

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
(повне найменування факультету)

Кафедра «Інформаційна безпека та наноелектроніка»
(повне найменування кафедри)

Пояснювальна записка

до дипломної роботи

бакалавр

(ступінь вищої освіти)

на тему Реалізація неінтерактивного протоколу доказу з нульовим
(назва теми)
розголошенням для використання у схемах автентифікації

Виконав: студент 4 курсу, групи БК-710

Спеціальності 125 Кібербезпека

(код і найменування спеціальності)

Освітня програма (спеціалізація)
системи технічного захисту інформації,
автоматизація її обробки

КАЦЮБА М.В.

(ПРИЗВИЩЕ та ініціали)

Керівник НЕЛАСА Г.В.

(ПРИЗВИЩЕ та ініціали)

Рецензент МАЛИЙ О.Ю.

(ПРИЗВИЩЕ та ініціали)

202 4

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Інформаційної безпеки та електронних комунікацій

Кафедра Інформаційна безпека та наноелектроніка

Ступінь вищої освіти бакалавр

Спеціальність 125 Кібербезпека

(код і найменування)

Освітня програма (спеціалізація) системи технічного захисту інформації,

(назва освітньої програми (спеціалізації))

автоматизація її обробки

ЗАТВЕРДЖУЮ

Завідувач кафедри ІБтаН

Андрій КОРОТУН

« » 2024 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТА

КАЦЮБИ Миколи Вікторовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема роботи Реалізація неінтерактивного протоколу доказу з нульовим розголошенням для використання в схемах автентифікації

керівник роботи канд. техн. наук, доцент, НЕЛАСА Ганна Вікторівна,

(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від «03» травня 2024 року №190

2. Строк подання студентом роботи 03.06.2024р.

3. Вихідні дані до роботи неінтерактивний протокол Plonky2, бібліотека plonky2, схема ідентифікації iden3

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз сучасних протоколів автентифікації з використанням доведень з нульовим розголошенням. 2. Розробка проєкту системи автентифікації. 3. Програмна реалізація.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Презентація PowerPoint (19 слайдів)

Ганна НЕЛАСА

АНОТАЦІЯ

Пояснювальна записка до дипломного проєкту: 63 с., 1 табл., 13 рис., 1 дод., 42 джерела.

АВТЕНТИФІКАЦІЯ, НЕІНТЕРАКТИВНІ СИСТЕМИ ДОВЕДЕННЯ З НУЛЬОВИМ РОЗГОЛОШЕННЯМ, НУЛЬОВЕ РОЗГОЛОШЕННЯ, PLONKY2, ПРОГРАМНА РЕАЛІЗАЦІЯ, RUST, ZERO KNOWLEDGE PROOF.

Об'єктом дослідження є процес автентифікації користувача в мережі з недовірою сторін.

Предметом дослідження є неінтерактивний протокол доведення з нульовим розголошенням Plonky2.

Метою роботи є реалізація схеми автентифікації, яка використовує сучасний неінтерактивний протокол доведення з нульовим розголошенням.

Практична цінність результатів проєкту полягає в тому, що неінтерактивні системи доведень з нульовим розголошенням можна використовувати у схемах автентифікації.

У даній дипломній роботі було досліджено можливість використання систем доведень з нульовим розголошенням для побудови систем автентифікації користувачів. Розглянуто готові рішення автентифікації, що використовують системи доведень з нульовим розголошенням. Розглянуто неінтерактивний протокол доведення з нульовим розголошенням Plonky2 і його можливість використання у схемах автентифікації. Створена програмна реалізація схеми автентифікації, яка використовує Plonky2, на мові програмування Rust.

ABSTRACT

Explanatory note to the diploma project: 63 p., 1 table., 13 fig., 1 app., 42 references.

AUTHENTICATION, NON-INTERACTIVE ZERO-KNOWLEDGE PROOF SYSTEMS, PLONKY2, RUST, SOFTWARE IMPLEMENTATION, ZERO KNOWLEDGE, ZERO-KNOWLEDGE PROOF.

The object of the work is the process of user authentication in a network with distrust of parties.

Subject of the work is Plonky2 non-interactive zero-knowledge proof protocol.

The goal of the work is implementation of an authentication scheme that uses a modern non-interactive proof protocol with zero knowledge.

The practical value of the project's results is that non-interactive zero-knowledge proof systems can be used in authentication schemes.

In this graduate work, the possibility of using proof systems with zero knowledge to build user authentication systems was investigated. Ready-made authentication solutions using zero-knowledge proof systems are considered. The non-interactive proof protocol with zero-knowledge Plonky2 and its possibility of use in authentication schemes are considered. Created a software implementation of the authentication scheme that uses Plonky2, in the Rust programming language.

