МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
(повне найменування факультету)
Кафедра інформаційної безпеки та наноелектроніки
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістр
(ступінь вищої освіти)

на тему Дослідження та розробка автоматизованих тестів
(назва теми)
смарт-контрактів Marketplace застосунку

Виконала: студентка 2 курсу, групи БК-813м

Спеціальності 125 Кібербезпека та захист
(код і найменування спеціальності)
Інформації

Освітня програма (спеціалізація)
Безпека інформаційних і комунікаційних систем

СИНЮШКІНА М.Д.
(ПРІЗВИЩЕ та ініціали)

Керівник НЕЛАСА Г.В.
(ПРІЗВИЩЕ та ініціали)
Рецензент САМОЙЛИК С.С.
(ПРІЗВИЩЕ та ініціали)

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
Кафедра інформаційної безпеки та наноелектроніки
Ступінь вищої освіти магістр
Спеціальність 125 Кібербезпека та захист інформації
(код і найменування)
Освітня програма (спеціалізація) Безпека інформаційних і комунікаційних систем
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІБтаН, к.ф.-м..н, доц.
Андрій Коротун
«_____» _____ 2024 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

СИНЮШКІНОЇ Марії Дмитрівни
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та розробка автоматизованих тестів смарт-контрактів marketplace застосунку

керівник проєкту (роботи) к.т.н., доцент НЕЛАСА Ганна Вікторівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від «05» грудня 2024 року №507

2. Строк подання студентом проєкту (роботи) 01 грудня 2024 року

3. Вихідні дані до проєкту (роботи) розробка юніт-тестів та статичного аналізатора коду для смарт-контрактів із використанням інструментів автоматизації тестування (GitHub Actions, Mocha, Chai, Truffle)

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз сучасних підходів тестування смарт-контрактів 2. Порівняльний аналіз та вибір оптимальних інструментів для тестування 3. Реалізація процесів тестування та безпеки для смарт-контрактів

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Презентація доповіді підготовлена у Microsoft Word

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
Розділи 1-3	НЕЛАСА Г.В., к.т.н. доцент каф. ІБтаН	01.10.2024	01.12.2024
Нормоконтроль	КОРОЛЬКОВ Р.Ю., к.т.н., доцент каф. ІБтаН		

7. Дата видачі завдання «01» _жовтня_____2024_року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	**Примітка**
1	Постановка завдання роботи.	01.10.2024 – 08.10.2024	Виконано
2	Аналіз предметної області.	09.10.2024 – 14.10.2024	Виконано
3	Аналіз вразливостей смарт-контрактів	15.10.2024 – 30.10.2024	Виконано
4	Програмна реалізація.	31.10.2024 – 14.11.2024	Виконано
5	Оформлення пояснювальної записки та відповідної документації.	15.11.2024 – 30.11.2024	Виконано
6	Нормоконтроль та рецензування.	01.12.2024 – 20.12.2024	Виконано
7	Захист дипломної роботи.	23.12.2024	Виконано

Студент(ка) _____ _Марія СИНЮШКІНА__
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ __Ганна НЕЛАСА_____
(підпис) (Ім'я ПРІЗВИЩЕ)

АНОТАЦІЯ

Пояснювальної записка до дипломного проєкту: 62 стор., 2 рис., 5 додатків, 20 джерел.

АВТОМАТИЗАЦІЯ, БЛОКЧЕЙН, СМАРТ-КОНТРАКТ, СТАТИЧНИЙ АНАЛІЗ, ТЕСТУВАННЯ, CHAI, CI/CD, GITHUB ACTIONS, UNIT-ТЕСТУВАННЯ, WEB3.JS

Об’єкт дослідження — процес верифікації та тестування смарт-контрактів, написаних мовою програмування Solidity, з використанням статичного аналізу коду та автоматизованого тестування.

Предмет дослідження — методи статичного аналізу смарт-контрактів, підходи до їх тестування, а також інструменти для автоматизації цих процесів, зокрема бібліотеки Chai і Web3.js.

Мета роботи — розробка та впровадження статичного аналізатора коду, а також автоматизованих тестів для смарт-контрактів, з використанням інструментів JavaScript та GitHub Actions.

Методи дослідження — програмування на JavaScript, модульне тестування з використанням Chai, статичний аналіз коду Solidity, CI/CD з GitHub Actions, а також бібліотека Web3.js для інтеграції зі смарт- контрактами.

В ході роботи було досліджено сучасні підходи до статичного аналізу смарт-контрактів та їх тестування. Реалізовано статичний аналізатор коду та набір автоматизованих модульних тестів для верифікації роботи смарт-контрактів.

Найефективнішими виявилися тести, які дозволили виявити критичні помилки в логіці контрактів, а статичний аналізатор забезпечив перевірку коду на відповідність стандартам безпеки Solidity.

ABSTRACT

Explanatory note to the master's thesis: 62 p., 2 figures, 5 appendices, 20 sources.

AUTOMATION, BLOCKCHAIN, CHAI, CI/CD, CODE SECURITY, GITHUB ACTIONS, SMART CONTRACTS, STATIC ANALYSIS, TESTING, UNIT TESTING, WEB3.JS

The object of research is the process of verification and testing of smart contracts written in the Solidity programming language using static code analysis and automated testing.

The subject of the research is the methods of static code analysis, approaches to testing smart contracts, and tools for automating these processes, particularly the Chai and Web3.js libraries.

The aim of the work is to develop and implement a static code analyzer and a set of automated unit tests for verifying the functionality of smart contracts using JavaScript and GitHub Actions tools.

To implement the research, programming in JavaScript, unit testing with Chai, static analysis of Solidity code, CI/CD with GitHub Actions, and the Web3.js library for smart contract integration were used.

During the course of the work, modern approaches to static analysis and testing of smart contracts were researched. A static code analyzer and a set of automated unit tests were implemented to verify the functionality of smart contracts.

The most effective tests proved to be those that allowed the identification of critical errors in contract logic, while the static analyzer ensured code compliance with Solidity security standards.

ЗМІСТ

Анотація 4

Abstract 5

Перелік скорочень 7

Вступ 9

1 Аналіз сучасних підходів тестування 12

1.1 Принципи взаємодії смарт-контрактів у блокчейн технології та децентралізація 12

1.2 Процес верифікації смарт-контрактів 17

1.3 Роль тестування у підвищенні надійності та безпеки Web3 Marketplace 20

додатків 20

2 Порівняльний аналіз та вибір оптимальних інструментів для тестування 25

2.1 Використання Web3JS бібліотеки та Solidity для реалізації роботи смарт-контрактів 25

2.2 Аналіз інструментів впровадження автоматизованого тестування 28

3 Реалізація процесів тестування для смарт-контрактів 31

3.1 Реалізація юніт-тестів для смарт-контрактів 31

3.2 Реалізація статичного аналізу смарт-контрактів 36

3.3 Налаштування CI/CD процесу 38

Висновки 45

Перелік джерел посилання 47

Додаток А 50

Додаток Б 54

Додаток В 56

Додаток Г 57

Додаток Д 60

ПЕРЕЛІК СКОРОЧЕНЬ

AR - Augmented Reality (доповнена реальність)

BEC – Batch Overflow Exploit (експлойт переповнення пакетів)

Business Process Management System (система управління бізнес-процесами)

CI/CD - Continuous Integration / Continuous Deployment (безперервна інтеграція та розгортання)

DAO - Decentralized Autonomous Organization (децентралізована автономна організація)

DApps – Decentralized Applications (децентралізовані додатки)

DeFi – Decentralized Finance (децентралізовані фінанси)

IPFS - InterPlanetary File System (міжпланетна файлова система)

IPC – Inter-Process Communication (міжпроцесна взаємодія)

NFT - Non-Fungible Token (невзаємозамінний токен)

VR - Virtual Reality (віртуальна реальність)

ВСТУП

З роками концепція Web 3.0 проходила крізь етапи еволюції, від початкової ідеї про семантичну мережу до більш децентралізованої версії Інтернету, побудованої за допомогою блокчейну. Концепція Web 3.0 вперше з'явилася в 1990-х роках. Бернерс-Лі назвав це семантичною мережею. Архітектура нового покоління повинна містити кілька основних компонентів. Однією з головних ідей Web 3.0 є переклад усього веб-контенту, написаного людськими мовами, у машиночитану форму, що дозволить алгоритмам і програмам визначати значення повідомлень і встановлювати з’єднання на їх основі.

Пізніше розуміння цього поняття у людей змінилося. У 2014 році співзасновник Ethereum Гевін Вуд опублікував статтю, в якій описав Web 3.0 з нової точки зору — більш децентралізовану версію Інтернету, побудовану за допомогою блокчейну. Його рекомендації в основному стосуються змін систем зберігання даних і підвищення анонімності користувачів.

У 2021 році, на тлі зростання популярності децентралізованих програм і NFT, термін Web 3.0 буде згадуватися знову. Під час обговорення інтернет-шторму став популярним і термін «Web3».

Важливою відмінністю між Web 3.0 і Web 2.0 є підвищений ступінь децентралізації на всіх рівнях, включаючи зберігання даних і використання програм. Програми епохи Web3 часто називають програмами, які мають один або кілька таких атрибутів:

- блокчейн і смарт-контракти інтегровані в різні функції продукту;
- вихідний код програми відкритий для громадськості, і сторонні розробники мають можливість;

- сервіс використовує технологію віртуальної (VR) або доповненої (AR) реальності;
- платіжні інструменти, які працюють з криптовалютами, вбудованими у зовнішній інтерфейс;
- використовуйте незамінні токени (NFT);
- система зберігання даних використовує протокол IPFS;
- децентралізована автономна організація (DAO) бере участь в управлінні проектом.

Використання технології блокчейн означає, що мережа рівною мірою буде належати кожному користувачеві та розробнику. Ніхто не може вкрасти чи скопіювати те, що ви створюєте.

У Web3 кожен вміст є цифровим активом. Ця концепція вже реалізована, і більшість людей чули про неї як про NFT. Ніхто інший не може завантажити вашу фотографію та продати її для реклами у своєму блозі, але вони можуть купити у вас права на цю фотографію, якщо захочуть. Завдяки Web3 більше нікому не ділитеся своїми конфіденційними даними. Навіть купувати або продавати банкам. З цієї причини з’явилися криптовалюти – ще одне відоме та всюдисуще явище. Платежі в криптовалюті пропонують сторонам свободу, якої не існувало в Web 2.0 – не потрібно нікому звітувати, хто це робить і на що витрачаються гроші.

При розробці додатків «нового покоління» дотримуються децентралізації організації даних, у тому числі їх зберігання. Децентралізація, у свою чергу, є ключем до успішного впровадження криптовалют і смарт-контрактів в економіку: усуває потребу в довірі, а отже, в посередниках і централізованих структурах.

Тому ідеальний стан Web3 – це ефективна бізнес-модель, яка не використовує ієрархічні структури та традиційні фінансові інструменти.

