МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики
Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота
бакалавр

на тему “Фреймворк для тестування смарт-контрактів”

Виконав: студент 4 курсу, групи МФ-42
спеціальність 122 «Комп’ютерні науки»
освітньо-професійна програма
«Інформатика»
Дроздов В. О.
Керівник Є.С. Меняйлов
Рецензент_____
(прізвище та ініціали)

Реферат

В даній дипломній роботі проведено детальне дослідження фреймворку Dotty, який є розробкою автора на мові програмування D. Робота має на меті оцінити особливості та потенціал цього фреймворку у контексті розробки програмного забезпечення.

У першій частині роботи було проведено огляд мови програмування D та її особливостей. Була розглянута історія розвитку мови D, відзначені її основні принципи та порівняння з іншими популярними мовами програмування. Цей огляд надав чітке уявлення про потенційні переваги використання мови D в порівнянні з іншими мовами програмування.

Друга частина роботи була присвячена огляду фреймворку Dotty. Розглянуті були його призначення та основні компоненти, включаючи компілятор, стандартну бібліотеку та інструменти розробки. Проведено порівняння фреймворку Dotty з іншими фреймворками та мовами програмування, а також визначено його переваги та потенційні сценарії застосування.

У третій частині роботи було розглянуто деталі реалізації фреймворку Dotty. Проведено дослідження архітектури фреймворку та аналіз основних компонентів та їх взаємодії. Були розглянуті внутрішні механізми фреймворку, зокрема система типів, компіляція та оптимізація. Це дослідження надало глибоке розуміння реалізації фреймворку Dotty та його внутрішніх процесів.

Окрема увага була приділена дослідженню бібліотеки Deth, яка є складовою частиною фреймворку Dotty. Були розглянуті основні можливості та функціональність бібліотеки, а також її вплив на розробку децентралізованих додатків на основі мови D.

У завершальній частині роботи були проведені експерименти та тестування фреймворку Dotty та бібліотеки Deth з метою оцінки їх продуктивності та ефективності в реальних сценаріях. Результати експериментів демонструють потенціал та переваги використання фреймворку Dotty та бібліотеки Deth у програмуванні децентралізованих додатків.

Ця дипломна робота сприяє розумінню можливостей та переваг фреймворку Dotty та його використання у програмуванні. Результати дослідження можуть бути корисні для розробників, які цікавляться використанням мови програмування D та фреймворку Dotty у своїх проектах.

1. Вступ.
2. Огляд предметної області.
    2.1. Платформа Ethereum та смарт-контракти.
    2.2. Тестування смарт-контрактів та обзор існуючих рішень.
    2.3. Обзор обраної мови програмування D.
3. Реалізація бібліотеки deth.
    3.1. Огляд функціональності бібліотеки deth.
    3.2. Опис основних модулів та їх функціональності.
    3.3. Патерни програмування у deth.
    3.4. Тестування бібліотеки.
    3.5. Хеш-функція keccak256.
    3.6. Еліптична криптографія на прикладі secp256k1.
4. Реалізація фреймворку dotty.
    4.1. Огляд функціональності бібліотики dotty.
    4.2. Реалізація модулю dotty:runner
    4.3. Реалізація модулю dotty:builder
    4.4. Процесс створення проекту за допомогую dotty.
5. Список використаних джерел, перелік літератури, статей, документації та інших джерел, використаних під час написання дипломної роботи.

Вступ

У сучасному світі розробки програмного забезпечення з'явилося нове поняття - смарт-контракти. Смарт-контракти є автоматизованими програмами, які регулюють виконання угод між сторонами без потреби посередників. Вони базуються на блокчейн-технології та забезпечують безпеку, надійність та прозорість угод. З іншого боку, програмування смарт-контрактів вимагає високої точності та надійності, оскільки помилки можуть призвести до втрати значних коштів або до інших негативних наслідків. Важливість цього питання породжує необхідність у тестуванні смарт-контрактів.

Тестування смарт-контрактів є надзвичайно важливим етапом у розробці. Воно дозволяє перевірити правильність роботи контрактів, виявити можливі помилки та недоліки. Тестування допомагає забезпечити якість програмного коду, підвищує впевненість у його коректності та забезпечує довіру до смарт-контрактів.

Однак, тестування смарт-контрактів має свої особливості і вимагає від розробників спеціалізованих інструментів. Метою даної дипломної роботи є детальне дослідження фреймворку Dotty для тестування смарт-контрактів, його функціональних можливостей та переваг. Робота також передбачає розробку прикладних тестових сценаріїв та проведення експериментів для оцінки ефективності фреймворку.

Дослідження цієї теми важливо, оскільки воно сприятиме покращенню процесу розробки смарт-контрактів та підвищенню їх якості.

2. Огляд предметної області.

2.1. Платформа Ethereum та смарт-контракти.

Ethereum — це децентралізована платформа відкритого коду, що використовує технологію блокчейн, і була створена з метою виконання peer-to-peer "смарт-контрактів". Ця платформа була запропонована в 2013 році, а її розвиток продовжується до сьогодні. В основі Ethereum лежить технологія блокчейн, що є прозорою і незмінною відкритою базою даних, яка підтримує та реєструє всі транзакції.

Смарт-контракти є ключовою особливістю Ethereum. Вони представляють собою комп'ютерні програми, що автоматично виконують умови договору, коли задані критерії виконуються. Це означає, що транзакції можуть бути автоматично виконані без необхідності в довірі або посередниках. Кожен смарт-контракт має власний баланс, і він може взаємодіяти з іншими контрактами, включаючи здійснення фінансових операцій.

Смарт-контракти в Ethereum пишуться на мові програмування Solidity, яка була спеціально розроблена для створення смарт-контрактів і має схожість із мовою JavaScript. Синтаксис Solidity було спроектовано так, щоб було легко розуміти його для розробників, які вже знайомі з JavaScript.

