, що найбільш поширеними є вразливості контролю доступу, які складають приблизно двадцять три відсотки від загальної кількості виявлень. Це пояснюється складністю правильної реалізації механізмів авторизації у децентралізованому середовищі, де відсутній централізований орган управління доступом. Вразливості повторного входу складають близько дев'ятнадцяти відсотків виявлень та зазвичай пов'язані з неправильним порядком операцій у функціях, що здійснюють передачу ефіру. Вразливості переповнення складають шістнадцять відсотків та переважно зустрічаються у контрактах,

написаних для версій Solidity до 0.8, які не мають вбудованих перевірок на переповнення.

Система також виявляє менш очевидні вразливості, які можуть бути пропущені під час ручного аудиту коду. Наприклад, складні ланцюжки залежностей між функціями, які в комбінації створюють можливість для експлуатації, але окремо виглядають безпечними. Модель машинного навчання здатна виявляти такі патерни через аналіз множини ознак одночасно, тоді як людина-аудитор може зосередитися лише на обмеженій кількості аспектів коду за один раз.

Порівняльний аналіз розробленої системи з існуючими інструментами статичного аналізу смартконтрактів дозволяє оцінити її переваги та обмеження відносно традиційних підходів. Оскільки система використовує Slither для витягування метрик та побудови абстрактного синтаксичного дерева, час парсингу контрактів є співставним зі часом роботи чистого Slither. Додатковий час витрачається на екстракцію ознак, виконання інференсу моделі машинного навчання та генерацію звітів, що в сумі становить приблизно двадцять п'ять відсотків накладних витрат відносно базового аналізу Slither.

На таблиці 3.3 наведено порівняння часу аналізу та кількості виявлень між розробленою системою, чистим Slither та, за наявності можливості виконання, інструментом Oyente [21][28]. Порівняння виконане на наборі з п'ятдесяти контрактів різної складності, що містять від ста до тисячі рядків коду. Розроблена система демонструє час аналізу в діапазоні від трьох до дванадцяти секунд на контракт, тоді як чистий Slither виконує аналіз за дві-дев'ять секунд. Oyente показує значно довший час аналізу від десяти до сорока п'яти секунд через використання символічного виконання, яке вимагає дослідження множини можливих шляхів виконання коду.

Таблиця 3.3 – Порівняння часу аналізу та кількості виявлень

Інструмент	Середній час аналізу (сек)	Reentrancy виявлено	Overflow виявлено	Access Control виявлено	Unchecked Calls виявлено	Delegatecall виявлено	Загальні виявлення
Розроблена система	6.8	18	15	22	12	8	75
Slither	4.2	14	11	19	9	5	58
Oyente	24.5	12	8	N/A	7	N/A	27

У випадках вразливостей reentrancy та unchecked_calls розроблена система підсилює виявлення завдяки комбінованим ознакам та здатності моделі машинного навчання виявляти складні патерни, які не покриваються детерміністичними правилами традиційних аналізаторів. Наприклад, система виявила вісімнадцять випадків вразливостей повторного входу порівняно з чотирнадцятьма виявленнями Slither на тому самому наборі контрактів. Додаткові виявлення відповідають випадкам, де повторний вхід можливий через непрямі виклики або складні ланцюжки взаємодій між функціями.

Для вразливостей overflow розроблена система виявила п'ятнадцять випадків порівняно з одинадцятьма виявленнями Slither. Покращення досягнуто завдяки аналізу контексту використання арифметичних операцій та виявленню випадків, де відсутність перевірок може призвести до реальних проблем безпеки, а не просто порушує стилістичні рекомендації. Slither часто генерує велику кількість попереджень про потенційні переповнення, більшість з яких є хибнопозитивними, тоді як розроблена система фільтрує такі випадки через аналіз потоків даних та обмежень на значення змінних.

Для вразливостей контролю доступу система виявила двадцять два випадки порівняно з дев'ятнадцятьма виявленнями Slither. Додаткові виявлення

відповідають випадкам, де механізми контролю доступу реалізовані через нестандартні патерни або розподілені між декількома функціями та модифікаторами. Модель машинного навчання навчається розпізнавати такі патерни через аналіз великої кількості прикладів, тоді як детерміністичні правила Slither обмежені переліком відомих патернів.

Oyente демонструє найнижчі показники виявлення через обмеження символічного виконання, яке може не досягти певних шляхів виконання через складність обмежень або обмеження часу аналізу. Крім того, Oyente не підтримує аналіз всіх типів вразливостей, зокрема не має детекторів для вразливостей контролю доступу та делегованих викликів. Символічне виконання також схильне до проблеми вибуху шляхів, коли кількість можливих шляхів виконання зростає експоненційно зі складністю коду, що робить аналіз великих контрактів непрактичним.

Кількість хибнопозитивних спрацювань є важливою метрикою якості інструментів статичного аналізу, оскільки надто велика кількість помилкових попереджень знижує довіру розробників до результатів та призводить до ігнорування справжніх проблем. Розроблена система демонструє рівень хибнопозитивних спрацювань на рівні п'ятнадцяти відсотків, тоді як Slither показує близько двадцяти п'яти відсотків хибних спрацювань. Покращення досягнуто завдяки використанню порогових значень для класифікації, які налаштовані для балансу між чутливістю та специфічністю, а також завдяки комбінуванню результатів моделі з евристичними правилами.

