МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики

Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота

магістр

на тему

”Аналіз коду з використанням методів штучного інтелекту”

Виконав: студент 2 курсу, групи МФ-61

спеціальність 122 «Комп’ютерні науки»

освітньо-наукова програма «Інформатика»

Дроздов В.О.

Керівник Дейнега О. А.

Рецензент_____

(прізвище та ініціали)

Харків – 2025 року

План дипломної роботи

1. ВСТУП

1.1. Формулювання мети роботи, задач та обґрунтування актуальності теми

1.2. Стислий огляд відомих результатів в області дослідження

1.3. Відомості про одержані результати та їх новизна

2. ОСНОВНА ЧАСТИНА

2.1. Постановка задачі

2.2. Розвинутий огляд сучасного стану справ в області дослідження

2.3. Методи дослідження

2.4. Описання та обґрунтування алгоритмів та результатів дослідження

2.5. Аналіз результатів

3. ВИСНОВКИ

3.1. Досягнуті результати

3.2. Практичне значення роботи

3.3. Напрями подальших досліджень

4. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

ВСТУП

1.1 Формулювання мети роботи, задач та обґрунтування актуальності теми

У сучасному світі цифрова трансформація відбувається зі стрімкістю, що змінює спосіб взаємодії людей, компаній і держав. Цифрові сервіси, онлайн-платформи та розподілені системи створюють нові можливості, але одночасно потребують високого рівня довіри та безпеки. Однією з ключових технологій, яка об’єднує автоматизацію та гарантію незмінності умов угод, стали смарт-контракти. Це самовиконувані програми, що працюють у розподілених блокчейн-мережах і забезпечують автоматичну обробку транзакцій за заздалегідь заданими правилами без участі третіх сторін.

Найпоширенішою платформою для розробки смарт-контрактів є Ethereum, а основною мовою – Solidity. Завдяки прозорості блокчейну, можливості аудиту коду та відсутності централізованого контролю, Ethereum-смарт-контракти знайшли застосування у багатьох галузях: фінансових додатках (DeFi), невзаємозамінних токенах (NFT), логістиці, децентралізованому голосуванні, геймінгу, страхуванні й системах управління ідентичністю. Зростаюча складність бізнес-логіки та інтерфейсів із зовнішніми сервісами підвищує ризики помилок у коді, що може призводити до значних фінансових втрат і підірвати довіру користувачів.

Історичний досвід свідчить про важливість безпечного написання контрактів. Наприклад, атака на The DAO у червні 2016 року, що спричинила втрату понад $60 млн, та баг у бібліотеках Parity у липні 2017 року, коли через помилку в коді було заморожено понад $150 млн активів, ілюструють потенціал загроз та масштаб наслідків. Крім того, дослідження показують, що близько 30% випущених смарт-контрактів містять критичні вразливості, які можуть бути експлуатовані зловмисниками.

Традиційні методи статичного (Slither, Mythril, SmartCheck) та динамічного аналізу виявляють широкий спектр помилок, але часто

пропускають складні патерни, пов’язані з контекстною взаємодією між контрактами або економічними атаками. Наприклад, контракти з багатокроковими сценаріями обміну активами чи адаптивним керуванням правами можуть містити логічні ланцюжки, що важко перевірити класичними інструментами.

Водночас сучасні моделі штучного інтелекту (LLM – Large Language Models) продемонстрували здатність аналізувати природно-мовні та програмні тексти, виявляти приховані закономірності та генерувати рекомендації щодо виправлення помилок. Поєднання генеративних можливостей LLM із зовнішніми джерелами знань через підхід Retrieval-Augmented Generation (RAG) дозволяє інтегрувати в процес аудиту як вбудовані знання моделі, так і релевантні дані з баз OWASP, CVE, GitHub або внутрішніх репозиторіїв прикладів атак. Це суттєво підвищує точність виводів та дає змогу отримувати аргументовані пояснення та кроки для усунення вразливостей.

Таким чином, існує нагальна потреба дослідити та впровадити AI-орієнтовані методи для аналізу безпеки смарт-контрактів, що можуть доповнити та покращити існуючі інструменти аудиту. Це не лише сприятиме розвитку наукової бази знань в області кібербезпеки децентралізованих додатків, а й надасть практичний інструмент для інженерів і аудитів.

Мета роботи: розробити та протестувати прототип системи для автоматизованого виявлення та класифікації вразливостей у Solidity-контрактах із використанням великих мовних моделей та техніки RAG.

Завдання:

1. Провести глибокий огляд та класифікацію типових вразливостей у Solidity-контрактах (реентрантність, переповнення, невірна авторизація, непроблемні виклики тощо);
2. Дослідити ефективність існуючих LLM у завданнях аналізу коду й порівняти їх із класичними інструментами;

3. Зібрати, анотувати та структурувати векторну базу знань із прикладами атак, уразливих патернів і способів їх виправлення;
4. Спроектувати та реалізувати RAG-систему для інтеграції векторної бази знань з генеративним модулем LLM;
5. Провести серію експериментів на базі популярних відкритих контрактів для оцінки точності;
6. Співставити результати AI-орієнтованого аналізу з висновками традиційних інструментів аудиту та надати рекомендації щодо інтеграції в CI/CD пайплайн.

1.2 Стислий огляд відомих результатів в області дослідження

Питання безпеки смарт-контрактів уже тривалий час перебуває в центрі уваги як академічних кіл, так і розробників на практиці. Із зростанням популярності децентралізованих фінансів (DeFi), токенізованих активів та нових підходів до організації взаємодії в Інтернеті, потреба у надійному та масштабованому аналізі коду на Solidity стала як ніколи актуальною. Дослідники зазначають, що стандартні методи перевірки не завжди виявляють складні логічні або міжфункціональні вразливості.

Інструменти типу Mythril, Slither, Securify отримали широке визнання за свою здатність до формального аналізу, однак навіть вони мають обмеження у виявленні так званих zero-day уразливостей або нових комбінацій уже відомих помилок. Особливо це стосується випадків, коли код має динамічні виклики або використовує складні шаблони проектування.

Останні дослідження показують, що використання методів машинного навчання, а зокрема великих мовних моделей (LLM), таких як GPT, LLaMA, Claude, дає змогу ідентифікувати патерни, які не завжди очевидні для статичного аналізатора. Більше того, моделі можуть не лише виявити проблему, а й запропонувати альтернативний безпечний варіант її реалізації, що значно підвищує цінність таких систем у реальному застосуванні.

Цікаво також, що сучасні дослідники вивчають гібридні підходи, які поєднують класичний аналіз із LLM. Наприклад, моделі можуть отримувати попередньо підготовлені структуровані дані – контрольні графи, графи викликів, шаблони небезпечної поведінки, а потім робити висновки на їх основі. Таким чином, модель не працює "в темряві", а має певну основу для контекстного міркування. Це відкриває шлях до пояснюваних рішень, що є критично важливим у сфері безпеки.

Особливу увагу отримав підхід Retrieval-Augmented Generation (RAG). Він дозволяє не тільки згенерувати відповідь на основі знань моделі, а й динамічно підвантажити додаткову інформацію з бази документів. У контексті смарт-контрактів це можуть бути документації OpenZeppelin, приклади вразливостей із CVE-баз, whitepaper проєктів, статті з блогів розробників. Такий підхід знижує ризик "галюцинацій" моделі та покращує точність.

Незважаючи на всі досягнення, галузь і далі стикається з викликами. Не всі вразливості можна виявити з високою точністю, а інтерпретованість відповідей моделі залишається важливим завданням. Тому сучасні підходи акцентують увагу не лише на точності, а й на зрозумілості, практичній корисності, інтеграції з розробницькими пайплайнами та адаптивності до нових типів загроз.

Останніми роками в академічному та промисловому середовищі було представлено низку інструментів для аналізу Solidity-коду: Mythril, Slither, які використовують статичний та динамічний аналіз. Інструменти типу Slither базуються на розборі абстрактного синтаксичного дерева (AST) та патернів коду, що дозволяє виявляти добре відомі уразливості, однак часто не виявляють більш складних або нових атак. Mythril використовує символічне виконання, що забезпечує глибший аналіз, але потребує більше обчислювальних ресурсів. Інструмент Securify, створений у Швейцарській вищій технічній школі Цюриха (ETH Zurich), фокусується на формальних методах верифікації.

Крім того, у наукових роботах активно обговорюються методи семантичного аналізу коду, зокрема аналіз потоку даних (data flow analysis) та міжпроцедурний аналіз, що дозволяють краще розуміти поведінку контрактів у різних станах. Також ведуться дослідження в напрямку графів викликів (call graphs), які є основою для побудови багатьох сучасних інструментів аналізу.