Однак, необхідно зазначити, що програмування смарт-контрактів вимагає високої точності та уваги до деталей. Помилки в коді можуть мати серйозні наслідки, оскільки після розміщення на блокчейні смарт-контракт не може бути змінений або видалений. Щодо можливості заміни коду смарт-контракту, в мові програмування Solidity існує певна можливість реалізувати механізм "Upgradability" або "Upgradeability". Цей механізм дозволяє замінювати або оновлювати логіку смарт-контракту після його розміщення на блокчейні. Принципово важливою у реалізації "Upgradability" є дотримання певних принципів та безпекових заходів, щоб забезпечити сумісність та безпеку під час оновлення контракту.

Однак, варто зауважити, що взагалі замінити код смарт-контракту на блокчейні не можливо. Оновлення смарт-контракту зазвичай виконується шляхом створення нового контракту, який містить оновлені функції або логіку, і перенесення даних зі старого контракту до нового. Цей підхід дозволяє зберегти попередні дані та стан контракту, але вимагає додаткових механізмів для забезпечення зв'язку між старим та новим контрактами.

2.2 Тестування смарт-контрактів та обзор існуючих рішень.

Тестування смарт-контрактів - важливий етап у розробці децентралізованих додатків. З огляду на незмінність смарт-контрактів після розгортання, критично важливо впевнитися, що вони працюють так, як задумано, щоб уникнути втрати активів або інших негативних наслідків.

Для тестування смарт-контрактів використовуються різні підходи. Одним з них є модульне тестування(unittests), де кожна функція контракту тестується окремо. Інший підхід - інтеграційне тестування(intergation tests), де перевіряється взаємодія між різними частинами контракту або навіть між різними контрактами. Також може бути використано формальну верифікацію, що полягає в математичному доказі коректності коду контракту, але цей метод часто є часозатратним і складним.

Існують різні інструменти та підходи для тестування смарт-контрактів, серед яких Truffle та Hardhat вирізняються своєю популярністю та ефективністю.

Truffle - це потужний розробницький фреймворк для Ethereum, що пропонує набір інструментів для написання, тестування та розгортання смарт-контрактів. Truffle включає в себе консоль для взаємодії зі смарт-контрактами, автоматизоване тестування, підтримку розгортання на різних мережах, а також інтеграцію з різними пакетами JavaScript, такими як Web3.js і Ethers.js. Однак, в Truffle можуть бути обмеження, особливо щодо швидкості виконання тестів та деяких складних сценаріїв тестування.

Hardhat, з іншого боку, є відносно новим інструментом у сфері розробки Ethereum. Це JavaScript фреймворк, який зосереджений на допомозі розробникам у розробці, тестуванні та відлагодженні смарт-контрактів. Hardhat включає в себе власну EVM (Ethereum Virtual Machine), яка дозволяє симулювати поведінку смарт-контрактів у

реальному світі, з можливістю зміни стану EVM, що дозволяє більш гнучкі сценарії тестування. Крім того, Hardhat дозволяє створювати плагіни для інтеграції з іншими інструментами та сервісами.

Обидва інструменти мають свої переваги та недоліки, а вибір між ними залежить від конкретних потреб розробника. Важливо зазначити, що обидва цих інструменти дозволяють реалізувати автоматизоване тестування, яке є критично важливим для забезпечення надійності та безпеки смарт-контрактів.

На даний момент не існує аналогічних інструментів, створених спеціально для мови програмування D.

2.3. Обзор обраної мови програмування D.

Мова програмування D є сучасною мовою програмування загального призначення, яка єбула створена з метою об'єднати продуктивність розробки, яку забезпечують такі мови як Python і JavaScript, із ефективністю виконання, характерною для мов низького рівня, таких як C і C++.

D включає багато сучасних особливостей програмування, таких як автоматичне управління пам'яттю, що включає в себе збирання сміття, систему типів, що забезпечує безпеку типів і поліморфізм, та об'єктно-орієнтоване програмування. З іншого боку, D дає можливість використовувати такі особливості низькорівневого програмування, як безпосередній доступ до пам'яті і вбудовані типи даних, що використовуються у системному програмуванні.

Важливою властивістю мови D є гнучкість її синтаксису і семантики, що дозволяє розробникам вибирати найбільш підходящі для конкретної задачі засоби програмування. D підтримує як процедурне, так і об'єктно-орієнтоване програмування, а також має можливість для функціонального і метапрограмування.

Метапрограмування та шаблонне програмування є важливими особливостями мови програмування D, що відрізняють її від багатьох інших мов. Метапрограмування в D дозволяє програмам змінювати себе під час компіляції, що забезпечує велику гнучкість та ефективність. Воно може бути реалізовано за допомогою таких механізмів, як виконання функції під час компіляції (CTFE), статичні if-оператори, статичні цикли та міксини.

Виконання функції під час компіляції (CTFE) дозволяє виконувати функції на етапі компіляції, що може бути використано для генерації коду або для виконання обчислень, які не міняються в рантаймі. Статичні if-оператори та цикли дають змогу виконувати умовні оператори та цикли на етапі компіляції. Міксини можуть бути використані для генерації коду з рядків або інших міксинів.

Шаблонне програмування в D дозволяє створювати шаблони функцій та структур, що можуть бути використані з різними типами даних. Шаблони у D є потужними та гнучкими, оскільки вони можуть приймати не лише типи, але й значення, як

параметри, і мають можливість умовної компіляції в залежності від цих параметрів. Така функціональність дозволяє створювати високоефективний спеціалізований код для різних типів та значень параметрів.

