КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота

на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

ЦИФРОВИЙ ПІДПИС У КРИПТОВАЛЮТАХ

Виконав: студент 4-го курсу
Олександр МАРЧУК _____
(підпис)

Науковий керівник:
професор кафедри теоретичної кібернетики
доктор фіз.-мат. наук, професор
Анатолій ПАШКО _____
(підпис)

Засвідчую, що в цій роботі
немає запозичень з праць інших авторів
без відповідних посилань.
Студент _____
(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри теоретичної
кібернетики
«____» _____ 2023 р.,
протокол № ___
Завідувач кафедри
доктор фіз.-мат. наук, професор
Юрій КРАК _____
(підпис)

КИЇВ – 2023

РЕФЕРАТ

Обсяг роботи 47 сторінок, 16 ілюстрацій, 20 джерел.

ЦИФРОВІ ПІДПИСИ, ЕЛІПТИЧНІ КРИВІ, ЕЛІПТИЧНІ КРИВІ ЕДВАРДСА, КРИПТОВАЛЮТА, JAVA.

Об’єктом аналізу є криптовалюти та алгоритми цифрового підпису, їх застосуванняв архітерктурі криптовалют. Об’єктом розробки є декілька з розглянутих алгоритмів цифрового підпису, які можуть бути застосовані і криптовалюті.

Метою дипломної роботи є проведення дослідження особливостей роботи криптовалют та алгоритми цифрового підпису, аналіз їх застосування та використання декількох з них в програмі, що використовуж цифрові підписи, а саме алгоритмів ECDSA та EdDSA.

Методи розроблення: аналіз архітектури криптовалют, аналіз існуючих алгоритмів цифрового підпису, проектування програмної реалізації цифрового підпису, покрокова розробка програми.

Інструменти розроблення: для розробки програмного додатку було використану мову програмування Java. У ролі середовища розробки використовувалось середовище IntelliJ IDEA.

Результат роботи: проведено аналіз архітекутри криптовалют та технології блокчейн, розглянуто потенційні алгоритми цифрового підпису, що можуть бути застосовані у криптовалюті. Серед розглянутих алгоритмів було обрані ті, які краще застосовувати в криптовалюті. Розроблено програмну реалізацію алгоритмів цифрового підпису з використанням еліптичних кривих, які використовуються у криптовалютах.

ЗМІСТ

ВСТУП

Оцінка сучасного стану об’єкта розробки. У сучасному світі криптовалюти здобувають все більшу популярність як альтернативна форма електронних платежів та зберігання активів. Криптовалюта є цифровим активом, що використовує криптографічні методи для забезпечення безпеки та контролю над транзакціями. Одним з ключових елементів криптовалют є цифровий підпис, який використовується для підтвердження автентичності та цілісності транзакцій.

Актуальність роботи та підстави для її виконання. Актуальність дослідження з цифрових підписів у криптовалютах пояснюється зростаючим використанням криптовалют як платіжного засобу, інвестиційного інструменту та цифрового активу. У такому контексті критично, щоб ці криптовалюти були безпечними, надійними та захищеними від шахрайства та несанкціонованого доступу. Цифрові підписи відіграють важливу роль у забезпеченні автентичності, цілісності та конфіденційності транзакцій, що робить їх дослідження та вдосконалення важливим завданням для розвитку безпечного та інструменту.

Мета й завдання роботи. Метою даної дипломної роботи є аналіз архітектури криптовалют та особливостей використання цифрового підпису в криптовалютах, аналіз основних алгоритмів цифрового підпису, що можуть бути використані Для досягнення цієї мети поставлено такі завдання:

а) розглянути архітектуру криптовалют та особливості використання цифрового підпису;

б) провести аналіз існуючих алгоритмів цифрового підпису проаналізувати їх структуру та безпеку, а також механізми, на яких вони ґрунтуються;

в) реалізувати програму що реалізовує цифровий підпис з використанням розглянутих алгоритмів цифрового підпису.

Об'єкт, методи й засоби реалізації. Об’єктом аналізу є криптовалюти, алгоритми цифрового підпису та їх застосування у криптовалютах.

Об'єктом даної розробки є створення програмного рішення для реалізації цифрового підпису з використанням еліптичних кривих. Вибір алгоритмів був здійснений на підставі аналізу різних алгоритмів цифрового підпису та особливостей їх застосування в криптовалютах.

Для реалізації програмного рішення було вибрано мову програмування Java. Java є об'єктно-орієнтованою мовою програмування, яка підтримує і процедурний, і об'єктно-орієнтований підхід до програмування. Java є популярною мовою програмування для серверних додатків, а також основною мовою програмування мобільних додатків для системи Android. Мова має широкий набір стандартних бібліотек, які включають в себе інструменти для роботи зі структурами даних, мережами. Для мови програмування Java існує велика кількість бібліотек з відкритим кодом для роботи з криптографією.

Під час реалізації програмного рішення було використано середовище IntelliJ IDEA. Безкоштовна версія має повну підтримку мови програмування Java, систем контролю версій git/svn.

Можливі сфери застосування. Розроблене програмне рішення може бути використане в системах, що використовують цифровий підпис, зокрема криптовалюти та блокчейн.

1 КРИПТОВАЛЮТА

1.1 Основні принципи криптовалюти

Криптовалюта – це форма цифрових або віртуальних валют, що використовує методи криптографії для забезпечення безпеки та контролю над створенням нових одиниць, а також для підтвердження та регулювання транзакцій. Вона базується на принципах криптографії, де математичні алгоритми забезпечують захист фінансових операцій та переведення власності[1].

Криптовалюти використовують децентралізовані системи, такі як блокчейн, для збереження інформації про власників та транзакції. Блокчейн – це публічний розподілений реєстр, який містить всю історію транзакцій у мережі. Він дозволяє учасникам мережі перевіряти та підтверджувати транзакції без необхідності довіряти централізованому органу. Це робить криптовалюти більш прозорими, незалежними від урядових органів та банків, і дає можливість користувачам здійснювати безпечні та анонімні транзакції.

Основна ідея криптовалют полягає в тому, що вони дозволяють користувачам обмінюватися цифровими активами без посередництва та безпеки централізованих фінансових установ. Криптовалюти можуть використовуватися для різних цілей, включаючи платежі за товари і послуги, інвестування, зберігання активів та трансграничні перекази.

У світі криптовалют існує безліч різних валют, таких як Bitcoin, Ethereum, Ripple та багато інших. Кожна з цих валют має свої особливості, протоколи та функціональні можливості. Принципи криптовалют можуть варіюватися в залежності від конкретної валюти, але загальні принципи безпеки, децентралізації та анонімності залишаються основними складовими будь-якої криптовалюти.

1.1.1 Децентралізація в криптовалютах

Децентралізація в криптовалютах є одним з основних принципів, що відрізняють їх від традиційних фінансових систем. В традиційній моделі фінансового обігу, банки та фінансові установи виступають посередниками, що контролюють та регулюють всі транзакції. У криптовалютних мережах, ця роль виконується самими учасниками мережі.

Децентралізована природа криптовалют означає, що не існує одного центрального органу або авторитету, що контролює всі операції. Замість цього, транзакції виконуються безпосередньо між учасниками мережі. Це забезпечує більшу свободу, незалежність від урядових регуляторів та більшу довіру до фінансових операцій.

Одним з ключових елементів децентралізованих криптовалют є технологія блокчейн. Блокчейн – це розподілена публічна книга, що містить записи всіх транзакцій, які були здійснені в мережі. Кожен блок містить інформацію про транзакції та попередній блок, утворюючи послідовний ланцюжок блоків.[2] Це забезпечує прозорість та недоступність для змінення вже збережених даних.

За допомогою блокчейна, учасники мережі можуть спільно підтверджувати транзакції та формувати консенсус щодо стану мережі без необхідності довіряти централізованому органу. Це забезпечує більшу безпеку та надійність мережі, оскільки жоден окремий учасник не має можливості контролювати

1.1.2 Журнал транзакцій та його роль у роботі криптовалют

Журнал транзакцій(або distibuted ledger) є ключовим елементом криптовалют та відіграє важливу роль у їхній роботі. Він є центральним компонентом блокчейн-технології, яка забезпечує децентралізований та безпечний обмін цифровими активами.[3]

Журнал транзакцій, є розподіленою публічною книгою, в якій зберігаються всі транзакції, що відбуваються в мережі криптовалюти. Кожна транзакція реєструється в окремому блоку, який потім додається до ланцюжка блоків. Кожен блок містить інформацію про попередній блок, що забезпечує послідовність та недоступність для змінення вже збережених даних. Це забезпечує прозорість та надійність системи, оскільки будь-які зміни в одному блоку відразу стають очевидними для всіх учасників мережі.

Роль журналу транзакцій полягає в тому, щоб фіксувати та підтверджувати всі транзакції, що відбуваються в мережі криптовалюти. Кожна транзакція, незалежно від свого походження, суми або учасників, отримує унікальний ідентифікатор і вноситься в блокчейн. Це дозволяє кожному учаснику мережі перевірити легітимність та статус будь-якої транзакції. Крім того, журнал транзакцій забезпечує історичну статистику та статус рахунків, що дає змогу вести аналіз, контролювати ризики та забезпечувати безпеку мережі.

Журнал транзакцій також відіграє важливу роль у механізмах консенсусу, які дозволяють учасникам мережі досягати згоди щодо стану системи. Завдяки блокчейну та журналу транзакцій, учасники можуть підтверджувати правильність транзакцій шляхом розрахунку хеш-сум та перевірки попередніх блоків. Цей механізм забезпечує безпеку, недоступність до змінення даних та запобігає можливим атакам на систему.