Метою є дослідження, розробка та реалізація автоматичних тестів для web3 Marketplace додатку з використанням мови програмування Solidity для створенних смарт-контрактів на платформі Ethereum, а також з використанням JavaScript та допоміжних інструментів для взаємодії та реалізації безперервної доставки та деплою. Мета полягає в демонстрації можливостей тестування для запобігання критичним помилкам у смарт-контрактах блокчейну та створення безпечного й ефективного електронного маркетплейсу на основі децентралізованих принципів. Для досягнення цієї мети необхідно виконати наступні завдання:

- зробити аналіз web3 як новітньої концепції;
- розглянути підхід до тестування смарт-контрактів;
- розробити смарт-контракти на мові Solidity, які дозволяють користувачам додавати товари;
- створити автоматизовані тести та аналізатор;
- впровадити CI/CD
- зробити висновки на основі дослідження;

За результатами роботи буде розроблено набір тестів для перевірки смарт-контрактів, що допоможе мінімізувати кількість помилок у коді, підвищити його надійність та безпеку. Також буде створено статичний аналізатор коду для автоматичного виявлення потенційних вразливостей, що сприятиме побудові більш безпечного та ефективного електронного маркетплейсу на основі децентралізованих технологій.

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ТЕСТУВАННЯ СМАРТ-КОНТРАКТІВ

1.1 Принципи взаємодії смарт-контрактів у блокчейн технології та децентралізація

Блокчейн існує вже понад десятиліття як перевірена технологія для запису транзакцій у децентралізованій, рівноправній мережі за допомогою розподіленої бази даних [1,2]. Це розглядається як розподілена парадигма обчислень, яка вирішує проблему довіри до одного суб'єкта. Таким чином, кілька вузлів працюють разом в мережі блокчейну для безпечного та надійного утримання розподіленого реєстру всіх минулих транзакцій. Сатоші Накамото запустив Біткоїн у 2008 році [3], який був першою запропонованою криптовалютою для представлення блокчейну як розподіленої інфраструктурної платформи. Це дало можливість будь-кому відправляти та отримувати біткоїни, вид криптовалюти, не потрібною залежністю від будь-якого центрального органу. Також були запропоновані інші системи на основі блокчейну для використання з монетою, включаючи Hyperledger Fabric [4] та Ethereum [5]. З використанням смарт-контрактів, на відміну від Біткоїна, стає можливим застосування технології блокчейну за межами звичайних контрактів шляхом автоматичного виконання умов угоди між двома чи більше особами в децентралізованому середовищі, якщо необхідні обставини задовольнені .

Із розвитком блокчейну смарт-контракти стали популярнішими. Смарт-контракт - це передова технологія, яку можна використовувати в екосистемі блокчейну для автоматичного ведення переговорів, виконання та забезпечення умов юридично обов'язкової угоди. Зменшення ризику, зниження вартості послуг та адміністрування, а також підвищена ефективність бізнес-процесів - це лише деякі з переваг смарт-контрактів перед звичайними. Зокрема, смарт-

контракти можуть викликати довіру між сторонами в умовах контрактів без довіри. У цьому відношенні вони можуть революціонізувати традиційні способи ведення бізнесу [6].

Технологія розподіленого реєстру (блокчейн) та смарт-контракти мають багато потенційних застосувань за межами фінансового сектору, включаючи обробку страхових випадків, управління ланцюгом постачання та забезпечення прав інтелектуальної власності. Використання смарт-контрактів та блокчейн-застосунків серед підприємств не вказує на зменшення темпів зростання. Однак застосунки блокчейну повинні бути уважно створені та ретельно протестовані для відповідності високим стандартам безпеки, масштабованості та ефективності. Це особливо важливо при розгляді нестандартних життєвих циклів програмного забезпечення, які використовуються при розробці смарт-контрактів, що ускладнює можливість оновлення або виправлення недоліків у встановлених програмах шляхом випуску оновленої версії програми. Створення смарт-контрактів відрізняється від традиційних форм інженерії програмного забезпечення і має свій унікальний набір труднощів. Наприклад, розробники повинні гарантувати безпеку коду для смарт-контрактів через незмінюваність блокчейну та чутливий характер цифрової інформації, якою часто керують, і вони повинні приділяти увагу споживанню газу через застосування механізму газу при виконанні смарт-контрактів у платформах блокчейну, таких як Ethereum. Життєвий цикл продукту та процес розробки програм смарт-контрактів та блокчейн-застосунків повинен враховувати унікальні обмеження та особливості, вимагані технологією блокчейну. Необхідно ідентифікувати та досліджувати всі ці зміни в продуктах програмного забезпечення та процесах, щоб створити повний корпус знань, базований на інженерії програмного забезпечення для блокчейну.

Смарт-контракти є значущим розвитком у сфері блокчейну. Смарт-контракти були вперше запропоновані в 1990-х як цифровий протокол транзакцій для виконання умов угоди. Смарт-контракти просто є контейнерами коду, які інкапсулюють та реплікують умови реальних угод у цифровому просторі. Угоди фундаментально є правовими зобов'язаннями між двома чи більше сторонами, кожна з яких зобов'язана виконати свої зобов'язання. Важливою є те, що угода повинна бути виконуваною за законом, часто через централізований правовий орган (організацію). Тим не менш, смарт-контракти замінюють довірених посередників чи посередників між контрактуючими сторонами. Вони використовують це за допомогою виконання коду, який автоматично розповсюджується та перевіряється вузлами мережі у децентралізованому блокчейні. Крім того, вони дозволяють здійснювати транзакції між ненадійними сторонами без необхідності прямого контакту між сторонами, залежності від третіх сторін та витрат на посередництво.

Порівняно з традиційними контрактами, смарт-контракти пропонують переваги у зменшенні ризику транзакцій, зниженні витрат на обслуговування та послуги, а також підвищенні ефективності корпоративних процесів, оскільки вони часто розташовані на та захищені блокчейном. У цьому відношенні передбачається, що смарт-контракти нададуть краще рішення для поточного механізму транзакцій у різних галузях бізнесу.

Блокчейн технологія заснована на принципі децентралізації, що передбачає відсутність центрального органу чи контролю, що вигідно відрізняє її від традиційних централізованих систем. У децентралізованих мережах прийняття рішень та обробка інформації розподілені серед всіх учасників мережі. Такий підхід зробив системи більш стійкими до зовнішніх втручань і цензури.

У випадку смарт-контрактів, їх виконання та функціонування базуються на принципах децентралізації. Коли смарт-контракт розгортається на блокчейні, він стає доступним для всіх учасників мережі, а його виконання стає відповідальністю розподіленої мережі вузлів. Це виключає можливість недобросовісних або зловживаючих дій з боку окремих учасників, оскільки всі події фіксуються і підтверджуються децентралізованою системою [7].

Одним з важливих аспектів децентралізації є незалежність від централізованих посередників або посередницьких організацій. У традиційних угодах часто використовується довіра до централізованого органу або правової системи для забезпечення виконання умов. Смарт-контракти, натомість, працюють в децентралізованому середовищі, уникнувши необхідності довіряти посередникам. Код смарт-контракту самостійно визначає та автоматично виконує умови угоди, зменшуючи ризик невиконання зобов'язань.

Процес роботи смарт-контрактів.

1. Сторони погоджуються з умовами. Створення смарт-контракту починається з угоди. Сторони, які бажають провести транзакцію чи обміняти товари чи послуги, повинні домовитися про умови угоди. Також сторонам треба вирішити, як працюватиме смарт-контракт, включаючи умови, які повинні бути виконані для виконання контракту, та чи він буде виконуватися автоматично.

2. Створюється смарт-контракт. Транзактуючі сторони мають кілька варіантів для створення смарт-контракту, від його власноручного кодування до співпраці з розробником смарт-контрактів. Умови угоди перекладаються на мову програмування для створення смарт-контракту, який визначає правила та наслідки так само, як традиційний юридичний контракт. Створення смарт-контракту може бути простим, але важливо зауважити, що погано розроблений

смарт-контракт є серйозним ризиком для безпеки. Важливо повністю перевірити його безпеку на цьому етапі.

3. Смарт-контракт розгортається. Коли безпечно розроблений смарт-контракт готовий, наступним кроком є його розгортання на блокчейн. Смарт-контракт транслюється на блокчейн так само, як будь-яка інша криптотранзакція, з кодом смарт-контракту, включеним у поле даних транзакції. Смарт-контракт активний на блокчейні після підтвердження транзакції і його не можна анулювати чи змінити. Ця частина є важливою. Розгортання смарт-контракту на блокчейні подібно до купівлі товару і свідомого викидання чека. Немає можливості повернення, повернення грошей чи обміну - без винятків.

4. Виконуються сприятливі умови. Смарт-контракт працює, спостерігаючи за блокчейн або іншим надійним джерелом інформації на предмет певних умов або сприятливих умов. Ці умови можуть включати практично будь-що, що можна перевірити цифрово - досягнута дата, завершений платіж, отримана щомісячна рахунок чи будь-яка інша перевірна подія. Умови сприятливих умов також можуть виконуватися, коли одна чи декілька сторін контракту виконують певну дію.

5. Виконується смарт-контракт. Коли умови сприятливих умов виконуються, смарт-контракт виконується. Смарт-контракт, який виконується автоматично, може виконати одну чи декілька дій, таких як переказ коштів продавцеві або реєстрація власності покупця на актив.

6. Результат контракту записується на блокчейні. *Виконання* смарт-контракту негайно транслюється на блокчейні. Мережа блокчейн перевіряє дії, виконані смарт-контрактом, записує його виконання як транзакцію і зберігає завершений смарт-контракт на блокчейні. Запис смарт-контракту, як правило, доступний для перегляду будь-кому в будь-який час.

1.2 Процес верифікації смарт-контрактів

Смарт-контракти [8] мають першорядне значення для розробки та реалізації бізнес-процесу, що значною мірою сприяє використанню блокчейну в системах управління бізнес-процесами (BPMS). Правильність і безпека смарт-контрактів є обов'язковими, оскільки збої смарт-контрактів можуть призвести до втрати мільйонів доларів. Таким чином, блокчейн-додатки, засновані на смарт-контрактах, повинні бути перевірені та підтверджені, щоб забезпечити правильність, безпеку та безпеку реалізацій смарт-контрактів.

Перевірка правильності стосується дотримання специфікацій, які визначають, як користувачі можуть взаємодіяти зі смарт-контрактами та як смарт-контракти повинні поводитися при правильному використанні. Для перевірки правильності використовуються два підходи: формальна перевірка та правильність програмування (рис. 1.1). Формальні методи перевірки базуються на формальних методах (математичні методи), тоді як методи правильності програмування базуються на забезпеченні правильності програмування як коду, що означає, що програма виконується без входу в цикл і видає правильні виходи для правильні входи. В основному було приділено увагу підходам формальної перевірки, тому що формальна перевірка є більш ретельною та надійною.

З іншого боку, аспект гарантії безпеки також важливий, ніж аспект правильності. Крім того, смарт-контракти є незмінними, тому будь-які баги чи помилки стануть постійними після публікації та можуть призвести до величезних економічних втрат. Щоб уникнути цього, досліджуються методи виявлення вразливостей, які спрямовані на підвищення безпеки смарт-контрактів шляхом вивчення вразливостей шляхом перевірки смарт-контрактів на список уже визначених і добре відомих шаблонів уразливостей. Виявлення

вразливостей дозволяє уникнути тих самих помилок, що робить розумні контракти більш безпечними.