Мова програмування D включає в себе вбудований тестовий запускач (test runner), що дозволяє виконувати модульні юніт-тести безпосередньо в рамках мови. Цей механізм є частиною D runtime і дозволяє автоматично виявляти та виконувати тестові функції.

Юніт-тести в D можна створити за допомогою спеціального ключового слова unittest. Юніт-тест це блок коду, який перевіряє правильність роботи окремого частини програми. Зазвичай, в рамках юніт-тесту, викликається функція, яка перевіряється, а потім перевіряється коректність результату цієї функції.

Якщо у вашій програмі є модуль з юніт-тестами, ви можете виконати всі юніт-тести, запустивши компілятор з флагом -unittest або dub з параметром test

Вбудований тестовий запускач в D простий у використанні та забезпечує пряму підтримку юніт-тестування в рамках самої мови. Такий підхід сприяє кращій організації коду та полегшує тестування.

DUB є стандартною системою збірки та менеджером пакетів для мови програмування D. Він дозволяє автоматизувати процес збірки проектів, управління залежностями, а також публікації та встановлення бібліотек D.

DUB також включає в себе репозиторій пакетів, який називається DUB Registry. Цей репозиторій містить велику кількість бібліотек D, які були опубліковані спільнотою. Це означає, що ви можете легко встановити та використовувати ці бібліотеки в своїх проектах, використовуючи команду dub add.

3.1. Огляд функціональності бібліотеки deth:

Бібліотека deth має клас Contract, який дозволяє взаємодіяти з контрактами Ethereum. Цей клас забезпечує генерацію коду (bindings) для методів контракту на основі його ABI (Application Binary Interface). ABI визначає структуру та типи параметрів методів контракту. Генерація bindings дозволяє зручно викликати методи контракту з додатків, що використовують бібліотеку deth.

Також, бібліотека містить клас RpcConnector для взаємодії з мережею Ethereum через RPC (Remote Procedure Call). RpcConnector дозволяє відправляти запити до мережі Ethereum, отримувати дані з блокчейна, створювати нові транзакції та отримувати стан контрактів.

Для роботи з транзакціями Ethereum, бібліотека deth надає функціональність оцінки необхідної кількості газу для виконання транзакції (функція estimate gas), підпису транзакцій з використанням приватного ключа та їх відправки.

Функція convTo використовується для зручної конвертації даних з одного типу в інший перед їх передачею або збереженням в контрактах Ethereum.

Бібліотека deth надає функції decode та encode для роботи з кодуванням та декодуванням даних в контексті Ethereum. Функція encode використовується для кодування параметрів методів контракту перед відправкою транзакції, а функція decode використовується для декодування отриманих даних від контракту.

Клас Wallet дозволяє працювати з приватними ключами в контексті Ethereum. За допомогою класу Wallet можна генерувати нові приватні ключі, виконувати підпис транзакцій та перетворювати приватні ключі в публічні адреси Ethereum.

3.2.

Модуль deth.contract в бібліотеці Deth містить класи і структури для взаємодії з Ethereum smart-контрактами.

Основні класи та структури, які визначені в цьому модулі:

1. **Contract** - цей клас представляє smart-контракт в Ethereum мережі. Це надає методи для виклику функцій контракту та передачі транзакцій.
2. **ContractABI** - структура, яка представляє ABI (Application Binary Interface) контракту. ABI визначає, як функції контракту можна викликати. ContractABI включає в себе функції для завантаження ABI з JSON-файлу.
3. **ContractFunction** та **ContractEvent** - ці структури представляють функції та події, визначені в контракті. Вони містять метаінформацію, необхідну для виклику функцій та обробки подій.
4. **Selector** - цей псевдонім використовується для представлення селектора функції, який є першими 4 байтами keccak256 хешу підпису функції.

Клас Contract використовує RPCConnector для взаємодії з Ethereum мережею через JSON-RPC. Він також надає методи для відправки транзакцій та виклику функцій контракту.

Особливістю цього модуля є те, що він генерує код для виклику функцій контракту на основі ABI контракту. Це дозволяє викликати функції контракту так, як би це були звичайні функції в D.

Модуль deth.rpcconnector призначений для взаємодії з Ethereum через його JSON-RPC API. Він імплементує декілька методів, включаючи отримання балансу, відправлення транзакцій, оцінку вартості газу, отримання кількості транзакцій, і т.д. Основні компоненти цього модуля описані нижче:

1. IEthRPC - це інтерфейс, який описує набір методів для взаємодії з Ethereum JSON-RPC API.
2. RPCConnector - це основний клас, який імплементує вищезгаданий IEthRPC інтерфейс і використовує його для взаємодії з Ethereum через JSON-RPC. Кожний метод у RPCConnector відповідає за виконання специфічного RPC виклику і надає додаткову логіку, яка спрощує взаємодію з Ethereum API.
3. HttpJsonRpcAutoClient!IEthRPC - це основний шаблон класу в D, який автоматично імплементує інтерфейс IEthRPC для виконання HTTP-запитів до Ethereum JSON-RPC API. Він бере на себе багато рутинних деталей імплементації HTTP клієнта, зокрема створення HTTP-запитів, обробку відповідей та обробку помилок.

Ця автоматизація значно спрощує код RPCConnector, оскільки класу потрібно лише надати URL ендпойнту Ethereum RPC і визначити додаткову логіку, специфічну для Ethereum. Це особливо важливо при роботі з Ethereum, оскільки його JSON-RPC API має багато методів, і ручна імплементація всіх цих методів була б досить трудомісткою.

Наприкінці модуля є два блоки модульних тестів, які демонструють використання RPCConnector для відправлення транзакцій до Ethereum мережі.

Ключовий момент, який важливо зазначити, це те, що цей модуль не включає в себе жодної логіки автентифікації або шифрування. Замість цього, він використовує приватні ключі, які зберігаються в wallet, для підпису транзакцій перед їх відправкою. Це означає, що захист цих ключів є важливою відповідальністю користувача.