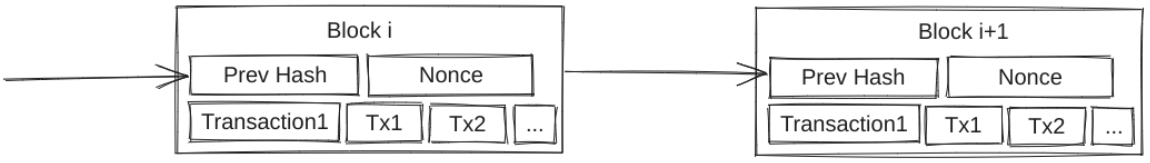

Рис.1 Два послідовні блоки блокчейну

1.1.3 Принципи консенсусу (proof of work, proof of stake)

«Proof of work» та «proof of stake» є двома основними механізмами консенсусу, які використовуються криптовалютами для перевірки нових транзакцій, додавання їх до блокчейну та створення нових токенів. «Proof of work», який вперше був впроваджений Bitcoin, використовує майнінг для досягнення цих цілей. «Proof of stake», який застосовується в Cardano, блокчейні ETH2 та інших криптовалютах, використовує стейкінг для досягнення тих самих результатів.

Децентралізованим мережам криптовалют необхідно переконатись, що ніхто не може витратити одні й ті ж гроші двічі без наявності центральної влади, такої як Visa або PayPal. Для досягнення цієї мети мережі використовують так званий «механізм консенсусу», який є системою, що дозволяє всім комп'ютерам у крипто-мережі згодитись щодо того, які транзакції є законними.

На сьогоднішній день існує два основних механізми консенсусу, які використовуються більшістю криптовалют. «Proof of work» є старішим з них і використовується Bitcoin, Ethereum 1.0 та багатьма іншими. Новіший механізм консенсусу називається «proof of stake» і використовується Ethereum 2.0, Cardano, Tezos та іншими криптовалютами.

«Proof of work» є першоджерелом механізму консенсусу у криптовалютах. Вперше використаний Bitcoin, він тісно пов'язаний з ідеєю майнінгу. Назва «proof of work» виникла через необхідність мережі великої обчислювальної потужності. Блокчейни, які використовують «proof of work», захищаються та верифікуються віртуальними майнерами з усього світу, які змагаються, щоб першими розв'язати математичну головоломку. Переможець отримує можливість оновити блокчейн із останніми верифікованими транзакціями та отримує винагороду від мережі визначеною кількістю криптовалюти.

«Proof of work» має кілька потужних переваг, особливо для відносно простих, але надзвичайно цінних криптовалют, таких як Bitcoin. Це

перевірений і надійний спосіб збереження безпечного децентралізованого блокчейну. Зі зростанням ціни на криптовалюту, більше майнерів мають стимул приєднуватись до мережі, що збільшує її потужність та безпеку. Через велику кількість обчислювальної потужності втручання окремої особи чи групи в блокчейн цінної криптовалюти стає непрактичним. [4]

З іншого боку, цей процес потребує значних енергетичних ресурсів і може мати проблеми з масштабуванням для обробки великої кількості транзакцій, які здатні забезпечувати блокчейни, сумісні зі смарт-контрактами, такі як Ethereum. Тому були розроблені альтернативи, найпопулярнішою з яких є «proof of stake».

Розробники Ethereum розуміли, що «proof of work» має обмеження щодо масштабованості, які з часом потрібно буде подолати. Зі зростанням популярності протоколів децентралізованої фінансової сфери (DeFi), побудованих на базі Ethereum, блокчейн почав мати проблеми зі швидкістю, що призвело до зростання комісій.

Відповідно до рішення розробників Ethereum, було розроблено зовсім новий блокчейн ETH2, який почав впроваджуватись у грудні 2020 року. У ETH2 використовується швидкий та менш енергозатратний механізм консенсусу, відомий як «proof of stake». Крім Ethereum, криптовалюти, такі як Cardano, Tezos та Atmos, використовують механізми консенсусу на основі «proof of stake» з метою максимізації швидкості та ефективності та зниження комісій.

У системі «proof of stake» стейкінг виконує подібну функцію до майнінгу в системі «proof of work», оскільки це процес, за яким учасник мережі вибирається для додавання останньої партії транзакцій до блокчейну та отримання криптовалюти в обмін.[5] Деталі можуть відрізнятися залежно від проекту, але, в цілому, блокчейни, що використовують «proof of stake», використовують мережу «валідаторів», які вносять власну криптовалюту в обмін за можливість перевірки нових транзакцій, оновлення блокчейну та отримання винагороди.

Мережа вибирає переможця на основі кількості криптовалюти, яку вніс кожен валідатор до пулу, та тривалості його перебування там, буквально винагороджуючи найбільш залучених учасників. Після того, як переможець перевірив останній блок транзакцій, інші валідатори перевіряють його роботу, щоб забезпечити консенсус, і якщо результати перевірки успішні, блок додається до блокчейну.

Загалом, «proof of stake» вважається більш енергоекономічним і швидким механізмом консенсусу порівняно з «proof of work». Однак він також має свої недоліки, такі як проблеми з безпекою при концентрації влади у великих валідаторах.

1.2 Застосування криптографії в криптовалютах

Криптографія є невід'ємною складовою криптовалют і блокчейн технології. Вона використовується для забезпечення безпеки та конфіденційності транзакцій, аутентифікації користувачів і захисту від несанкціонованого доступу. Криптографічні алгоритми, такі як еліптична крива криптографія та хеш-функції, дозволяють генерувати приватні та публічні ключі, підписувати та перевіряти транзакції, а також забезпечувати безпечне зберігання активів.

1.2.1 Генерація приватних та публічних ключів.

Bitcoin використовує еліптичну криптографію (ECC) та алгоритм безпечного хешування SHA-256 для генерації публічних ключів з відповідних приватних ключів.

Публічний ключ використовується для створення адреси криптогаманця для отримання вхідних транзакцій, тоді як приватний ключ необхідний для підпису транзакцій та підтвердження власності коштів.

Приватний ключ – це важлива частина ключової пари та зберігається в криптогаманці. Технічно, криптогаманець зберігає доступ особи до їх криптовалюти, а не саму криптовалюту. Самі кошти є лише записами

даних, збереженими в блокчейні, і можуть бути ідентифіковані та розблоковані за допомогою ключів, збережених у вашому гаманці.

1.2.2 Еліптична криптографія

Еліптична криптографія – це використання особливої математичної кривої, яка має горизонтальну симетрію. Якщо ви проведете будь-яку лінію через цю криву, вона перетне криву максимум тричі. Еліптичні криві є важливою частиною криптовалюти і дозволяє користувачам генерувати публічний ключ.

Для генерації ключової пари Bitcoin спочатку потрібно створити приватний ключ.

Приватний ключ Bitcoin – це випадково згенероване 256-бітне число. Це число зазвичай генерується в момент створення криптогаманця.[6]

Публічний ключ потім генерується з цього числа за допомогою множення на еліптичну криву. Це включає в себе вибір початкової точки на еліптичній кривій (відомої як точка генератора) та множення її на випадкове число приватного ключа для отримання нової точки на кривій.

Ця нова точка стає публічним ключем зі специфічними координатами x та y. Знаходження приватного ключа за відомим публічним ключем майже неможливе через складність задачі дискретного логарифмування.

Теоретично, для знаходження цього числа протягом одного дня потрібен квантовий комп'ютер з більш ніж 13,000,000 фізичних кубітів. На сьогоднішній день один з найбільш сучасних квантових комп'ютерів у світі, процесор IBM Osprey, має лише 433 кубітів.

Іншими словами, системи, які використовуються криптовалютами є наразі повністю безпечними.

1.2.3 Хеш-функції

Криптографічна хеш-функція ("cryptographic hash function") – це алгоритм хешування (відображення довільного двійкового рядка в бінарний рядок фіксованого розміру n-бітів).[7] Ідеальна криптографічна геш-функція має три основні властивості:

а) Ймовірність отримання певного значення хешу (хеш-значення) розміру n-бітів для випадкового вхідного рядка ("повідомлення") становить 2^{-n} (як у будь-якого надійного хешування), тому хеш-значення може використовуватись як представник повідомлення.

б) Знаходження вхідного рядка, який відповідає заданому значенню хешу (попередній образ), є недосяжним, якщо значення не вибране з відомого попередньо розрахованого словника ("rainbow table"). Стійкість до такого пошуку характеризується як стійкість попереднього образування, а криптографічний хеш з хеш-значенням розміром n бітів має очікувану стійкість попереднього образування в розмірі n бітів.

в) Знаходження будь-якої пари різних повідомлень, що мають однакове значення хешу (колізія), також є недосяжним. Криптографічний хеш очікується мати стійкість до колізій в розмірі n/2 бітів (менша через парадокс днів народження).

1.2.3.1 Властивості криптографічних хеш-функцій

Більшість криптографічних хеш-функцій призначені для приймання рядка будь-якої довжини як вхідних даних та генерації хеш-значення фіксованої довжини.

Криптографічна хеш-функція повинна витримувати всі відомі типи криптоаналітичних атак. У теоретичній криптографії рівень безпеки криптографічної хеш-функції визначається наступними властивостями:

а) Стійкість до відтворення. Для заданого значення хешу h важко знайти будь-яке повідомлення m таке, що $h = hash(m)$. Це поняття пов'язане з односторонньою функцією. Функції, які не мають цієї властивості, вразливі до атак на відтворення.

б) Стійкість до вторинного відтворення. Для заданого вхідного значення m1 важко знайти інше вхідне значення $m2$ таке, що $hash(m1) = hash(m2)$. Ця властивість іноді називається слабкою стійкістю до колізій. Функції, які не мають цієї властивості, вразливі до атак на вторинне відтворення.