З появою великих мовних моделей (LLM), таких як GPT-3/4, Claude, Gemini, Mistral, почали з’являтись дослідження та проєкти, які інтегрують ці моделі в роботу аналітиків безпеки. Вони дозволяють зменшити навантаження на ручний аудит, генеруючи опис вразливостей та можливі способи їх усунення. Моделі демонструють хорошу здатність до узагальнення, особливо в умовах few-shot learning – коли вони отримують кілька прикладів для орієнтації в завданні.

У межах таких досліджень також вивчається ефективність підходу Retrieval-Augmented Generation (RAG), коли модель має доступ до зовнішньої бази знань – наприклад, документів OWASP, прикладів з GitHub або внутрішніх баз аналітичних компаній. Це дозволяє покращити достовірність та точність відповідей моделей і забезпечити прозорість рішень.

Незважаючи на прогрес у цій сфері, відкритими залишаються питання довіри до моделей, пояснюваності результатів, визначення повноти аналізу та відсутності помилкових спрацьовувань (false positives / false negatives). Саме тому ця галузь потребує подальших досліджень і систематичного підходу до валідації моделей.

1.3 Відомості про одержані результати

У межах цієї дипломної роботи було реалізовано комплексний підхід до виявлення вразливостей у Solidity-контрактах, що поєднує класичні методи аналізу з сучасними підходами штучного інтелекту. Основні досягнення:

- Проведено порівняльний аналіз існуючих інструментів перевірки Solidity-контрактів (Slither, Mythril, Securify), виявлено їх переваги та обмеження.
- Створено спеціалізований датасет вразливостей, який містить приклади небезпечного коду, пояснення проблеми, а також безпечні версії того ж фрагменту коду. Датасет структуровано у форматі JSON, що дозволяє його використання в автоматизованих системах.
- Побудовано прототип RAG-системи, в якій GPT-модель звертається до векторної бази знань (створеної за допомогою FAISS) для покращення якості аналізу. Це дозволило поєднати знання моделі з фактичними прикладами з літератури, документації та open-source.
- Розроблено кастомні промпти для аналізу коду, які дають можливість генерувати не лише виявлену помилку, а й пояснення, рівень ризику та конкретну інструкцію щодо усунення вразливості. У результаті відповіді моделі стали більш зрозумілими та структурованими.
- Проведено тестування на контрактах з Ethernaut, OpenZeppelin Contracts, а також зібрано приклади з etherscan.io, щоб забезпечити реалістичні сценарії використання.
- Здійснено кількісну та якісну оцінку результатів: визначено precision/recall на основі ручної верифікації, а також зібрано відгуки розробників щодо зручності та практичної користі інструменту.

2. ОСНОВНА ЧАСТИНА

2.1 Постановка задачі

У контексті постійного зростання популярності смарт-контрактів та їх широкого впровадження у реальні бізнес-процеси постає проблема забезпечення їхньої надійності та безпеки. Як було відзначено у вступі, велика кількість інцидентів, пов’язаних із вразливостями в коді смарт-контрактів, призводить до фінансових втрат, втрати довіри до технології блокчейну та створення правових ризиків. Проблема ускладнюється тим, що навіть

досвідчені розробники не завжди здатні виявити складні логічні помилки або врахувати всі аспекти взаємодії контракту з іншими частинами системи. У деяких випадках помилки виникають не лише на рівні реалізації, а й через неправильне розуміння специфікацій чи непередбачувані сценарії використання контракту в середовищі взаємодіючих смарт-контрактів.

У зв’язку з цим виникає нагальна потреба у створенні інструменту, який би дозволяв швидко, ефективно та автоматизовано виявляти типові й нетипові вразливості у смарт-контрактах, написаних мовою Solidity. Такий інструмент має відповідати сучасним вимогам як до точності результатів, так і до зручності використання, забезпечуючи гнучкість, масштабованість, інтерпретованість відповідей і можливість інтеграції в розробницький пайплайн.

Особлива увага в межах цього дослідження приділяється аналізу потенціалу великих мовних моделей (LLM) у поєднанні з Retrieval-Augmented Generation (RAG) як інноваційного інструменту в галузі безпеки коду. Метою є не лише ідентифікація проблемних ділянок коду, а й пояснення знайдених помилок, генерація виправлень і формування структурованих аналітичних звітів.

Постановка задачі дипломної роботи включає такі ключові напрями:

- сформулювати вимоги до системи виявлення вразливостей у Solidity-контрактах з урахуванням актуальних типів атак, таких як reentrancy, integer overflow/underflow, unchecked calls, використання tx.origin тощо;
- побудувати архітектуру рішення, що поєднує мовну модель з джерелами достовірної інформації (наприклад, прикладами CVE, документацією OpenZeppelin, дослідницькими статтями);
- створити набір прикладів вразливостей та безпечних реалізацій, анотований у форматі JSON для подальшого використання в тестуванні;

- реалізувати систему генерації звітів, яка буде повертати список вразливостей з інформацією про лінії коду, тип помилки, її опис, рівень ризику та запропонований спосіб усунення;
- провести порівняльний аналіз з існуючими інструментами (наприклад, Slither, Mythril), визначивши переваги та недоліки кожного з підходів;
- оцінити продуктивність, точність і застосовність створеного рішення для реального використання в задачах аудитів смарт-контрактів.

Поставлені задачі охоплюють як дослідницький аспект – вивчення можливостей сучасних LLM та підходу RAG у контексті аналізу коду, так і прикладний: створення діючого прототипу інструменту, який дозволить суттєво спростити процес аудиту, зменшити кількість помилок та покращити загальний рівень безпеки у Web3-розробці.

2.2 Розгорнутий огляд сучасного стану справ в області дослідження

2.2.1 Загальні підходи до аналізу коду

Аналіз коду є фундаментальним етапом у забезпеченні якості, надійності та безпеки програмного забезпечення. Існують два основні підходи до аналізу коду: **статичний** та **динамічний**. Кожен із них має свої переваги, обмеження та специфічні області застосування. У сукупності вони забезпечують більш повну картину потенційних проблем у програмному забезпеченні.

Статичний аналіз коду

Статичний аналіз коду (англ. Static Code Analysis) – це метод дослідження програмного забезпечення, який передбачає перевірку коду без його виконання. Він дозволяє виявляти синтаксичні помилки, порушення стандартів кодування, потенційні вразливості та недоліки логіки на ранніх етапах розробки. Такий аналіз зазвичай проводиться автоматизовано за допомогою спеціалізованих інструментів.

Загальні підходи та алгоритми:

- **Абстрактна інтерпретація (Abstract Interpretation):** метод, який імітує виконання програми у скороченому вигляді, дозволяючи оцінювати поведінку змінних без фактичного запуску коду. Його основна перевага полягає в тому, що він забезпечує звичайно консервативні, але корисні оцінки поведінки програми. Метод застосовується у випадках, коли потрібно забезпечити доказ правильності певних властивостей, зокрема відсутності переповнень або звернень до нульових покажчиків.
- **Потоковий аналіз (Data Flow Analysis):** вивчає, як дані передаються між змінними, функціями та модулями. Завдяки йому можна виявити мертві змінні, непрямі залежності, порушення інваріантів. Різновиди цього методу – аналіз доступу до змінних, живих змінних, досягності значень тощо.
- **Контроль потоку виконання (Control Flow Analysis):** дозволяє створити граф керування виконанням (CFG), який представляє всі можливі шляхи виконання через програму. Такий аналіз необхідний для розуміння логіки функцій, перевірки циклів, гілок умов, виявлення нескінченних петльових структур або unreachable code. Його також використовують для побудови автоматів для формального верифікування.
- **Аналіз залежностей (Dependency Analysis):** визначає, які об'єкти, функції або змінні взаємозалежні. Це дає змогу виявити тісно пов’язані частини системи, сприяти декомпозиції, покращити модульність, а також допомагає уникнути побічних ефектів у процесі змін.
- **Семантичний аналіз:** заглиблюється у зміст конструкцій, логіку, контексти виконання. Його основна задача – з'ясування смислових зв'язків між частинами коду, що важливо для виявлення логічних помилок, а не лише синтаксичних або стилістичних. Такий аналіз іноді використовує елементи логіки першого порядку або абстрактних інтерпретаторів.

Популярні інструменти:

- **SonarQube:** платформа для безперервного аналізу якості коду. Підтримує понад 20 мов, інтегрується з CI/CD системами, надає детальні дашборди та виявляє баги, уразливості, code smells і проблеми з підтримкою.
- **PVS-Studio:** потужний інструмент для виявлення помилок у C, C++, C#, Java. Пропонує глибокий аналіз низькорівневих помилок, інтегрується з Visual Studio, IntelliJ IDEA, VS Code.
- **PMD:** сканер коду для Java, Apex, Visualforce, JavaScript. Виявляє дублікати, мертвий код, некоректні конструкції, слабкі місця у логіці.
- **Checkstyle:** перевірка відповідності Java-коду визначеним стилістичним правилам. Підходить для підтримки єдиного стилю в команді.
- **SpotBugs:** статичний аналізатор Java-коду, successor до FindBugs. Використовує байт-код для виявлення потенційно помилкової поведінки, як-от null dereference, недосяжний код тощо.
- **Pylint:** Python-аналізатор, що виконує linting, оцінку стилю, перевірку на помилки, використання змінних і потенційні проблеми безпеки.
- **Bandit:** спеціалізований на виявленні вразливостей Python-коду, наприклад, використання небезпечних функцій (exec, eval), hardcoded credentials.
- **Semgrep:** комбінує можливості класичного лінтера та кастомних шаблонів на основі синтаксичних дерев. Підходить для безпеки коду, рев'ю та відповідності стандартам.