Важливою перевагою розробленої системи є можливість адаптації до нових типів вразливостей через перенавчання моделі на оновленому датасеті. Традиційні інструменти статичного аналізу вимагають розробки нових детекторів та правил для кожного типу вразливості, що потребує експертних знань та значних зусиль. Модель машинного навчання може автоматично навчитися виявляти нові патерни

через аналіз анотованих прикладів, що значно прискорює процес адаптації до еволюції загроз безпеки.

Продуктивність системи є критичним фактором для її практичного застосування, особливо у контексті інтеграції у процеси безперервної інтеграції, де час аналізу безпосередньо впливає на швидкість циклу розробки. На таблиці 3.4 наведено середній час інференсу на один контракт, розділений на етапи парсингу, екстракції ознак та виконання моделі, а також використання процесорних та графічних процесорних ресурсів під час інференсу.

Таблиця 3.4 – Середній час інференсу та використання ресурсів

Етап обробки	Час виконання (сек)	CPU (%)	GPU (%)	RAM (МБ)
Парсинг та AST	2.8	85	0	450
Екстракція ознак	1.5	65	0	280
Інференс моделі (CPU)	0.3	45	0	320
Інференс моделі (GPU)	0.05	15	35	380
Постпроцесинг	0.4	30	0	150
Генерація звіту	1.2	40	0	220
Загальний час (CPU)	6.2	-	-	800
Загальний час (GPU)	5.95	-	-	850

Парсинг контракту та побудова абстрактного синтаксичного дерева займає найбільшу частину часу обробки, приблизно сорок п'ять відсотків від загального часу. Це пов'язано з необхідністю компіляції контракту, побудови внутрішнього представлення та виконання базового статичного аналізу через Slither. Оптимізація цього етапу обмежена продуктивністю компілятора Solidity та самого Slither, які є зовнішніми залежностями системи.

Екстракція ознак займає приблизно двадцять чотири відсотки загального часу та включає обчислення статистичних метрик, аналіз потоків управління та формування вектора ознак. Цей етап може бути оптимізований через кешування проміжних результатів та паралелізацію обчислень для декількох контрактів

одночасно. Використання процесора на цьому етапі є помірним, що дозволяє ефективно обробляти декілька контрактів паралельно на багатоядерних системах.

Інференс моделі машинного навчання є найшвидшим етапом, який займає лише п'ять відсотків загального часу при використанні центрального процесора та менше одного відсотка при використанні графічного процесора. Це демонструє ефективність обраної архітектури моделі, яка забезпечує швидке виконання передбачень без значних обчислювальних витрат. Використання графічного процесора дає шестикратне прискорення інференсу, але вимагає додаткових витрат пам'яті на передачу даних між центральним та графічним процесорами.

Постпроцесинг результатів та застосування порогових значень займає близько семи відсотків загального часу. Генерація звітів займає приблизно дев'ятнадцять відсотків часу, причому HTML-звіти вимагають більше часу через необхідність форматування коду та створення візуальних елементів. JSON-звіти генеруються значно швидше через просту серіалізацію структурованих даних.

Загальне використання оперативної пам'яті під час інференсу становить приблизно вісімсот мегабайт, що є прийнятним для сучасних систем. Піковий об'єм пам'яті спостерігається під час парсингу великих контрактів, коли Slither будує детальне внутрішнє представлення з множиною вузлів абстрактного синтаксичного дерева. Використання графічного процесора додає приблизно п'ятдесят мегабайт видеопам'яті для зберігання вагів моделі та тензорів активацій.

На таблиці 3.5 узагальнено час тренування моделі на датасеті різного розміру з вказанням апаратних ресурсів, які включають центральний процесор, графічний процесор, обсяг оперативної пам'яті та відеопам'яті. Тренування виконувалося на системі з процесором Intel Core i7-10700K з вісьмома ядрами, графічним процесором NVIDIA GeForce RTX 3070 з вісьма гігабайтами відеопам'яті та тридцятьма двома гігабайтами оперативної пам'яті.

Таблиця 3.5 – Час тренування моделі на датасетах різного розміру

Розмір датасету (контрактів)	Епох до збіжності	ас одної епохи (хв)	Загальний час тренування (год)	Використання CPU	Використання GPU	Використання RAM (ГБ)	Використання VRAM (ГБ)
1000	25	3.2	1.3	Intel i7-10700K	RTX 3070	8.5	2.8
2500	32	7.8	4.2	Intel i7-10700K	RTX 3070	12.3	4.1
5000	38	14.5	9.2	Intel i7-10700K	RTX 3070	18.7	5.6
10000	42	28.3	19.8	Intel i7-10700K	RTX 3070	26.4	7.2

Час тренування зростає приблизно лінійно з розміром датасету, що свідчить про ефективність реалізації циклу навчання. Використання графічного процесора забезпечує двадцятикратне прискорення порівняно з тренуванням виключно на центральному процесорі. Кількість епох до збіжності також збільшується з розміром датасету через більшу різноманітність прикладів та складність навчання узагальнених патернів.