в) Стійкість до колізій. Важко знайти два різних повідомлення $m1$ і $m2$ такі, що $hash(m1) = hash(m2)$. Така пара називається криптографічною колізією. Ця властивість іноді називається сильною стійкістю до колізій. Вона вимагає значення хешу принаймні вдвічі довшого, ніж потрібно для стійкості до відтворення, в іншому випадку колізії можуть бути знайдені за допомогою атаки «днів народження».

1.2.3.2 Криптографічної хеш-функції в криптовалютах

Криптографічні функції SHA – це сімейство криптографічних функції. Наприклад, для створення адреси біткойн-гаманця використовується алгоритм SHA-256, який хешує публічний ключ гаманця.

Цей криптографічний хеш-алгоритм був розроблений та опублікований Національною службою безпеки США (NSA) у 2001 році і змінює будь-який вхідний параметр (у цьому випадку, координати публічного ключа) в унікальний код фіксованої довжини 256 біт.

SHA-3 (Secure Hash Algorithm 3) – останній представник сімейства стандартів Secure Hash Algorithm, який був випущений NIST 5 серпня 2015 року.

Ethereum використовує KECCAK-256[8], що є аналогом алгоритму SHA-3. Слід зазначити, що він не використовує стандарт, заснований на FIPS-202 (відомий також як SHA-3), оскільки імплементація відрізняється.

2 ЦИФРОВІ ПІДПИСИ

2.1 Загальний опис

Цифровий підпис – це математична схема для перевірки автентичності цифрових повідомлень або документів. Дійсний цифровий підпис, за умови виконання передбачених умов, надає отримувачу високу впевненість у тому, що повідомлення було створено відомим відправником (автентичність) і не було змінено під час передачі (цілісність)[9].

Цифрові підписи є стандартним елементом більшості наборів криптографічних протоколів і широко використовуються для розповсюдження програмного забезпечення, фінансових транзакцій, управління договорами та інших випадках, де важливо виявити підробку або втручання.

Цифрові підписи використовують асиметричну криптографію. У багатьох випадках вони забезпечують шар перевірки та безпеки повідомлень, які надсилаються через незахищений канал: правильно реалізований цифровий підпис дає отримувачу підставу вірити, що повідомлення було надіслано зазначеним відправником. Цифрові підписи еквівалентні традиційним рукописним підписам в багатьох відношеннях, але правильно реалізовані цифрові підписи складніше підробити, ніж рукописні. Цифрові підписи базуються на криптографії і мають бути правильно реалізовані для ефективності. Вони також можуть забезпечувати незаперечність, що означає, що підписник не може успішно стверджувати, що він не підписав повідомлення, при цьому стверджуючи, що його приватний ключ залишається секретним. Крім того, деякі схеми незаперечності пропонують відмітку часу для цифрового підпису, щоб, навіть якщо приватний ключ став відомим, підпис залишався дійсним. Цифрові підписи можуть стосуватися будь-яких даних, які можна представити у вигляді бітового рядка, наприклад, електронної пошти,

договорів або повідомлення, надісланого за допомогою іншого криптографічного протоколу.

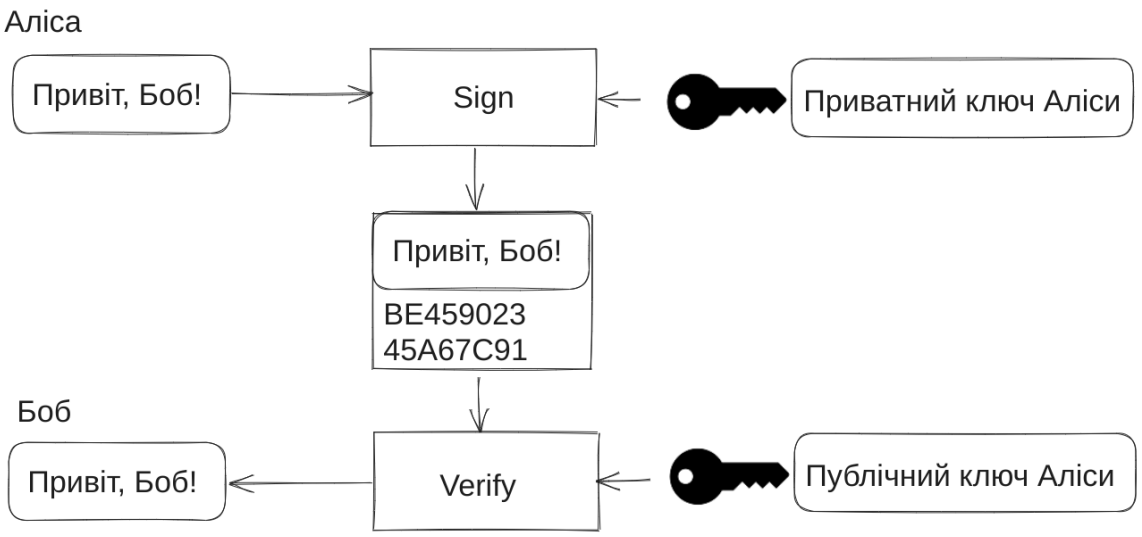

Рис.2 Схематичне зображення підпису та верифікації повідомлення

2.1.1 Формальне означення

Схема цифрового підпису це трійка $\Sigma = (Gen, Sign, Vfy)$ трьох ймовірнісних поліноміальних алгоритмів:

а) $Gen(1^k) \rightarrow (pk, sk)$ – алгоритм генерації пари публічного та приватного ключів

б) $Sign(sk, m) \rightarrow \sigma$ – алгоритм створення підпису

в) $Vfy(pk, m, \sigma) \rightarrow \{0, 1\}$ – алгоритм верифікації цифрового підпису

Де $k \in N$ це параметр безпеки(зазвичай довжина ключа); sk – приватний ключ; pk – публічний ключ; m – повідомлення, що підписужться та верифікується схемою цифровго підпису, σ – результат цифрового підпису.[10]

Трійка алгоритмів повинна задовільняти наступну умову:

$\forall k, m \in \{0, 1\}^* , \forall (pk, sk) \leftarrow Gen(1^k) : Vfy(pk, m, Sign(sk, m)) = 1$

Параметр безпеки схеми цифрового підпису визначається цілим додатним числом n. Алгоритм генерації ключів G використовує це число n як вхідний параметр, який визначає довжину вихідних даних – пару

рядків. Кожна пара $(pk,\ sk)$ у діапазоні $Gen(1^n)$ представляє відповідні ключі для підписання та верифікації, тобто пару приватного та публічного ключі криптосистеми.

Підпис S(s, m) вказує на підпис документа або повідомлення m, створений з використанням приватного ключа підписання s. Аналогічно, коли V(v, m, β) = 1, це означає, що β є правомірним підписом для m з використанням публічного ключа перевірки v.

Пара приватного та публічного ключів $(pk,\ sk)$, згенерована алгоритмом генерації ключів Gen, разом із трійкою алгоритмів $\Sigma = (Gen,\ Sign,\ Vfy)$ утворює екземпляр схеми цифрового підпису $\Sigma = (Gen,\ Sign,\ Vfy)$.

Схема цифрового підпису використовується наступним чином. Припустимо, що Аліса бажає підписати повідомлення. Вона використовує алгоритм генерації приватного та публічного ключа $Gen(1^n)$ і отримує пару $(pk,\ sk)$. Згідно з вимогами схеми цифрового підпису, будь-яка інша сторона може отримати легітимну копію публічного ключа pk. Тому Аліса може розповсюджувати свій публічний ключ pk.

Коли Аліса бажає підписати повідомлення m, вона використовує алгоритм підпису $Sign$ і обчислює підпис $Sign(sk,\ m)$ за допомогою свого приватного ключа sk. Тепер Аліса може надіслати пару (m, Sign(sk, m)). Отримувач, знаючи відкритий публічний ключ pk, може перевірити автентичність повідомлення m, застосовуючи алгоритм верифікації Vfy і перевіряючи, чи $Vfy(pk,\ m,\ Sign(sk,\ m)) = 1$. Таким чином Боб розуміє, що саме Аліса відправила йому повідомлення і це повідомлення не було модифіковане під час передачі.

2.2.1 RSA

Схема RSA цифрового підпису складається з трьох алгоритмів $\Sigma = (Gen,\ Sign,\ Vfy)$, які виконують наступні дії. Алгоритм генерації

ключів $Gen(1^n)$ обирає два великі прості числа p і q довжиною n біт, обчислює їх добуток $N = pq$, а потім обирає число e, яке менше $\varphi(N)$ і взаємнопросте з $\varphi(N)$. За допомогою розширеного алгоритму Евкліда знаходиться число d, що задовольняє рівняння $ed \equiv 1(mod\varphi(N))$. Отримані значення (N, e) утворюють публічний ключ, а (N, d) – приватний ключ. [11]

Алгоритм підписання повідомлення S отримує на вхід приватний ключ (N, d) та повідомлення m. Він обчислює підпис $s = S((N, d), m) = m\hat{}d \bmod N$.

Алгоритм верифікації підпису Vfy отримує на вхід публічний ключ (N, e), підпис s та повідомлення m. Він обчислює $m' = se \bmod N$ та перевіряє, чи $m' = m$. Якщо ця рівність виконується, то підпис s вважається дійсним для повідомлення m відносно публічного ключа (N, e).

Хоча схема RSA вважається безпечною від атак класичного комп'ютера при достатньо великому параметрі безпеки, квантовий алгоритм Шора може ефективно розв'язати задачу факторизації за поліноміальний час.

2.2.1.1 Вразливості криптосистеми RSA

Криптосистема RSA є вразливою до двох основних видів потенційно можливих атак: атаки цілочисельної факторизації та атаки з використанням мультиплікативної властивості RSA.