Динамічний аналіз коду (англ. Dynamic Code Analysis) – це метод дослідження, який включає запуск програмного забезпечення в контрольованому середовищі для виявлення помилок, що виникають під час виконання. Цей підхід дозволяє виявляти такі проблеми, як витоки пам’яті, переповнення буфера, умови гонки, некоректне керування ресурсами, які не можуть бути знайдені статичним аналізом.

Загальні підходи та алгоритми:

- **Аналіз покриття (Code Coverage Analysis):** визначає, які частини коду були виконані під час тестів. Розрізняють покриття рядків, гілок, умов, шляхів – усе це дозволяє оцінити повноту тестування. Аналіз допомагає забезпечити, щоб жодна критична частина коду не залишилась без тестового покриття.
- **Фаззінг (Fuzz Testing):** автоматичне генерування випадкових або некоректних вхідних даних для тестування стабільності програми. Він широко використовується у сфері безпеки для виявлення zero-day вразливостей. Цей метод може бути як чорним (без знання про структуру програми), так і білим (з частковим розумінням структури).
- **Моніторинг виконання:** спостереження за поведінкою програми під час її виконання. Це може бути пасивне логування або активний контроль метрик: споживання ресурсів, навантаження, затримки. Такий аналіз дозволяє оптимізувати продуктивність та виявити «вузькі місця».
- **Інструментування коду (Instrumentation):** модифікація вихідного або байт-коду з метою відстеження виконання. Це може бути як ручна вставка логів, так і автоматичне додавання викликів у ключові точки виконання. Інструментування корисне для динамічного профілювання, виявлення змін у часі відгуку та діагностики складних багів.

Популярні інструменти:

- **Valgrind:** набір утиліт для профілювання та виявлення помилок у пам’яті, таких як use-after-free, витоки пам’яті, невизначене використання змінних. Найбільш корисний для C/C++.
- **AppDynamics:** APM-платформа, що надає телеметрію для продуктивності додатків, дозволяє стежити за станом сервісів у режимі реального часу, виявляти аномалії, будувати аналітику на основі логів.
- **Dynatrace:** аналог AppDynamics з AI-модулем Davis, що автоматично ідентифікує root-cause проблем, підтримує повну трасування транзакцій.

- **JaCoCo:** засіб аналізу покриття коду для Java, інтегрується з Maven, Gradle, Jenkins, ідеально підходить для перевірки ефективності юніт-тестів.
- **Selenium:** набір бібліотек для автоматизації браузерного тестування. Підтримує різні браузери, дозволяє імітувати дії користувача, створювати тестові сценарії.
- **JMeter:** Apache JMeter призначений для навантажувального тестування веб-додатків і сервісів. Підтримує HTTP, FTP, SOAP, JDBC, дозволяє створювати складні сценарії взаємодії.

Поєднання статичного та динамічного аналізу забезпечує повніше покриття можливих вразливостей і дозволяє досягти вищої надійності, безпеки та якості програмного забезпечення на всіх етапах його життєвого циклу. Ці два підходи слугують не як конкуренти, а як взаємодоповнюючі інструменти, що разом формують цілісну систему контролю якості. У великих проєктах, де помилки можуть мати серйозні наслідки, така комбінація є не просто бажаною, а обов'язковою.

Крім того, інтеграція аналітичних підходів у процес розробки – зокрема в рамках DevSecOps – дозволяє виявляти дефекти ще до їхнього потрапляння в продакшн. Систематичне використання аналізу коду сприяє підвищенню рівня культури програмування в команді, покращує документацію, спрощує супровід і модернізацію ПЗ. У підсумку, це не лише знижує витрати на подальшу підтримку, але й формує основу для довгострокового успіху проєкту в умовах динамічного розвитку IT-індустрії.

2.2.2 Аналіз коду за допомогою штучного інтелекту

Аналіз коду за допомогою штучного інтелекту (ШІ) стає дедалі більш популярним напрямом у сучасній розробці програмного забезпечення. Розвиток мовних моделей і машинного навчання відкриває нові горизонти для автоматизації рутинних завдань, підвищення продуктивності розробників, а також покращення якості та безпеки програмного коду. У цьому підрозділі буде

розглянуто ключові інструменти та підходи, які застосовують ШІ для аналізу коду.

AI-асистенти для розробників

Останніми роками з'явився ряд інструментів, які вбудовуються безпосередньо в середовища розробки (IDE) і допомагають програмістам під час написання коду.

- **GitHub Copilot** – один з найвідоміших AI-асистентів, створений компанією GitHub спільно з OpenAI. Використовуючи модель GPT, він пропонує продовження коду, автоматично генерує функції, коментарі та навіть тести.
- **Tabnine** – інструмент, що базується на меншій, спеціалізованій моделі, адаптованій для роботи з кодом. Його фокус – надання контекстно-залежних підказок на основі попереднього коду та стилю програміста.
- **Amazon CodeWhisperer** – AI-асистент від AWS, який інтегрується з хмарними сервісами Amazon. Він допомагає створювати безпечний код, розпізнаючи потенційно небезпечні шаблони програмування.

Ці інструменти не лише пришвидшують процес програмування, але й зменшують ймовірність помилок завдяки автозаповненню та рекомендаціям.

AI-інструменти для аналізу коду

Окрім допомоги під час написання коду, ШІ також широко використовується для його подальшого аналізу. Сучасні AI-моделі здатні знаходити складні помилки, вразливості та місця для рефакторингу в коді. Вони поєднують в собі елементи статичного та семантичного аналізу, навчаючись на мільйонах рядків коду різних мов програмування.

- **CodeQL** – аналітичний інструмент від GitHub, який дозволяє описувати запити для знаходження вразливостей у коді. Він працює на базі створення графів програм і використовує мову запитів для

виявлення шаблонів помилок. CodeQL добре підходить для складних проєктів з відкритим кодом і активно використовується у багатьох програмах bug bounty.

- **DeepCode (тепер частина Snyk)** – інструмент, що базується на використанні нейромереж для виявлення потенційних проблем у коді. Його сильна сторона – здатність аналізувати великий контекст програми, розуміти логіку і пропонувати не лише виправлення, а й пояснення, чому саме фрагмент коду є проблемним.
- **OpenAI Codex API** – мовна модель, що здатна аналізувати, інтерпретувати та покращувати код на десятках мов програмування. Її гнучкість дозволяє створювати кастомізовані рішення, наприклад, інтеграцію з системами RAG (Retrieval-Augmented Generation), щоб знаходити вразливості, виправляти помилки або навіть писати юніт-тести автоматично.
- **SonarQube з Machine Learning** – деякі сучасні версії SonarQube підтримують ML-аналіз для оцінки технічного боргу. Це поєднання класичних правил перевірки та навчання на історичних даних репозиторіїв дозволяє автоматично адаптувати правила під конкретний проєкт.
- **CodeGuru від Amazon** – інструмент, який аналізує Java- та Python-код, фокусуючись на продуктивності та безпеці. Використовується в екосистемі AWS для виявлення неоптимальних конструкцій та дорогих у виконанні запитів до баз даних.

Додаткові переваги таких інструментів:

- Інтеграція з CI/CD пайплайнами (GitHub Actions, GitLab CI, Jenkins), що дозволяє запускати аналіз автоматично після кожного коміту.
- Візуалізація проблем і рекомендацій у вигляді інтуїтивно зрозумілих звітів.
- Можливість виявлення zero-day патернів на основі моделі, натренованої на схожих прикладах коду.

Водночас варто зазначити і виклики:

- Не всі інструменти є повністю відкритими або безкоштовними, а інтеграція з існуючим стеком може вимагати часу.
- Різні інструменти показують різну ефективність залежно від мови програмування (наприклад, CodeGuru оптимізований для Java та Python, але не підтримує C++).
- У деяких випадках потрібне глибоке налаштування для досягнення прийнятної точності в конкретному проєкті.

Ці засоби відкривають можливості для глибшого, багатовимірного аналізу коду з меншими витратами часу, підвищуючи якість продукту ще на ранніх стадіях розробки. Завдяки гнучкості налаштування, можливості розширення та сумісності з різними інфраструктурами, AI-інструменти стають не просто допоміжними утилітами, а повноцінними компонентами сучасного пайплайну розробки. Вони дозволяють формувати культуру якісного коду в команді, зменшувати технічний борг, а також забезпечувати відповідність стандартам безпеки та надійності. Використання таких інструментів дедалі більше інтегрується в практики DevSecOps, де безпека та якість коду розглядаються не як етап наприкінці, а як постійний процес, що супроводжує всю розробку програмного забезпечення.