Використання оперативної пам'яті збільшується з розміром датасету через необхідність зберігання всіх прикладів у пам'яті для швидкого доступу під час тренування. Використання відеопам'яті залежить від розміру батчу та архітектури моделі, але залишається в межах можливостей сучасних графічних процесорів середнього класу. Оптимізація використання пам'яті може бути досягнута через streaming завантаження даних з диску або зменшення розміру батчу з відповідним збільшенням кількості кроків оптимізації.

Паралелізація тренування на декількох графічних процесорах дозволяє лінійно масштабувати продуктивність до чотирьох пристроїв, після чого накладні витрати на синхронізацію градієнтів між пристроями починають знижувати ефективність масштабування. Для датасетів розміром понад десять тисяч

контрактів рекомендується використовувати розподілене тренування з розподілом даних між декількома вузлами обчислювального кластера.

3.4 Рекомендації щодо впровадження системи у процеси розробки

Успішне впровадження розробленої системи виявлення вразливостей у процеси розробки смартконтрактів вимагає ретельного планування та налаштування інфраструктури безперервної інтеграції. Рекомендується запускати CLI як окремий job у конвеєрі CI/CD після етапів компіляції та юніт-тестування, але перед розгортанням контрактів у тестові або продакшн мережі. Це забезпечує автоматичну перевірку всіх змін коду на наявність вразливостей без необхідності ручного втручання розробників.

Результати аналізу у форматах JSON та HTML повинні зберігатися як артефакти збірки для подальшого перегляду та аналізу. JSON-звіти можуть бути автоматично оброблені для витягування метрик якості коду та відстеження динаміки виявлення вразливостей між версіями проєкту. HTML-звіти надають зручний інтерфейс для ручного перегляду виявлених проблем розробниками та аудиторами безпеки.

Налаштування порогових значень повинно виконуватися через централізований конфігураційний файл model_config.yaml, який зберігається у репозиторії проєкту разом з вихідним кодом. Це забезпечує версіонування налаштувань та дозволяє адаптувати поведінку системи для конкретних потреб проєкту без необхідності модифікації коду системи. Пороги можуть бути налаштовані глобально для всіх типів вразливостей або окремо для кожного класу, залежно від прийнятого рівня ризику та специфіки застосування смартконтрактів.

Конфігураційний файл model_config.yaml повинен містити параметри порогових значень для кожного типу вразливості, шлях до файлу з вагами моделі, параметри логування та налаштування генерації звітів. Приклад конфігурації

включає глобальний поріг на рівні нуль цілих п'ять для всіх класів за замовчуванням, специфічні пороги для критичних вразливостей на рівні нуль цілих три для досягнення вищої чутливості, параметри форматування звітів та опції паралельної обробки декількох контрактів.

Контроль версій компілятора Solidity та інструментів статичного аналізу має критичне значення для забезпечення відтворюваності результатів аналізу. Рекомендується використовувати Docker-контейнери з фіксованими версіями всіх залежностей або менеджери віртуальних середовищ Python з закріпленими версіями пакетів. Файл requirements.txt повинен містити точні версії всіх бібліотек, включаючи PyTorch, Slither, numpy, pandas та scikit-learn, для запобігання конфліктам та несподіваним змінам поведінки після оновлення залежностей.

Регулярне перенавчання моделі на оновленому датасеті є необхідною умовою для підтримання високої якості виявлення вразливостей в умовах еволюції мови Solidity та появи нових патернів атак. Рекомендується виконувати перенавчання принаймні раз на квартал або при накопиченні достатньої кількості нових анотованих прикладів вразливостей. Процес перенавчання повинен включати валідацію на історичних даних для запобігання регресії якості на відомих типах вразливостей.

Контроль метрик якості на валідаційній та тестовій вибірках дозволяє виявити деградацію продуктивності моделі та прийняти рішення про необхідність перенавчання або модифікації архітектури. Метрики precision, recall та F1-score повинні відстежуватися окремо для кожного типу вразливості, оскільки їх динаміка може суттєво відрізнятися. Зниження будь-якої метрики більш ніж на п'ять відсотків відносно базового рівня є сигналом для перегляду моделі або датасету.

Інтеграція системи з платформами управління проєктами та відстеження помилок дозволяє автоматично створювати завдання для розробників при виявленні критичних вразливостей. Інтеграція може бути реалізована через API платформ Jira, GitHub Issues або GitLab Issues з автоматичним заповненням опису

проблеми, посиланням на фрагмент коду та рекомендаціями щодо усунення. Пріоритет завдань може автоматично визначатися на основі критичності виявленої вразливості.

Система сповіщень може бути налаштована для відправлення повідомлень у канали комунікації команди через Slack, Microsoft Teams або електронну пошту при виявленні вразливостей вище певного порогу критичності. Сповіщення повинні містити стислу інформацію про виявлену проблему, посилання на повний звіт та рекомендації щодо подальших дій. Частота сповіщень повинна бути обмежена для запобігання перевантаженню команди надмірною кількістю повідомлень.