Продовжуємо детальний аналіз структури D коду з наступного модуля deth.wallet, який реалізує маніпуляції з приватними ключами.

Цей модуль визначає структуру Wallet, яка включає в себе асоціативний масив addrs, що використовує адреси як ключі і об'єкти secp256k1 як значення. Об'єкти secp256k1 представляють приватні ключі Ethereum, які зберігаються в гаманці.

Метод addPrivateKey дозволяє додати новий приватний ключ до гаманця. Цей метод є шаблонним і може приймати ключ у різних форматах, включаючи Hash, string і bytes. Метод автоматично конвертує ключ в потрібний формат і створює новий об'єкт secp256k1.

Метод remove дозволяє видалити одну або декілька адрес з гаманця.

Метод addresses повертає список всіх адрес, збережених в гаманці.

Метод signTransaction використовує один з приватних ключів в гаманці, щоб підписати транзакцію. Якщо транзакція вже має вказане поле from, метод використовує ключ цієї адреси. Інакше використовується ключ вказаного підписанта. Метод повертає RLP-кодовану підписану транзакцію.

метод signTransaction здатний обробляти два типи підписування транзакцій, що залежить від наявності значення chainid.

EIP-155 (Ethereum Improvement Proposal) введений для уникнення певних типів атак на мережу Ethereum, зокрема від replay attacks, де транзакції можуть бути повторно використані на інших Ethereum-сумісних блокчейнах. Включення chainid дозволяє розрізнити транзакції, здійснені на різних мережах.

Важливо зауважити, що цей модуль використовує бібліотеку secp256k1, яка є основою для генерації ключів і підписання транзакцій в Ethereum.

Далі розглянемо модулі в папці util

deth.util.rlp пов'язаний з RLP кодуванням. RLP (Recursive Length Prefix) є основним методом кодування структурованих двійкових даних в Ethereum, використовується для серіалізації об'єктів у блокчейні Ethereum.

Функції модуля:

1. rlpEncode: ця функція приймає масив байтів і повертає RLP-кодовані байти. Вона перебирає кожен елемент у вхідному масиві, перевіряє його довжину і додає відповідний префікс, згідно з правилами RLP.
2. lenToRlp: ця приватна функція приймає довжину і оффсет, і повертає RLP-кодовану довжину. Вона використовується в rlpEncode для обчислення RLP-кодованих довжин.
3. cutBytes: ця приватна функція приймає масив байтів і "відрізає" початкові нулі, повертаючи новий масив байтів, що починається з першого ненульового байта. Ця функція важлива для підтримки вірного представлення байтів в контексті RLP.

У модулі також є два блоки unittest, що демонструють роботу функцій модуля і дають приклади їх використання. Вони забезпечують перевірку роботи коду і слугують як документація по використанню функцій.

Наступний, який називається "deth.util.abi", є частиною бібліотеки для обробки данних у форматі Ethereum ABI. ABI, або Application Binary Interface, описує кодування/декодування даних і викликів функцій для Ethereum Smart Contracts.

Він включає наступні ключові частини:

1. encode - функція, що кодує вхідні аргументи у байтовий рядок для передачі до смарт-контракту. Вона використовує різні способи кодування в залежності від типу аргументів.
2. encodeUnit - допоміжна функція, яка виконує кодування для окремих одиниць даних. Ця функція використовує різні способи кодування в залежності від типу даних.
3. tuplelize та tuplelizeT - ці дві функції використовуються для обробки даних у форматі кортежу.
4. decode - ця функція використовується для декодування даних, що були закодовані за допомогою вищезазначених методів.

Цей модуль також включає декілька модульних тестів (unittests) для перевірки правильності роботи функцій кодування та декодування.

Модуль deth.util.transaction призначений для обробки та відправки транзакцій в рамках Ethereum blockchain.

Основні елементи модуля:

- Структура Transaction: ця структура представляє собою модель транзакції Ethereum. Вона містить такі поля: from, to, gas, gasPrice, value, data, nonce, chainid. Всі ці поля можуть бути не визначені (Nullable), оскільки вони не обов'язково мають бути встановлені при створенні транзакції.
- Метод toJSON(): цей метод перетворює структуру Transaction у JSON формат.
- Метод serialize(): цей метод використовується для серіалізації даних транзакції в байти.
- Структура SendableTransaction: ця структура містить Transaction і RPCConnector і використовується для відправки транзакцій. Метод send() відправляє транзакцію, після того як всі параметри транзакції були встановлені.
- Шаблон NamedParameter: цей шаблон використовується для створення не обов’язкових параметрів метода send

У цілому, цей модуль відповідає за відправку транзакцій в Ethereum blockchain.

Модуль "deth.util.types" має застосування в мові програмування D та використовується в рамках розробки блокчейн-застосунків, зокрема тих, що працюють з Ethereum. Основний функціонал цього модуля - надання набору типів даних і корисних функцій для роботи з цими типами.

Основні типи даних, які надає цей модуль:

- Address: тип даних для представлення Ethereum адреси. Це просто масив з 20 байтів.
- Hash: тип даних для представлення хешу, що складається з 32 байтів.
- bytes: тип даних для представлення послідовності байтів.

Модуль також надає функції для перетворення між цими типами даних та рядками, які представляють гексадецимальні значення.

Є інші важливі функції, такі як hexToBytes, яка перетворює гексадецимальний рядок в послідовність байтів, і convTo, яка дозволяє перетворити значення одного типу в значення іншого типу.

Додатково, модуль містить визначення кількох структур, які представляють різні типи транзакцій Ethereum:

- TransactionReceipt: містить інформацію про результати виконання транзакції.
- TransactionInfo: містить інформацію про транзакцію.
- Log: структура, яка описує подію, іметовану у транзакції.