Переваги та обмеження AI в аналізі коду

Використання ШІ в аналізі коду має низку важливих переваг:

- **Швидкість** – AI може миттєво знаходити проблеми, які людина виявила б лише після ретельного огляду.
- **Масштабованість** – AI можна застосовувати до великих обсягів коду без зниження якості аналізу.
- **Контекстність** – сучасні моделі можуть враховувати широкий контекст програми, включаючи залежності між модулями та попередній код.

Однак існують і обмеження:

- **Хибно позитивні результати** – AI іноді помилково позначає правильний код як потенційно проблемний.
- **Непрозорість** – результати аналізу ШІ не завжди можна легко пояснити або інтерпретувати.
- **Залежність від якості даних** – точність аналізу сильно залежить від якості даних, на яких навчалася модель.

Таким чином, штучний інтелект виступає потужним інструментом у руках розробників, але для досягнення найкращих результатів його слід використовувати у поєднанні з традиційними методами перевірки та рецензування коду. Він не замінює людину повністю, але суттєво розширює її можливості. Як помічник, аналітик чи наглядач, штучний інтелект здатен виявляти нестандартні помилки, логічні упущення або запропонувати ефективніші рішення. Його застосування підвищує не тільки технічну якість коду, а й культуру командної взаємодії, змушуючи розробників бути уважнішими до архітектури, читаємості та документації. У майбутньому, зі зростанням складності проєктів, значення таких інструментів лише зростатиме, адже вони дозволяють тримати контроль над усе більшими обсягами коду, забезпечуючи швидку реакцію на потенційні загрози та проблеми.

2.2.3 Аналіз коду смарт-контрактів без використання AI

Особливості смарт-контрактів

Смарт-контракти – це автономні програми, що працюють на блокчейн-платформах, таких як Ethereum, і автоматично виконують умови договорів без участі третіх сторін. Основною мовою програмування для створення смарт-контрактів на Ethereum є Solidity. Унікальність смарт-контрактів полягає в їхній незмінності після розгортання, що робить особливо важливим ретельний аналіз їхнього коду перед деплоєм. Оскільки будь-яка помилка в коді може призвести до втрати коштів або експлуатації

контракту зловмисниками, питання безпеки відіграє ключову роль у процесі розробки.

Традиційні інструменти аналізу

До появи інструментів, що базуються на штучному інтелекті, аналіз смарт-контрактів проводився за допомогою класичних статичних аналізаторів коду. Два найпоширеніші інструменти в цій категорії – **Mythril** та **Slither**.

- **Securify2** – це ще один важливий інструмент для аналізу смарт-контрактів, який використовує формальні методи для забезпечення безпеки коду. Він здійснює перевірку властивостей контракту на основі формальної верифікації, виявляючи такі помилки, як потенційні порушення безпеки, невідповідність специфікації та некоректні логічні операції. Перевагою Securify2 є можливість автоматизованого доказу безпеки для великого класу контрактів, хоча він також може вимагати значних ресурсів і додаткових налаштувань.
- **Solhint** – це популярний лінтер для Solidity, призначений для підтримки стилістичної узгодженості коду і дотримання найкращих практик програмування. Solhint перевіряє код на відповідність загальноприйнятим правилам та допомагає запобігати помилкам, які можуть вплинути на безпеку контракту, наприклад, виявляючи потенційні проблеми з іменуванням змінних, некоректні використання функцій або ігнорування рекомендованих шаблонів розробки.
- **Mythril** – це інструмент для аналізу байткоду Ethereum-контрактів. Він використовує техніку символічного виконання (symbolic execution), яка дозволяє виявляти помилки без реального виконання коду. Mythril вміє знаходити вразливості, такі як переповнення чисел (integer overflow), повторні виклики (reentrancy), та інші поширені проблеми безпеки.
- **Slither** – потужний статичний аналізатор для Solidity-контрактів, який працює на рівні вихідного коду. Slither генерує абстрактне представлення коду у вигляді проміжного подання (intermediate

representation), що дозволяє проводити складні правила перевірки. Він може виявляти десятки типів помилок, а також забезпечує інтеграцію з CI/CD пайплайнами.

Окрім них, також використовуються такі інструменти, як **Oyente**, **Solhint** та **Surya**, кожен з яких виконує свою специфічну роль у процесі перевірки смарт-контрактів.

Обмеження традиційних методів

Попри високу точність деяких інструментів, традиційні методи аналізу мають низку обмежень. По-перше, вони часто обмежені жорстко запрограмованими правилами, що унеможливлює виявлення нових або складних вразливостей, які не були враховані під час створення інструмента. По-друге, символічне виконання, яке використовують деякі з них, є вкрай ресурсоємним і не завжди масштабується для великих або складних контрактів. По-третє, результати аналізу іноді містять велику кількість false positive-помилок, що ускладнює ефективне використання таких систем на практиці.

Більш того, традиційні аналізатори не завжди здатні контекстуалізувати логіку контракту або зрозуміти наміри розробника, що знижує якість аналізу, особливо для складних бізнес-логік.

У зв’язку з цими обмеженнями, зростає інтерес до застосування методів штучного інтелекту, які можуть адаптуватися до нових патернів вразливостей і навчатися на великому обсязі прикладів, що і буде розглянуто у наступних підрозділах.

2.2.4 Аналіз коду смарт-контрактів з використанням AI

Використання штучного інтелекту для аналізу коду смарт-контрактів є одним із сучасних і перспективних напрямків у сфері безпеки блокчейн-додатків. AI-підходи дозволяють значно автоматизувати процеси пошуку вразливостей, оптимізації та рефакторингу коду. Застосування AI до

аналізу смарт-контрактів дозволяє розробникам оперативно і з високою точністю виявляти типові помилки, які можуть призвести до значних фінансових втрат або компрометації безпеки.

Існує кілька перспективних напрямків застосування штучного інтелекту у сфері аналізу смарт-контрактів. Одним із ключових підходів є використання моделей машинного навчання та глибокого навчання для автоматичного виявлення вразливостей, таких як reentrancy-атаки, переповнення цілих чисел, небезпечні виклики зовнішніх функцій тощо. Зокрема, застосовуються такі підходи як нейронні мережі типу Transformer, Graph Neural Networks (GNN), та LSTM-мережі, які можуть аналізувати структурні та семантичні особливості коду.

Серед найбільш відомих інструментів, які активно використовують AI для аналізу смарт-контрактів, варто зазначити:

- **SmartBugs** – фреймворк, який інтегрує різні методи аналізу, включаючи AI-моделі, для комплексної перевірки смарт-контрактів на вразливості.
- **ContractWard** – інструмент, що використовує глибоке навчання для виявлення та класифікації вразливостей у Solidity-контрактах.
- **SolidityAI** – розширення до середовища розробки Visual Studio Code, яке застосовує нейронні мережі для автоматичного виявлення проблем у коді безпосередньо під час написання контрактів.

У науковій літературі також активно розглядаються підходи на основі AI. Наприклад, праці, що використовують Graph Neural Networks, демонструють високу ефективність у визначенні залежностей між різними компонентами смарт-контрактів, що суттєво покращує якість аналізу.

Переваги застосування штучного інтелекту в аналізі смарт-контрактів очевидні. По-перше, AI здатний швидко обробляти великі об'єми коду, виявляючи вразливості значно швидше і точніше порівняно з ручним аналізом або класичними статичними методами. По-друге, моделі штучного інтелекту

можуть навчатися на нових даних, постійно вдосконалюючи свої алгоритми та адаптуючись до нових типів атак і загроз. Нарешті, використання AI дозволяє не лише виявляти вразливості, а й пропонувати конкретні шляхи їх виправлення, що значно спрощує роботу розробників і підвищує загальний рівень безпеки блокчейн-додатків.

2.3. Методи дослідження

2.3.1. Загальна характеристика методів дослідження

Для досягнення мети дипломної роботи, яка полягає у виявленні вразливостей у Solidity-контрактах, були обрані сучасні методи аналізу програмного коду. Центральне місце серед цих методів посідають технології, що базуються на штучному інтелекті (AI). Зокрема, у роботі використовується бібліотека LangChain, що дозволяє ефективно взаємодіяти з моделями GPT від OpenAI через API.

Підхід передбачає застосування двох основних стратегій аналізу: класичний метод аналізу коду на основі промптів до мовних моделей, а також більш просунутий метод Retrieval-Augmented Generation (RAG), який передбачає використання додаткових контекстних даних для покращення якості аналізу. Вибір таких методів обумовлений їх здатністю ефективно та точно ідентифікувати широкий спектр можливих вразливостей у контрактах, що є критично важливим для забезпечення безпеки блокчейн-додатків.