ЗМІСТ

Перелік скорочень	8
Вступ.....	9
1 Аналіз сучасних протоколів автентифікації з використанням доведень з нульовим розголошенням.....	11
1.1 Визначення систем доведення з нульовим розголошенням.....	11
1.2 Класифікація систем доведення з нульовим розголошенням	13
1.3 Протокол Plonky2.....	14
1.3.1 Загальне про протокол	14
1.3.2 Вибір поля	15
1.3.3 Побудова арифметичної схеми	16
1.3.4 Нульове знання	17
1.3.5 Попередня обробка.....	17
1.3.6 Основний протокол	18
1.3.7 Протокол FRI	19
1.4 Приклади схем автентифікації, які використовують системи доведення з нульовим розголошенням	21
1.4.1 Протокол безпечного віддаленого пароля.....	21
1.4.2 Протокол автентифікації Sediciі	22
1.4.3 NuCypher	23
1.4.4 Aleo	23
1.4.5 Автентифікація в Iden3	24
1.5 Постановка задачі	26
2 Розробка проекту системи автентифікації.....	27
2.1 Проектування системи автентифікації за допомогою UML.....	27
2.1.1 Діаграма варіантів використання.....	27
2.1.2 Діаграма кооперації.....	28
2.1.3 Діаграма послідовності.....	30

2.1.4 Діаграма класів	31
2.1.5 Діаграма розгортання.....	32
2.2 Вибір мови програмування.....	32
3 Програмна реалізація	37
3.1 Огляд бібліотеки plonky2	37
3.2 Розробка програмного коду модулів системи автентифікації	38
3.3 Опис роботи програмної реалізації.....	40
3.4 Результати та оцінка виконання програми.....	43
Висновки	45
Перелік джерел посилання	46
Додаток А. Код програмної реалізації на Rust.....	50
А.1 Код бібліотеки auth_lib.....	50
А.2 Код тестової програми AuthPlonky2	58
А.3 Код програми, яка робила вимірювання швидкості генерації і перевірки доведення	60

ПЕРЕЛІК СКОРОЧЕНЬ

FRI (Fast RS IOPP, або Fast Reed—Solomon Interactive Oracle Proofs of Proximity) – швидкі інтерактивні оракульні доведення близькості на основі коду Ріда-Соломона

iZKP (interactive Zero-Knowledge Proof) – інтерактивні системи доведення з нульовим розголошенням

MITM (Man-in-the-middle) – атака «людина посередині»

NIZK (Non-interactive Zero Knowledge) – неінтерактивні системи доведення з нульовим розголошенням

PAKE (Password Authenticated Key Exchange) - обмін ключами із автентифікацією пароля

Prover (P) – сторона, що доводить деяке твердження

SRP (Secure Remote Password) – протокол безпечного віддаленого паролю

UML (Unified Modeling Language) – уніфікована мова моделювання

Verifier (V) – сторона, що перевіряє доведення доказувача

ZEXE (Zero-Knowledge EXEcution) – виконання нульового розголошення

zkSNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) - стислий неінтерактивний аргумент знань із нульовим знанням

zkSTARK (Zero-Knowledge Scalable Transparent Argument of Knowledge) - масштабований прозорий аргумент знання з нульовим розголошенням

ZKP (Zero-Knowledge Proof) – доведення з нульовим розголошенням

ВСТУП

На сьогоднішній день існує багато способів автентифікації користувачів [1]. Традиційним вважається парольний спосіб автентифікації, тобто вхід у систему здійснюється за допомогою паролю. Системи, що використовують даний спосіб автентифікації найпростіше всього реалізувати, але при цьому безпека даних у більшій мірі залежить від самого користувача, що придумав пароль. Крім того, бази даних, що зберігають хешовані паролі, часто зламують, і, як наслідок, виникають витoki баз паролів. Також існують біометричні та апаратні методи автентифікації. Біометрична автентифікація здійснюється на основі фізіологічних та поведінкових характеристик людини (відбитки пальців, райдужна оболонка ока, геометрія руки й обличчя, рукописний почерк, клавіатурний почерк, розпізнавання голосу тощо). Апаратна автентифікація здійснюється за допомогою унікальних предметів (магнітні карти, смарт-карти, USB-токени тощо) [2]. Біометрична й апаратна автентифікація забезпечують більш надійний захист, ніж парольна автентифікація, проте вони вимагають додаткового апаратного й програмного забезпечення. Досить поширеним є багатофакторна автентифікація, яка здійснюється в результаті комбінації двох або більше різних однофакторних методів (наприклад, пароль і відбиток пальця, пароль і SMS з одноразовим кодом підтвердження тощо). Даний вид автентифікації забезпечує надійний захист, але його основним недоліком є складність використання непідготовленими користувачами.

Одним із сучасних методів автентифікації є автентифікація з використанням систем доведення з нульовим розголошенням. Завдяки такому методу автентифікації користувач замість паролю відправляє на перевірку згенероване доведення, яке у свою чергу не розкриває другій стороні додаткової або конфіденційної інформації про користувача. Цей метод є простим у використанні і забезпечує потрібний рівень надійності даних.