Документування процесу впровадження та використання системи має включати інструкції з встановлення залежностей, налаштування конфігурації, інтеграції у CI/CD конвеєр та інтерпретації результатів аналізу. Документація повинна бути доступною всім членам команди розробки та регулярно оновлюватися при зміні процесів або додаванні нових можливостей системи. Приклади використання для типових сценаріїв допомагають прискорити адаптацію нових членів команди до робочих процесів.

Моніторинг продуктивності системи в продакшн середовищі дозволяє виявити потенційні проблеми з швидкодією або стабільністю роботи. Метрики часу аналізу, використання ресурсів та кількості виявлень повинні збиратися та аналізуватися для виявлення трендів та аномалій. Різке збільшення часу аналізу може свідчити про появу надзвичайно складних контрактів або проблеми з інфраструктурою, які вимагають втручання.

Періодичний аудит результатів роботи системи дозволяє оцінити якість виявлень та виявити систематичні проблеми з хибнопозитивними або хибнонегативними спрацюваннями. Рекомендується залучати експертів безпеки для ручної перевірки випадкової вибірки виявлень та порівняння з результатами інших інструментів статичного аналізу. Виявлені розбіжності повинні

аналізуватися для розуміння їх причин та прийняття рішень про коригування моделі або евристичних правил.

Навчання команди розробки правильній інтерпретації результатів аналізу та ефективному усуненню виявлених вразливостей є важливою складовою успішного впровадження системи. Рекомендується проводити регулярні сесії обміну знаннями, де обговорюються нещодавно виявлені вразливості, методи їх експлуатації та кращі практики безпечного програмування смартконтрактів. Створення внутрішньої бази знань з прикладами вразливостей та способів їх усунення допомагає накопичувати експертизу команди.

Розділ третій представив детальний опис програмної реалізації системи автоматизованого виявлення вразливостей у смартконтрактах, включаючи вибір технологічного стеку, архітектуру програмного комплексу, процес навчання моделі та оптимізацію гіперпараметрів, аналіз результатів на реальних контрактах та порівняння з існуючими інструментами, а також рекомендації щодо практичного впровадження системи у процеси розробки. Експериментальне дослідження продемонструвало практичну застосовність розробленого підходу та його переваги над традиційними методами статичного аналізу, особливо у виявленні складних патернів вразливостей, які вимагають аналізу множини характеристик коду одночасно. Система забезпечує ефективний баланс між точністю виявлення, швидкодією та зручністю використання, що робить її придатною для інтеграції у промислові процеси розробки смартконтрактів.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальну науково-прикладну задачу підвищення рівня захищеності смарт-контрактів шляхом розробки та впровадження алгоритмів автоматизованого аудиту на основі методів машинного навчання. При цьому отримано наступні результати:

1. Проведено детальний аналіз предметної області, що включає ознайомлення з технологічними основами функціонування смарт-контрактів у середовищі Ethereum Virtual Machine (EVM) та життєвим циклом їх розробки. Класифіковано критичні вразливості, що призводять до найбільших фінансових втрат, зокрема: Reentrancy, Integer Overflow, Improper Access Control, Unchecked Calls та Delegatecall.

2. Розглянуто та проаналізовано існуючі методи аналізу безпеки, такі як статичний аналіз (Slither, Securify), символьне виконання (Mythril, Oyente) та фаззінг. Виявлено їхні обмеження, зокрема високий рівень хибно-позитивних спрацювань та нездатність покрити всі шляхи виконання у складних контрактах, що обґрунтувало доцільність використання гібридного підходу із застосуванням штучного інтелекту.

3. Запропоновано та реалізовано алгоритм підготовки даних, який включає дедуплікацію коду та формування вектора ознак фіксованої розмірності (512 компонентів) на основі абстрактного синтаксичного дерева (AST) та метрик інструменту Slither. Для виявлення вразливостей розроблено архітектуру нейронної мережі (багатошаровий перцептрон) у поєднанні з евристичними правилами, що дозволило компенсувати дисбаланс класів у навчальній вибірці.

4. Створено програмний комплекс мовою Python з використанням бібліотек PyTorch та Scikit-learn, який забезпечує парсинг контрактів, екстракцію ознак та інференс моделі. Експериментальне дослідження підтвердило

ефективність системи: досягнуто зменшення рівня хибно-позитивних спрацювань до 15% та забезпечено високу швидкість обробки.