2.3.2. Теоретичні методи дослідження

У рамках теоретичного обґрунтування дослідження було здійснено аналіз сучасних наукових публікацій, дослідницьких статей та практичних рішень, присвячених проблематиці виявлення вразливостей у коді за допомогою штучного інтелекту. Особлива увага була приділена методам генерації відповідей великими мовними моделями (LLM, Large Language Models), а також підходам до поєднання цих моделей із зовнішніми джерелами знань.

Великі мовні моделі (LLM)

LLM, такі як GPT-4, Claude або CodeLLaMA, є прикладами трансформерних архітектур, що здатні виконувати широкий спектр завдань – від генерації тексту до складного аналізу коду. В основі таких моделей лежить механізм self-attention, який дозволяє враховувати контекст у довгих послідовностях. У сфері аналізу коду LLM здатні виявляти логічні помилки, порушення інваріантів, небезпечні шаблони, а також пропонувати варіанти виправлень.

LLM побудовані на основі трансформерної архітектури, запропонованої Vaswani et al. у 2017 році. Центральним компонентом трансформера є механізм самоуваги (self-attention), який дозволяє кожному елементу вхідної послідовності взаємодіяти з усіма іншими елементами, зважуючи їхню важливість для поточного контексту. Це забезпечує гнучке та масштабоване представлення тексту, дозволяючи моделі розуміти як локальні, так і глобальні залежності в даних.

Внутрішня структура LLM включає в себе багатошаровий стек енкодерів і/або декодерів, кожен з яких містить багатоголовкову увагу (multi-head attention), механізми нормалізації, залишкові з'єднання (residual connections), а також позиційні механізми, що дозволяють моделі враховувати порядок слів і структурні залежності. Багатоголовкова увага дає змогу моделі аналізувати кожен токен у контексті всіх інших, що суттєво покращує здатність до розуміння складних зв’язків у тексті або коді. Позиційне кодування компенсує відсутність рекурентності, додаючи інформацію про порядок елементів у послідовності.

Кожен шар моделі поглиблює її розуміння вхідних даних, забезпечуючи дедалі вищий рівень абстракції. Наприклад, нижчі шари зосереджуються на синтаксичних зв’язках, тоді як вищі формують семантичне уявлення. Завдяки цьому модель здатна розпізнавати шаблони, що відповідають помилкам програмування або нестандартній логіці.

Під час тренування модель оптимізує свою здатність передбачати наступне слово або токен у тексті, використовуючи функцію втрат – зазвичай перехресну ентропію. Це дає змогу створити гнучку статистичну модель мови, яка може генерувати новий текст або кодувати існуючий, формуючи контекстуально релевантні виводи. У результаті модель здатна ефективно переносити знання з одного домену в інший і виконувати нові завдання із мінімальною адаптацією.

Загальною задачею великих мовних моделей є навчання універсального представлення знань про мову та світ на основі великих корпусів тексту. Таке представлення дозволяє LLM адаптуватися до різноманітних завдань: класифікація, генерація, резюмування, переклад, відповідь на запити, аналіз коду тощо.

Однак навіть найпотужніші мовні моделі мають обмеження. Вони базуються на знаннях, що були отримані під час навчання, і не завжди володіють актуальною або специфічною інформацією. Це особливо критично при аналізі коду, де важливу роль відіграє поточний контекст, специфіка контракту або поєднання шаблонів. Тому все частіше використовуються підходи з доповненням пам’яті (наприклад, Retrieval-Augmented Generation), які дозволяють моделі динамічно отримувати актуальні дані з зовнішніх джерел.

Ембедінги (Embeddings)

Ембедінги – це спосіб представлення текстових або кодових елементів у вигляді векторів фіксованої розмірності. Метою такого представлення є перенесення семантичних властивостей об'єктів (наприклад, функцій або змінних у коді) в числовий простір, де можна обчислювати подібність, класифікувати або знаходити найближчі за змістом елементи. У випадку аналізу коду використовуються спеціалізовані моделі, такі як OpenAI Text Embeddings або CodeBERT, які дозволяють перетворювати фрагменти програмного коду у вектори, що враховують як синтаксис, так і семантику. Це дозволяє ефективно виконувати задачі семантичного пошуку, кластеризації, а

також використовувати ці вектори як основу для Retrieval-Augmented Generation (RAG). Завдяки ембедінгам стає можливою побудова масштабованих систем аналізу коду з точним пошуком релевантних прикладів або шаблонів вразливостей.

Векторні бази даних (Vector Databases)

Щоб ефективно зберігати та шукати ембедінги, використовуються векторні бази даних – системи, оптимізовані для роботи з великою кількістю векторів у високовимірному просторі. Одним із популярних рішень є FAISS (Facebook AI Similarity Search), яке дозволяє виконувати швидкий пошук найближчих сусідів (nearest neighbors) до заданого вектору. У системах типу RAG векторна база знань містить ембедінги фрагментів коду, описів вразливостей, документації або прикладів атак. Коли модель отримує запит, вона спочатку перетворює його у вектор, після чого здійснюється пошук найбільш подібних записів у базі. Ці знайдені фрагменти повертаються моделі як контекст для генерації, що суттєво покращує її точність і обґрунтованість.

Retrieval-Augmented Generation (RAG)

Щоб подолати обмеження моделей у сучасних знаннях, було розроблено підхід Retrieval-Augmented Generation (RAG). Його суть полягає в тому, що модель перед генерацією відповіді отримує додатковий контекст – релевантні документи, витягнуті з векторної бази знань. Таким чином, процес генерації стає не лише більш точним, а й обґрунтованим з позиції реальних прикладів, документації або специфічних патернів уразливостей.

Процес роботи RAG можна описати наступними етапами:

1. Запит користувача або фрагмент коду надходить на вхід системи.
2. За допомогою ембедінг-моделі (наприклад, OpenAI Embeddings або BERT) текст перетворюється на векторне представлення.
3. Вектор запиту використовується для пошуку найбільш релевантних документів у векторній базі даних (Vector DB), наприклад FAISS.

4. Найбільш релевантні документи (контексти) додаються до промпта – модель отримує не лише запит, а й підкріплення у вигляді реальних прикладів.
5. LLM обробляє розширений промпт та генерує відповідь.
6. Відповідь повертається у форматі, зручному для аналізу (наприклад, структурований JSON).

Архітектура системи на основі RAG

На **рисунку 2.1** представлена узагальнена архітектура системи, яка реалізує підхід Retrieval-Augmented Generation. Схема ілюструє основні етапи: від надходження запиту до генерації фінального результату.