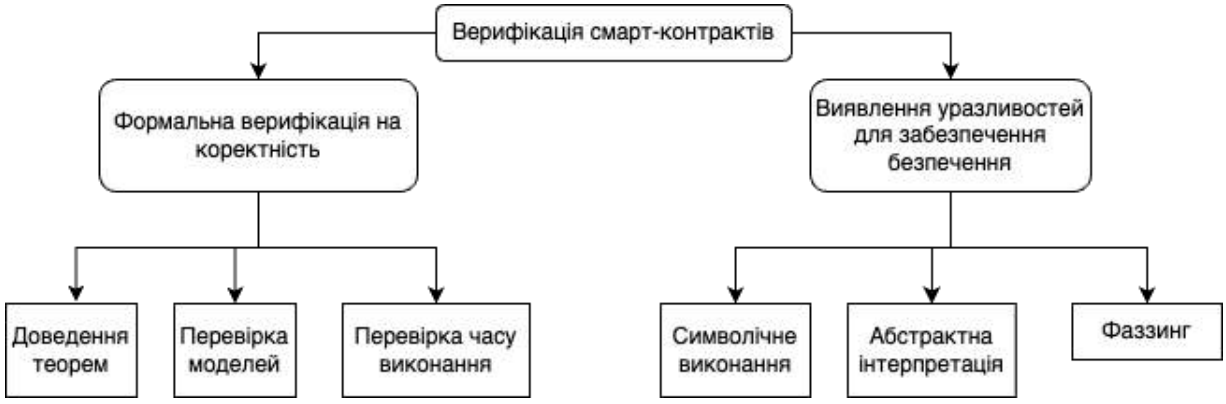

Рисунок 1.1 - Таксономія методів верифікації смарт-контрактів

Концепція смарт-контрактів у системах розподіленої книги вважається безпечним способом забезпечення виконання угод між сторонами-учасниками. Однак, на відміну від юридичних контрактів, які говорять про поведінку, яка має здійснюватися, і наслідки поведінки інакше, виконуваний код, вбудований в смарт-контракти, дає чіткі інструкції щодо досягнення відповідності. Отже, забезпечення правильності виконуваних кодів є важливим і складним завданням, оскільки інциденти можуть призвести до величезних фінансових втрат через помилки, порушення та недоліки в смарт-контрактах. Наприклад, як зазначено в статті [7], зловмиснику вдалося вичерпати понад 3,6 мільйона ефірів, скориставшись недоліком у вихідному коді смарт-контракту розподіленої автономної організації (DAO).

У вересні 2017 року мультипідписний гаманець Parity зазнав атаки в Ethereum, що призвело до розкрадання понад 150000 Ether (близько 30 мільйонів доларів). У квітні 2018 року через атаку BEC (Batch Overflow Exploit) було вкрадено близько 900 мільйонів доларів. Зіткнувшись із цими втратами

перевірка смарт-контрактів перед розгортанням на блокчейні привернула велику увагу.

Можна визначити принаймні три причини для застосування формальних специфікацій і перевірок до смарт-контрактів:

- по-перше, вразливості смарт-контрактів, через незмінну природу смарт-контрактів будь-які баги або помилки стануть постійними після публікації та можуть призвести до величезних втрат. Для цього потрібно забезпечити безпеку смарт-контрактів. Уразливості в смарт-контрактах призвели до кількох гучних експлойтів у технології блокчейн;

- по-друге, смарт-контракти часто є низькорівневими реалізаціями високорівневого робочого процесу, який містить стан машина з різними діями, що визначаються відповідним контролем доступу, щоб визначити, хто має дозвіл на виконання певної дії [7]. Відсутність у програмістів знань щодо семантики програмування, яка відображає робочі процеси високого рівня, може призвести до багатьох проблеми неправильного уявлення. В результаті отримуємо «несправедливі контракти», які є синтаксично технічно правильними, але не реалізують бажану бізнес-логіку. Таким чином, існує сильна потреба в мові специфікації високого рівня, щоб виразити мету робочого процесу в розумному контракті;

- по-третє, що стосується правильності, багато парадигм програмування, які використовуються для написання смарт-контрактів, не призначені для використання в контексті середовища блокчейн [9], що призводить до багатьох проблем під час виконання.

1.3 Роль тестування у підвищенні надійності та безпеки Web3 Marketplace додатків

Для забезпечення високої якості та надійності функціоналу web3 Marketplace додатку важливо використовувати систему тестів. Ця система охоплює різноманітні аспекти розгортання, взаємодії з продуктами та транзакціями на блокчейні [10]. Тести є необхідним елементом розробки, оскільки вони дозволяють виявити та усунути помилки ще на ранніх етапах, забезпечуючи високу якість продукту та надійність його функціонування.

Важливість тестування визначається кількома ключовими факторами.

1. Раннє виявлення помилок.Тести дозволяють ідентифікувати дефекти та недоліки в коді на ранніх етапах розробки, що сприяє значному зниженню витрат на виправлення помилок [11]. Оскільки блокчейн-транзакції незворотні, виправлення помилок після релізу може бути надзвичайно складним або навіть неможливим.

2. Перевірка інтеграції. Оскільки web3 Marketplace додаток працює на основі смарт-контрактів і взаємодіє з блокчейном, важливо протестувати правильну інтеграцію додатку з блокчейн-мережею. Наприклад, необхідно перевіряти, чи коректно відправляються і обробляються транзакції, чи виконуються всі смарт-контракти відповідно до вимог.

3. Забезпечення стабільності та безпеки. Блокчейн-додатки часто працюють з фінансовими активами або важливими даними, тому надійність і безпека є ключовими аспектами. Тести допомагають виявляти можливі вразливості або некоректну поведінку додатку при обробці транзакцій, доступу до даних або взаємодії з користувачами [12].

4. Автоматизоване тестування. Для децентралізованих додатків, які часто розгортаються на публічних блокчейнах, важливо автоматизувати

тестування на етапі розробки. Це включає не тільки юніт-тести, але й функціональні, інтеграційні та стрес-тести. Функціональні тести дозволяють переконатися, що всі бізнес-логічні процеси працюють коректно, тоді як стрес-тести оцінюють, як додаток поводиться під навантаженням або в нестандартних ситуаціях, таких як збої в мережі.

5. Юніт-тести. У процесі розробки web3 Marketplace додатку використовувалися юніт-тести для перевірки окремих компонентів програми або частин коду на коректність їх роботи. Юніт-тести ізолюють окремі частини програми та перевіряють їх роботу без взаємодії з іншими компонентами, що дозволяє швидко виявляти і виправляти помилки. Це забезпечує стабільність роботи всіх елементів додатку.

6. Тестування смарт-контрактів. Оскільки смарт-контракти є серцевиною блокчейн-додатків, їхнє тестування є критично важливим. Це включає перевірку коректності бізнес-логіки, захист від повторних атак (reentrancy), правильність обробки транзакцій і захист від переповнень. Завдяки такому підходу можна уникнути великих ризиків, пов'язаних з фінансовими втратами або компрометацією системи [13].

7. Контроль за оновленнями. Блокчейн-додатки зазвичай розгортаються на постійних мережах, що ускладнює внесення змін після релізу. Тому тестування відіграє ключову роль у тому, щоб будь-які зміни та оновлення, які впроваджуються до смарт-контрактів або децентралізованих додатків, були повністю перевірені до моменту їх впровадження.

8. Безперервна інтеграція (CI) та розгортання (CD). Важливо впровадити механізми безперервної інтеграції та розгортання для web3 Marketplace додатку. CI/CD забезпечує автоматичний запуск тестів при кожній зміні коду, що дозволяє швидко виявляти помилки та зберігати високу якість програмного

продукту. У випадку з децентралізованими додатками, це допомагає уникати проблем на етапі деплою смарт-контрактів у живій мережі (рис. 1.3).

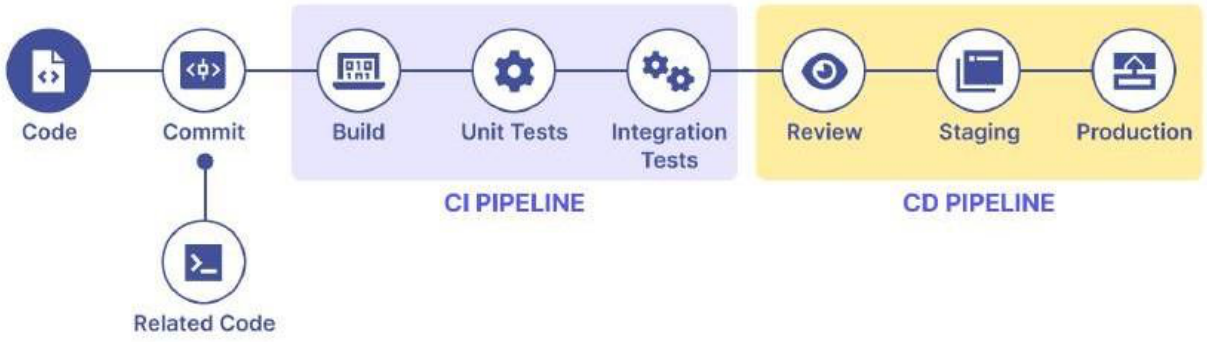

Рисунок 1.3 - Безперервна інтеграція (CI) та розгортання (CD)

Впровадження системи тестування у процес розробки дозволяє знизити ризики та забезпечити стабільну, ефективну роботу web3 Marketplace додатку в умовах реальних навантажень, гарантуючи безпеку та зручність для кінцевих користувачів.

Ще однією перевагою проведення регулярного тестування є підвищення довіри користувачів до додатку. Знання того, що система пройшла повний цикл перевірок і тестувань перед запуском, додає впевненості у її надійності. У випадку з додатками, що працюють з блокчейном та криптовалютами, де йдеться про безпеку транзакцій і збереження коштів, це має вирішальне значення для залучення користувачів і підтримки їх довіри [14].

Крім того, тестування сприяє більш швидкому та плавному впровадженню нових функцій. Завдяки автоматизованим тестам розробники можуть бути впевнені, що внесені зміни не порушать існуючий функціонал, що дозволяє прискорити процес розробки та оновлення додатку.

Смарт-контракти, як самовиконуючі угоди зписані прямо в блокчейн, стали невід'ємною частиною сучасних децентралізованих додатків. Їхня прозорість, автоматизація та безпека роблять їх привабливими для різноманітних галузей. Однак, складність їхньої логіки та високі ставки, пов'язані з фінансовими операціями, вимагають ретельного тестування.

Юніт-тестування є фундаментом для забезпечення якості смарт-контрактів. Воно дозволяє ізольовано перевіряти окремі функції контракту, гарантуючи їхню коректну роботу. Особливістю юніт-тестування смарт-контрактів є необхідність враховувати специфіку блокчейн-середовища, зокрема високі витрати на транзакції та незмінність коду після розгортання [15].

Інтеграційне тестування виходить за межі окремих функцій і перевіряє взаємодію смарт-контракту з іншими компонентами системи. Це дозволяє виявити помилки, пов'язані з передачею даних між різними контрактами або зовнішніми системами [16]. Інтеграційні тести часто проводяться в тестових мережах, щоб максимально наблизити умови до реального використання.

Функціональне тестування спрямоване на перевірку того, чи відповідає поведінка смарт-контракту очікуванням, визначеним у вимогах. Це включає в себе перевірку різних сценаріїв використання, обробку помилок та граничних умов. Метою функціонального тестування є забезпечення того, що смарт-контракт виконує всі необхідні функції.

Статичний аналіз коду є проактивним підходом до виявлення потенційних вразливостей у смарт-контрактах. За допомогою спеціалізованих інструментів можна автоматично перевіряти код на наявність типових вразливостей, таких як переповнення стеку, умови гонки та несанкціонований доступ до коштів. Цей метод дозволяє виявити проблеми ще на ранніх етапах розробки.

Нові підходи до тестування смарт-контрактів, такі як формальна верифікація та моделювання загроз, набирають все більшої популярності [17]. Формальна верифікація використовує математичні методи для доведення коректності роботи смарт-контракту, забезпечуючи високий рівень впевненості в його безпеці. Моделювання загроз дозволяє імітувати атаки зловмисників для виявлення потенційних вразливостей.