Атака цілочисельної факторизації полягає у тому, що якщо зловмисник може факторизувати, тобто розкласти на прості множники, публічний модуль N, то потім він зможе обчислити $\varphi(N)$ та, використовуючи алгоритм Евкліда, знайти приватний ключ d з $\varphi(N)$ та публічної експоненти e, розв’язавши рівняння $ed \equiv 1 \bmod \varphi$. Щоб уникнути можливості такої атаки, p та q повинні бути такими, щоб факторизація $N = pq$ була дуже складною задачею.

Мультиплікативну властивість RSA іноді називають гомоморфною властивістю. Якщо $s1 = m1\ d\ mod\ N$ та $s2 = m2\ d\ mod\ N$ є підписами до повідомлень $m1$ та $m2$ відповідно, то $s = s1s2$ має таку властивість, що $s = (m1m2)d\ mod\ n$. Якщо $m = m1m2$, то s є дійсним підписом до повідомлення m.

Вирішенням цієї проблеми є підписання не повідомлення *m*, а підписання натомість $m' = R(m)$, де *R* – функція надмірності, яка переводить повідомлення в простір $\mathbb{Z}_n$. Відповідно, при відновленні зашифрованого повідомлення на останньому етапі *m* обчислюється як $m = R - 1(m')$. Важливо, щоб ця функція була не мультиплікативною, тобто для усіх $a,\ b$, що належать простору допустимих повідомлень, $R(ab) \neq R(a)R(b)$.

2.2.2 DSA

Алгоритм цифрового підпису DSA базується на концепціях підпису Ель-Гамаля та Шнорра і використовує складність обчислення логарифмів у скінченних полях. Цей алгоритм був розроблений Національним інститутом стандартів та технологій США і отримав патент у 1991 році[12].

Переваги схеми DSA включають скорочену довжину підпису порівняно з схемою Ель-Гамаля при однаковому рівні безпеки, більш швидкі обчислення порівняно з схемою Ель-Гамаля та менші потреби в просторі для зберігання даних.

Недолік схеми DSA полягає у тому, що верифікація підпису може включати складні операції з залишками, що негативно впливає на швидкість обчислень.

Схема цифрового підпису DSA складається з трьох алгоритмів $\Sigma = (Gen,\ Sign,\ Vfy)$, які можна описати наступним чином.

Алгоритм генерації ключів Gen має два етапи. На першому етапі встановлюються параметри системи. Для цього обираються криптографічна геш-функція H, яка має стійкість до колізій (найчастіше це алгоритми з хешування SHA). Потім вибирається просте число q, розмір якого відповідає розмірності значень геш-функції $Hash(x)$ і позначається як N. Обирається також просте число p, яке є дільником $(p - 1)$. Параметр L відповідає бітовій довжині p, і виконується нерівність $2^{L-1} < p < 2^{L}$. Наступним кроком є вибір числа g, яке має мультиплікативний порядок q за модулем p. Цей вибір здійснюється за формулою $g = h^{(p-1)/q}$, де h – довільне число з діапазону $(1, p - 1)$ і $g \neq 1$. Часто використовується значення $h = 2$. Трійка (p, q, g) може бути використана будь-якими користувачами системи. Важливими параметрами безпеки є геш-функція $Hash$, бітова довжина вхідної послідовності N і параметр L. Стандарт цифрових підписів рекомендує такі можливі значення пар (L, N): (2048, 224), (3072, 256), (2048, 256), (1024, 160), з використанням сімейства геш-функцій SHA. На другому етапі генеруються приватний і публічний ключі для окремого користувача. Випадковим чином вибирається значення x, де $0 < x < q$. Далі обчислюється $y = g^{x \bmod p}$. Отже, відкритим ключем є структура (p, q, g, y), а закритим ключем – значення x.

Алгоритм підпису повідомлення $Sign$ здійснює наступні дії для підпису повідомлення m.

а) Випадково обирається значення k, де $1 < k < q$.

б) Обчислюється $s \equiv (Hash(m) + xr)k^{-1} \ (mod\ q)$ та $r \equiv g \ (mod\ q)$.

в) У випадку, якщо $s = 0$ або $r = 0$, алгоритм підпису повторюється.

г) Підписом повідомлення є пара (r, s).

Алгоритм верифікації підпису Vfy приймає публічний ключ $(p,\ q,\ g,\ y)$, підпис $(r,\ s)$ та повідомлення m. Він перевіряє, чи $0\ <\ r\ <\ q$ та $0\ <\ s\ <\ q$, якщо ні, то підпис не є дійсним. Потім обчислюються значення $w\ =\ s^{-1}\ (mod\ q)$,$u1\ =\ (Hash(m)w)\ (mod\ q)$, $u2\ =\ (rw)\ (mod\ q)$,$v\ =\ ((g^{u1}y^{u2})\ (mod\ p))\ (mod\ q)$. Алгоритм Vfy підтверджує, що s – дійсний підпис повідомлення m, тоді і тільки тоді, коли $v\ =\ r$.

Алгоритм є коректним у тому сенсі, що згенеровані підписи завжди можна перевірити, і вони дійсно свідчать про те, чи було насправді підписане повідомлення. Якщо $g\ =\ h^{(p-1)/q}$, то $g^q \equiv h^{(p-1)} \equiv 1\ (mod\ p)$ за малою теоремою Ферма. Оскільки $g\ >\ 1$ та q – просте, g має порядок q. Підписант обчислює $s \equiv (Hash(m)\ +\ xr)k^{(-1)}(mod\ q)$, тому $k \equiv Hash(m)s^{(-1)}\ +\ xrs^{(-1)}\ =\ Hash(m)w\ +\ xrw$. Оскільки g має порядок $q\ (mod\ p)$: $g \equiv g^{Hash(m)w}g^{xrw} \equiv g^{u1}g^{u2}\ (mod\ p)$. Нарешті, коректність DSA випливає з наступної рівності:

$r\ =\ (g^k\ (mod\ p))\ (mod\ q)\ =\ (g^{u1}y^{u2}\ (mod\ p))\ (mod\ q)\ =\ v$.[13]

2.2.3 ECDSA

ECDSA, або Elliptic Curve Digital Signature Algorithm, є алгоритмом цифрового підпису, який був запропонований в 1992 році.[14] Він використовує принципи DSA, але замість кільця цілих чисел використовує криптографію на еліптичних кривих. Тобто безпека алгоритму ґрунтується на складності обчислення дискретного логарифму в полі цілих чисел.

Перевагами ECDSA є його придатність для використання в менших полях, порівняно з алгоритмом DSA; відсутність проблем з продуктивністю; швидкість процесу підписування та перевірки підпису; відповідність постійно зростаючим вимогам безпеки; підтримку національних стандартів безпеки.

ECDSA – цифровий підпис з трьох алгоритмів $\Sigma = (Gen, Sign, Vfy)$, що використовують криптографічну геш-функцію $Hash$ (зазвичай обирають геш-функцію сімейства SHA).

Для створення пар відкритого та закритого ключів за допомогою доменних параметрів ECDSA. Доменні параметри $(Ep(a, b), G, n)$ включають обрану еліптичну криву з полем та рівнянням, генератор G та просте ціле число n, що визначає порядок мультиплікативної підгрупи точки Gen. Для генерації ключів, G випадково обирає приватний ключ d, і публічний ключ Q обчислюється як $Q = d * G$. Для підпису повідомлення m, обчислюється його геш-значення $l = Hash(m)$, вибирається випадкове k, обчислюється точка $(x1, y1) = k * G, r = x1 \, mod \, n, s = k^{(-1)}(l + rd) \, mod \, n$. Підпис представляється як (r, s), а $(r, -s \, mod \, n)$ також є дійсним підписом.

Алгоритм верифікації підпису Vfy перевіряє, чи є точка Q припустимою на еліптичній кривій, шляхом вивчення трьох утверджень: точка Q не є ідентичним елементом O і має допустимі координати, точка Q лежить на кривій, і добуток порядку генератора G, n, та точки Q дорівнює ідентичному елементу, тобто $n * Q = 0$. Якщо хоча б одне з цих утверджень є неправдивим, алгоритм верифікації підпису Vfy вважає підпис недійсним. У противному випадку, перевіряється, чи належать частини підпису (r, s) діапазону $(1, n)$, тобто чи виконуються нерівності $1 < r < n$ та $1 < s < n$ одночасно. Якщо це не так, підпис вважається недійсним, інакше продовжуються наступні кроки. Обчислюється геш-значення повідомлення m, позначимо його як $l = Hash(m)$. Для перевірки можуть використовуватись крайні ліві біти гешу l, кількість яких дорівнює бітовій довжині порядку групи n. Обчислюються значення $u1 = ls^{-1} mod \, n$ та $u2 = rs^{-1} mod \, n$. Знаходиться точка на кривій $(x1, y1) = u1 * G + u2 * Q$. Якщо

$(x1,\ y1)\ =\ 0$, підпис вважається недійсним. Підпис вважається дійсним тільки тоді, коли $r \equiv x1\ (mod\ n)$.