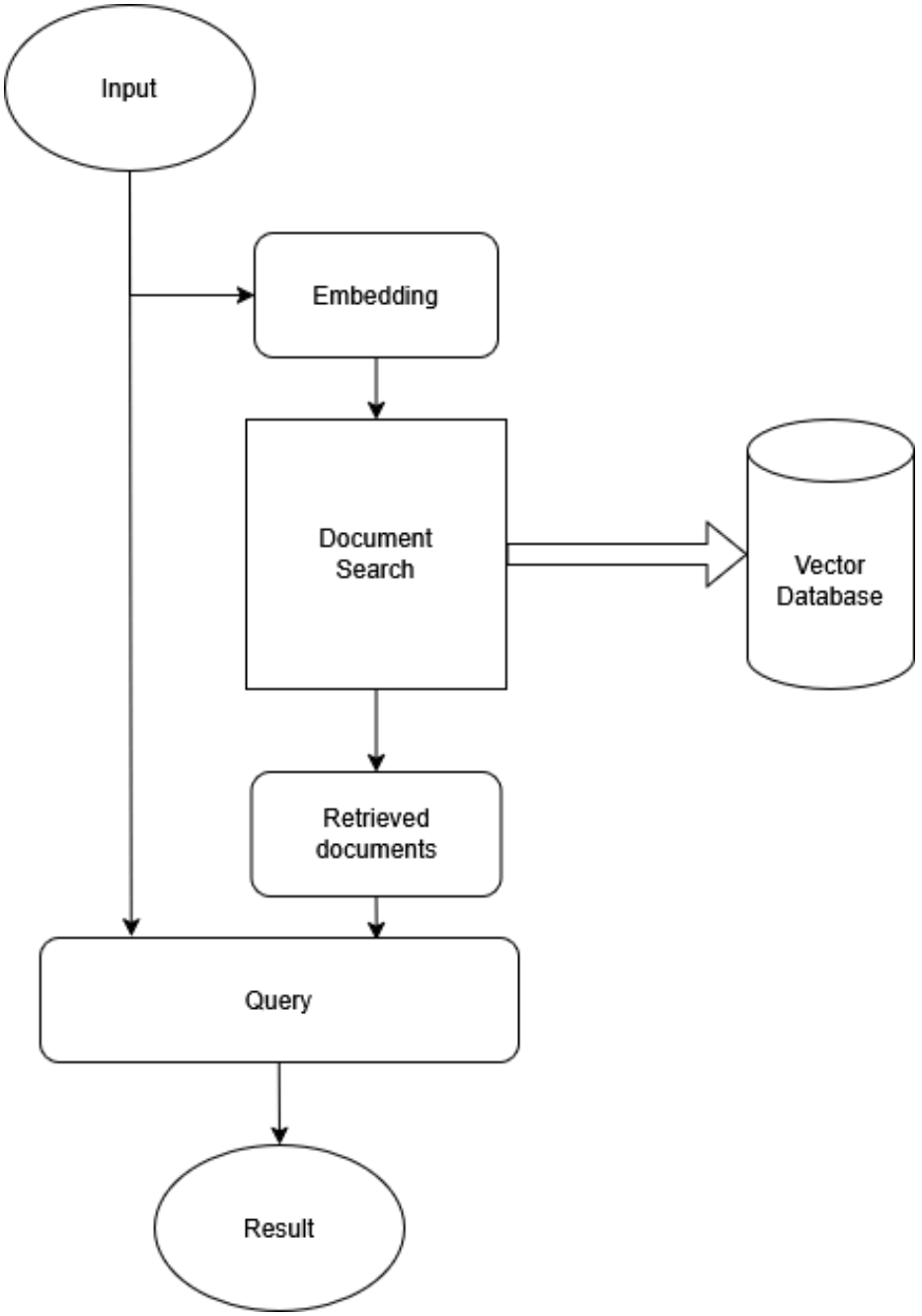

Рисунок 2.1 – Архітектура системи аналізу коду з використанням Retrieval-Augmented Generation (RAG)

Цей підхід забезпечує гнучкість, розширюваність та підвищену точність виявлення вразливостей. Завдяки тому, що база знань легко доповнюється, система може адаптуватися до нових шаблонів атак або нових мов програмування без потреби перевчання самої LLM.

Практичне значення

RAG дозволяє будувати системи, здатні:

- Враховувати специфіку кожного запиту;
- Мінімізувати "галюцинації" моделі;
- Залучати приклади з перевірених джерел (SWC Registry, CVE, OpenZeppelin Docs);
- Генерувати не лише виявлення, а й **пояснення** проблем та пропозиції фіксів.

Таким чином, поєднання LLM з механізмами пошуку релевантних знань відкриває нові горизонти в задачах аналізу коду та дозволяє реалізувати системи, які значно перевищують традиційні статичні аналізатори як у точності, так і в пояснюваності результатів.

2.3.3. Експериментальні методи дослідження

У рамках експериментальної частини дослідження було реалізовано два незалежних підходи до аналізу Solidity-контрактів: із використанням Retrieval-Augmented Generation (RAG) та без нього. Кожен з підходів реалізовано у вигляді окремого скрипта на мові Python: analysis.py (без RAG) та analysisrag.py (із RAG).

Було створено датасет із прикладами смарт-контрактів (10–20 файлів), що зберігаються у папці vuln_files. Ці приклади використовувалися як навчальна база для побудови векторного простору, який потім застосовувався у RAG-підході. Для цього використовувалася бібліотека FAISS, а ембедінги обчислювалися за допомогою OpenAI Embeddings.

У RAG-реалізації було створено векторну базу знань, яка дозволяє LangChain здійснювати пошук найбільш релевантної інформації для кожного запиту до моделі GPT. У класичному підході модель аналізує код лише на основі сформованого промпту без додаткового контексту.

Також було створено механізм оцінки якості результатів – за допомогою скрипту run_test.ipynb, який порівнює фактичні виводи моделі з очікуваними та обчислює точність виявлення вразливостей.

2.3.4. Практична реалізація експериментів

Реалізація експериментів здійснювалася за допомогою мови програмування Python. Було створено декілька окремих модулів, кожен з яких виконує певну функцію в межах аналізу безпеки смарт-контрактів. Файл embedings.py відповідає за побудову векторної бази з текстових прикладів вразливостей, що зберігаються в каталозі vuln_files. За допомогою LangChain та OpenAI Embeddings здійснюється обробка цих документів та збереження їх у форматі FAISS.

Файл analysisrag.py реалізує повноцінний RAG-ланцюг: від завантаження векторної бази до побудови prompt-шаблону, взаємодії з LLM (GPT-4o-mini), обробки результатів та парсингу у форматі JSON. У свою чергу, anaylysis.py дозволяє виконувати аналіз без застосування векторної бази – лише через інструкцію у форматі prompt.

Обидва підходи дозволяють отримати структуровану відповідь від моделі у форматі JSON, що містить інформацію про рядок коду, тип вразливості, її опис, рівень критичності та спосіб усунення. Для коректної обробки відповідей використовується окремий модуль json_parser.py, який автоматично витягує JSON-зміст з відповіді моделі. Результати аналізу виводяться на консоль або зберігаються для подальшої оцінки точності в тестах.

2.3.5. Методи оцінки якості результатів

Оцінка ефективності реалізованих методів аналізу здійснювалася шляхом порівняння результатів виявлення вразливостей у Solidity-контрактах із заздалегідь відомими правильними відповідями. Для цього був створений окремий файл тестування (run_test.ipynb), який автоматизує процес перевірки результатів моделей.

Відповідь моделі порівнюється із очікуваними результатами, що зберігаються у тестовому наборі даних. У процесі оцінювання використовуються базові метрики точності: частка правильних виявлень серед

усіх запропонованих (accuracy), повнота (recall), а також можливе обчислення F1-score для загальної оцінки балансу між точністю (precision) і повнотою.

Також проводилося порівняння результатів двох підходів: класичного (без RAG) і з використанням Retrieval-Augmented Generation. Це дозволяє виявити переваги RAG-підходу щодо покращення релевантності аналізу, особливо при роботі з фрагментами коду, які вимагають додаткового контексту для точного визначення вразливості.

2.3.6. Процедура обробки та аналізу даних

Після отримання відповідей від мовної моделі, важливим етапом стало забезпечення коректної обробки даних та перетворення їх у структурований формат. Для цього було реалізовано спеціальний модуль json_parser.py, який дозволяє автоматично витягувати JSON-фрагменти з відповіді моделі OpenAI. Цей парсер забезпечує стійкість до синтаксичних помилок, таких як незакриті лапки або неправильне форматування JSON.

Усі вхідні Solidity-файли, що використовуються для аналізу, попередньо проходять стадію обробки та, у разі потреби, фрагментації на менші логічні блоки. Для цього застосовується функціонал розбиття документів у LangChain (RecursiveCharacterTextSplitter), що дозволяє оптимізувати обробку великих обсягів коду, зберігаючи релевантність запитів.

Також здійснюється завантаження та збереження векторної бази знань у форматі FAISS, яка забезпечує ефективний пошук релевантного контексту під час генерації відповіді в рамках RAG-підходу. Збереження бази у локальній директорії (vuln_index) дозволяє багаторазово використовувати її без необхідності повторного обчислення ембедінгів.

Таким чином, повний процес аналізу охоплює не лише генерацію відповідей, але й комплексну обробку, структуризацію та збереження результатів у зручному форматі для подальшого аналізу, тестування та візуалізації.

2.3.3. Застосування Chain-of-Thought reasoning для підвищення пояснюваності

Одним із викликів у застосуванні великих мовних моделей (LLM) для аналізу коду є непояснюваність результатів. Навіть при високій точності, розробникам і аудиторам часто не вистачає аргументованих пояснень – чому саме модель вважає певний фрагмент коду уразливим.

Для подолання цієї проблеми в останні роки активно використовується техніка, відома як Chain-of-Thought (CoT) reasoning – “ланцюжок міркувань”. Цей підхід передбачає, що мовна модель не просто видає відповідь, а поетапно формує логічний ланцюжок роздумів, який веде до результату. Таким чином, результат є не просто "висновком", а аргументованим поясненням.

Приклад:

Запит: Чи містить цей код потенційну reentrancy-вразливість?

Міркування:

1. Функція здійснює зовнішній виклик через call().

2. Баланс користувача оновлюється після зовнішнього виклику.

3. Це порушує шаблон "checks-effects-interactions".

→ Висновок: так, функція містить reentrancy-вразливість.

Застосування в рамках дипломної роботи

У реалізованій системі з LLM при генерації результатів було протестовано два підходи:

- Без CoT: модель одразу повертає відповідь у форматі JSON.

- З CoT: у prompt включено інструкцію пояснювати логіку аналізу, наприклад:

"Проаналізуй код і поясни крок за кроком, чому в ньому є або немає уразливостей, перш ніж сформувати відповідь."

Результати показали, що точність відповідей залишалася стабільною, проте зрозумілість та довіра до результатів значно зросли. Особливо це помітно у випадках із логічними або контекстно-залежними вразливостями.