Комбінація різних методів тестування є ключовою для забезпечення високої якості та безпеки смарт-контрактів. Ретельне тестування допомагає запобігти фінансовим втратам, репутаційним ризикам та іншим негативним наслідкам, пов'язаним з помилками в коді смарт-контрактів.

2 ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА ВИБІР ОПТИМАЛЬНИХ ІНСТРУМЕНТІВ ДЛЯ ТЕСТУВАННЯ

2.1 Використання Web3JS бібліотеки та Solidity для реалізації роботи смарт-контрактів

Розгорнуті на блокчейні віртуальні активи, криптовалюти, програмовані токени та смарт-контракти є важливими складовими децентралізованих додатків (DApps). Однак для взаємодії з цими елементами on-chain, транзакції повинні бути створені на блокчейні. Вузол повинен передати транзакцію основній мережі пір-до-піра, щоб off-chain програма могла встановити її на блокчейні. Web3.js допомагає розробникам взаємодіяти з цими on-chain елементами, створюючи зв'язок з вузлами Ethereum.

Наразі Ethereum є основною платформою для децентралізованих додатків (dApps) та децентралізованих фінансів (DeFi). Її екосистема складається з базового блокчейну, великої кількості смарт-контрактів, розгорнутих на ньому широкого спектру цінних активів (насамперед, взаємозамінних і не взаємозамінних токенів, якими керують смарт-контракти), а також Ethereum Foundation, яка координує зусилля спільноти ентузіастів і компаній, що підтримують її. Хоча розробники роблять все можливе, щоб уникнути проблем з безпекою, навіть експерти виявляють або не помічають серйозні вразливості. Не дивно, що Ethereum стикається з атаками з високими ставками і втратами, тому реальні і потенційні вразливості є основним предметом занепокоєння. [9] Така ситуація мотивує дослідників і практиків розробляти методи та інструменти для розробки безпечних смарт-контрактів, щоб уникнути пасток з самого початку, а також для перевірки смарт-контрактів на вразливості. Ці

зусилля включають документування вразливостей, збір найкращих практик та автоматизоване виявлення проблем.

На Ethereum транзакції отримуються вузлом через інтерфейс JSON RPC. Це розшифровується як протокол видалених процедурного виклику і є текстовою структурою кодування, яка дозволяє процесам отримувати транзакційні дані. Вузли мережі Ethereum можуть використовувати цей інтерфейс різними способами, залежно від конфігурації та реалізації базового програмного забезпечення. Популярними альтернативами є IPC, WebSockets та HTTP-з'єднання. Наприклад, за допомогою опції командного рядка "geth -rpc," реалізацію Go Ethereum (geth) можна налаштувати для надання інтерфейсу RPC веб3-експертом.

Копія всієї інформації та коду на блокчейні зберігається вузлами пір-до-піра мережі Ethereum. Аналогічно використанню jQuery з JSON API, Web3.js отримує та записує дані в мережу, надсилаючи запити JSON RPC до вузла Ethereum.

Web3.js використовує постачальника і реалізує функцію запиту, відповідальну за виклик методу Ethereum RPC для перетворення коду JavaScript на запити json-rpc. Вищезазначена специфікація легко реалізовується за допомогою Web3.js і зроблена доступною через web3.providers, HttpProvider, WebsocketProvider та IpcProvider.

Web3.js та Solidity утворюють необхідну інфраструктуру для взаємодії з блокчейном Ethereum та розробки смарт-контрактів. Копія інформації та коду на блокчейні зберігається вузлами мережі, а Web3.js дозволяє розробникам взаємодіяти з цими даними, використовуючи JSON RPC інтерфейс. Завдяки Web3.js можливе спрощене взаємодія з Ethereum nodes, що робить процес розробки додатків та виконання транзакцій більш доступним [18].

Solidity, у свою чергу, становить високорівневий інструмент для написання смарт-контрактів на базі Ethereum. Ця мова програмування є

статично типізованою та підтримує ряд складних функцій, таких як успадкування та користувацькі типи даних. Web3.js та Solidity разом уможливлюють розробку децентралізованих додатків, взаємодія яких з блокчейном та виконання смарт-контрактів стає ефективним та простим процесом.

Solidity є статично типізованою і підтримує складні програмні можливості, такі як успадкування та користувацькі типи даних. Її можна використовувати для різних випадків використання смарт-контрактів, таких як голосування, гаманці з багатьма підписами та збір коштів. Його розробники регулярно оновлюють репозиторій з новими можливостями та виправленнями помилок.

Solidity розроблена на синтаксисі ECMAScript, щоб її було легко зрозуміти та кодувати. Вона підтримує ієрархічні структури та C3 лініаризацію для багатократного успадкування. Її відображення виконуються через структури даних ключ-значення.

Також її можна використовувати з криптовалютними гаманцями, такими як Mist та MetaMask. A Truffle для розробки та розгортання високорівневого коду на Solidity. Такі інструменти дозволяють інженерам створювати децентралізовані додатки (DApps) з мінімальними зусиллями.

Багато інженерів блокчейну віддають перевагу Solidity [19] через те, що вона працює з Ethereum, фактичним протоколом блокчейну для смарт-контрактів. Крім того, вона дозволяє їм створювати складні додатки з багатофункціональністю, оскільки Solidity є тьюринг-повною (тобто вона може виконувати будь-яке завдання, якщо їй надано правильні інструкції, достатньо часу та обчислювальної потужності).

2.2 Аналіз інструментів впровадження автоматизованого тестування

Вибір інструментів для автоматизації тестування – це важливе рішення, яке впливає на ефективність та якість процесу розробки. Від правильно обраного стеку інструментів залежить швидкість знаходження дефектів, стабільність програмного забезпечення та його готовність до релізу. У цьому розділі ми проведемо порівняльний аналіз популярних інструментів і обґрунтуємо наш вибір на користь JavaScript, GitHub Actions, Chai.

Сучасний ринок пропонує широкий спектр інструментів для автоматизації тестування, які відрізняються за мовою програмування, функціональністю, рівнем автоматизації та іншими параметрами. Вибір конкретного набору інструментів визначається вимогами до проєкту, доступними ресурсами, технічними обмеженнями та очікуваним рівнем автоматизації. Далі розглянемо особливості кожного з інструментів, на основі яких було прийнято рішення для проєкту.

1. JavaScript. універсальна мова програмування, яка отримала популярність завдяки своїй широкій підтримці як на стороні клієнта, так і на стороні сервера (через Node.js). JavaScript дозволяє створювати тести, які можуть виконуватись у різних середовищах, що є великою перевагою для проектів, які розробляються як для фронтенду, так і для бекенду. Це дозволяє знизити поріг входу для нових учасників команди та забезпечує єдність технологічного стеку.

2. GitHub Actions. платформа для автоматизації робочих процесів, яка надає безліч можливостей для налаштування CI/CD. Завдяки GitHub Actions можна автоматизувати процеси тестування, збірки, розгортання та моніторингу проєкту безпосередньо на GitHub, що забезпечує зручність і доступність. Це дозволяє запускати тести автоматично при кожному коміті, що прискорює зворотній зв’язок і сприяє безперервному контролю якості коду.

3. Chai. фреймворк для створення асертів у JavaScript, який надає широкий набір зручних функцій для опису очікуваної поведінки коду. Chai дозволяє зробити тести читабельними, що полегшує їх підтримку та розуміння іншими розробниками. Завдяки інтуїтивному синтаксису, Chai дозволяє легко реалізовувати складні сценарії тестування, особливо в поєднанні з іншими інструментами тестування, такими як Mocha або Jest.

Кожен із цих інструментів був обраний завдяки своїй функціональності, простоті використання, легкій інтеграції з іншими компонентами та широкому застосуванню в індустрії. Ці фактори є особливо важливими для проекту, який включає технології Web3.js та Solidity, оскільки специфіка розробки смарт-контрактів вимагає інструментів, які здатні взаємодіяти з блокчейн-середовищем, а також легко інтегруються з компонентами, що виконують автоматичні тести. [20]. Такий підхід до автоматизації тестування забезпечує стабільну, швидку та ефективну перевірку якості коду, що критично для проектів з високими вимогами до надійності.

Серед альтернативних інструментів для автоматизації тестування можна розглянути такі варіанти.

1. Python (з pytest). Python є одним із найпопулярніших мов для автоматизації тестування. Його бібліотека pytest надає широкий набір функцій для тестування, включаючи потужні інструменти для роботи з асерціями, параметризації тестів та забезпечення читаємості коду. Проте, для Web3.js та Solidity, де велика частина інтеграції з блокчейном та розробка відбувається на JavaScript, використання Python вимагатиме додаткових інтерфейсів і викликає труднощі у підтримці єдності технологічного стеку.

2. Jest (для тестування JavaScript). Jest – популярний фреймворк для тестування JavaScript-коду, який має підтримку модульного тестування та зручний у використанні. Однак, Jest не надає прямої підтримки для тестування смарт-контрактів і взаємодії з блокчейн-середовищем, що обмежує його

застосування в проектах, які потребують інтенсивної роботи з Solidity та Web3.js.

3. Hardhat (з Mocha та Chai). Hardhat є спеціалізованим інструментом для розробки та тестування смарт-контрактів на Solidity. Він відмінно підходить для Solidity-проектів, оскільки включає вбудовану підтримку Mocha та Chai для тестування смарт-контрактів. Хоча Hardhat є потужним рішенням, його використання було відкинуто, оскільки проект уже має певну інфраструктуру, інтегровану з JavaScript і GitHub Actions, а GitHub Actions забезпечує більш гнучкі можливості для автоматизації CI/CD процесів у порівнянні з Hardhat.

Таким чином, JavaScript, GitHub Actions, Chai забезпечують ідеальний баланс між функціональністю, зручністю та інтеграцією, що є важливим для проекту на Web3.js та Solidity. Вибраний стек дозволяє ефективно автоматизувати тести для смарт-контрактів, підтримуючи швидку перевірку якості коду та стабільність проекту.

3 РЕАЛІЗАЦІЯ ПРОЦЕСІВ ТЕСТУВАННЯ ДЛЯ СМАРТ-КОНТРАКТІВ

3.1 Реалізація юніт-тестів для смарт-контрактів

Код юніт тестів (додаток А) містить кілька тестових блоків, кожен із яких перевіряє різні аспекти роботи контракту та забезпечує, що функціонал відповідає очікуванням.

На початку коду оголошуються зовнішні модулі та бібліотеки:

`Marketplace` використовується для імпорту контракту

- Marketplace.sol`, що є головним об’єктом тестування;
- `fs` та `path` є модулями для роботи з файловою системою, які потрібні для перевірки JSON-виводу контракту;
- `chai` – бібліотека для асерцій (перевірок), яка перевіряє результати тестів; вона також підключає плагін `chai-as-promised` для роботи з асинхронними операціями;

```
contract('Marketplace', ([deployer, seller, buyer]) => {
```

Тут оголошується тестова функція для контракту `Marketplace`, яка приймає три користувачі: `deployer`, `seller` і `buyer`. Ці змінні представляють різні ролі, необхідні для тестування сценаріїв, зокрема сценаріїв купівлі-продажу.

```
before(async () => {
    marketplace = await Marketplace.deployed();
```

```
});
```

Функція `before` виконується один раз перед запуском усіх тестів. Вона ініціалізує змінну `marketplace`, зберігаючи в ній екземпляр розгорнутого контракту `Marketplace`. Це дозволяє уникнути повторного розгортання контракту перед кожним тестом, що економить час і ресурси.

```
describe('deployment checks', async () => {
```

Блок `describe('deployment checks')` містить тести для перевірки, чи контракт було успішно розгорнуто та чи відповідає його базова конфігурація очікуванням.