3.3. Патерни програмування у deth.

Builder Pattern: В модулі deth.util.transaction використовується структура SendableTransaction, яка використовується для побудови складних об'єктів типу Transaction з використанням іменованих параметрів у методі send. Це дозволяє зручно задавати різні параметри транзакції без необхідності вказувати всі параметри в конструкторі.

Proxy Pattern: Клас Contract діє як проксі для взаємодії з реальним контрактом у Ethereum. Це дозволяє забезпечити зручний програмний інтерфейс для роботи з контрактом та обробляти складні деталі перетворення типів та викликів функцій.

3.4. Тестування бібліотеки.

Тести в модулі rpcconnector призначені для перевірки функціональності класу RPCConnector. Ці тести перевіряють два сценарії відправки транзакцій.

Перший тест, "**sending legacy tx**", використовує легасі транзакцію для відправлення деякої кількості токенів з однієї адреси на іншу. Тест встановлює з'єднання з Ethereum RPC-клієнтом, додає приватний ключ до гаманця коннектора і відправляє транзакцію з використанням методу send() класу SendableTransaction. Після відправки транзакції перевіряється наявність інформації про транзакцію за її хешем за допомогою методу getTransaction() і чекається, поки транзакція буде включена в блок за допомогою методу waitForTransactionReceipt(). У кінці тесту перевіряється наявність TransactionReceipt для транзакції.

Другий тест, "**sending eip-155 tx**", використовує транзакцію з протоколом EIP-155 для відправлення токенів. Він має схожий сценарій, але додатково вказується ідентифікатор мережі (chainId) у транзакції. Це необхідно для забезпечення правильного підпису транзакції з врахуванням EIP-155.

Тести в модулі util.abi перевіряють функції encode та decode:

Перший тест, **"solidity ABI encode"** : Цей тест перевіряє роботу функції encode, яка виконує кодування аргументів для Solidity-функції за допомогою ABI-кодування. В цьому тесті перевіряється правильність кодування різних типів даних і списків аргументів. Кожен виклик runTest перевіряє, чи правильно закодовані аргументи. Закодовані значення порівнюються з очікуваними значеннями.

Другий тест , **"solidity ABI decode",** перевіряє роботу функції decode, яка виконує декодування результатів Solidity-функцій за допомогою ABI-кодування. У цьому тесті перевіряється правильність декодування різних типів даних і списків результатів. Кожен виклик runTestDecode зпочатку енкодує надані аргументи, після чого декодує результат, та звіряє з наданим аргументом.

Перший тест, "cutting null bytes": Цей тест перевіряє роботу функції cutBytes, яка виконує видалення нульових байтів з початку масиву байтів. У цьому тесті створюється масив cases, який містить структури Case з вхідними значеннями a і очікуваними результатами b. Для кожного елементу c в масиві cases перевіряється, чи результат виклику cutBytes для a дорівнює очікуваному значенню b.

```
struct Case
{
   bytes a, b;
}
Case[] cases = [
   Case([1, 2, 3], [0, 0, 1, 2, 3]),
   Case([1, 1, 0, 0, 1, 2, 3], [0, 0, 1, 1, 0, 0, 1, 2, 3]),
   Case([45, 128, 0, 0, 1, 2, 3], [0, 0, 45, 128, 0, 0, 1, 2, 3]),
   Case([1, 0], 256.convTo!bytes),
   Case([1, 0, 0, 0, 0], (1L << 32).convTo!bytes),
];
foreach (c; cases)
{
   assert(c.a.dup.cutBytes == c.b.dup.cutBytes);
   assert(c.a.dup.cutBytes == c.a.dup.cutBytes);
}
```

Цей тест дозволяє перевірити правильність роботи функції cutBytes для різних вхідних значень і забезпечує відповідність отриманих результатів очікуваним значенням.

Другий тест, "rlp encode": Цей тест перевіряє роботу функції rlpEncode, яка виконує кодування даних за допомогою RLP-кодування (Recursive Length Prefix). У цьому тесті виконуються декілька викликів rlpEncode з різними вхідними даними, а результати порівнюються з очікуваними значеннями.

```
assert(rlpEncode([
      cast(bytes) "cat", cast(bytes) "dog", cast(bytes) "dogg\0y",
      cast(bytes) "man"
   ]).toHexString == "D38363617483646F6786646F67670079836D616E");
```

Файл deth.util.types містить кілька функцій та структур, а також два набори unittests для перевірки їх роботи.

Набір unittests для функції hexToBytes перевіряє правильність розкодування шістнадцяткового рядка в масив байтів. В цьому наборі тестів порівнюється результат виклику функції hexToBytes з вхідним рядком та очікуваним масивом байтів.

Набір unittests для функції convTo перевіряє різні варіанти перетворення значень з одного типу у інший. В цьому наборі тестів перевіряється, чи відбувається коректне перетворення значень і чи компілюється код для невизначених пар типів.

Набір unittests для функції ox перевіряє, чи додається префікс "0x" до рядків. В цьому наборі тестів перевіряється, чи результат виклику функції ox з різними рядками дорівнює очікуваному значенню з доданим префіксом "0x".

Результати тестування:

✓ deth.util.rlp cutting null bytes
✓ deth.util.types 0x prefix
✓ deth.util.types convTo
✓ deth.util.types hexToBytes
✓ deth.util.rlp rlp encode
✓ deth.util.decimals decimals conv
✓ deth.util.abi solidity ABI decode
✓ deth.contract type convertor toDType
✓ deth.util.abi solidity ABI encode
✓ deth.rpcconnector sending eip-155 tx
✓ deth.rpcconnector sending legacy tx

Summary: 11 passed, 0 failed in 5778 ms

3.5. Хеш-функція keccak256.