Переваги CoT reasoning:

- Покращує інтерпретованість результатів;
- Дозволяє виявити логічні помилки самої моделі;
- Є базою для подальшої інтеграції з Explainable AI (XAI)
- Сприяє навчанню розробників – пояснення можна використовувати як освітні приклади.

Обмеження:

- Збільшує довжину відповіді;
- Вимагає додаткової обробки тексту перед витягом JSON-даних;
- Не завжди дає чітку структуру – залежить від prompt engineering.

Таким чином, включення CoT reasoning у систему аналізу смарт-контрактів дозволяє підвищити довіру до висновків AI та зробити їх більш придатними для використання в реальних аудитах.

2.4. Описання та обґрунтування алгоритмів та результатів дослідження

2.4.1. Алгоритм виявлення вразливостей у смарт-контрактах за допомогою LLM

У даній роботі основною метою є створення системи, яка здатна автоматично

аналізувати код смарт-контрактів, написаних мовою Solidity, та знаходити потенційні вразливості ще до їх експлуатації зловмисниками. Це має особливу важливість у контексті зростаючої кількості атак у сфері децентралізованих фінансів (DeFi) та блокчейн-платформ загалом. Для вирішення поставленої задачі використовується велика мовна модель (LLM) від OpenAI, що здатна аналізувати код не лише з синтаксичної, а й з семантичної точки зору. Взаємодія з моделлю здійснюється через спеціально сконструйовані запити, що включають інструкції та приклади.

Розроблений алгоритм пошуку вразливостей включає наступні ключові етапи:

- попереднє зчитування та базова обробка коду смарт-контракту, включаючи вилучення основних структур, функцій і модифікаторів;

- створення запиту до LLM, який містить формалізовані інструкції щодо аналізу, а також one-shot або few-shot приклади типових вразливостей (наприклад, reentrancy, integer overflow, unchecked call return value);

- передача згенерованого запиту до мовної моделі;

- отримання відповіді у структурованому форматі JSON, де кожна знайдена вразливість має вказаний номер рядка, проблемний фрагмент коду, класифікацію вразливості, ступінь небезпеки, розгорнуте пояснення та пропозиції щодо усунення.

Таким чином, LLM виконує роль інтелектуального аналізатора коду, здатного виявити широкий спектр проблем – від синтаксичних і стилістичних недоліків до критичних логічних помилок, які можуть бути експлуатовані в реальному середовищі. Наприклад, модель ефективно розпізнає reentrancy-уразливості, аналізуючи порядок змін стану контракту перед

здійсненням зовнішнього виклику. Крім того, модель виявляє потенційні помилки авторизації, неправильно ініціалізовані змінні, порушення інваріантів та інші аспекти, які складно виявити за допомогою традиційного статичного аналізу. Застосування LLM дозволяє значно підвищити рівень автоматизації безпеки у процесі розробки смарт-контрактів.

2.4.2. Архітектура системи з використанням векторного представлення коду та шаблонів вразливостей

З метою покращення точності виявлення вразливостей та зменшення кількості помилкових спрацьовувань, у даній роботі реалізовано компонент retrieval-augmented generation (RAG). Цей підхід базується на поєднанні генеративних можливостей великих мовних моделей із механізмами пошуку релевантної інформації у попередньо підготовленій базі знань. Завдяки цьому модель отримує додатковий контекст, який дозволяє їй робити більш точні й обґрунтовані висновки щодо аналізованого коду.

Система побудована за модульною архітектурою з використанням таких компонентів:

- кожен приклад вразливості структуровано у вигляді шаблону (template), який включає приклад коду, короткий опис уразливості, її категорію та спосіб усунення;

- всі шаблони векторизуються за допомогою попередньо натренованих OpenAI Embeddings, що дозволяє перевести їх у числове представлення для подальшого пошуку;

- отримані вектори організовуються у базу FAISS – швидкий та ефективний індекс для пошуку схожих векторів у великих наборах даних;

- при кожному запиті відбувається побудова векторного представлення коду, який потрібно проаналізувати, після чого

здійснюється пошук найбільш релевантних шаблонів у базі;

- знайдені шаблони інтегруються у prompt разом із основним кодом, створюючи повний запит, що надсилається до LLM.

Ключова перевага цього підходу полягає в тому, що модель не "вигадує" відповідь з нуля, а має змогу опиратись на приклади, які були попередньо перевірені й структуровані експертами або зібрані з надійних джерел. Це дозволяє підвищити точність аналізу, зробити відповіді більш обґрунтованими та зменшити ризик пропущених або неправильно класифікованих вразливостей.

Крім того, такий підхід забезпечує гнучкість у налаштуванні системи: база шаблонів може доповнюватися новими кейсами без потреби перенавчання моделі, що робить систему адаптивною до нових типів атак і дозволяє легко масштабувати її функціональність.

2.4.3. Використання prompt engineering та few-shot learning для покращення точності аналізу

Особливу увагу в рамках цієї роботи було приділено побудові ефективних prompt-ів (запитів) для взаємодії з LLM. Зміст і структура цих запитів мають вирішальне значення для того, як модель інтерпретує поставлене завдання та які результати повертає. Запити до моделі формувалися з урахуванням принципів контекстуального навчання і включали чіткі інструкції щодо аналізу коду, а також приклади (few-shot) типових уразливостей разом із поясненнями і варіантами їх усунення. Це дає змогу моделі орієнтуватися у форматі відповіді, стилі викладу і типових шаблонах, які слід розпізнавати.

Було проведено серію експериментів для визначення впливу різних типів prompt-ів на якість виявлення вразливостей. Серед протестованих варіантів були:

- прості запити без прикладів (zero-shot), які передбачають повну генерацію відповіді лише на основі інструкції;

- prompt-и з одним прикладом (one-shot), що дають мінімальний контекст для орієнтації моделі;

- prompt-и з трьома повноцінними прикладами різних типів вразливостей, кожен з яких включав код, опис проблеми, ступінь ризику та запропоноване виправлення.

У процесі тестування виявлено, що саме few-shot варіант з трьома прикладами демонструє найвищі показники точності (accuracy) і повноти (recall). Крім цього, окремо досліджувалося, як порядок розміщення прикладів впливає на результат. Виявилося, що побудова запиту за принципом від простого до складного дозволяє моделі краще зрозуміти логіку класифікації й типізацію помилок, а також генерувати більш релевантні висновки.

Також було помічено, що додавання до запиту невеликих підказок – наприклад, коментарів до прикладів, вказівок на тип вразливості або навіть підкреслення потенційно небезпечного коду – додатково підвищує ефективність моделі. У сукупності всі ці фактори свідчать про важливість prompt engineering як критичного компоненту системи AI-аналізу безпеки коду.

2.4.4. Обговорення отриманих результатів і їх верифікація

У процесі експериментального дослідження було протестовано кілька конфігурацій промптів та мовних моделей, серед яких GPT-4o-mini, GPT-3.5-turbo та o4-mini. Для кожної з моделей використовувалися різні підходи: BASIC, FEWSHOT, SIMPLE, а також варіації з Retrieval-Augmented Generation (+RAG). Метою тестування було оцінити ефективність виявлення вразливостей у Solidity-контрактах за допомогою різних архітектур запитів.

Верифікація результатів здійснювалася на основі метрик: TP (істинно позитивні), FP (хибно позитивні), FN (хибно негативні), TN (істинно негативні), що дозволило обчислити загальні показники Accuracy (точність), Recall (повнота) та F1-score.

Найкращі результати за F1-метрикою показали конфігурації:

- **GPT-3.5-turbo + SIMPLE+RAG: F1 = 0.61, Recall = 0.49**
- **GPT-4o-mini + BASIC+RAG: F1 = 0.60, Recall = 0.46**
- **GPT-4o-mini + SIMPLE+RAG: F1 = 0.60, Recall = 0.46**
- **o4-mini + FEWSHOT+RAG: F1 = 0.62, Recall = 0.45**

Ці результати свідчать про значний вплив додаткового контексту (RAG) на здатність моделі виявляти вразливості. Порівняно з базовими підходами, конфігурації з +RAG покращують як повноту, так і загальну збалансованість виявлення проблем (F1).

Особливо помітним є прогрес моделей типу o4-mini, які досягають високих показників навіть без значної кількості хибно позитивних відповідей (наприклад, 0 FP у FEWSHOT+RAG).

Загалом, аналіз результатів демонструє стабільне покращення в роботі моделей за рахунок інженерії промптів та додавання контекстуальних знань через RAG.

2.5. Аналіз результатів

Перед представленням детальних результатів варто пояснити використані в таблицях метрики. TP (True Positives) – це кількість правильно ідентифікованих вразливостей; FP (False Positives) – хибні спрацювання, коли модель помилково виявила вразливість там, де її не було; FN (False Negatives) – вразливості, які модель не змогла виявити; TN (True Negatives) – правильно класифіковані безпечні фрагменти коду. З цих метрик обчислюються наступні показники: Accuracy (точність) – частка правильних відповідей серед усіх;

Recall (повнота) – частка виявлених вразливостей серед усіх реальних вразливостей; F1-score – гармонійне середнє між точністю та повнотою.