1. Перевірка успішного розгортання контракту:

```
    it('deploys successfully', async () => {
        const address = await marketplace.address; assert.notEqual(address, 0x0);
        assert.notEqual(address, '');
        assert.notEqual(address, null);
        assert.notEqual(address, undefined);
    });
```

У цьому тесті перевіряється, чи адреса контракту не є порожньою або недійсною, що підтверджує успішне розгортання.

Використовується `assert.notEqual`, щоб гарантувати, що значення `address` не дорівнює типово порожнім значенням.

2. Перевірка наявності назви контракту:

```
    it('has a name', async () => {
```

```
const name = await marketplace.name();
assert.equal(name, "Mariia's Marketplace");
});
```

Тест викликає функцію `name()` контракту, щоб перевірити, чи правильно встановлено його назву, яка в цьому випадку повинна бути `"Mariia's Marketplace"`.

```
describe('products', async () => {
```

Цей блок містить тести для функціоналу створення та продажу продуктів, щоб перевірити правильність їх реалізації.

3. Створення продуктів:

```
it('creates products', async () => {                    });
```

У цьому тесті спочатку створюється продукт за допомогою функції `createProduct`. Після цього перевіряється, що:

- `productCount` дорівнює 1, що підтверджує створення одного продукту.
- Дані продукту, такі як `name`, `price`, `owner`, збігаються з очікуваними значеннями.

Також перевіряється, що створення продукту з порожнім ім'ям або нульовою ціною буде відхилено, використовуючи метод `.should.be.rejected`.

4. Продаж продуктів:

```
it('sells products', async () => {
  //...code
```

```
});
```

У цьому тесті модель поведінки продажу перевіряється шляхом виконання транзакції `purchaseProduct`, де покупець надсилає суму, еквівалентну ціні продукту. Після покупки перевіряється, що:

- `owner` продукту змінено на адресу покупця;
- `purchased` продукту позначено як `true`;
- баланс продавця збільшився на вартість продукту;

Тест також містить кілька негативних перевірок, щоб переконатися, що помилкові дії (наприклад, покупка з недостатньою сумою або повторна покупка вже купленого продукту) відхиляються контрактом.

```
describe('JSON output checks', async () => {
```

Цей блок перевіряє коректність JSON-файлу контракту, що містить його ABI, байт-код, AST та метадані.

5. Перевірка наявності ABI:

```
it('should have ABI', async () => { const abi =
    jsonOutput.abi;
    assert.isArray(abi, 'ABI should be an array');
});
```

У цьому тесті перевіряється, чи ABI є масивом. Це важливо, оскільки ABI є структурою, яка містить опис функцій і подій контракту, необхідних для його інтеграції.

6. Перевірка байт-коду:

```
it('should have bytecode', async () => {
 bytecode = jsonOutput.bytecode;
    assert.isString(bytecode, 'Bytecode should be a string');
    assert.isNotEmpty(bytecode, 'Bytecode should not be empty');
});
```

Перевіряється, що байт-код є непорожнім рядком, оскільки це машинний код, що представляє контракт в блокчейні.

7. Перевірка AST та метаданих:

```
it('should have an AST', async () => { const ast =
    jsonOutput.ast;
    assert.isObject(ast, 'AST should be an object');
});
```

Тут перевіряється, що AST (Abstract Syntax Tree) є об’єктом, що дозволяє проаналізувати структуру коду контракту.

Усі ці тести забезпечують всебічну перевірку контракту `Marketplace`, забезпечуючи коректність його функціоналу та структури для подальшого використання.

3.2 Реалізація статичного аналізу смарт-контрактів

Для забезпечення високої якості та безпеки смарт-контрактів на платформі Ethereum, важливо мати ефективні інструменти, що здатні оперативно виявляти потенційні вразливості на етапі розробки. Зважаючи на те, що вразливості можуть мати серйозні наслідки, такі як втрата коштів або порушення функціональності контрактів, роль статичних аналізаторів є критичною.

Однак більшість існуючих інструментів зосереджуються на загальних аспектах безпеки, що не завжди дає точні результати в специфічних випадках, особливо коли йдеться про складні сценарії застосування смарт- контрактів. Реалізований статичний аналізатор (додаток Б) для перевірки смарт-контрактів є інноваційним завдяки своєму підходу до виявлення специфічних проблем у контрактах на Solidity, що дозволяє значно покращити безпеку та якість коду. Існуючі інструменти, такі як Mythril або аналізатор від ConsenSys Diligence, зосереджуються на загальних вразливостях, часто даючи хибнопозитивні результати або працюючи повільно через використання складних перевірок. Новий підхід полягає у точнішому фокусуванні на конкретних проблемах, таких як перевірка балансу, обробка видимості функцій та попередження про можливість реентрантних атак, що робить цей аналізатор більш ефективним і зручним для практичного використання.

Основні покриті тестами компоненти безпеки.

1. Перевірка на реентрантні атаки через функції call, send, transfer.

Реентрантність (можливість зробити повторний виклик) є однією з найбільших загроз у смарт-контрактах, особливо коли контракт викликає інші контракти та дозволяє їм змінювати свій стан. Використання функцій, таких як call, send, transfer, без відповідних заходів безпеки може дозволити зловмиснику повторно викликати функцію контракту і змінити стан контракту

перед завершенням попереднього виклику. Аналізатор перевіряє, чи є в контракті виклики цих функцій без захисту від реентрантних атак (наприклад, використання "порядку змінюваних даних" або "замків"). Це покриває важливий аспект безпеки, зокрема захист від реентрантних атак, що є критично важливим для контрактів, які здійснюють фінансові операції.

2. Перевірка видимості функцій.

Смарт-контракти Solidity мають різні рівні доступу до своїх функцій, визначені модифікаторами видимості, такими як public, internal, external і private. Якщо функція не має чіткого модифікатора видимості, це може створити небезпеку для контракту, оскільки інші контракти або зовнішні користувачі можуть випадково або навмисно взаємодіяти з цією функцією. Модифікатор видимості важливий для захисту від несанкціонованого доступу до функцій. Аналізатор перевіряє, чи всі функції мають належну видимість, і видає попередження, якщо якась функція не має модифікатора. Це допомагає уникнути потенційних проблем з небажаним доступом до внутрішніх функцій контракту, що підвищує конфіденційність та цілісність контракту.

3. Перевірка на використання tx.origin.

Використання tx.origin для авторизації транзакцій може бути небезпечним, оскільки цей параметр може бути підроблений у разі виконання транзакції через інші контракти. Зловмисники можуть маніпулювати tx.origin, щоб обійти механізми безпеки, такі як перевірка підписів. Оскільки tx.origin доступний для всіх функцій, це може дозволити атаки типу phishing або replay attack. Аналізатор перевіряє наявність використання tx.origin і попереджає про можливі безпекові проблеми, пов'язані з його використанням. Це покриває важливу частину безпеки, а саме захист від несанкціонованого доступу та запобігання підробці транзакцій.

4. Перевірка на відсутність подій (Events).

В Solidity події використовуються для реєстрації важливих змін у стані контракту та забезпечують прозорість для зовнішніх користувачів або додатків, що взаємодіють з контрактом. Відсутність подій може призвести до того, що користувачі або інші системи не зможуть відслідковувати важливі зміни. Це знижує трасованість і прозорість смарт-контракту. Аналізатор перевіряє, чи мають контракти необхідні події, і повідомляє про відсутність таких подій, що допомагає підвищити прозорість і відслідковуваність операцій, що відбуваються в системі.

5. Перевірка правильності параметрів функцій.

Контракт може бути вразливим, якщо функції не перевіряють правильність переданих параметрів. Наприклад, функція, яка приймає ціну, може працювати з нульовими значеннями, що може спричинити несподівані результати. Аналізатор перевіряє наявність правильних перевірок параметрів, зокрема на порожні значення або неправильний тип даних, що дозволяє уникнути неправильних транзакцій і зловживань, покращуючи тим самим надійність контракту.

3.3 Налаштування CI/CD процесу

CI/CD (Continuous Integration/Continuous Delivery або Continuous Deployment) — це сучасна практика у розробці програмного забезпечення, яка передбачає автоматизацію етапів збірки (build), тестування (test) і розгортання (deploy). Основна ціль цієї методології — прискорити створення продукту та покращити його якість завдяки автоматизованим процесам та неперервному циклу інтеграції, тестування та випуску нових версій. Основними складовими

CI/CD є неперервна інтеграція (Continuous Integration) та неперервна доставка (Continuous Delivery).

– Continuous Integration (CI). Ця методологія передбачає регулярне додавання змін у спільний репозиторій коду (наприклад, на платформі управління версіями, такій як Git). Після кожного внесення змін автоматично запускаються збірка і тестування, що допомагає швидко виявити конфлікти та помилки

– Continuous Delivery (CD). Практика, яка забезпечує готовність програмного продукту до релізу в будь-який момент. Всі етапи, включаючи збірку, тестування та розгортання, автоматизовані. Це дозволяє розробникам випускати нові версії відразу після успішного проходження тестування. Розширенням Continuous Delivery є Continuous Deployment, де нові версії автоматично розгортаються у виробничому середовищі одразу після проходження перевірок.

Згідно з дослідженням щодо стану CI/CD за 2024 рік [7], більшість організацій активно впроваджують ці практики. Зокрема, 83% розробників застосовують підходи DevOps, що свідчить про високий рівень автоматизації. Однак, менш досвідчені команди рідше використовують повний спектр інструментів DevOps, що може негативно впливати на ефективність їхньої роботи.

Також звіт підкреслює важливість включення тестування безпеки у CI/CD процеси, адже використання таких практик позитивно впливає на результати за ключовими метриками DORA. CI/CD конвеєри дають змогу миттєво виявляти проблеми у коді, що дозволяє командам уникати витрат часу на повторне опрацювання старих помилок та знижує ризики.

Автоматизація тестування у CI/CD спрощує перевірку нових змін, дозволяючи швидко інтегрувати їх до існуючого коду. Це сприяє стабільності програмного забезпечення та мінімізації кількості помилок у випущених

версіях. Завдяки цьому бізнес швидше адаптується до змін на ринку, а розробники можуть зосередитися на створенні нових функцій без тривалих перерв.

Згідно з дослідженням щодо стану CI/CD за 2024 рік [7], більшість організацій активно впроваджують ці практики. Зокрема, 83% розробників застосовують підходи DevOps, що свідчить про високий рівень автоматизації. Однак, менш досвідчені команди рідше використовують повний спектр інструментів DevOps, що може негативно впливати на ефективність їхньої роботи.

Також звіт підкреслює важливість включення тестування безпеки у CI/CD процеси, адже використання таких практик позитивно впливає на результати за ключовими метриками DORA. CI/CD конвеєри дають змогу миттєво виявляти проблеми у коді, що дозволяє командам уникати витрат часу на повторне опрацювання старих помилок та знижує ризики.

Автоматизація тестування у CI/CD спрощує перевірку нових змін, дозволяючи швидко інтегрувати їх до існуючого коду. Це сприяє стабільності програмного забезпечення та мінімізації кількості помилок у випущених версіях. Завдяки цьому бізнес швидше адаптується до змін на ринку, а розробники можуть зосередитися на створенні нових функцій без тривалих перерв.