У dub-пакеті keccak-tiny реалізована хеш-функція, яка є стандартною хешфункцією у Ethereum та Bitcoin. Хеш-функція keccak256 є частиною криптографічного стандарту Keccak, який був вибраний у 2012 році в якості стандарту SHA-3 (Secure Hash Algorithm 3) у результаті конкурсу проведеного NIST (National Institute of Standards and Technology) у США. Хеш-функція keccak256, як і інші варіанти Keccak, використовує нестандартний для блочних шифрів "сферичний" дизайн, що надає йому неперевершену стійкість до криптоаналізу.

Keccak256 використовує розмір блоку 1600 біт, а вихідний розмір хешу становить 256 біт, що забезпечує високий рівень криптографічної стійкості, достатній для більшості застосувань. Ця функція вважається безпечною від злому за допомогою квантових комп'ютерів, що робить її особливо актуальною в сучасних умовах.

Keccak256 широко використовується в сучасному криптографічному програмному забезпеченні. Вона є основною хеш-функцією в Ethereum, де використовується для обчислення адрес, визначення унікальних ідентифікаторів транзакцій, і створення цифрових підписів.

Незважаючи на високу стійкість, keccak256, як і будь-яка інша хеш-функція, піддається атакам перебору (brute force attacks) і атакам на основі зіткнень (collision attacks). Однак, у випадку keccak256, для успішного здійснення таких атак потрібно здійснити величезну кількість обчислень, що вважається практично неможливим з урахуванням сучасних технологій.

keccak256 використовує функцію keccakf для оновлення State, який базується на вхідних даних, і в кінці видає хеш вхідних даних.

Функція keccakf - це основна функція криптографічного хешування алгоритму Keccak, використовується в стандарті SHA-3.

1. **Theta**: Це перший крок криптографічного преобразування Keccak, який працює з бітами, що розташовані в одному і тому ж стовпці. Це допомагає забезпечити дифузію бітів по всьому стану.
2. **Rho and pi**: Ці два кроки виконуються разом. Rho виконує циклічний зсув для кожної ланки (елементів матриці), а рі переставляє ланки. Це допомагає забезпечити перетворення (або перемішування) бітів в межах кожного стовпця.
3. **Chi**: Цей крок виконує нелінійне преобразування, яке опрацьовує кожен рядок незалежно. Він додає додаткову складність до алгоритму.
4. **Iota**: Це останній крок, який додає змінну до першої клітинки, що допомагає уникнути симетрії.

Ці чотири етапи складають один цикл Keccak. Цей процес повторюється 24 рази. У кожному циклі використовуються деякі додаткові константи, визначені в матриці RC.

3.6. Еліптична криптографія на прикладі secp256k1

Еліптична криптографія (Elliptic Curve Cryptography, ECC) це підхід до публічного ключового шифрування, заснований на математиці еліптичних кривих. ECC пропонує значно більшу безпеку за ключ на символ при порівнянні з іншими методами шифрування, такими як RSA. Це означає, що ключі ECC можуть бути коротшими, що забезпечує менший обсяг переданих даних та швидше обчислення, зберігаючи при цьому той же рівень безпеки.

В даному розділі ми розглянемо ECC на прикладі стандарту secp256k1, який часто використовується в криптографічних застосуваннях та системах, зокрема в Bitcoin або Ethereum.

Secp256k1 - це визначення еліптичної кривої, яка є основою для ECC, що використовується в системі Bitcoin. Вона була вибрана засновником Bitcoin, Сатоші Накамото, і є однією з родини кривих, визначених стандартами SECG (Standards for Efficient Cryptography Group).

Крива secp256k1 визначається наступним рівнянням:

$$y^2 = x^3 + 7$$

Приватний ключ в ECC на базі secp256k1 - це число, вибране з діапазону від 1 до n - 1, де n - порядок кривої, що є фіксованим числом. Публічний ключ є точкою на кривій, яка є результатом множення приватного ключа на базову точку кривої (де операція множення визначена особливим чином в еліптичній кривій).

Використовуючи ECC, і в частості secp256k1, можна виконувати безпечне шифрування, генерацію цифрового підпису та інші криптографічні операції.

Secp256k1 також широко використовується в криптовалютах, зокрема в Bitcoin, для генерації ключів та цифрових підписів, що забезпечують безпеку і невід'ємність транзакцій.

Secp256k1 вибрана для використання в Bitcoin та інших криптовалютах через її високий рівень безпеки та ефективність. Хоч і ключі secp256k1 коротші за ключі RSA, вони забезпечують порівнянний або кращий рівень безпеки.

Окрім цього, secp256k1 має ефективні алгоритми для виконання операцій на кривій, що забезпечує швидкість та ефективність, необхідні для високопродуктивних криптографічних систем.

DUB пакет secp256k1 дає обгортку для популярної бібліотеки з bitcoin-core, написана мовою програмування C та asm.

Клас secp256k1 реалізує основні операції, пов'язані з еліптичною кривою secp256k1, яка використовується в Ethereum. Він містить ряд методів та властивостей, що описують нижче:

- Конструктори: Клас містить три конструктори, кожен з яких використовується для створення нового екземпляра класу secp256k1. Перший конструктор створює новий екземпляр з випадковим секретним ключем. Другий конструктор створює новий екземпляр з вказаним секретним ключем. Третій конструктор створює новий екземпляр з вказаним публічним ключем.
- sign: Цей метод використовує секретний ключ для підпису даних. Він повертає структуру Signature, що містить підписані дані.
- signHash: Цей метод використовує секретний ключ для підпису хешу даних. Він також повертає структуру Signature.
- verify: Цей метод використовує публічний ключ для перевірки підпису даних. Він повертає true, якщо підпис є дійсним, та false в іншому випадку.
- address: Ця властивість повертає Ethereum-адресу, що відповідає публічному ключу. Адреса є першими 20 байтами хешу

Структуру Signature: Ця структура визначає формат підпису в secp256k1, який включає ідентифікатор відновлення та два 32-байтових масиви, 'r' та 's', які представляють криптографічний підпис.