У даному дипломному проєкті було розглянуто готові рішення

автентифікації, що використовують системи доведень з нульовим розголошенням. Також розглянуто один із сучасних протоколів доведення з нульовим розголошенням Plonky2 і створено програмну реалізацію схеми автентифікації, яка використовує цей протокол, на мові програмування Rust.

Дипломний проєкт складається з трьох розділів. У першому розділі йдеться мова про визначення систем доведення з нульовим розголошенням та їхні типи, а також було приведено неінтерактивний протокол Plonky2. Розглянуто готові рішення автентифікації з використанням ZKP. Зроблено постановку задачі.

У другому розділі виконано проєктування системи автентифікації за допомогою діаграм UML та обрано мову програмування для виконання реалізації.

У третьому розділі приведено програмну реалізацію системи автентифікації. Розглянуто використану бібліотеку, модулі, поля й методи реалізації. Розглянуто принцип роботи програмної реалізації та результати роботи програми.

1 АНАЛІЗ СУЧАСНИХ ПРОТОКОЛІВ АВТЕНТИФІКАЦІЇ З ВИКОРИСТАННЯМ ДОВЕДЕНЬ З НУЛЬОВИМ РОЗГолоШЕННЯМ

1.1 Визначення систем доведення з нульовим розголошенням

Доведення із нульовим розголошенням — це криптографічний протокол, який дозволяє одній стороні (тому, хто доводить) переконати іншу сторону (верифікатора) у правдивості твердження, не розкриваючи жодної додаткової інформації про себе (наприклад, не надавати жодних подробиць про те, що саме було доведено чи звідки взята ця інформація).

Доведення з нульовим розголошенням були вперше запропоновані Шафі Голдвассером (Shafi Goldwasser), Сільвіо Мікалі (Silvio Micali) та Чарльзом Ракоффом (Charles Rackoff) у 1985 роках у статті «Складність знань інтерактивних систем доведення» (англ. «The Knowledge Complexity of Interactive Proof-Systems») [3]. Протягом останніх десятиліть їхні конструкції стали дуже практичними та зробили величезний вплив у різних практичних застосуваннях [4].

Пояснити нульове розголошення можна на прикладі казки про печеру (рис. 1.1) [5]. У печері, що має форму кільця, є потайні двері, яку можна відкрити тільки той, хто знає секретні слова. Тут дійовими особами є Пеггі та Віктор. Пеггі стверджує, що знає ці секретні слова, що дозволяє їй відкрити ці двері у печері. Вона хоче переконати Віктора, що це дійсно так, але при цьому не збирається йому повідомляти самі секретні слова.

Щоб довести це, Пеггі входить в печеру і йде до дверей, при цьому вона випадковим чином вибирає один з двох можливих шляхів (або до точки С, або до точки D), який саме - Віктор не знає, тому що він чекає в цей час біля входу (точка А). Як тільки Пеггі зникає з виду, Віктор заходить в печеру і на розгалуженні (точка В) кричить: «Пеггі, вийди з лівого коридору!» або ж «Пеггі, вийди з правого коридору!». Необхідність відкрити двері, щоб вийти в потрібному напрямку, покаже знання нею секретного ключа, в іншому випадку

Пеггі вийде з того боку, звідки прийшла. Так як Віктор випадково вибирає шлях, існує ймовірність, що Пеггі не знає секрету. Але якщо повторити цю процедуру багато разів, то її шанси приховати незнання секрету зменшуються практично до нуля. Отже, Пеггі зможе переконати Віктора в знанні пароля, не розкриваючи його.

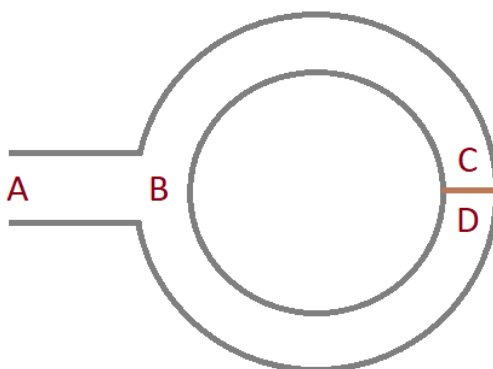


Рисунок 1.1 – Печера нульового розголошення

Доведення з нульовим розголошенням повинні відповідати таким вимогам:

- повнота: доказуючий завжди може переконати перевіряючого, якщо приводить істинні твердження;
- коректність: доказуючий не може переконати перевіряючого, якщо приводить неправдиві твердження;
- нульове знання: перевіряючий не дізнається від доведення нічого, крім того факту, що твердження істинне.

Якщо застосувати ці вимоги до вище вказаного прикладу, то ми побачимо, що цей приклад цілком їм відповідає. Повнота забезпечується тим, що Пеггі завжди може переконати Віктора, що знає секрет, якщо дійсно буде виходити з усіх напрямків, обраних Віктором, не зважаючи на її заздалегідь вибраний шлях. Якщо ж Пеггі не знає секретних слів, вона практично ніколи не зможе переконати Віктора у протилежному, тобто, таким чином забезпечується обґрунтованість. І якщо все ж таки Пеггі вдається довести Віктору, що знає секрет, то той отримає тільки доведення, а сам секрет все одно знатиме тільки Пеггі, тобто,

забезпечується нульове знання.