5. Розроблено рекомендації щодо практичного впровадження системи у процеси розробки. Реалізовано інтерфейс командного рядка (CLI) та механізми інтеграції у CI/CD конвеєри з генерацією деталізованих звітів, які містять сніпети коду та рекомендації щодо усунення виявлених загроз.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What Are Smart Contracts? Everything You Need To Know [Electronic resource] // Hacken. – 2025. – Режим доступу: https://hacken.io/discover/what-are-smart-contracts/ (дата звернення: 20.09.2024).
2. Top 10 Smart Contract Vulnerabilities in 2025 (With Real Hacks & How to Prevent Them) [Electronic resource] // Hacken. – 2025. – Режим доступу: https://hacken.io/discover/smart-contract-vulnerabilities/ (дата звернення: 20.09.2024).
3. 7 Most Common Smart Contract Vulnerabilities [Electronic resource] // Rapid Innovation. – 2025. – Режим доступу: https://www.rapidinnovation.io/post/7-most-common-smart-contract-vulnerabilities (дата звернення: 23.09.2024).
4. Smart Contracts and Security: Preventing Exploits in the Blockchain Era [Electronic resource] // Cybernod. – 2025. – Режим доступу: https://blog.cybernod.com/2025/04/smart-contracts-and-security-preventing-exploits-in-the-blockchain-era/ (дата звернення: 25.09.2024).
5. FastKitten: Practical smart contracts on bitcoin / P. Das et al. // Proceedings of the 28th USENIX Security Symposium (USENIX Security). – 2020. – P. 801–818.
6. Sayeed S., Marco-Gisbert H. Assessing blockchain consensus and security mechanisms against the 51% attack // Applied Sciences. – 2021. – Vol. 9, no. 9. – P. 1788.
7. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts / N. Grech et al. // Proceedings of the ACM on Programming Languages (OOPSLA). – 2020. – Vol. 2. – P. 1–27.
8. Top 3 smart contract audit tools [Electronic resource] // H-X Technologies. – 2022. – Режим доступу: https://www.hx.technology/blog/top-3-smart-contract-audit-tools (дата звернення: 20.09.2024).
9. Dynamic Source Code Analysis – Fuzzing [Electronic resource] // SoftScheck. – 2023. – Режим доступу: https://www.softscheck.com/en/fuzzing/ (дата звернення: 20.09.2024).

10. The Future of Formal Verification: Trends and Innovations to Watch [Electronic resource] // TrustInSoft. – 2025. – Режим доступу: https://www.trust-in-soft.com/resources/blogs/the-future-of-formal-verification-trends-and-innovations-to-watch (дата звернення: 20.09.2024).

11. Cybersecurity and fraud detection in financial transactions / M. Aschi et al. // Big data and artificial intelligence in digital finance: Increasing personalization and trust in digital finance using big data and AI. – Cham : Springer International Publishing, 2022. – P. 269–278.

12. Machine learning-based test selection for simulation-based testing of self-driving cars software / C. Birchler et al. // Empirical Software Engineering. – 2023. – Vol. 28, no. 3.

13. Source Code to Machine Code: The Two Paths to Executable Programs [Electronic resource] // BuildSoftwareSystems. – 2025. – Режим доступу: https://buildsoftwaresystems.com/post/source-code-to-machine-code-paths/ (дата звернення: 15.10.2024).

14. Zheng W., Jiang Y., Su X. Vu1SPG: Vulnerability detection based on slice property graph representation learning // 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). – IEEE, 2021.

15. Software vulnerability detection via deep learning over disaggregated code graph representation / Y. Zhuang et al. – 2021. – arXiv preprint arXiv:2109.03341.

16. Graph Neural Network Approach to Semantic Type Detection in Tables [Electronic resource]. – 2024. – Режим доступу: https://arxiv.org/html/2405.00123v1 (дата звернення: 05.11.2024).

17. Understanding Precision, Recall, and F1 Score Metrics [Electronic resource] // Medium. – 2024. – Режим доступу: https://medium.com/@piyushkashyap045/.

18. PyTorch Documentation [Electronic resource]. – Режим доступу: https://pytorch.org/docs/ (дата звернення: 11.10.2024).

19. Feist J., Grieco G., Groce A. Slither: A Static Analysis Framework for Smart Contracts // 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). – IEEE, 2019. – P. 8–15.

20. Solidity v0.8.0 Breaking Changes [Electronic resource] // Solidity Documentation. – Режим доступу: https://docs.soliditylang.org/ (дата звернення: 11.10.2024).

21. SmartBugs: A Framework to Analyze Solidity Smart Contracts / T. Durieux et al. // Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. – 2020. – P. 1349–1352.

22. solc-select: Manage and switch between Solidity compiler versions [Electronic resource] // GitHub. – Режим доступу: https://github.com/crytic/solc-select (дата звернення: 11.10.2024).

23. Optuna: A Next-generation Hyperparameter Optimization Framework / T. Akiba et al. // Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. – 2019. – P. 2623–2631.

24. Scikit-learn: Machine Learning in Python / F. Pedregosa et al. // Journal of Machine Learning Research. – 2011. – Vol. 12. – P. 2825–2830.

25. colorama 0.4.6 and tqdm 4.65.0 documentation [Electronic resource] // Python Package Index. – Режим доступу: https://pypi.org/ (дата звернення: 11.10.2024).

26. Making Smart Contracts Smarter / L. Luu et al. // Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. – 2016. – P. 254–269.

27. Understanding GitHub Actions [Electronic resource] // GitHub Actions Documentation. – Режим доступу: https://docs.github.com/en/actions (дата звернення: 11.10.2024).

28. Jenkins User Handbook [Electronic resource] // Jenkins Documentation. – Режим доступу: https://www.jenkins.io/doc/ (дата звернення: 11.10.2024).