4. Реалізація фреймворку dotty.

4.1. Огляд функціональності бібліотики dotty.

До ключових можливостей Dotty належать:

1. **Компіляція Solidity контрактів**: Dotty використовує Solidity компілятор (solc) для компіляції контрактів на Solidity в байт-код Ethereum і генерації абстрактного бінарного інтерфейсу (ABI).
2. **Генерація прив'язок контракту**: Після компіляції контрактів, Dotty генерує код DLang для взаємодії з цими контрактами через Ethereum вузли.
3. **Тестування контрактів**: Dotty містить шаблон тесту та спеціальний раннер тестів пристосований для смарт-контрактів.
4. **Інтеграція з Dub**: Dotty інтегрований з Dub, системою збірки і менеджером пакетів DLang, для автоматизації процесу компіляції і тестування контрактів.

4.2. Реалізація модулю dotty:runner

Модуль runner.runner надає функціональність для запуску тестових сценаріїв. Він замінює стандртну функцію тестів на свою. Він містить наступні основні елементи:

shared static this() - метод, який ініціалізує тестове оточення. У цьому методі відбувається ініціалізація ExtentedTestRunner, обробка параметрів запуску. Цей метод визивається перед функцією main під час заспуску командой dub test

TestMassive - структура, яка представляє собою тестову групу, що містить об'єкт тестування ContractTest та масив тестів.

Test - структура, яка описує окремий тестовий сценарій. Вона містить інформацію про повне ім'я тесту, ім'я тестового методу, розташування тесту в початковому коді та вказівник на функцію, що виконує тест.

getTests - функція, яка виконує виявлення тестових сценаріїв. Вона проходить по модулях у dub_test_root та для кожного модуля перевіряє наявність класів, відмічених атрибутом ContractTest. Якщо клас є тестовим, створюється екземпляр класу, а для кожного методу класу, що задовольняє певним умовам, створюється об'єкт Test та додається до масиву тестів у відповідній тестовій групі. До кожного тесту додаються методи beforeEach та afterEach, до та після виконня відповідно.

runTests - функція, яка виконує всі тестові групи, передані у якості аргументів. Вона запускає функцію executeTest для кожного тестового сценарію та виводить загальну статистику про кількість пройдених і провалених тестів.

executeTest - функція, яка виконує конкретний тестовий сценарій. Вона викликає функцію ptr з об'єкта Test та обробляє винятки, що виникають під час виконання тесту. Якщо тест завершується успішно, він виводить повідомлення "Test [ім'я тесту] passed". В іншому випадку, він виводить повідомлення про помилку та деталі винятку.

4.3.Реалізація модулю dotty:builder

Модуль dotty:builder надає функціональність для побудови та генерації артефактів контрактів з використанням мови Solidity. Він містить наступні основні елементи:

ContractBuilder - клас, що відповідає за побудову та генерацію артефактів контрактів. Він містить налаштування, такі як шлях до папки з контрактами, шлях до папки з побудованими артефактами та команда для виклику компілятора Solidity. Клас також зберігає дані про контракти у внутрішній змінній contracts.

build - метод класу ContractBuilder, який виконує побудову контрактів. Він знаходить усі файли з розширенням .sol у вказаній папці з контрактами, формує аргументи для виклику компілятора Solidity та отримує результати компіляції у вигляді об'єкту JSONValue. Далі, він перебирає кожен контракт у результаті компіляції та зберігає їх дані у змінній contracts.

generateBindings - метод класу ContractBuilder, який генерує зв'язки для контрактів. Він створює модулі D з відповідними зв'язками для кожного контракту у змінній contracts та зберігає їх у відповідних файлових шляхах.

generateArtifacts - метод класу ContractBuilder, який генерує артефакти контрактів. Він створює JSON-файли з даними контрактів у вказаній папці з побудованими

артефактами. Артефакти зберігають інформацію о байткоді та ABI контрактів у вигляді JSON файлу.

parseLibs - метод класу ContractBuilder, який здійснює обробку бібліотек. Під час компіляції контрактів створюються плейсхолдери для адрес звонішніх бібліотек. Ці плейсхолдери мають бути замінені функцію link. Метод ParseLibs створює асоціфтивний масив, який зберігає інформацію про те, які спейсхолдери до якої бібліотеки відносяться. За стандартом плейсхолдер виглядає так __$(Хеш назви бібліотеки)$__.

main - точка входу програми, де створюється об'єкт ContractBuilder і викликаються методи build, generateArtifacts та generateBindings для побудови та генерації артефактів контрактів.

Цей модуль є частиною системи для автоматизованої розробки контрактів та допомагає розробникам виконувати процес побудови та генерації артефактів контрактів на основі коду Solidity.

4.4. Процесс створення проекту за допомоги dotty.

Щоб розпочати solidity проект на dotty. Спочатку треба встановити dub та D compiler. Їх можна скачати з офіційного сайту dlang.org або з github releases або за допомоги вашего пакетного менеджеру(apt, pacman, brew і т.п.) Є декілька різних компіляторів, але краще обрати dmd або ldc. Також потрібен компілятор solc для solidity.

Для створення проекту треба проініціалізувати dub пакет за шаблоном dotty

```
dub init -t dotty НазваПроєкту
```

Ця команда визиває dub-пакет під назвою dotty:init-exec.

Цей пакет створює необхідні файли для dotty: Папку з контрактами, конфігурацію dotty та dub необхідні для запуску тестів.

Зразу після init можна запустити dub test та запустити тести які були додані у шаблон.