1.2 Класифікація систем доведення з нульовим розголошенням

Системи доведення з нульовим розголошенням можна поділити на дві групи: інтерактивні (iZKP) та неінтерактивні (NIZK).

В інтерактивних системах доведення доказуючий і перевіряючий знаходяться у безпосередній взаємодії в раундах, при цьому інформація передається в обох напрямках.

Перевагою таких систем є висока надійність. Проте їхнім основним недоліком є необхідна одночасна доступність двох сторін, а також вони вимагають декілька раундів для взаємодії.

Нижче приведено приклади iZKP:

- протокол Шнорра [6];
- схема ідентифікації Фейге–Фіат–Шаміра [7];
- протокол Гіллу — Кіскатра [8].

Неінтерактивні системи доведення позбавлені необхідності одночасної доступності обох сторін. У цих системах доведення складається лише з одного повідомлення, якого достатньо для переконання перевіряючої сторони

Нижче приведено приклади NIZK:

- підпис Шнорра [6];
- евристика (перетворення) Фіата-Шаміра [9];
- протоколи zkSNARK та zkSTARK.

Серед неінтерактивних систем доведення стали поширеними протоколи zkSNARK та zkSTARK. zkSNARK буквально означає стислий неінтерактивний аргумент знань із нульовим знанням. Характеристика «стислий» означає, що доведення нульового знання в цьому протоколі можна перевірити протягом декількох мілісекунд, а довжина такого доведення - лише кілька сотень байт,

навіть для досить великого розміру вихідного твердження [10].

На момент написання роботи існує багато реалізацій zkSNARK, які використовують різноманітні підходи для побудови та верифікації доведень. Ось деякі з них:

- Pinocchio [11];
- Groth16 [12];
- Sonic [13];
- Halo [14];
- PLONK [15] тощо.

zkSTARK, на відміну від zkSNARK, є масштабованим (системи з використанням zkSTARK легко масштабуються в термінах швидкості обчислень та розміру пам'яті на завдання з великою кількістю обчислень) і прозорим (не вимагає попереднього налаштування). zkSTARK виграє перед більшістю протоколів zkSNARK завдяки прозорості і стійкістю до квантових атак. Проте «вузьким горлом» для використання zkSTARK у реальних проєктах стає розмір доведення, що може вимірюватись кілобайтами, що забагато для багатьох проєктів і нині існуючих блокчейнів [16].

Одним із сучасних реалізацій zkSNARK є Plonky2 [17]. Його особливість полягає у тому, що він використовує інструменти zkSTARK, і при цьому генерує досить маленькі доведення і зберігає достатню швидкість для генерації. У наступному підрозділі приведено опис цього протоколу.

1.3 Протокол Plonky2

1.3.1 Загальне про протокол

Plonky2 – сучасний неінтерактивний протокол доведення з нульовим розголошенням, оснований на PLONK [15]. Його сильною стороною є швидкодія, оскільки він:

- використовує FRI [18] замість схем на еліптичних кривих;
- є рекурсивним протоколом: розбиває доведення на окремі доведення, які обчислюються паралельно, і потім об'єднуються в один доведення;
- використовує основне просте поле Галуа за модулем $p = 2^{64} - 2^{32} + 1$, яке дозволяє оптимізувати обчислення на апаратному рівні (про це детальніше у наступному підрозділі) [17].

1.3.2 Вибір поля

Дані для свідка кодуються у простому полі F_p , де $p = 2^{64} - 2^{32} + 1$. Це поле вибрано з міркувань швидкості обчислень: його елементи вміщаються в 64-бітове слово, а структура p призводить до ефективного методу скорочення.

Щоб зрозуміти, чому скорочення за модулем p ефективно, необхідно зауважити, що

$$2^{64} \equiv 2^{32} - 1 \pmod{p}, \quad (1.1)$$

і, отже

$$\begin{aligned} 2^{96} &\equiv 2^{32}(2^{32} - 1) \pmod{p} \\ &\equiv 2^{64} - 2^{32} \pmod{p} \\ &\equiv -1 \pmod{p}. \end{aligned} \quad (1.2)$$

Щоб скоротити 128-бітове число n , необхідно n переписати як $n_0 + 2^{64}n_1 + 2^{96}n_2$, де n_0 - це 64 біти, а n_1 і n_2 - по 32 біти кожен. Потім

$$n \equiv n_0 + 2^{64}n_1 + 2^{96}n_2 \pmod{p}$$

$$\equiv n_0 + (2^{32} - 1)n_1 - n_2 \pmod{p}. \quad (1.3)$$

Вибір p також зменшує потребу у зберіганні констант у регістрах. Зокрема, при переповненому додаванні необхідно віднімати p як виправлення. Віднімання p еквівалентне додаванню $2^{64} - p = 2^{32} - 1$ в 64-бітовій арифметиці. Зручно, що у багатьох архітектурах є інструкція для встановлення регістра 0 чи $2^{32} - 1$ залежно від прапора переноса; наприклад, у x86 є SBB і ARM64 є CSETM. Таким чином, хоча p утворюється як константа в алгоритмі скорочення, її не потрібно зберігати у регістрі. Це знижує навантаження на регістри та накладні витрати на завантаження констант [17].