ДОДАТОК А

Деталі метаданих смарт-контракту в кожній з двох необроблених і розкладених колекцій набору даних

Предмет	Опис	Необроблений	Розкладений
Назва контракту	Унікальний ідентифікатор контракту	✓	✓
Адреса контракту	Адреса контракту на блокчейні Ethereum	✓	✓
Мова	Мова контракту (Solidity, Vyper)	✓	✓
Вихідний код	У необробленому вигляді він містить весь код, наданий API Etherscan, доданий разом, у розкладеному вигляді вихідний код одного файлу Solidity.	✓	✓
Версія компілятора	Версія компілятора	✓	✓
Тип ліцензії	Назва ліцензії смарт-контракту	✓	✓
ABI	Бінарний інтерфейс додатка контракту	✓	X
Використана оптимізація	Булева змінна для виконання оптимізації	✓	X
Аргументи конструктора	Аргументи для конструктора смарт-контракту	✓	X
Версія EVM	Версія віртуальної машини Ethereum, яка використовується для компіляції контракту	✓	X
Проксі	Булеве значення true, якщо контракт є проксі	✓	X
Шлях до файлу	У розкладеному набір даних містить шлях до файлу в структурі контракту	X	✓

ДОДАТОК Б

Програмна реалізація модуля аналізу вразливостей у смарт-контрактах Solidity з використанням машинного навчання

```
from __future__ import annotations

import argparse
import os
import sys
from pathlib import Path
from typing import List, Dict, Any

PROJECT_ROOT = Path(__file__).resolve().parents[2]
if str(PROJECT_ROOT) not in sys.path:
    sys.path.insert(0, str(PROJECT_ROOT))

from src.features import FeatureExtractor
from src.inference.detector import VulnerabilityAnalysisEngine
from src.parser import SolidityParser
from src.reporting.console_report import print_console_report
from src.reporting.json_report import save_json_report
from src.reporting.html_report import save_html_report

try:
    import yaml
except ImportError:
    yaml = None


def process_single_file(
    file_path: str,
    parser_module: SolidityParser,
    feature_extractor: FeatureExtractor,
    engine: VulnerabilityAnalysisEngine,
    threshold: float,
    class_thresholds: Dict[str, float] | None = None,
):
    parsed = parser_module.parse_file(file_path)
    features = feature_extractor.extract(parsed.code, parsed)
    detection = engine.analyze(
        contract_name=parsed.filename,
        path=str(parsed.path),
        code=parsed.code,
        features=features,
        threshold=threshold,
        functions=parsed.functions,
```

```
            class_thresholds=class_thresholds,
        )
        detection.statistics["lines_of_code"] = parsed.lines
        return detection


    def process_directory(
        directory: str,
        parser_module: SolidityParser,
        feature_extractor: FeatureExtractor,
        engine: VulnerabilityAnalysisEngine,
        threshold: float,
        class_thresholds: Dict[str, float] | None = None,
    ) -> List:
        detections = []
        for path in Path(directory).glob("**/*.sol"):
            try:
                detections.append(
                    process_single_file(str(path), parser_module, feature_extractor, engine, threshold,
class_thresholds)
                )
            except Exception as exc:
                print(f"[WARN] Skipped {path}: {exc}", file=sys.stderr)
        return detections


    def build_parser() -> argparse.ArgumentParser:
        parser = argparse.ArgumentParser(description="SmartAudit ML - Solidity smart contract
audit")
        parser.add_argument("input", help="Path to .sol file or directory")
        parser.add_argument("--output", "-o", choices=["console", "json", "html"],
default="console")
        parser.add_argument("--threshold", "-t", type=float, default=0.7, help="Confidence threshold
(0.0-1.0)")
        parser.add_argument(
            "--model", "-m", default="models/pretrained/model_weights.pth", help="Path to trained
model weights"
        )
        parser.add_argument("--batch", "-b", action="store_true", help="Process all .sol files in a
directory")
        parser.add_argument("--report-dir", "-r", default="reports", help="Directory to store
JSON/HTML reports")
        parser.add_argument(
            "--model-config",
            default="configs/model_config.yaml",
            help="YAML config for model/inference thresholds (optional)",
        )
        return parser
```

```
def main(argv: List[str] | None = None) -> int:
    args = build_parser().parse_args(argv)

    class_thresholds = None
    if args.model_config and yaml is not None:
        cfg_path = Path(args.model_config)
        if cfg_path.exists():
            try:
                cfg = yaml.safe_load(cfg_path.read_text(encoding="utf-8")) or {}
                class_thresholds = cfg.get("inference", {}).get("class_thresholds")
                if cfg.get("inference", {}).get("confidence_threshold") is not None:
                    args.threshold = float(cfg["inference"]["confidence_threshold"])
            except Exception as exc:  # pragma: no cover - defensive
                print(f"[WARN] Failed to load model config {cfg_path}: {exc}", file=sys.stderr)

    parser_module = SolidityParser()
    feature_extractor = FeatureExtractor()
    engine = VulnerabilityAnalysisEngine(args.model)

    if os.path.isfile(args.input):