Для власного CI/CD було створено YAML файл який описує налаштування у GitHub Actions для тестування смарт-контрактів (додаток В).

Запуск workflow:

```
name: Test Contracts on:
    push:
    workflow_dispatch:
```

– `on`: Вказує, коли запускається workflow;

– `push`: Запускає workflow при кожному пуші в репозиторій;

– `workflow_dispatch`: Дозволяє вручну запускати workflow через інтерфейс GitHub;

Налаштування завдання (job):

```
jobs:
  Test-React-App:
    runs-on: ubuntu-latest
```

– `jobs`: Усі завдання, які виконуються в рамках workflow;

– `Test-React-App`: Назва конкретного завдання;

– `runs-on: ubuntu-latest`: Вказує, що завдання виконується на віртуальній машині з останньою версією Ubuntu;

Кроки виконання (steps):

```
- name: Checkout repository uses:
  actions/checkout@v3
```

– `uses: actions/checkout@v3`: Офіційний GitHub Action для клонування (checkout) репозиторію у віртуальне середовище;

Налаштування Node.js:

```
- name: Set up Node.js
  uses: actions/setup-node@v3 with:
    node-version: '18'
```

– `uses: actions/setup-node@v3`: Встановлює Node.js у віртуальне середовище;

– `node-version: '18'`: Вказує версію Node.js (18);

Встановлення залежностей:

```
- name: Install dependencies run:
  npm install
- `run: npm install`: Встановлює залежності, визначені у файлі `package.json`.
```

Перевірка залежності `solc`:

```
- name: list solc
  run: npm list solc
```

– `run: npm list solc`: Виводить у консоль список встановлених версій `solc` (Solidity Compiler), що допомагає переконатися у правильній установці;

Запуск статичного аналізатора

```
- name: run analyzer
  run: node analizer.js
```

– `run: node analizer.js`: Запускає файл `analizer.js`, який, ймовірно, є вашим статичним аналізатором коду для смарт-контрактів;

Запуск локальної блокчейн-мережі (Ganache)

```
- name: Start ganache-cli
  run: npx ganache-cli -p 7545 -i 1337 > /dev/null &
```

- `npx ganache-cli`: Запускає локальну блокчейн-мережу Ganache на вказаному порту.

```
```

– `-p 7545`: Використовує порт 7545;

– `-i 1337`: Встановлює ідентифікатор мережі на 1337;

– `> /dev/null &`: Відправляє вихідний лог Ganache у фон;

Затримка для Ganache:

```
- name: Wait for Ganache run:
  sleep 5
```

– `sleep 5`: Затримує виконання на 5 секунд, щоб дати час для повного запуску Ganache.

Запуск тестів Truffle:

```
- name: Run truffle test
  run: npm run truffle:test
```

– `npm run truffle:test`: Виконує команди, визначені у `package.json` (ймовірно, команда запускає `truffle test` для виконання тестів смарт-контрактів);

ВИСНОВКИ

В ході виконання дипломної роботи був розроблений статичний аналізатор для перевірки смарт-контрактів, а також створено модульні тести з використанням бібліотеки Chai, що забезпечують надійність смарт-контрактів.

Головна мета роботи полягала у створенні інструментів для автоматизації процесу перевірки коду смарт-контрактів на мові Solidity. Реалізація статичного аналізатора дала змогу виявляти помилки та вразливості ще на ранніх етапах розробки, що мінімізує ризики під час розгортання контрактів у реальних умовах. Крім цього, модульні тести дозволили перевірити функціональність смарт-контрактів у різних сценаріях, що забезпечує високу якість та відповідність бізнес-логіці.

Особливу увагу було приділено інтеграції створених рішень у процес CI/CD за допомогою GitHub Actions. Це дозволило автоматизувати перевірку та тестування коду кожного разу, коли вносяться зміни до репозиторію, забезпечуючи безперервний контроль якості та безпеки. Такий підхід значно підвищує ефективність командної роботи та сприяє уникненню помилок, які могли б виникнути внаслідок людського фактора.

У процесі виконання роботи було вивчено можливості Web3.js для інтеграції смарт-контрактів з фронтенд-додатками, що спрощує взаємодію користувачів із блокчейном. Також було реалізовано приклади використання розроблених інструментів, які підтвердили їхню ефективність у реальних умовах.

Проект має практичне значення для розвитку безпечних децентралізованих додатків. Впроваджені рішення сприяють зниженню кількості критичних помилок, забезпечують прозорість і довіру до виконання

смарт-контрактів, що є ключовими факторами для їх успішного використання в різних сферах – від електронної комерції до фінансових послуг.

Результати роботи демонструють, що автоматизація перевірки та тестування смарт-контрактів не лише спрощує процес їхньої розробки, але й підвищує рівень безпеки в децентралізованих екосистемах. Отримані знання та практичний досвід у розробці інструментів для перевірки коду можуть бути використані для подальших досліджень у галузі блокчейну, створення нових інструментів і впровадження інноваційних підходів у розробку децентралізованих систем.

Розроблені інструменти демонструють практичне застосування теоретичних знань і сприяють підвищенню безпеки, прозорості та ефективності автоматизації бізнес-процесів. Вони можуть бути адаптовані до різних сфер діяльності, таких як фінанси, логістика, охорона здоров’я та інші галузі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review, 2008, 21260 [Електронний ресурс]. – Режим доступу: https://bitcoin.org/en/bitcoin-paper – Дата доступу: 22.11.2024.

2. Vieira, G., Zhang, J. Peer-to-peer energy trading in a microgrid leveraged by smart contracts. Renewable and Sustainable Energy Reviews, 2021, 143, 110900.

3. Swan, M. Blockchain: Blueprint for a New Economy. Sebastopol, CA, USA: O'Reilly Media, Inc., 2015.

4. Осіпова, Т. І., Венгер, О. В. Блокчейн: технологія розподіленого реєстрування. Київ: НУХТ, 2018.

5. Завадська, Т. М., Гомельский, В. В., Семенов, В. В. Сучасний взгляд на розвиток глобальних технологій та ефективність їх використання. Економічний вісник університету, 2019, 41, 101–108.

6. Rameder H., Di Angelo M., Salzer G. Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum // Frontiers in Blockchain. — 2022. — Vol. 5.

7. Гнатишин, Я. М. Смарт-контракти як інструмент оптимізації бізнес-процесів. Інформаційні технології та комп’ютерна інженерія, 2018, 3(59), 39-43.

8. Wood, D. D. Ethereum: a secure decentralised generalised transaction ledger [Електронний ресурс]. – 2014. – Режим доступу: https://ethereum.org/en/whitepaper/ – Дата доступу: 22.11.2024.

9. Gelvez, M. Explaining the DAO Exploit for Beginners in Solidity [Електронний ресурс]. – 2016. – Режим доступу: https://medium.com/@MyPaoG/explaining-the-dao-exploit-for-beginners-in-solidity-80ee84f0d470 – Дата доступу: 22.11.2024.

10. The State of Continuous Integration and Continuous Delivery 2024: Report. Continuous Delivery Foundation [Електронний ресурс]. – Режим доступу: https://cd.foundation/state-of-cicd-2024/ – Дата доступу: 22.11.2024.

11. Ante, L. Smart Contracts on the Blockchain – A Bibliometric Analysis and Review. Telematics and Informatics, 2021, 57.

12. Dika, A. Ethereum Smart Contracts: Security Vulnerabilities and Security Tools. Trondheim, Norway: Norwegian University of Science and Technology, Department of Computer Science, 2017.

13. Lahiri S. K., Chen S., Wang Y., Dillig I. Formal Specification and Verification of Smart Contracts for Azure Blockchain // arXiv:1812.08829 [cs.SE]. — 2018. — Режим доступу: http://arxiv.org/abs/1812.08829 (дата доступу: 22.11.2024).

14. Kalra S., Goel S., Dhawan M., Sharma S. Zeus: Analyzing Safety of Smart Contracts // Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 2018), San Diego, California, USA, February 18–21, 2018. — Режим доступу: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf (дата доступу: 22.11.2024)

15. Ray P. P. Web3: A comprehensive review on background, technologies, applications, zero-trust architectures, challenges and future directions // Internet of Things and Cyber-Physical Systems. — 2023. — Vol. 3. — P. 213–248. — Режим доступу: https://www.sciencedirect.com/science/article/pii/S2667345223000305 (дата доступу: 22.11.2024).

16. 16. Grishchenko I., Maffei M., Schneidewind C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts // Cham: Springer, 2018. — Режим доступу**:** https://link.springer.com/chapter/10.1007/978-3-319-89722-6_10 (дата доступу: 22.11.2024).

17. 17. Almakhour M., Sliman L., Samhat A. E., Mellouk A. Verification of smart contracts: A survey // Pervasive and Mobile Computing. — 2020. — Vol. 67. — P. 101227. — Режим доступу: https://doi.org/10.1016/j.pmcj.2020.101227 (дата доступу: 22.11.2024).

18. 18. Almasoud A. S., et al. Smart contracts for blockchain-based reputation systems: A systematic literature review // Journal of Network and Computer Applications. — 2020. — Режим доступу**:** https://doi.org/10.1016/j.jnca.2020.102625 (дата доступу: 22.11.2024).

19. Антонопулос, А. Блокчейн. Від теорії до практики. – Київ: Видавництво "Техніка", 2018.

20. Меркулов, Н. Смарт-контракти. Програмування блокчейнів Ethereum і Bitcoin. – Київ: Видавництво "ІТ-Професіонал", 2019.