1.3.3 Побудова арифметичної схеми

Арифметична схема – арифметична схема, з якої генеруються доведення. Спосіб формування арифметичних схем полягає в тому, щоб виразити математичний вираз як послідовність операторів у вигляді присвоювання змінної та математичних двійкових операцій, таких як множення та додавання [18].

На рисунку зображено візуальний приклад арифметичної схеми виразу

$$x^3 + x + 5 = 35 \quad (1.4)$$

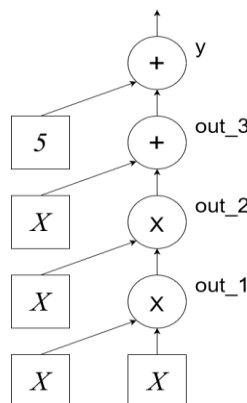


Рисунок 1.2 – Арифметична схема для прикладу

З цього випливає, що арифметичну схему можна зобразити послідовністю операторів, які, як зазначено вище, мають дві форми:

$x = y$ (де y може бути змінна або число);

$x = y \text{ op } z$ (де op може бути $+$, $-$, $\cdot$, $\div$, а y і z можуть бути змінними, числами або виразами).

У нашому випадку, даний вираз можна зобразити таким чином:

$$\begin{aligned} out_1 &= x \cdot x, \\ out_2 &= out_1 \cdot x, \\ out_3 &= out_2 + x, \\ y &= out_3 + 5. \end{aligned} \tag{1.5}$$

1.3.4 Нульове знання

Для досягнення нульового знання у Plonky2, як і у PLONK, поліноми доказувача необхідно приховати додаванням до траси рядків, які заповнені випадковими елементами.

Далі використовуються стандартні техніки для забезпечення нульового розголошення протоколу FRI.

1. Оскільки відправка оцінок поліномів доказуючого на H прямо розкриває інформацію про свідоктві, відправляються оцінки на суміжну підмножину H .

2. Дерева Меркла перетворюються на векторні зобов'язання приховування шляхом обгортання кожного листа зобов'язанням на основі хеша [17].

1.3.5 Попередня обробка

Нижче приведено інтерактивний варіант протоколу Plonky2.

Неінтерактивний варіант виходить застосуванням перетворення Фіат-Шаміра [9].

Нехай r буде параметром повторення. Це кількість повторів певних перевірок, особливо тих, які містять помилку достовірності $y/|F_p|$, де y - це деяка «маленька» величина, така як ступінь схеми або кількість обмежень.

Нехай $Com(\vec{p})$ буде позначенням прихильності до вектора поліномів. Саме він є «шапкою» дерева Меркла, де кожен лист відповідає одній точці $x \in H$, і містить оцінки $p_1(x), \dots, p_k(x)$.

1. P, V будують схему і обчислюють $\vec{C}$, набір поліномів, які кодують константи, налаштовані для кожного вентиля, а також $\vec{\sigma}$, який кодує «розширену» перестановку [15].

2. P, V будують дерево Меркла, що містить $\vec{C}, \vec{\sigma}$.

3. P зберігає це дерево Меркла як ключ доведення.

4. V зберігає $Com(\vec{C}, \vec{\sigma})$ як ключ перевірки.

1.3.6 Основний протокол

1. P генерує свідка $\vec{w}_i$ відправляє $Com(\vec{w})$, зобов'язання до кожного поліному дроту $w(x)$.

2. V вибирає $\beta_1, \dots, \beta_r, \gamma_1, \dots, \gamma_r \in F_p$ - випадкові значення, які використовуються в аргументі перестановки.

3. P відправляє $Com(\vec{Z}, \vec{\pi})$ - зобов'язання до кожного полінома перестановки $Z(x)$ і полінома часткового добутку $\pi(x)$.

4. V вибирає $\alpha_1, \dots, \alpha_r \in F_p$ - випадкові значення, що використовуються для комбінування обмежень.

5. P відправляє $Com(\vec{q})$, зобов'язання до кожного поліному-діляника $q(x)$.
Зокрема,

$$q_i(x) = C_i(x)/Z_H(x), \quad (1.6)$$

де $C_i(x)$ є комбінованим обмеженням, який був зменшений з використанням ступенів α_i , і багаточлен $Z_H(x) = x^n - 1$, який дорівнює нулю в H .

6. **V** вибирає $\zeta \in F_p(\varphi)$ - випадкова точка, в якій повинні перевірятися поліноміальні тотожності.