Алгоритм є коректним, оскільки згенеровані підписи завжди можна перевірити, і вони дійсно підтверджують, чи було насправді підписане повідомлення. Позначимо точку на кривій з алгоритму верифікації підпису як $P\ =\ (x1,\ y1)\ =\ u1\ *\ G\ +\ u2\ *\ Q$. Враховуючи визначення, $Q\ =\ d\ *\ G$, маємо, що $P\ =\ u1\ *\ G\ +\ u2\,d\ *\ G$, і з властивостей точок еліптичних кривих отримуємо, що $P\ =\ (u1\ +\ u2d)\ *\ G$. Використовуючи визначення $u1$ та $u2$, отримуємо $P\ =\ (ls^{-1}\ +\ rs^{-1}d)\ *\ G\ =\ (l\ +\ rd)s^{-1}\ *\ G$. У алгоритмі підписання повідомлення *S*, ми маємо $S\ =\ k\ -\ 1(Hash(m)\ +\ rd)\ mod\ n$, тому $P\ =\ (l\ +\ rd)(l\ +\ rd)^{(-1)}(k^{-1})\ \ *\ G$. Оскільки обернене значення оберненого значення елемента є самим елементом, а добуток елемента на обернений утворюють нейтральний елемент, отримуємо $P\ =\ k\ *\ G$, що перевіряється в алгоритмі верифікації, де $r \equiv x1(mod\ n)$. Таким чином, правильний підпис завжди є дійсним. [15]

2.2.3.1 Чому біткоїн використовує secp256k1

Еліптична криптографія є основою для серії криптографічних схем з використанням публічного ключа, наприклад, схем підпису, шифрування та передачі ключів, а також схем узгодження ключів. Взагалі, ці схеми передбачають арифметичні операції на еліптичній кривій над скінченним полем. Secp256k1 та secp256r1 є двома поширеними кривими. Hyperledger/Fabric, розроблений IBM, використовує secp256r1, тоді як Bitcoin використовує secp256k1. Яка різниця між ними, і чому Сатоші вирішив використовувати secp256k1, що вважалося несподіваним вибором на той час?

Основна відмінність між secp256k1 та secp256r1 полягає в тому, що

secp256k1 є кривою Кобліца, яка визначена у скінченному полі характеристики 2, тоді як secp256r1 є кривою над простим полем. Зверніть увагу, що просте поле та скінченне поле характеристики 2 є лише двома типами скінченних полів, які використовуються SECG. Криві secp256k1 є недовільними, тоді як secp256r1 має псевдовипадкову структуру. Навіть якщо загалом відомо, що криві Кобліца є на кілька бітів менш безпечними, ніж криві над простим полем, в разі 256-бітових кривих це майже не впливає.

Secp256k1 є чистою кривою SECG, тоді як secp256r1 є так званою кривою NIST. Криві NIST ширше використовуються і отримали більше уваги, ніж інші криві SECG. Це загалом вважається причиною, чому Сатоші не використовував secp256r1. Зокрема, витеклі документи від підрядника Національного агентства з безпеки та розкривача інформації, що NSA використовувало свій вплив на NIST, щоб використати вразливість у генератор випадкових чисел, який використовується в стандартах еліптичної криптографії. Не знаючи про це, Сатоші хотів знизити ризик наявності вразливості в кривій, яку він реалізує, а оскільки NIST і NSA дуже близькі, він, ймовірно, віддав би перевагу чистій кривій SECG. [16]

2.2.4 EdDSA

EdDSA (Edwards-curve Digital Signature Algorithm)) - це сучасний та безпечний алгоритм цифрового підпису, який базується на оптимізованих з точки зору продуктивності еліптичних кривих, таких як 255-бітова крива Curve25519 і 448-бітова крива Curve448. Підписи EdDSA використовують форму Едвардса еліптичних кривих (з метою поліпшення продуктивності), відповідно edwards25519 і edwards448. Алгоритм EdDSA базується на алгоритмі підпису Шнорра та покладається на складність задачі дискретного логарифмування на еліптичних кривих (ECDLP).

Алгоритм підпису EdDSA та його варіанти Ed25519 і Ed448 технічно описані в RFC 8032. [17][18]

2.2.4.1 Генерація ключів EdDSA

Ed25519 та Ed448 використовують невеликі приватні ключі (відповідно 32 або 57 байт), невеликі публічні ключі (відповідно 32 або 57 байт) та невеликі підписи (відповідно 64 або 114 байт), забезпечуючи високий рівень безпеки (відповідно 128 або 224 біт).
Передбачається, що еліптична крива для алгоритму EdDSA має точку-генератор G та порядок підгрупи q для точок на криві, що генеруються з точки G.
Пара ключів EdDSA складається з:

а) приватного ключа (ціле число): $privKey$

б) публічного ключа (точка еліптичної кривої):

$pubKey = privKey * G$

Приватний ключ генерується з випадкового цілого числа, відомого як "seed" (яке повинно мати подібну довжину в бітах, як порядок кривої). "Seed" спочатку хешується, потім останні кілька бітів, що відповідають фактору кривої (8 для Ed25519 та 4 для Ed448), очищаються, потім найстарший біт очищається, а другий найстарший біт встановлюється. Ці перетворення гарантують, що приватний ключ завжди буде належати до тієї самої підгрупи точок еліптичної кривої та приватні ключі завжди матимуть подібну довжину в бітах (щоб захиститися від атак на основі витоку часу). Для Ed25519 приватний ключ складається з 32 байтів. Для Ed448 приватний ключ складається з 57 байтів.

Публічний ключ pubKey є точкою на еліптичній кривій, обчислений шляхом множення точки еліптичної кривої: $pubKey = privKey * G$ (приватний ключ, помножений на точку-генератор G для кривої).
Публічний ключ кодується як стисла точка еліптичної кривої: y-координата, поєднана з найнижчим бітом (парністю) x-координати. Для

Ed25519 публічний ключ складає 32 байти. Для Ed448 публічний ключ складає 57 байтів.

2.2.4.2 Алгоритм створення підпису EdDSA

Алгоритм підпису EdDSA (RFC 8032) приймає на вхід текстове повідомлення msg та приватний ключ EdDSA privKey і генерує на виході пару цілих чисел $(R,\ s)$. Процес підпису EdDSA виконується наступним чином: $sign(msg,\ privKey)\ \rightarrow\ (R,\ s\ )$

1) Обчислити $pubKey\ =\ privKey\ *\ G$
2) Обчислити секретне ціле число $r\ =\ hash(hash(privKey)\ +\ msg)\ mod\ q$
3) Обчислити публічний ключ R, множачи його за допомогою генератора кривої: R = r * G
4) Обчислити $h\ =\ hash(R\ +\ pubKey\ +\ msg)\ mod\ q$
5) Обчислити $s\ =\ (r\ +\ h\ *\ privKey)\ mod\ q$
6) Повернути підпис $(R,\ s)$

Отриманий цифровий підпис складає 64 байти (32 + 32 байти) для Ed25519 та 114 байтів (57 + 57 байтів) для Ed448. Він містить стислу точку R та ціле число s (підтвердження того, що підписник знає повідомлення та приватний ключ).

2.2.4.3 Алгоритм верифікації підпису EdDSA

Алгоритм перевірки підпису EdDSA (RFC 8032) приймає на вхід текстове повідомлення msg, публічний ключ підписника EdDSA pubKey та підпис EdDSA $(R,\ s)$ і генерує на виході булеве значення (правильний або неправильний підпис): $verify(msg,\ pubKey,\ (\ R,\ s\ )\ )\ \rightarrow\ \{true,\ false\}$.

Процес перевірки EdDSA виконується наступним чином:

1) Обчислити $h\ =\ hash(R\ +\ pubKey\ +\ msg)\ mod\ q$
2) Обчислити $P1\ =\ s\ *\ G$
3) Обчислити $P2\ =\ R\ +\ h\ *\ pubKey$
4) Повернути $P1\ ==\ P2$

2.2.4.4 Принцип роботи EdDSA

Під час перевірки обчислюється точка $P1$ за формулою: $P1\ =\ s\ *\ G$. Під час підписування $s\ =\ (r\ +\ h\ *\ privKey)\ mod\ q$. Тепер підставимо s в формулу:

$$P1 = s * G = (r + h * privKey) \bmod q * G =$$
$$= r * G + h * privKey * G = R + h * pubKey$$

Вищезазначене дорівнює іншій точці $P2$. Якщо ці точки $P1$ і $P2$ є одною точкою на еліптичній кривій, це означає, що точка $P1$, обчислена за допомогою приватного ключа, відповідає точці $P2$, створеній його відповідним публічним ключем.

2.2.5 Різниця між ECDSA та EdDSA

Якщо порівняти алгоритми підпису та верифікації для EdDSA, то зрозуміло, що EdDSA є простішим за ECDSA, його легше розуміти та реалізувати. Обидва алгоритми підпису мають подібну стійкість до атак при використанні кривих з подібною довжиною ключа.[19] Для найпопулярніших кривих (наприклад, curve25519 і curve448) алгоритм EdDSA трохи швидший за ECDSA, але це сильно залежить від конкретних кривих та реалізації. На відміну від ECDSA, підписи EdDSA не надають можливості відновлення публічного ключа від підпису та повідомлення. [20]

3 РЕАЛІЗАЦІЇ ПРОГРАМИ ДЛЯ ВИКОРИСТАННЯ ЦИФРОВИХ ПІДПИСІВ

3.1 Опис реалізації

На підставі попередніх розділів було вибрано два алгоритм схеми цифрового підпису: алгоритм ECDSA з кривою secp256k1 та алгоритм EdDSA з кривою curve25519. Цей вибір зумовлений тим, що ці алгоритми є основою багатьох сучасних схем цифрового підпису, що використовуються в криптовалютах. Вони відповідає основним потребам криптовалют. Реалізації ци алгоритмів існують в багатьох бібліотеках з відкритим кодом.

Використовуючи зазначені алгоритми, ми реалізували програму командного рядка, яка використовуючи алгоритми цифрового підпису, може генерувати пару публічного та приватного ключів, читати ключі з файлу, писати в ключі файл, підписувати та верифікувати повідомелння з підписом.

Для реалізації цього алгоритму була вибрана мова програмування Java. Для розробки програми було обрано інтегроване середовище програмування Intellij IDEA, яке надає повний набір необхідних інструментів для розробки та тестування програм, написаних на Java. Intellij IDEA підтримує підсвітку коду, інтеграцію з системою контролю версій та різні сценарії запуску програм. Це середовище розробки доступне безкоштовно для студентів вищих навчальних закладів для некомерційної розробки. Також доступна безкоштовна версія середовища.