Код Контракту зі шаблона:

```
contract Registry {
    address public owner;
    constructor(address _owner){
        owner = _owner;
    }
    mapping(bytes32 => address) addresses;

    function get(string memory name) public view returns (address) {
        return addresses[keccak256(abi.encode(name))];
    }

    function set(string memory name, address a) public {
        require(msg.sender == owner);
        addresses[keccak256(abi.encode(name))] = a;
    }
}
```

Контракт "Registry" дозволяє зберігати адреси за їх назвами і отримувати адреси за назвами. Лише власник контракту може встановлювати адреси за назвами за допомогою функції set. Інші користувачі можуть отримувати адреси за назвами за допомогою функції get. Подібні контракти часто з’євляються у проектах на Solidity.

Код тесту зі шаблона:

```
import dotty;
import contracts: Registry;

class ExampleTest: ContractTest
{
   Registry registry;

   mixin ContractTestContructor;

   override void before(){
      registry = Registry.deploy(conn, conn.remoteAccounts[1]);
   }

   @("registry should setable")
   void test()
   {
      registry.set("registry", registry.address).send(conn.remoteAccounts[1].From);
      assert(registry.get("registry") == registry.address, "addr must be in registry");
   }
}
```

В цьому файлі імпортується біндинг контракту, та створюється класс з тестом для Registry.
Він перевіряє можливість встановлення значення адреси у Registry. Назва методу тесту може бути будь-якою, але кожен метод має бути без аргументів та мати унікальну назву. Всі загальні для тесту змінні мають бути полями класу, та ініціалюватися у спеціальному методі before, який виконуються перед усіма тестами у класі. Також можна використовувати beforeEach. Тоді тест буде проводитися у уновленому середовищі.

Solidity немає стандартної бібліотеки, тому в проектах на часто використовують сторонні бібліотеки, тому спробуємо додати таку.
Для цього створюємо npm проект прямо в середені dub проекту:
npm init
Після цього інтегруємо бібліотеки у наш контракт.

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/structs/EnumerableMap.sol";

contract Registry is Ownable {
    using EnumerableMap for EnumerableMap.UintToAddressMap;

    EnumerableMap.UintToAddressMap private addresses;

    function get(string memory name) public view returns (address) {
        bytes32 key = keccak256(abi.encode(name));
        return addresses.get(uint256(key));
    }

    function set(string memory name, address a) public onlyOwner {
        bytes32 key = keccak256(abi.encode(name));
        addresses.set(uint256(key), a);
    }
}
```

та також змінюємо деплой контракту у тесті(так як змінили конструктор)

```
    override void before(){
        registry = Registry.deploy(conn, conn.remoteAccounts[1].From);
    }
```

Тепер додаємо нову функцію та тест.
Наступна функція має повернути всі адреса які зберігається в registry.

```
    function getAddresses() public view returns (address[] memory) {
        address[] memory result = new address[](addresses.length());
        for (uint256 i = 0; i < addresses.length(); i++) {
            (, address value) = addresses.at(i);
            result[i] = value;
```

```
        }
        return result;
    }
```

Адаптовані тести:

```
import dotty;
import contracts: Registry;

class ExampleTest: ContractTest
{
    Registry registry;
    Address owner;
    mixin ContractTestContructor;
    override void before(){
        owner = conn.remoteAccounts[1];
        registry = Registry.deploy(conn, owner.From);
    }
    @("registry should setable")
    void test()
    {
        registry.set("registry", registry.address).send(owner.From);
        assert(registry.get("registry") == registry.address, "addr must be in registry");
    }

    @("getAddress should return array of addresses in registry")
    void testGetAdressses(){
        assert(registry.getAddresses() == [registry.address]);
        registry.set("owner",owner).send(owner.From);
        assert(registry.getAddresses() == [registry.address, owner] );
    }
}
```

В цьому коді додан тест функції getAdresses який перевіряє змінення результату після додаваня новуї адреси у контракт.

Результат:

```
✓ registrytest.ExampleTest registry should setable
✓ registrytest.ExampleTest getAddress should return array of addresses in registry

Summary: 2 passed, 0 failed in 1041 ms
```

Дотті - це фреймворк, який надає базові можливості для тестування контрактів. Ось деякі плюси та мінуси використання цього фреймворку:

Плюси:

Перевірка синтаксису перед запуском тестів: Дотті перевіряє синтаксис вашого коду перед запуском тестів, що дозволяє виявляти помилки на ранніх етапах тестування.

Вбудовані оператори для роботи з BigNumber та Address: Dlang на відміну від JS надає вбудовані оператори, такі як +, -, == та інші, для зручної роботи з числами великої точності, адресами контрактів, масивами, тощо.

Зручний імпорт контрактів як звичайних класів: Дотті дозволяє імпортувати контракти як звичайні класи, що полегшує їх використання та тестування.

Паралельний запуск тестів: Дотті дозволяє запускати тести для різних контрактів паралельно, що забезпечує швидше виконання тестових наборів.

Мінуси:

Довгий час компіляції: Компіляція контрактів може займати значний час, особливо якщо в проекті присутні складні контракти або багато контрактів.

5. Список літератури.

1. Документація Solidity - https://docs.soliditylang.org/
2. https://medium.com/coinmonks/introduction-to-blockchains-bedrock-the-elliptic-curve-secp256k1-e4bd3bc17d
3. https://medium.com/0xcode/hashing-functions-in-solidity-using-keccak256-70779ea55bb0
4. https://medium.com/@angellopozo/ethereum-signing-and-validating-13a2d7cb0ee3
5. https://ernestognw.medium.com/what-is-a-smart-contracts-abi-anyways-a-guide-to-understand-client-blockchain-communication-7b68fb6ae466
6. Документація Dlang - https://dlang.org/documentation.html