7. **P** відправляє $\vec{P}(\zeta)$ - передбачувана оцінка кожного полінома в ζ , де $\vec{P}$ являє собою конкатенацію всіх вищезгаданих поліномів: $\vec{P} = (\vec{C}, \vec{\sigma}, \vec{w}, \vec{Z}, \vec{\pi}, \vec{q})$

8. Щоб перевірити ці відкриття, **P**, **V** використовують пакетний протокол FRI за наступним списком передбачуваних кодових слів:

$$\vec{B} = \left(\frac{p_i(x) - p_i(\zeta)}{x - \zeta} \mid p_i \in \vec{P} \right). \quad (1.7)$$

9. **V** використовує $\vec{P}(\zeta)$, щоб вивести оцінку комбінованих поліномів обмежень $c_1(\zeta), \dots, c_r(\zeta)$, і стверджує, що кожен $c_i(\zeta) = q_i(x)Z_H(x)$.

1.3.7 Протокол FRI

Протокол FRI, який використовується у Plonky2, практично ідентичний стандартному протоколу [20], з невеликими відмінностями, такими як використання «шапки» Меркла (cap of the Merkle tree) замість коренів Меркла.

1. **V** вибирає $\alpha \in F_p(\varphi)$, випадкове число, що використовується зменшення пакета Bs в одне передбачуване кодове слово.

2. **P** відправляє $Com(h_0)$, де h_0 - поліном композиції DEEP,

$$h_0(x) = \sum_{i=0}^{|\vec{B}|-1} \alpha^i \vec{B}_i(x) \quad (1.8)$$

3. **P**, **V** виконують фазу фіксації FRI. Для кожної арності приведення l_i

(а) **V** вибирає $\alpha \in F_p(\varphi)$

(б) **P** переписує $h_i(x)$ як

$$h_i(x) = \sum_{j=0}^{l_i-1} x^j h_{i,j}(x^{l_i}) \quad (1.9)$$

де кожен $h_{i,j}$ містить коефіцієнти $h_i(x)$, показники ступеня якого конгруентні $j \bmod l_i$. **P** потім відправляє $Com(h_{i+1})$, зобов'язання до поліному

$$h_{i+1}(x) = \sum_{j=0}^{l_i-1} \beta^j h_{i,j}(x) \quad (1.10)$$

4. **V** вибирає $\tau \in F_p^4$.

5. **P** виконує «подрібнення» (grinding) [20] і надсилає своєму свідку доведення роботи μ .

6. **V** стверджує, що $H(\tau, \mu)$ містить як мінімум proof_of_work_bits провідних нулів у його двійковому поданні

7. **V** вибирає num_query_rounds випадкових індексів $q_1, \dots, q_k \in \{0, \dots, n-1\}$

8. Для кожного індексу запиту q_i

(а) **P** відправляє оцінки $\vec{P}(x)$, разом із кожним $h_i(x)$, де x – це i -я точка суміжного класу, у якому визначено кодові слова.

(б) **V** виконує серію перевірок узгодженості на x , спочатку між $\vec{P}(x)$ та $h_0(x)$, а потім між кожною парою $(h_i(x), h_{i+1}(x))$.

1.4 Приклади схем автентифікації, які використовують системи доведення з нульовим розголошенням

1.4.1 Протокол безпечного віддаленого пароля

Протокол безпечного віддаленого пароля (SRP) — це широко перевірений і широко розгорнутий протокол обміну ключами із автентифікацією пароля (PAKE), який взаємно автентифікує як користувача, так і сервер, не надсилаючи жодних еквівалентних паролю даних через мережу. Безпека SRP, як і Діффі-Хеллмана, базується на складності проблеми дискретного логарифмування [21].

SRP, як протокол з нульовим знанням на основі перевірки, стійкий до словникових атак, пропонує низку нових переваг для розробників системи паролів [22]:

- зловмисник, який не має ні пароля користувача, ні файлу паролів хоста, не може здійснити словникову атаку на пароль. У цьому сценарії досягається взаємна автентифікація;
- зловмисник, який захоплює файл паролів хоста, не може безпосередньо скомпрометувати автентифікацію між користувачем і отримати доступ до хосту без дорогого пошуку в словнику;
- зловмисник, який скомпрометував хост, не отримує пароль із законної спроби автентифікації;
- зловмисник, який захопив сеансовий ключ, не може ним скористатися щоб здійснити словникову атаку на пароль;
- зловмисник, який захоплює пароль користувача, не може використовувати його для зламу сеансових ключів минулих сеансів.

Цей протокол є прикладом інтерактивної системи доведення з нульовим розголошенням, тобто обидві сторони знаходяться у постійній взаємодії.