3.2 Створення проекту

Для створення нового проекту необхідно вибрати меню “File”, далі обираємо меню “New” та “Project”. У вікні створення нового проекту треба

вибрати мову програмування Java, та систему Gradle для збірки проекту.

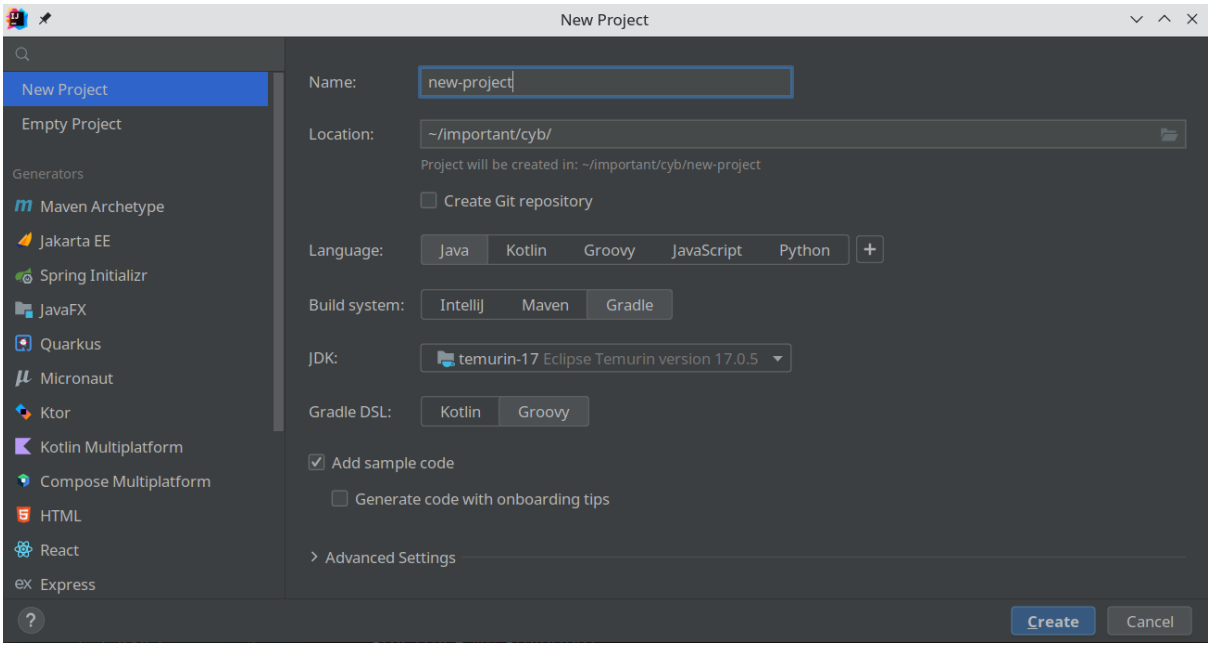

Рис.3 Вікно створення нового проекту

3.2.1 Вибір та підключення бібліотек

Bouncy Castle є відкритою криптографічною бібліотекою для мови програмування Java, що надає реалізацію різноманітних криптографічних алгоритмів і протоколів безпеки. Ця бібліотека надає розширений функціонал, включаючи генерацію ключів, шифрування, розшифрування, хешування, підписання та перевірку підпису даних. Bouncy Castle підтримує багато різних алгоритмів шифрування та хешування, включаючи стандартизовані алгоритми і експериментальні прототипи. Ця бібліотека широко використовується в різних областях, де потрібна криптографічна функціональність, таких як захист даних, електронний підпис, SSL/TLS, цифрові валюти та багато іншого.

Для додавання Bouncy Castle до Java проекту з використанням системи збірки Gradle, потрібно додати залежність до файлу build.gradle. Після додавання залежності Bouncy Castle можна імпортувати класи з бібліотеки та використовувати їх у своєму Java коді.

```
dependencies {
    implementation("org.bouncycastle:bcprov-jdk15on:1.70")
}
```

Рис.4 Підключення криптографічної бібліотеки

3.3 Реалізація програми

Для програми, яка використовує декілька різних алгоритмів цифрового підпису з однаковим контрактом, доцільно мати абстракцію для роботи з цифровими підписами. Інтерфейс для цифрового підпису дозволяє забезпечити загальний контракт, що спрощує структуру програми та забезпечує гнучкість при додаванні нових алгоритмів підпису.

3.3.1 Інтерфейс для роботи з цифровим підписом

Створимо інтерфейс для роботи з цифровими підписами.

```
public interface DigitalSignatureInterface {
    void generateKeyPair() throws KeyAlreadyExist;

    void setPrivateKey(byte[] privateKey) throws KeyAlreadyExist;

    void setPublicKey(byte[] publicKey) throws KeyAlreadyExist;

    byte[] getPrivateKey() throws KeyDoesNotExist;

    byte[] getPublicKey() throws KeyDoesNotExist;

    byte[] sign(byte[] msg) throws KeyDoesNotExist;

    boolean verify(byte[] msg, byte[] signatureData) throws KeyDoesNotExist;

    String keysToString();
}
```

Рис.5 Інтерфейс для роботи з алгоритмом цифрового підпису

Даний інтерфейс DigitalSignatureInterface описує контракт для роботи з цифровим підписом. Він включає наступні методи:

а) generateKeyPair(): Генерує нову пару ключів (приватний та публічний). Виклик цього методу може викликати виняток KeyAlreadyExist, якщо ключі вже існують.

б) setPrivateKey(byte[] privateKey): Встановлює приватний ключ на основі заданого масиву байтів. Виклик цього методу може викликати виняток KeyAlreadyExist, якщо ключ вже встановлений.

в) setPublicKey(byte[] publicKey): Встановлює публічний ключ на основі заданого масиву байтів. Виклик цього методу може викликати виняток KeyAlreadyExist, якщо ключ вже встановлений.

г) getPrivateKey(): Повертає приватний ключ у вигляді масиву байтів. Якщо ключ не існує, метод може викликати виняток KeyDoesNotExist.

ґ) getPublicKey(): Повертає публічний ключ у вигляді масиву байтів. Якщо ключ не існує, метод може викликати виняток KeyDoesNotExist.

д) sign(byte[] msg): Підписує задане повідомлення, передане як масив байтів, за допомогою приватного ключа і повертає підпис у вигляді масиву байтів. Якщо ключ не існує, метод може викликати виняток KeyDoesNotExist.

е) verify(byte[] msg, byte[] signatureData): Перевіряє підписане повідомлення, передане як масив байтів, за допомогою публічного ключа і підпису, переданого як масив байтів. Повертає true, якщо перевірка успішна, і false в іншому випадку. Якщо ключ не існує, метод може викликати виняток KeyDoesNotExist.

є) keysToString(): Повертає рядок, який представляє приватний та публічний ключі у зручному для читання форматі.

Цей інтерфейс надає базовий контракт для роботи з цифровим підписом, але не визначає конкретну реалізацію алгоритмів підпису. Конкретна реалізація може бути виконана з використанням різних алгоритмів, таких як RSA, ECDSA, EdDSA тощо.

3.3.2 Обробка винятків та неправильної поведінки

Клас KeyAlreadyExist і KeyDoesNotExist є винятками , які успадковані від класу Exception в Java. Вони використовуються для вираження спеціальних ситуацій або помилок, пов'язаних з публічним та приватним ключами.

Ці винятки використовуються для сигналізування про стан ключів в програмі. Клас KeyAlreadyExist використовується для інформування про те, що ключ вже існує, тобто виникла спроба створити новий ключ, коли він вже присутній. Клас KeyDoesNotExist використовується для показу того, що ключ не існує, коли він очікується або потрібний для виконання певних операцій. Використання цих винятків забезпечує більш чіткий та контрольований потік виконання програми. Наприклад, для об’єкту класу, що успадкований від DigitalSignatureInterface не можна два рази згенерувати ключі.

```
3   public class KeyAlreadyExist extends Exception {
4   }
5
```

Рис.6 Виняток KeyAlreadyExist

3.3.3 Реалізація інтерфейсу для одного з алгоритмів

Створимо клас Curve25519DigitalSignature, що імплементує інтерфейс DigitalSignatureInterface. Він матиме приватні поля для публічного та приватного ключів.

```
12  public class Curve25519DigitalSignature implements DigitalSignatureInterface {
13      Ed25519PrivateKeyParameters privateKey;
14      Ed25519PublicKeyParameters publicKey;
```

Рис.7 Клас, що імплементує DigitalSignatureInterface

Далі опишемо імплементацію основних методів. Метод generateKeyPair() реалізує генерацію пари ключів (приватний і публічний) для алгоритму Ed25519. Основні кроки імплементації методу наступні:

а) Перевірка, чи не існує вже приватного або публічного ключа. Якщо ключі вже існують, повертаємо виняток KeyAlreadyExist.

б) Створення генератора пари ключів Ed25519KeyPairGenerator.

в) Ініціалізація генератора для генерації ключів Ed25519.

г) Виклик методу generateKeyPair() генератора, який повертає пару ключів (AsymmetricCipherKeyPair).