Для прикладу, якщо модель має Recall = 0.50 та Precision = 0.60 (який можна отримати по формулі (як TP / (TP + FP)), то F1-score обчислюється за формулою:

$F1 = 2 \times (Precision \times Recall) / (Precision + Recall) = 2 \times (0.60 \times 0.50) / (0.60 + 0.50) \approx 0.545$

Цей показник дозволяє отримати більш збалансовану оцінку ефективності моделі в умовах, коли важливе не лише покриття, але й точність передбачень.

У цьому підрозділі проведено детальний аналіз точності моделей на основі отриманих метрик. Було протестовано 18 різних конфігурацій (6 промптів × 3 моделі), де для кожної моделі окремо вимірювалися значення Accuracy, Recall та F1-score.

Результати тестування моделей для аналізу коду

Accuracy для gpt-4o-mini

Prompt Type	Accuracy	Recall	F1
BASIC	0.55	0.48	0.59
BASIC+RAG	0.58	0.46	0.60
FEWSHOT	0.51	0.35	0.49
FEWSHOT+RAG	0.55	0.45	0.57
SIMPLE	0.49	0.34	0.48
SIMPLE+RAG	0.58	0.46	0.60

Кількість результатів для gpt-4o-mini

Prompt Type	TP	FP	FN	TN
BASIC	31	9	34	21
BASIC+RAG	30	5	35	25
FEWSHOT	23	5	42	25
FEWSHOT+RAG	29	7	36	23
SIMPLE	22	5	43	25
SIMPLE+RAG	30	5	35	25

Accuracy для gpt-3.5-turbo

Prompt Type	Accuracy	Recall	F1
BASIC	0.52	0.37	0.51
BASIC+RAG	0.43	0.25	0.37
FEWSHOT	0.49	0.29	0.44
FEWSHOT+RAG	0.55	0.40	0.55
SIMPLE	0.52	0.34	0.49
SIMPLE+RAG	0.57	0.49	0.61

Кількість результатів для gpt-3.5-turbo

Prompt Type	TP	FP	FN	TN
BASIC	24	5	41	25
BASIC+RAG	16	5	49	25
FEWSHOT	19	2	46	28
FEWSHOT+RAG	26	4	39	26
SIMPLE	22	3	43	27
SIMPLE+RAG	32	8	33	22

Accuracy для o4-mini

Prompt Type	Accuracy	Recall	F1
BASIC	0.59	0.43	0.59
BASIC+RAG	0.57	0.38	0.55
FEWSHOT	0.55	0.37	0.53
FEWSHOT+RAG	0.62	0.45	0.62
SIMPLE	0.58	0.43	0.58
SIMPLE+RAG	0.60	0.43	0.60

Кількість результатів для o4-mini

Prompt Type	TP	FP	FN	TN
BASIC	28	2	37	28
BASIC+RAG	25	1	40	29
FEWSHOT	24	2	41	28
FEWSHOT+RAG	29	0	36	30
SIMPLE	28	3	37	27
SIMPLE+RAG	28	1	37	29

У випадку GPT-4o-mini, спостерігається чітке покращення при використанні підходу +RAG у кожному з режимів промптів. Особливо це помітно у SIMPLE+RAG та BASIC+RAG, які досягають найвищого F1-score (0.60). Це свідчить про збалансовану здатність виявляти вразливості при мінімальній кількості хибних спрацювань.

Для GPT-3.5-turbo можна побачити, що найбільш ефективною конфігурацією є SIMPLE+RAG, яка має найвищі показники як Recall (0.49), так і F1-score (0.61). Це свідчить про те, що навіть менш потужні моделі можуть досягати високих результатів за рахунок правильно підібраної інженерії промптів і додаткового контексту.

У моделі o4-mini найвищі значення точності та F1-score показує конфігурація FEWSHOT+RAG. Цікаво, що в цій конфігурації взагалі відсутні хибно позитивні спрацювання (FP = 0), що робить її надзвичайно привабливою для практичного застосування у виявленні вразливостей. Загалом модель демонструє високу стабільність результатів незалежно від типу промпту, що свідчить про її зрілість і надійність.

Таким чином, результати підтверджують гіпотезу про доцільність використання Retrieval-Augmented Generation та адаптивного підбору промптів для підвищення ефективності аналізу безпеки Solidity-контрактів. Особливо це актуально для нових моделей, таких як o4-mini та GPT-4o-mini.

3. ВИСНОВКИ

У рамках даної дипломної роботи було реалізовано інноваційну систему для глибокого аналізу смарт-контрактів, написаних мовою Solidity, яка базується на сучасних методах штучного інтелекту. Основна мета дослідження полягала у створенні гнучкого, масштабованого та ефективного інструменту, здатного автоматично виявляти потенційні вразливості в коді смарт-контрактів. Для цього було застосовано сучасні мовні моделі, зокрема великі мовні моделі (LLM), а також інтегровано підхід Retrieval-Augmented Generation (RAG), що дозволяє підвищити точність і релевантність аналізу за рахунок динамічного включення зовнішніх знань у процес обробки запиту. Запропонований підхід орієнтований на підвищення безпеки децентралізованих додатків, полегшення роботи аудиторам коду та автоматизацію процесів, що раніше потребували значних людських ресурсів і технічної експертизи.

В процесі виконання роботи було досягнуто наступних результатів:

- Побудовано архітектуру системи, яка поєднує в собі векторне представлення відомих вразливостей, шаблонів коду та можливість інтеграції з LLM для семантичного аналізу вхідного коду.

- Реалізовано два варіанти аналізу: зі звичайним запитом до LLM та з використанням механізму RAG, який підвищує якість відповідей за рахунок релевантного контексту.

- Проведено експерименти з використанням датасету з прикладів Solidity-контрактів, що містять типові вразливості, та порівняно точність виявлення помилок у різних режимах роботи системи.

Запропонований підхід продемонстрував високу ефективність у виявленні широкого спектру вразливостей у коді смарт-контрактів, включаючи як базові, так і складні логічні помилки, які часто залишаються непоміченими

при використанні традиційних інструментів аналізу. Ретельний аналіз експериментальних результатів показав, що використання RAG-механізму відіграє ключову роль у підвищенні якості виявлення вразливостей. Зокрема, спостерігалося суттєве покращення метрик точності: Recall та F1-score значно зросли в порівнянні з варіантами без використання додаткового контексту. Це свідчить про здатність системи ефективно виявляти загрози без істотного збільшення кількості хибно позитивних спрацювань. Крім того, застосування RAG дозволяє не лише точніше визначати потенційно небезпечні фрагменти коду, але й генерувати більш обґрунтовані пояснення для кожної виявленої проблеми, що підвищує інтерпретованість та практичну цінність системи для розробників і аудиторів смарт-контрактів.

Розроблена система має низку переваг:

- Гнучкість та масштабованість для додавання нових шаблонів та типів вразливостей.
- Можливість розширення підтримки інших мов програмування шляхом адаптації векторного індексу та навчального контексту.
- Використання стандартного API (OpenAI), що спрощує інтеграцію з іншими інструментами.

Подальші напрямки дослідження:

- Інтеграція Graph Neural Networks (GNN) для аналізу залежностей у коді та покращення виявлення логічних вразливостей.
- Дослідження мульти-модальних підходів, які поєднують текст, AST-структури, та графи потоку даних.
- Підтримка інших мов смарт-контрактів, таких як Vyper або Move.
- Використання інструкцій у природній мові для пояснення виявлених проблем, що покращить інтерпретованість системи.

Таким чином, результати роботи підтверджують доцільність використання AI-підходів у сфері безпеки смарт-контрактів та відкривають перспективи для подальшого розвитку даного напрямку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ConsenSys Diligence. "Top 10 Smart Contract Vulnerabilities." Medium, medium.com/consensys-diligence/top-10-smart-contract-vulnerabilities
2. OpenZeppelin. "Secure Smart Contract Guidelines." OpenZeppelin Blog, blog.openzeppelin.com/secure-smart-contract-guidelines
3. Koczwara, Marcin. "Using GPT-4 to Analyze Code for Vulnerabilities." Medium, medium.com/@mckoczwara/analyzing-code-with-gpt-4
4. Raschka, Sebastian. "Prompt Engineering with GPT-4: Tips and Techniques." Sebastian Raschka's Blog, sebastianraschka.com/blog/2023/prompt-engineering.html
5. "Smart Contract Weakness Classification Registry (SWC)." SWC Registry, swcregistry.io
6. "Retrieval-Augmented Generation (RAG): A Gentle Introduction." Towards Data Science, Medium, towardsdatascience.com/retrieval-augmented-generation-rag-explained