1.4.2 Протокол автентифікації Sedicii

Автентифікація Sedicii базується на перевіреному протоколі ZKP для забезпечення безпечного входу без передачі та зберігання особистих даних користувача. У рамках процесу автентифікації Sedicii ZKP застосовується наступним чином [23]:

- користувач жодного разу не надсилає особисту інформацію верифікатору, що означає, що за допомогою Sedicii приватні дані користувача ніде не передаються та не зберігаються. Оскільки приватні дані не передаються, будь-яка інформація, отримана під час передачі, не має значення для хакерів, щоб вчиняти шахрайство;

- автентифікація Sedicii виконується 20, 30 або більше разів проти комбінації перестановок, і кожна використана перестановка не повторюється в наступних аутентифікаціях;

- у реалізації Sedicii використовується інформація про перестановку дійсна лише один раз і протягом певного часу, щоб запобігти повторним відтворенням і атакам грубої сили.

Sedicii має переваги перед іншими системами автентифікації завдяки тому, що не вимагає додаткового обладнання та працює у звичайній веб-програмі клієнт-сервер. Технологія Sedicii використовує стандартні функції HTML 5, що усуває потребу в плагіні браузера. Коли користувач входить до сервера перевірки ідентифікації або сервера автентифікації з підтримкою Sedicii, із сервера в браузер користувача надсилається ряд математичних завдань, які вимагають відповіді. Інформація перевіряється лише тоді, коли браузер користувача правильно відповідає на всі виклики. Для кожної нової спроби перевірки пропонується інший набір завдань. Таку саму методологію можна застосувати до кількох типів приватної інформації, як-от авторизації платежів кредитною картою, щоб фактичні дані картки ніколи не передавалися через Інтернет [23].

1.4.3 NuCypher

NuCypher – це програмне вирішення, що має на меті гарантувати додаткову безпеку й конфіденційність децентралізованим застосункам, побудованим на публічних блокчейнах, наприклад Ethereum. NuCypher дає змогу розробникам зберігати, розкривати й адмініструвати приватні дані та має на меті створити інтероперабельний рівень безпеки для різних блокчейнів, де розробники можуть надавати дозвіл на доступ до конфіденційної інформації в різних децентралізованих застосунках. NuCypher надає криптографічні послуги кільком протоколам, безпечно зберігаючи приватні дані публічних блокчейнів [24].

Платформа пропонує систему управління секретами й послуги динамічного контролю доступу через Umbral – власну схему шифрування, а також Ursula – мережу операційних вузлів [24].

Umbral – це схема шифрування NuCypher, яка дає змогу користувачам як зберігати дані в приватний спосіб, так і безпечно обмінюватися інформацією [24].

Власники даних надають права на розшифрування їхньому одержувачу в процесі, що виконується вузлами-посередниками (так званими Ursula-вузлами), які повторно шифрують дані для одержувачів. (Слід зазначити, що в посередників немає доступу до базових даних або ключів розблокування: вони просто застосовують додаткові умови, які повинні виконуватися для розблокування інформації) [24].

1.4.4 Aleo

Aleo використовує криптографію з нульовим знанням для забезпечення конфіденційності та програмованості. Він використовує це як основу системи під назвою ZEXE (Zero-Knowledge EXEcution). У ZEXE користувачі виконують

переходи між станами в автономному режимі. Цей процес створює доказ, який об'єднується в транзакцію в ланцюжку. Ця транзакція оновлює стан системи шляхом споживання/створення записів у ланцюжку. Ця система дає нам надійні гарантії конфіденційності, оскільки транзакції містять лише докази, а не вхідні дані, які їх створили. ZEXE може підтримувати смарт-контракти, які дозволяють користувачам взаємодіяти або передавати значення заздалегідь визначеним способом [25].

Aleo має досить потужний інструментарій, серед яких [25]:

- мова програмування Leo, яка абстрагує низькорівневі криптографічні концепції;
- менеджер пакетів Aleo. Це перший менеджер пакунків для схем із нульовим знанням. Він включає функції спільної роботи, наприклад команди та організації.

1.4.5 Автентифікація в Iden3

Iden3 — це рішення Self-Sovereign Identity (SSI), яке дозволяє користувачам використовувати свої вже існуючі перевірені ідентифікатори. Завдяки SSI вони можуть довести, що вони є тими, за кого себе видають, на основі доведень нульового знання. Одним із прямих застосувань технології iden3 є дозвіл веб-додаткам повторно використовувати ці ідентифікатори для входу на свої портали [26]. На рис. 2.1 представлено процес входу в iden3.