г) Збереження згенерованих ключів відповідними приватним (privateKey) та публічним (publicKey) полями об'єкту.

```
@Override
public void generateKeyPair() throws KeyAlreadyExist {
    if (privateKey != null || publicKey != null) {
        throw new KeyAlreadyExist();
    }

    SecureRandom secureRandom = new SecureRandom();
    Ed25519KeyPairGenerator keyPairGenerator = new Ed25519KeyPairGenerator();
    keyPairGenerator.init(new Ed25519KeyGenerationParameters(secureRandom));
    AsymmetricCipherKeyPair keyPair = keyPairGenerator.generateKeyPair();
    privateKey = (Ed25519PrivateKeyParameters) keyPair.getPrivate();
    publicKey = (Ed25519PublicKeyParameters) keyPair.getPublic();
}
```

Рис.8 Імплементація методу generateKeyPair()

Методи getPrivateKey() та getPublicKey() повертають приватний та публічний ключі у вигляді масиву байтів. Кожен із методів перевіряє чи ключ був ініціалізований. Ці два методи необхідні для того, щоб зберегти ключ в файл та використовувати його під час наступного виконання програми.

```
@Override
public byte[] getPrivateKey() throws KeyDoesNotExist {
    if (privateKey == null) {
        throw new KeyDoesNotExist();
    }
    return this.privateKey.getEncoded();
}
@Override
public byte[] getPublicKey() throws KeyDoesNotExist {
    if (publicKey == null) {
        throw new KeyDoesNotExist();
    }
    return this.publicKey.getEncoded();
}
```

Рис.9 Імплементація методів getPrivateKey() та getPublicKey()

Метод sign підписує повідомлення msg за допомогою алгоритму Ed25519. Він перевіряє наявність приватного ключа, і якщо ключ існує, ініціалізує підписувач, оновлює його змістом повідомлення та генерує підпис. Підпис повертається у вигляді масиву байтів. У випадку відсутності приватного ключа генерується виняток.

```java
@Override
public byte[] sign(byte[] msg) throws KeyDoesNotExist {
    if (privateKey == null) {
        throw new KeyDoesNotExist();
    }

    Ed25519Signer signer = new Ed25519Signer();
    signer.init(true, privateKey);
    signer.update(msg, 0, msg.length);
    return signer.generateSignature();
}
```

Рис.10 Імплементація методу sign

Метод verify перевіряє підпис повідомлення msg за допомогою алгоритму Ed25519. Він перевіряє наявність публічного ключа, і якщо ключ існує, ініціалізує перевіряючий об'єкт, оновлює його змістом повідомлення та перевіряє підпис. Результат перевірки (чи вірний підпис чи ні) повертається в булевому значенні. У випадку відсутності публічного ключа генерується виняток.

```java
public boolean verify(byte[] msg, byte[] signatureData) throws KeyDoesNotExist {
    if (publicKey == null) {
        throw new KeyDoesNotExist();
    }

    Ed25519Signer verifier = new Ed25519Signer();
    verifier.init(false, publicKey);
    verifier.update(msg, 0, msg.length);
    return verifier.verifySignature(signatureData);
}
}
```

Рис.11 Імплементація методу sign

3.3.4 Реалізація класу для виконання в командному рядку

Статичний публічний метод main є точкою входу для виконання програми в мові Java. У класі на рис. 11 визначено метод main, який є статичним і головним методом програми. Виконання програми розпочинається з цього методу.

```java
public class Main {
    public static void main(String[] args) {
        new Runner().run();
    }
}
```

Рис.12 Метод main

У методі main створюється об'єкт класу Runner. Runner є іншим класом в програмі, який містить метод run(). Основна задача цього класу –

це зчитування вводу користувача та використання інтерфейсу DigitalSignatureInterface. Реалізацію класу Runner наведено в додатку А.

3.4 Інтерфейс програми

Програма має 10 команд: 1 команда показує повідомлення зі списком команд, 1 команда змінює алгоритм цифрового підпису, 8 команд відповідають методам DigitalSignatureInterface.

Детальний опис кожної з команд

1) help – показати це повідомлення з усіма командами
2) algo – вибрати алгоритм для цифрового підпису.
3) generatepair – згенерувати та вивести на екран публічний та приватний ключ для обраного алгоритму.
4) writepubkey – зберегти публічний ключ у файл.
5) writeprivkey – зберегти приватний ключ у файл.
6) readpubkey – прочитати публічний ключ з файлу.
7) readprivkey – прочитати приватний ключ з файлу.
8) sign – підписати повідомлення та зберегти підпис у файл.
9) verify – перевірити підпис, який збережений у файлі.
10) info – вивести інформацію про завантажені(або згенеровані) публічний та приватний ключі.

Розглянемо на прикладі методу verify його реалізацію. Реалізація інших методів є аналогічною. Коли користувач вводить назву команди “verify” в командному рядку, із методу run буде викликано метод doVerify, який друкує підказку користувачу. Метод очікує поки користувач введе назву файлу з підписом. Метод викикає наступний метод verify, який прочитає підпис та повідомлення з файлу та викличе метод verify в об’єкту, клас якого імплементує DigitalSignatureInterface.

```
98  @       private void doVerify(Scanner in) {
99              print("Input path with saved signature:");
100             String path = in.nextLine();
101             verify(path);
102     }
```

Рис.13 Допоміжний метод doVerify

3.5 Приклад роботи програми

Програму можна запустити декількома способами. Наприклад, за допомогою системи gradle, можна створити jar файл, який можна далі

запустити з командного рядка. Інший простий спосіб запустити програму – використовуючи інтерфейс Intellij IDEA.

Введемо команду help, щоб показати список всіх команд:

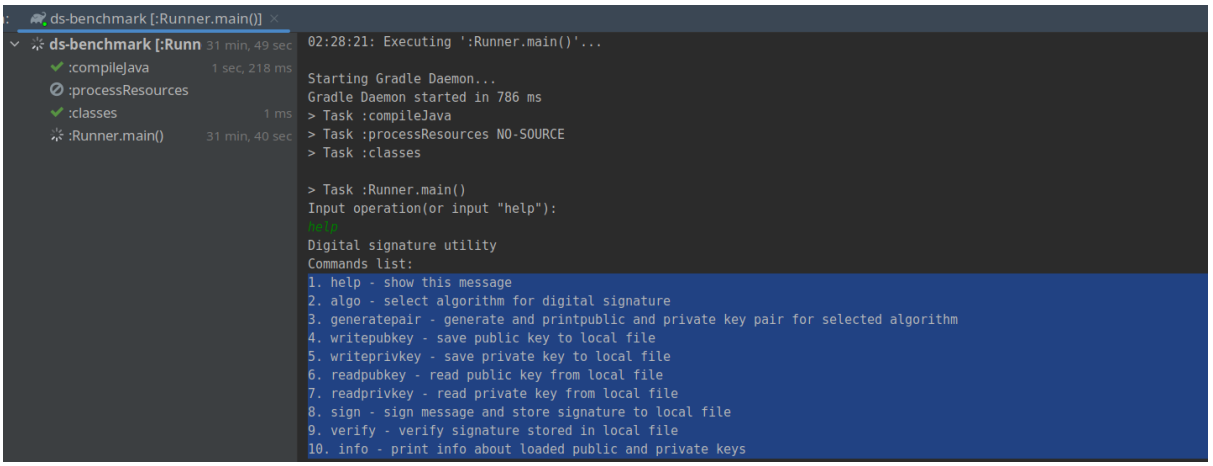

Рис.14 Результат команди help

Далі оберемо алгоритм та згенеруємо пару ключів командами “algo” та “generatepair” відповідно. Програма надрукувала публічний та приватний ключі в шістнадцятковій системі числення.

Рис.15 Вибір алгоритму та генерація ключів

За допомогою команд “writepubkey” та “writeprivkey” збережемо ключі в файл, щоб завантажити їх при наступному запуску програми та верифікувати підпис.

Підпишемо повідомлення командою sign. Після підпису, програма зберігає підпис та повідомлення в файл. Цей файл треба використовувати для верифікації підпису повідомлення.

Тепер запустимо програму ще раз, завантажимо тільки публічний ключ командною readpubkey та верифікуємо повідомлення командою verify. Програма повернула повідомлення “Signature is valid!”.

Рис.16 Завантаження ключа та верифікація повідомлення

3.6 Підсумки реалізації

Під час написання кваліфікаційної роботи бакалавра було імплементовано додаток командного рядка для використання цифрових підписів. Вона може бути розширена додаванням інших алгоритмів цифрового підпису, використовуватися як модуль для в інших додатках або для подальших досліджень.

ВИСНОВКИ

У даній кваліфікаційній роботі бакалавра було проведено аналіз архітектури криптовалют та особливостей використання цифрового підпису в криптовалютах. Було розглянуто основні принципи криптовалют, зокрема децентралізацію, роль журналу транзакцій та принципи консенсусу, такі як proof of work та proof of stake. Також було досліджено застосування криптографії в криптовалютах, зокрема генерацію приватних та публічних ключів, еліптичну криптографію та хеш-функції.

Для досягнення мети було проведено аналіз основних алгоритмів цифрового підпису. Були розглянуті алгоритми RSA, DSA, ECDSA та EdDSA. Досліджено структуру та безпеку цих алгоритмів, а також механізми, на яких вони ґрунтуються. Виявлено, що EdDSA є простішим у реалізації та розумінні порівняно з ECDSA, при цьому забезпечуючи подібний рівень безпеки.

У третьому розділі було реалізовано програму для використання цифрових підписів з використанням обговорених алгоритмів. Було описано процес реалізації програми, вибір та підключення необхідних бібліотек, а також розроблено інтерфейси та функціонал для роботи з цифровим підписом.