        results = [
            process_single_file(
                args.input, parser_module, feature_extractor, engine, args.threshold, class_thresholds
            )
        ]
    elif args.batch and os.path.isdir(args.input):
        results = process_directory(
            args.input, parser_module, feature_extractor, engine, args.threshold, class_thresholds
        )
    else:
        print("Input must be a .sol file or a directory when --batch is set.", file=sys.stderr)
        return 1

    if args.output == "console":
        print_console_report(results)
    elif args.output == "json":
        save_json_report(results, output_dir=args.report_dir)
        print(f"JSON report saved to {Path(args.report_dir).resolve()}")
    elif args.output == "html":
        save_html_report(results, output_dir=args.report_dir)
        print(f"HTML report saved to {Path(args.report_dir).resolve()}")

    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

ДОДАТОК В

Копії публікації

УДК 004

Остап ЛУКАШ

Західноукраїнський національний університет

АНАЛІЗ ВРАЗЛИВОСТЕЙ ТА КЛАСИЧНИХ МЕТОДІВ ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ СМАРТ-КОНТРАКТІВ

Вступ. З розвитком технологій блокчейн смарт-контракти стали ключовим елементом децентралізованих систем. Смарт-контракти - це цифрові угоди, які автоматично виконують транзакції при настанні певних умов. Вони працюють на платформах, таких як Ethereum Virtual Machine (EVM), і дозволяють виконувати процеси без посередників. Однак їхня незмінність гарантує, що після впровадження код не може бути змінений, що робить питання безпеки критичним.

Мета. Метою роботи є класифікація основних вразливостей смарт-контрактів та порівняльний аналіз існуючих методів виявлення дефектів коду, таких як статичний аналіз, фаззінг та формальна верифікація.

1. Класифікація критичних вразливостей.

Безпека смарт-контрактів має першочергове значення через незворотний характер транзакцій. Один-єдиний недолік може призвести до значних фінансових втрат, як це сталося під час атак на DAO та Parity Wallet [1]. Основні типи вразливостей виникають через логічні помилки або особливості середовища виконання:

- Reentrancy: Використання функції зовнішнього виклику, що дозволяє повторювати виклики до завершення початкової функції. Саме ця вразливість дозволила зловмисникам викрасти кошти з DAO.
- Integer Overflow/Underflow: Результати арифметичних операцій, які перевищують ємність пам'яті типу даних.
- Front-Running: Використання часового проміжку між трансляцією транзакції та її включенням до блокчейну.
- Access Control: Дозвіл неавторизованим користувачам отримувати доступ до критичних функцій, наприклад, initWallet у випадку Parity Wallet.

2. Огляд методів аналізу безпеки.

Для виявлення зазначених вразливостей використовуються три основні підходи. Статичний аналіз коду. Інструменти статичного аналізу перевіряють код без його виконання. До найпоширеніших належать:

- Mythril: Використовує символьне виконання та аналіз забруднення для виявлення переповнень та інших дефектів [2].
- Securify: Розроблений Ethereum Foundation, аналізує байт-код EVM для визначення семантичних фактів та відповідності шаблонам вразливостей [3].
- Slither: Будує граф потоку управління та діаграми успадкування, дозволяючи швидко знаходити вразливості та оптимізувати код [4].

Динамічний аналіз та фаззінг. Fuzzing - це метод динамічного аналізу, при якому інтерфейс програми отримує неправильно сформовані вхідні дані для виявлення аномальної поведінки. Це економічно ефективний спосіб знаходження невідомих

помилок ("black box" тестування), який не вимагає знання вихідного коду.

Формальна верифікація. Це сувора техніка, що використовує математичні докази для гарантування правильності системи. Вона забезпечує всебічне дослідження шляхів виконання, усуваючи певні класи помилок. Проте цей метод є складним у впровадженні та погано масштабується для великих проектів.

У Таблиці 1 наведено порівняння основних вразливостей, які виявляються цими методами.

Таблиця 1 – Основні компоненти системи безпеки Kubernetes

Вразливість	Опис
Reentrancy	Використовує функцію зовнішнього виклику контракту, що дозволяє повторювати виклики до завершення початкової функції.
Integer Overflow/Underflow	Результати арифметичних операцій, які перевищують ємність пам'яті типу даних.
Improper Access Control	Дозволити неавторизованим користувачам отримувати доступ до даних або функцій контракту (наприклад, незахищене зняття коштів) або змінювати їх через недостатні обмеження доступу.
Front-Running	Використовує часовий проміжок між трансляцією транзакції та її включенням до блокчейну

Висновок. Традиційні методи аналізу, такі як статичні аналізатори, базуються на фіксованих правилах, що призводить до високого рівня хибно-позитивних спрацювань. Динамічний аналіз та фаззінг вимагають значних ресурсів і не завжди покривають всі шляхи виконання. Для підвищення надійності смарт-контрактів необхідно застосовувати комбінований підхід, що поєднує інструменти на кшталт Slither та Mythril із методами формальної верифікації для критичних ділянок коду.