ДОДАТОК А

Реалізовані автоматичні юніт-тести

```
const Marketplace = artifacts.require('./Marketplace.sol')
const fs = require('fs')
const path = require('path')
const chai = require('chai')
chai.use(require('chai-as-promised')).should()
const assert = chai.assert

contract('Marketplace', ([deployer, seller, buyer]) => {
  let marketplace

  before(async () => {
    marketplace = await Marketplace.deployed()
  })

  describe('deployment checks', async () => {
    it('deploys successfully', async () => {
      const address = await marketplace.address
      assert.notEqual(address, 0x0)
      assert.notEqual(address, '')
      assert.notEqual(address, null)
      assert.notEqual(address, undefined)
    })

    it('has a name', async () => {
      const name = await marketplace.name()
      assert.equal(name, "Mariia's Marketplace")
    })
  })

  describe('products', async () => {
    let result, productCount

    before(async () => {
      result = await marketplace.createProduct(
        'iPhone X',
        web3.utils.toWei('1', 'Ether'),
        { from: seller },
      )
```

```
productCount = await marketplace.productCount()
    })

    it('creates products', async () => {
      assert.equal(productCount, 1)
      const event = result.logs[0].args
      assert.equal(
        event.id.toNumber(),
        productCount.toNumber(),
        'id is correct',
      )
      assert.equal(event.name, 'iPhone X', 'name is correct')
      assert.equal(event.price,   '1000000000000000000',   'price   is
correct')
      assert.equal(event.owner, seller, 'owner is correct')
      assert.equal(event.purchased, false, 'purchased is correct')

      await     marketplace.createProduct('',     web3.utils.toWei('1',
'Ether'), {
        from: seller,
      }).should.be.rejected
      await  marketplace.createProduct('iPhone  X',  0,  {  from:  seller
}).should.be
        .rejected
    })

    it('sells products', async () => {
      let oldSellerBalance = await web3.eth.getBalance(seller)
      oldSellerBalance = new web3.utils.BN(oldSellerBalance)

      result = await marketplace.purchaseProduct(productCount, {
        from: buyer,
        value: web3.utils.toWei('1', 'Ether'),
      })

      const event = result.logs[0].args
      assert.equal(
        event.id.toNumber(),
        productCount.toNumber(),
        'id is correct',
```

```
 const           contractPath           =           path.join(__dirname,
'../src/abis/Marketplace.json')
      jsonOutput = JSON.parse(fs.readFileSync(contractPath, 'utf8'))
    })

    it('should have ABI', async () => {
      const abi = jsonOutput.abi
      assert.isArray(abi, 'ABI should be an array')
    })

    it('should have bytecode', async () => {
      const bytecode = jsonOutput.bytecode
      assert.isString(bytecode, 'Bytecode should be a string')
      assert.isNotEmpty(bytecode, 'Bytecode should not be empty')
    })

    it('should have an AST', async () => {
      const ast = jsonOutput.ast
      assert.isObject(ast, 'AST should be an object')
    })

    it('should have metadata', async () => {
      const metadata = jsonOutput.metadata
      assert.isString(metadata, 'Metadata should be a string')
      assert.isNotEmpty(metadata, 'Metadata should not be empty')
    })

    it('should have expected structure and values', async () => {
      const abi = jsonOutput.abi
      const hasProductCreatedEvent = abi.some(
        (item)   =>   item.type   ===   'event'   &&   item.name   ===
'ProductCreated',
      )
      const hasProductPurchasedEvent = abi.some(
        (item)   =>   item.type   ===   'event'   &&   item.name   ===
'ProductPurchased',
      )
      const hasCreateProductFunction = abi.some(
        (item)   =>   item.type   ===   'function'   &&   item.name   ===
'createProduct',
```

```
      )
      const hasPurchaseProductFunction = abi.some(
        (item)   =>   item.type   ===   'function'   &&   item.name   ===
'purchaseProduct',
      )

      assert.isTrue(
        hasProductCreatedEvent,
        'Contract should have ProductCreated event',
      )
      assert.isTrue(
        hasProductPurchasedEvent,
        'Contract should have ProductPurchased event',
      )
      assert.isTrue(
        hasCreateProductFunction,
        'Contract should have createProduct function',
      )
      assert.isTrue(
        hasPurchaseProductFunction,
        'Contract should have purchaseProduct function',
      )
    })
  })
})
```

ДОДАТОК Б

Розроблений статичний аналізатор

```
const { readFileSync } = require('fs');
const { parse, visit } = require('@solidity-parser/parser');

function analyzeContract(filePath) {
  let content;
  try {
    content = readFileSync(filePath, 'utf8');
  } catch (error) {
    throw new Error(`Error reading file at ${filePath}:
${error.message}`);
  }

  let ast;
  try {
    ast = parse(content);
  } catch (error) {
    throw new Error(`Error parsing Solidity file at ${filePath}:
${error.message}`);
  }

  let events = [];

  function isBalanceCheck(condition) {
    return (
      condition.type === 'BinaryOperation' &&
      condition.operator === '>=' &&
      ((condition.left.type === 'MemberAccess' &&
condition.left.expression.name === 'msg' && condition.left.memberName
=== 'value') ||
       (condition.right.type === 'MemberAccess' &&
condition.right.expression.name === 'msg' && condition.right.memberName
=== 'value'))
    );
  }

  visit(ast, {
    FunctionDefinition(node) {
      const functionName = node.name || 'constructor';
```

```
 }
    }
  });

  // Report on events defined in the contract
  if (events.length === 0) {
    console.warn(`⚪ No events defined in the contract at ${filePath}.`);
  }

  // Check for required events dynamically (if needed, could be adjusted
based on context)
  events.forEach((event) => {
    if (!events.includes(event)) {
      throw new Error(`Required event "${event}" is missing from the
contract.`);
    }
  });

  console.log('✅ Contract analysis completed');
}

try {
  analyzeContract('./src/contracts/Marketplace.sol');
} catch (error) {
  console.error(error.message);
}

try {
  analyzeContract('./src/contracts/Migrations.sol');
} catch (error) {
  console.error(error.message);
}
```

ДОДАТОК В

Реалізація процесів безперервної інтеграції та доставки

```
name: Test Contracts

on:
  push:
  workflow_dispatch:

jobs:
  Test-React-App:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Install dependencies
        run: npm install

      - name: list solc
        run: npm list solc

      - name: run analyzer
        run: node analizer.js

      - name: Start ganache-cli
        run: npx ganache-cli -p 7545 -i 1337 > /dev/null &

      - name: Wait for Ganache
        run: sleep 5

      - name: Run truffle test
        run: npm run truffle:test
```

ДОДАТОК Г

Тестовані Marketplace контракти

Marketplace.sol

```
pragma solidity >=0.4.21 <0.6.0;
pragma solidity ^0.5.0;

contract Marketplace {
    string public name;
    mapping(uint => Product) public products;
    uint public productCount = 0;

    constructor() public {
        name = "Mariia's Marketplace";
    }

    struct Product {
        uint id;
        string name;
        uint price;
        address payable owner;
        bool purchased;
    }

    event ProductCreated(
        uint id,
        string name,
        uint price,
        address payable owner,
        bool purchased
    );

    event ProductPurchased(
        uint id,
        string name,
        uint price,
        address payable owner,
        bool purchased
    );
```

```
    function createProduct(string memory _name, uint _price) public {
        require(bytes(_name).length > 0, "Product name must be
provided.");
        require(_price > 0, "Product price must be greater than zero.");

        productCount++;
        products[productCount] = Product(productCount, _name, _price,
msg.sender, false);

        emit ProductCreated(productCount, _name, _price, msg.sender,
false);
    }

    function purchaseProduct(uint _id) public payable {
        // Fetch the product
        Product memory _product = products[_id];
        address payable _seller = _product.owner;

        // Validate product ID range
        require(_id > 0 && _id <= productCount, "Invalid product ID.");

        // Check for sufficient payment
        require(msg.value >= _product.price, "Not enough Ether to
purchase this product.");

        // Ensure product has not been purchased already
        require(!_product.purchased, "Product has already been
purchased.");

        // Check ownership to ensure buyer is not the seller
        require(_seller != msg.sender, "Buyer cannot be the seller.");

        // Transfer ownership to the buyer
        _product.owner = msg.sender;
        _product.purchased = true;

        // Update the product
        products[_id] = _product;

        // Transfer funds to the seller
```

```
Migrations.sol
pragma solidity >=0.4.21 <0.6.0;

contract Migrations {
  address public owner;
  uint public last_completed_migration;

  constructor() public {
    owner = msg.sender;
  }

  modifier restricted() {
    require(msg.sender == owner, "Only the owner can call this
function.");
    _;
  }

  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }

  function upgrade(address new_address) public restricted {
    Migrations upgraded = Migrations(new_address);
    upgraded.setCompleted(last_completed_migration);
  }
}
    _seller.transfer(msg.value);

         emit ProductPurchased(_id, _product.name, _product.price,
 msg.sender, true);
     }
 }
```

ДОДАТОК Д

Презентація

Рис. Д.1 – Титульний лист

Рис. Д.2 – Вступ

Метою є дослідження, розробка та реалізація автоматичних тестів для web3 Marketplace додатку з використанням мови програмування Solidity для створенних смарт-контрактів на платформі Ethereum, а також з використанням JavaScript та допоміжних інструментів для взаємодії та реалізації безперервної доставки та деплою. Мета полягає в демонстрації можливостей тестування для запобігання критичним помилкам у смарт-контрактах блокчейну та створення безпечного й ефективного електронного маркетплейсу на основі децентралізованих принципів.

Для досягнення цієї мети необхідно виконати наступні завдання:

- зробити аналіз web3 як новітньої концепції;
- розглянути підхід до тестування смарт-контрактів;
- розробити смарт-контракти на мові Solidity, які дозволяють користувачам додавати товари;
- створити автоматизовані тести та аналізатор;
- впровадити CI/CD
- зробити висновки на основі дослідження;

Рис. Д.3 – Мета та задачі

Процес роботи смарт-контрактів:

1) Сторони погоджуються з умовами
2) Створюється смарт-контракт
3) Смарт-контракт розгортається
4) Виконуються сприятливі умови
5) Виконується смарт-контракт
6) Результат контракту записується на блокчейні