У підсумку можна зазначити, що аналіз архітектури криптовалют та особливостей використання цифрового підпису в криптовалютах дозволив отримати глибше розуміння принципів функціонування цих систем. Реалізація програми з використанням цифрових підписів дозволила перевірити практичну застосовність аналізованих алгоритмів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Кравченко П. БЛОКЧЕЙН І ДЕЦЕНТРАЛІЗОВАНІ СИСТЕМИ / П. Кравченко, Б. Скрябін, О. Дубініна. – Харків, 2019. – 474 с.
2. Bitcoin and cryptocurrency technologies: a comprehensive introduction. / [A. Narayanan, J. Bonneau, E. Felten та ін.]. – Princeton, New Jersey: Princeton University Press., 2016. – 336 с. – (Princeton University Press).
3. SETH S. What Is a Cryptocurrency Public Ledger, How It Works, Risks [Електронний ресурс] / SHOBHIT SETH. – 2021. – Режим доступу до ресурсу: https://www.investopedia.com/tech/what-cryptocurrency-public-ledger/.
4. PROOF-OF-WORK (POW) [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/.
5. PROOF-OF-STAKE (POS) [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.
6. Private key [Електронний ресурс] – Режим доступу до ресурсу: https://en.bitcoin.it/wiki/Private_key.
7. Menezes A. Handbook of Applied Cryptography / A. Menezes, P. van Oorschot, S. Vanstone., 1996.
8. Introduction to Ethereum’s Keccak-256 Algorithm [Електронний ресурс] – Режим доступу до ресурсу: https://wiki.rugdoc.io/docs/introduction-to-ethereums-keccak-256-algorithm/.
9. Katz J. Digital Signatures / Katz., 2010.
10. Goldreich O. The Foundations of Cryptography - Volume 2 / Oded Goldreich. – New York: Cambridge University Press. – 452 с.
11. R. L. Rivest, A. Shamir, L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, 1977.

12. Digital Signature Standard (DSS). Federal Information Processing Standards publication #186-2. U. S. Department of Commerce, National Institute of Standards and Technology, 2000.
13. Schneier B. Applied Cryptography / Bruce Schneier., 2015. – 784 с.
14. S. Vanstone, “Responses to NIST’s Proposal”, Communications of the ACM, 35, July 1992, с 50-52.
15. Johnson D. The Elliptic Curve Digital Signature Algorithm (ECDSA) [Електронний ресурс] / D. Johnson, A. Meneze, S. Vanston – Режим доступу до ресурсу: https://www.cs.miami.edu/home/burt/learning/Csc609.142/ecdsa-cert.pdf.
16. Secp256k1 [Електронний ресурс] – Режим доступу до ресурсу: https://en.bitcoin.it/wiki/Secp256k1.
17. Josefsson. Edwards-Curve Digital Signature Algorithm (EdDSA) [Електронний ресурс] / Josefsson, Liusvaara – Режим доступу до ресурсу: https://www.rfc-editor.org/rfc/rfc8032.
18. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang. High-speed high-security signatures. Journal of Cryptographic Engineering 2 (2012), с 77–89.
19. What’s The Difference Between ECDSA and EdDSA? [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://medium.com/asecuritysite-when-bob-met-alice/whats-the-difference-between-ecdsa-and-eddsa-e3a16ee0c966.
20. A Bluffers Guide To ECDSA and EdDSA [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://medium.com/asecuritysite-when-bob-met-alice/a-bluffers-guide-to-ecdsa-and-eddsa-ab65a82a2180.

ДОДАТОК А

Реалізація класу Runner:

```
package com.marchuk0;

import org.bouncycastle.crypto.digests.SHA3Digest;
import org.bouncycastle.util.encoders.Hex;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.Scanner;

public class Runner {
   // Ed25519 is default algorithm
   DigitalSignatureInterface digitalSignatureInterface = new
Curve25519DigitalSignature();

   public void run() {
       Scanner in = new Scanner(System.in);

       while (true) {
           print("Input operation(or input \"help\"):");
           String op = in.nextLine();
           if ("help".equals(op)) {
               printHelp();
           } else if ("algo".equals(op)) {
               digitalSignatureInterface = doAlgoSelect(in);
           } else if ("generatepair".equals(op)) {
               generateKeyValuePair();
           } else if ("writepubkey".equals(op)) {
               doWritePubKey(in);
           } else if ("writeprivkey".equals(op)) {
               doWritePrivKey(in);
           } else if ("readpubkey".equals(op)) {
               doReadPubKey(in);
           } else if ("readprivkey".equals(op)) {
               doReadPrivKey(in);
           } else if ("sign".equals(op)) {
               doSign(in);
           } else if ("verify".equals(op)) {
               doVerify(in);
           } else if ("info".equals(op)) {
               printKeys();
           }
       }
   }
```

```
    private void printHelp() {
        StringBuilder sb = new StringBuilder();
        sb.append("Digital signature utility\n");
        sb.append("Commands list:\n");
        sb.append("1. help - show this message\n");
        sb.append("2. algo - select algorithm for digital signature\n");
        sb.append("3. generatepair - generate and printpublic and private key pair
for selected algorithm\n");
        sb.append("4. writepubkey - save public key to local file\n");
        sb.append("5. writeprivkey - save private key to local file\n");
        sb.append("6. readpubkey - read public key from local file\n");
        sb.append("7. readprivkey - read private key from local file\n");
        sb.append("8. sign - sign message and store signature to local file\n");
        sb.append("9. verify - verify signature stored in local file\n");
        sb.append("10. info - print info about loaded public and private keys\n");
        print(sb.toString());
    }

    private void doWritePubKey(Scanner in) {
        print("Input path for public key:");
        String path = in.nextLine();
        writePublicKeyToFile(path);
    }

    private void doWritePrivKey(Scanner in) {
        print("Input path for private key:");
        String path = in.nextLine();
        writePrivateKeyToFile(path);
    }

    private void doReadPubKey(Scanner in) {
        print("Input path for public key:");
        String path = in.nextLine();
        readPublicKeyFromFile(path);
    }

    private void doReadPrivKey(Scanner in) {
        print("Input path for private key:");
        String path = in.nextLine();
        readPrivateKeyFromfile(path);
    }

    private void doSign(Scanner in) {
        print("Input message:");
        String message = in.nextLine();
        print("Input path to write signature:");
```

```
        String path = in.nextLine();
        sign(message, path);
    }

    private void doVerify(Scanner in) {
        print("Input path with saved signature:");
        String path = in.nextLine();
        verify(path);
    }

    private DigitalSignatureInterface doAlgoSelect(Scanner in) {
        print("Select algorithm to test(1 or 2): \n 1) ECDSA(secp256k1) \n 2)
EdDSA(curve25519)");
        String algo = in.nextLine();
        if ("1".equals(algo)) {
            return new Secp256k1DigitalSignature();
        } else if ("2".equals(algo)){
            return new Curve25519DigitalSignature();
        } else {
            print("Unknown algo");
            return digitalSignatureInterface;
        }
    }

    private void generateKeyValuePair() {
        try {
            digitalSignatureInterface.generateKeyPair();
            print(digitalSignatureInterface.keysToString());
        } catch (KeyAlreadyExist e) {
            print("Public or private Key is already present. Reselect algorithm to
generate new pair");
        }
    }

    private void writePublicKeyToFile(String path) {
        File file = new File(path);
        try {
            file.createNewFile();
        } catch (IOException e) {
            print("Cannot create file\n" + e.toString());
        }

        try {
            Files.write(file.toPath(), digitalSignatureInterface.getPublicKey());
        } catch (IOException e) {
            print("Cannot write file\n" + e.toString());
        } catch (KeyDoesNotExist e) {
```

```
            print("Pub key does not exist");
        }
    }

    private void writePrivateKeyToFile(String path) {
        File file = new File(path);
        try {
            file.createNewFile();
        } catch (IOException e) {
            print("Cannot create file\n" + e.toString());
        }

        try {
            Files.write(file.toPath(), digitalSignatureInterface.getPrivateKey());
        } catch (IOException e) {
            print("Cannot write file\n" + e.toString());
        } catch (KeyDoesNotExist e) {
            print("Priv key does not exist");
        }
    }

    private void readPublicKeyFromFile(String path) {
        File file = new File(path);

        try {
            byte[] b = Files.readAllBytes(file.toPath());
            digitalSignatureInterface.setPublicKey(b);
        } catch (IOException e) {
            print("Cannot write file\n" + e.toString());
        } catch (KeyAlreadyExist e) {
            print("Pub key already exist");
        }
    }

    private void readPrivateKeyFromfile(String path) {
        File file = new File(path);

        try {
            byte[] b = Files.readAllBytes(file.toPath());
            digitalSignatureInterface.setPrivateKey(b);
        } catch (IOException e) {
            print("Cannot write file\n" + e.toString());
        } catch (KeyAlreadyExist e) {
            print("Pub key already exist");
        }
    }
```

```
private void sign(String message, String writePath) {
    File sigFile = new File(writePath);
    File msgFile = new File(writePath + ".message");

    byte[] msgHash = sha3(message);
    try {
        byte[] signature = digitalSignatureInterface.sign(msgHash);
        Files.write(sigFile.toPath(), signature);
        Files.write(msgFile.toPath(), message.getBytes());

        print("Signature: ");
        print(Hex.toHexString(signature) + " " + "saved to file " + sigFile);
    } catch (KeyDoesNotExist e) {
        print("Private key already exist");
    } catch (IOException e) {
        print("Cannot write to file");
    }

}

private void verify(String writePath) {
    File sigFile = new File(writePath);
    File msgFile = new File(writePath + ".message");
    try {
        byte[] sign = Files.readAllBytes(sigFile.toPath());
        String message = new String(Files.readAllBytes(msgFile.toPath()));
        byte[] msgHash = sha3(message);
        if (digitalSignatureInterface.verify(msgHash, sign)) {
            print("Message: " + message +"\nSignature is valid!");
        } else {
            print("Message: " + message +"\nInvalid signature!");
        }
    } catch (KeyDoesNotExist e) {
        print("Private key already exist");
    } catch (IOException e) {
        print("Cannot read message or signature from file");
    }
}

private void printKeys() {
    print(digitalSignatureInterface.keysToString());
}

private byte[] sha3(byte[] input) {
    SHA3Digest digest = new SHA3Digest(256);

    byte[] output = new byte[digest.getDigestSize()];
```

```
        digest.update(input, 0, input.length);
        digest.doFinal(output, 0);

        return output;
    }

    private byte[] sha3(String message) {
        return sha3(message.getBytes());
    }

    private void print(String s) {
        System.out.println(s);
    }
}
```