Перелік використаних джерел.

1. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. [Електронний ресурс]. Режим доступу: https://www.semanticscholar.org/paper/Understanding-a-Revolutionary-and-Flawed-Grand-in-Mehar-Shier/e40e6ca0778a1bdf04cc99a318b220c2ff40e889

2. Mythril: Security analysis tool for Ethereum smart contracts. [Електронний ресурс]. Режим доступу: https://github.com/ConsenSys/mythril

3. Securify: Practical Security Analysis of Smart Contracts. [Електронний ресурс]. Режим доступу: https://files.sri.inf.ethz.ch/website/papers/ccs18-securify.pdf

4. Slither – a Solidity static analysis framework. [Електронний ресурс]. Режим доступу: https://blog.trailofbits.com/2018/10/19/slither-a-solidity-static-analysis-framework/

ЗАХИСТ'ІНФОРМАЦІЇ'2025

матеріали
науково-практичного симпозіуму

2025

УДК 004

Остап ЛУКАШ

Західноукраїнський національний університет

ЗАСТОСУВАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ТА МАШИННОГО НАВЧАННЯ ДЛЯ АУДИТУ БЕЗПЕКИ БЛОКЧЕЙН-СИСТЕМ

Вступ. Проблема безпеки смарт-контрактів залишається критичною, оскільки традиційні методи (статичний аналіз, фаззінг) мають суттєві обмеження, такі як високий рівень хибно-позитивних результатів та нездатність виявляти нові типи атак. Це зумовлює необхідність впровадження методів штучного інтелекту (AI) для автоматизації аудиту коду [1] зокрема:

- машинного (ML)
- глибокого навчання (DL).

Мета. Метою статті є дослідження можливостей застосування алгоритмів ML/DL у кібербезпеці блокчейну, а також порівняльний аналіз підходів на основі обробки природної мови (NLP) та графів потоку керування (CFG).

1. Штучний інтелект у задачах аналізу коду.

ML та DL трансформували підходи до безпеки, дозволяючи системам знаходити закономірності у даних без прямого втручання людини.

Machine Learning (ML) використовує алгоритми класифікації та регресії. Вдосконалення моделей значною мірою залежить від якості та кількості даних.

Deep Learning (DL) базується на штучних нейронних мережах і здатне обробляти великі масиви даних, проте часто характеризується як «чорна скринька».

2. Порівняння NLP та CFG підходів.

При аналізі коду смарт-контрактів виділяють два основні напрямки представлення даних.

- Обробка природної мови (NLP).
- Графи потоку керування (CFG).

NLP підхід розглядає код як послідовність символів або токенів (Text-based). Використовуються ембеддінги (наприклад, CodeBERT [2]) та n-грами. До переваг є низька/середня обчислювальна складність, ефективність для великих обсягів коду, стосовно недоліків - низька стійкість до обфускації (зміна імен змінних сильно впливає на результат), втрата інформації про потік даних.

Підхід CFG розглядає код як структуру, де вузли - це базові блоки, а ребра - переходи керування (Graph-based). Ключовою перевагою є висока стійкість до обфускації, збереження логічної структури програми, а до недоліків - висока обчислювальна складність, оскільки порівняння графів є NP-складною задачею.

Сучасні дослідження пропонують об'єднувати ці методи [3]. В таблиці 1 наведено порівняння розглянутих підходів.

Таблиця 1 – Порівняння NLP та CFG підходів

Критерій порівняння	Підхід NLP	Підхід CFG
Представлення	Послідовність токенів (CodeBERT).	Графова структура (вузли та ребра).
Стійкість до обфускації	Низька. Чутливість до перейменування.	Висока. Структура залишається незмінною.
Складність	Лінійна, швидка обробка.	Висока (NP-складна задача).
Втрата інформації	Втрачається потік даних між функціями.	Втрачається контекст (назви, коментарі).

Графові нейронні мережі (GNN) використовують структуру з CFG, але кожному вузлу присвоюють векторні ознаки, отримані через NLP. Схема з довгостроковою короткочасною пам'яттю (LSTM) для вбудовування базових блоків та індуктивним навчанням на основі GraphSAGE дозволяє отримати як точність графів, так і семантичне розуміння коду. Це вирішує проблему накладних витрат, характерну для чистих CFG методів [4].

Висновок. Аналіз смарт-контрактів вимагає гібридного погляду. Код слід розглядати не просто як текст (NLP) і не тільки як математичну структуру (CFG), а як послідовність інструкцій із чіткими логічними зв'язками. Використання GNN у поєднанні з методами глибокого навчання дозволяє створити системи самоадаптивної кібербезпеки, здатні прогнозувати потенційні атаки ефективніше за традиційні інструменти.

Перелік використаних джерел:

1. Smart Contract Vulnerability Detection using Graph Neural Network. [Електронний ресурс]. Режим доступу: https://scispace.com/pdf/smart-contract-vulnerability-detection-using-graph-neural-40egxpqrcq.pdf

2. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. [Електронний ресурс]. Режим доступу: https://arxiv.org/abs/2002.08155

3. Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection. [Електронний ресурс]. Режим доступу: https://messi-q.github.io/projects/papers/tkde/tkde.pdf

4. GraphSA: Smart Contract Vulnerability Detection Combining Graph Neural Networks and Static Analysis. [Електронний ресурс]. Режим доступу: https://www.researchgate.net/publication/374309511_GraphSA_Smart_Contract_Vulnerability_Detection_Combining_Graph_Neural_Networks_and_Static_Analysis