Рис. Д.4 – Процес роботи смарт-контрактів

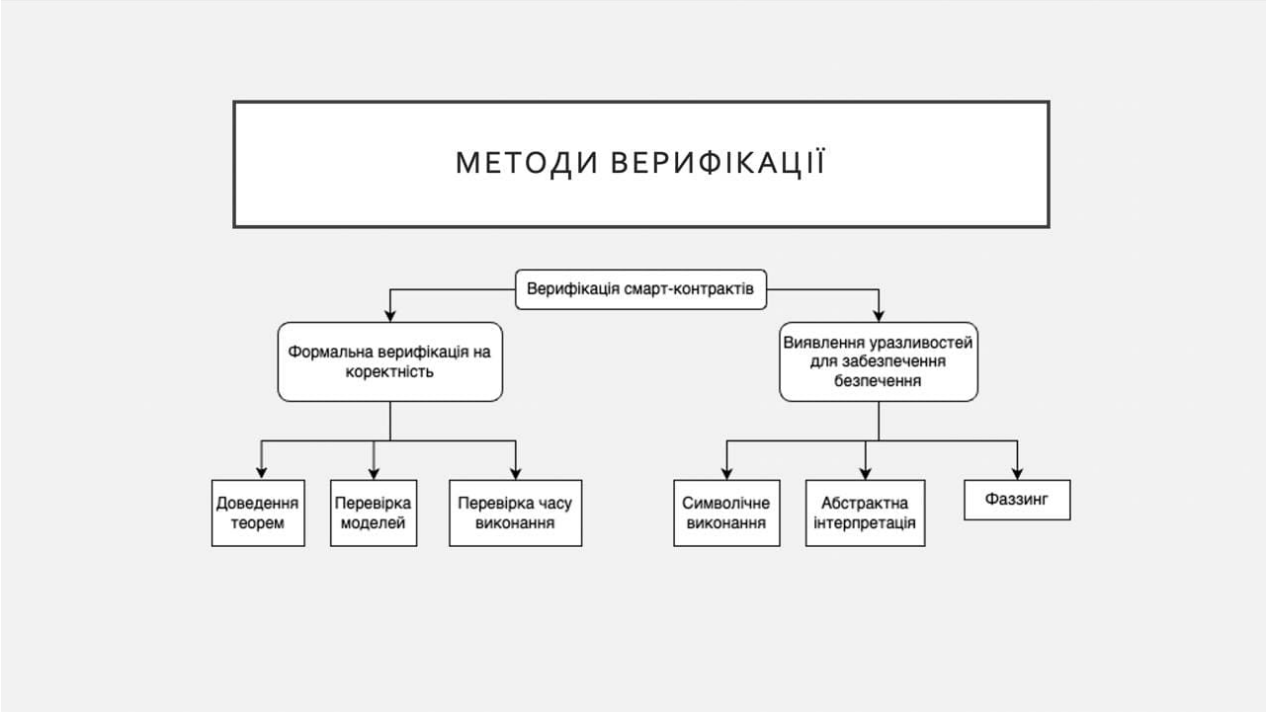

Рис. Д.5 – Методи верифікації смарт-контрактів

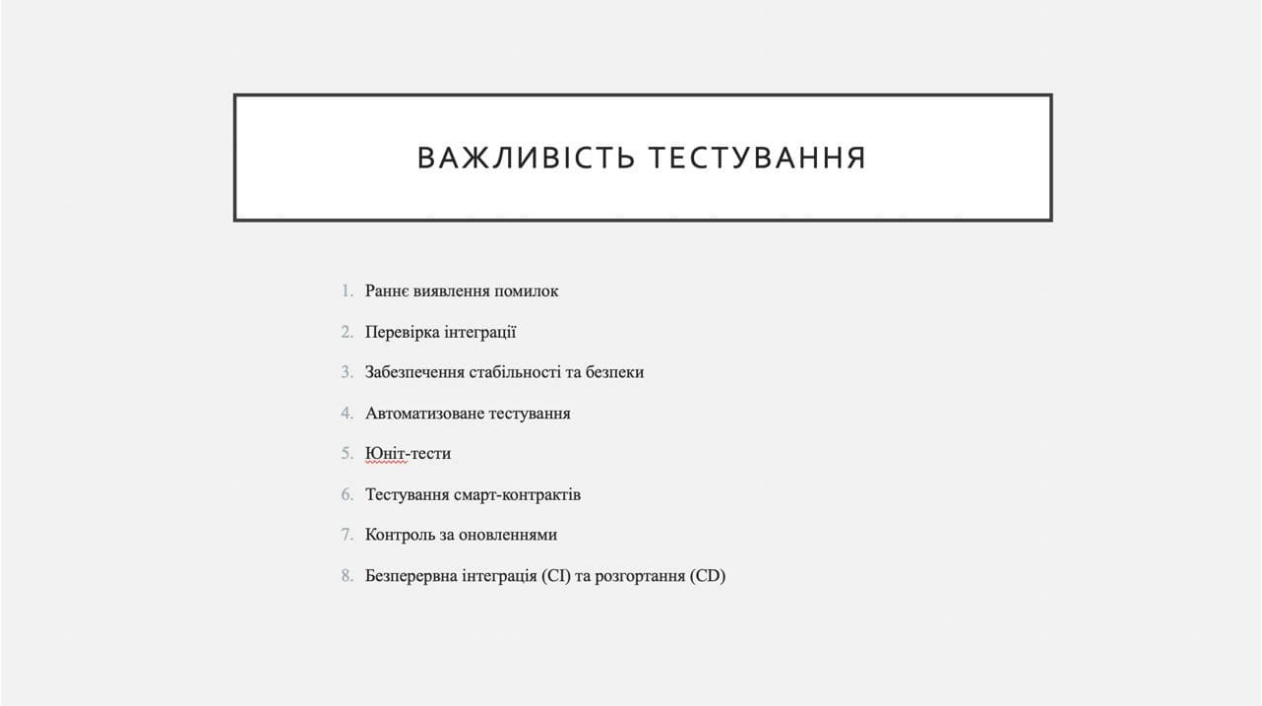

Рис. Д.6 – Аспекти важливості тестування

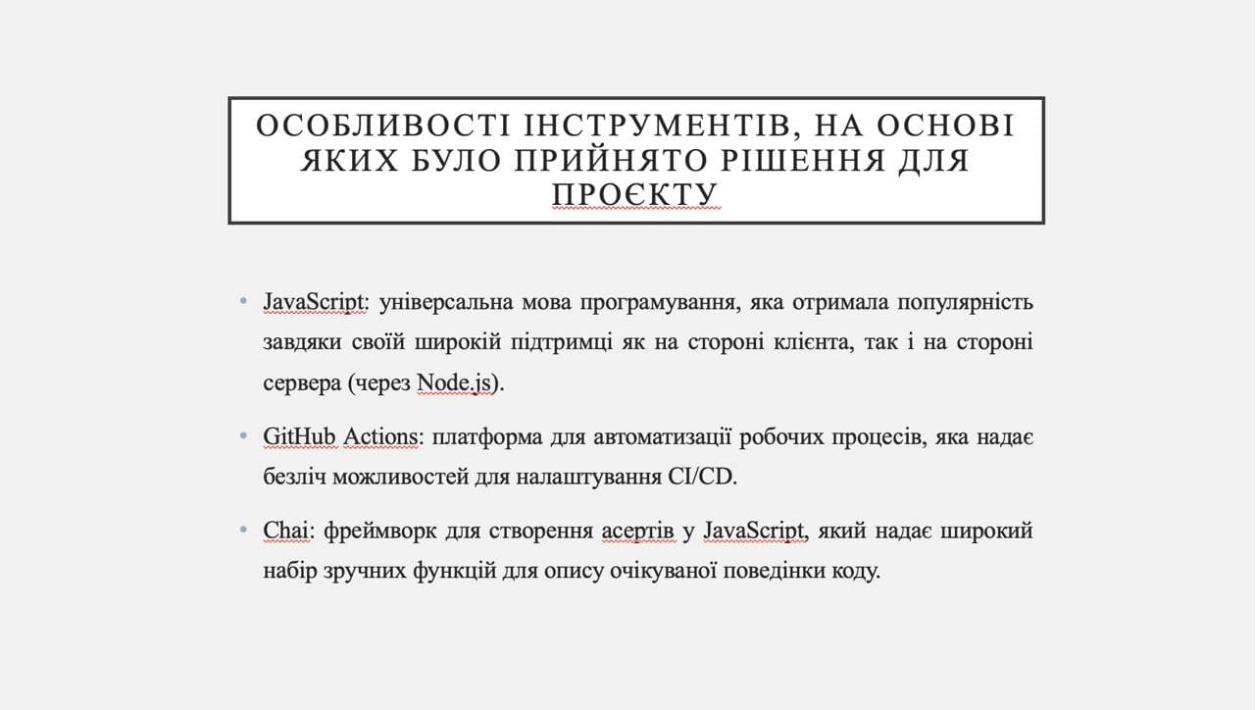

Рис. Д.7 – Вибір інструментів

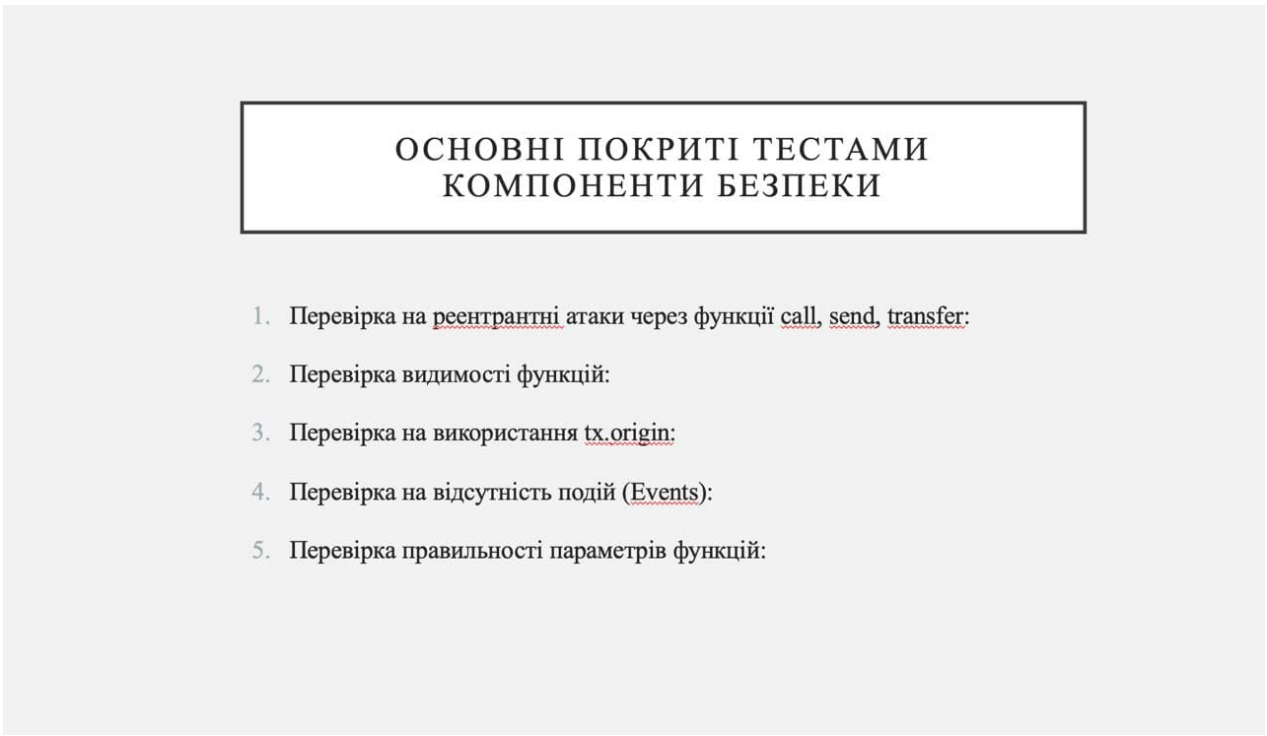

Рис. Д.8 – Покриті компоненти безпеки

НАЛАШТУВАННЯ CI/CD ПРОЦЕСУ

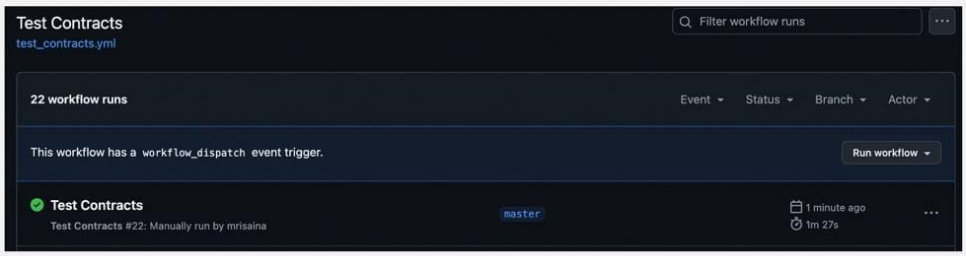

Рис. Д.9 – Налаштування CI/CD

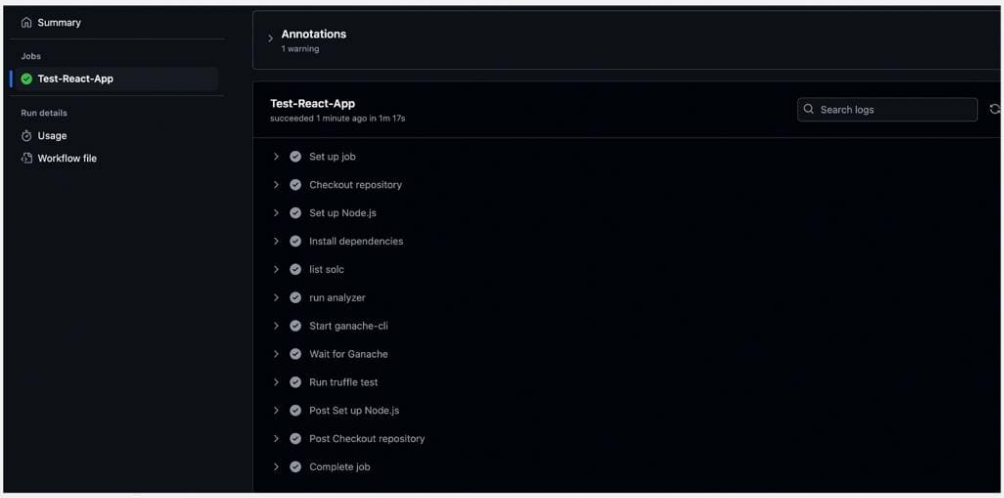

Рис. Д.10 – Налаштування CI/CD

Рис. Д.11 – Висновки