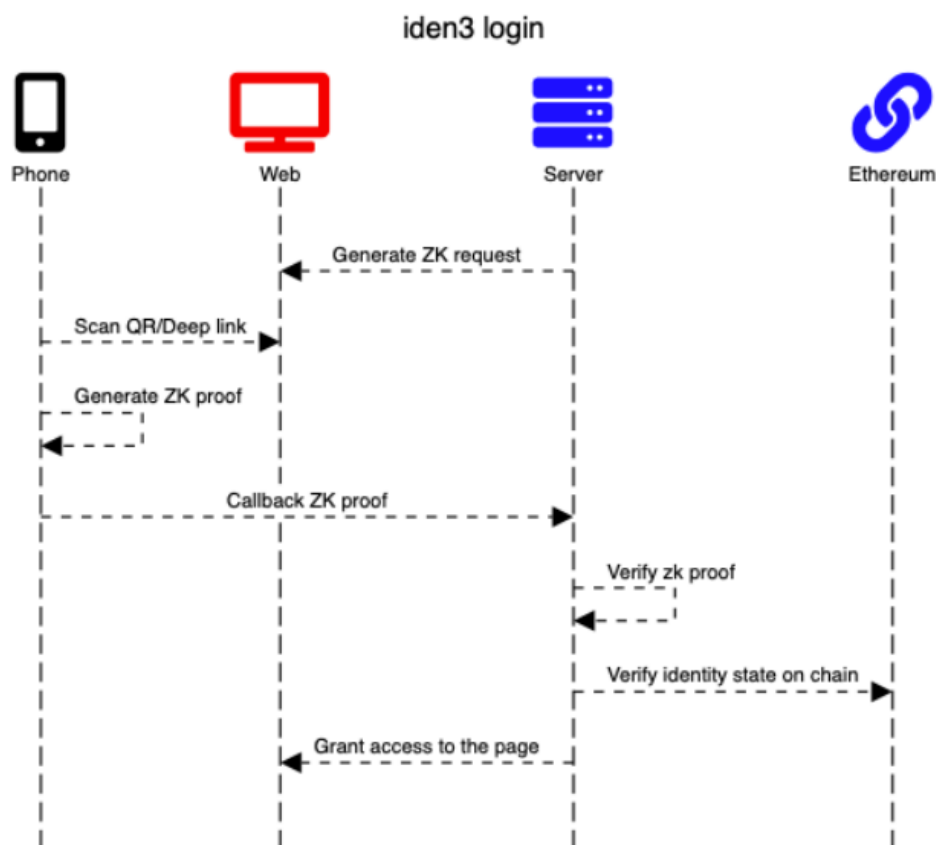


Рисунок 1.3 – Процес входу в iden3 [26]

Даний процес можна описати таким чином:

Крок 1. Сервер генерує запит на автентифікацію. Він може бути доставлений користувачеві через різні канали зв'язку: QR-код, електронну пошту, Deep Links тощо.

Крок 2. Клієнт аналізує запит, генерує доведення і надсилає відповідь на URL-адресу зворотного виклику.

Крок 3. Сервер перевіряє доведення(а також identity state на блокчейні Ethereum) і, в залежності від результату перевірки, приймає або відхиляє авторизацію.

Процес автентифікації в iden3 використовує принцип неінтерактивної взаємодії. В автентифікації Iden3 використовуються доведення Groth16 [12].

У цій дипломній роботі дана схема автентифікації була взята за основу для реалізації.

1.5 Постановка задачі

Реалізація Iden3 є досить успішною та надійною системою автентифікації з нульовим розголошенням. Проте її недоліком є те, що вона використовує Groth16 для генерації доведення. Цей протокол зарекомендував себе як надійний і досить потужний протокол. Проте цей протокол вимагає довірчого налаштування для його успішної роботи. Протокол Plonky2 позбавлений необхідності в довірчому налаштуванні. Окрім того, цей протокол оптимізований на апаратному рівні й використовує сучасні інструменти для генерації доведення. Тому програмна реалізація системи автентифікації використовуватиме саме цей протокол.

Отже, постановка задачі: зробити програмну реалізацію системи автентифікації, яка використовує Plonky2 для генерації та перевірки доведень. Взято за основу схему автентифікації від Iden3.

2 РОЗРОБКА ПРОЄКТУ СИСТЕМИ АВТЕНТИФІКАЦІЇ

2.1 Проєктування системи автентифікації за допомогою UML

Для відображення базових сутностей розроблюваної системи було вибрано зручну у використанні та зрозумілу у синтаксисі уніфіковану мову моделювання (Unified Modeling Language – UML). Це мова для специфікації, візуалізації, конструювання і документування програмних систем, а також бізнес-моделей та інших не програмних систем. UML є об'єднанням інженерних прийомів, які раніше успішно використовувалися під час моделювання великих і складних систем [27].

2.1.1 Діаграма варіантів використання

Візуальне моделювання в UML можна представити як деякий процес спуску по рівнях від найбільш загальної і абстрактної концептуальної моделі вихідної системи до логічної, а потім і до фізичної моделі відповідної програмної системи. Для досягнення цих цілей спочатку будується модель у формі так званої діаграми варіантів використання, яка описує функціональне призначення системи або, іншими словами, те, що система буде робити в процесі свого функціонування. Діаграма варіантів використання (use case) є вихідним концептуальним уявленням або концептуальною моделлю системи в процесі її проєктування і розроблення. Діаграма варіантів використання являє собою найбільш загальну контекстну модель проєктованої системи, яка є вихідною для всіх інших діаграм [28].

На рис. 2.1 зображено діаграму варіантів використання для схеми автентифікації.

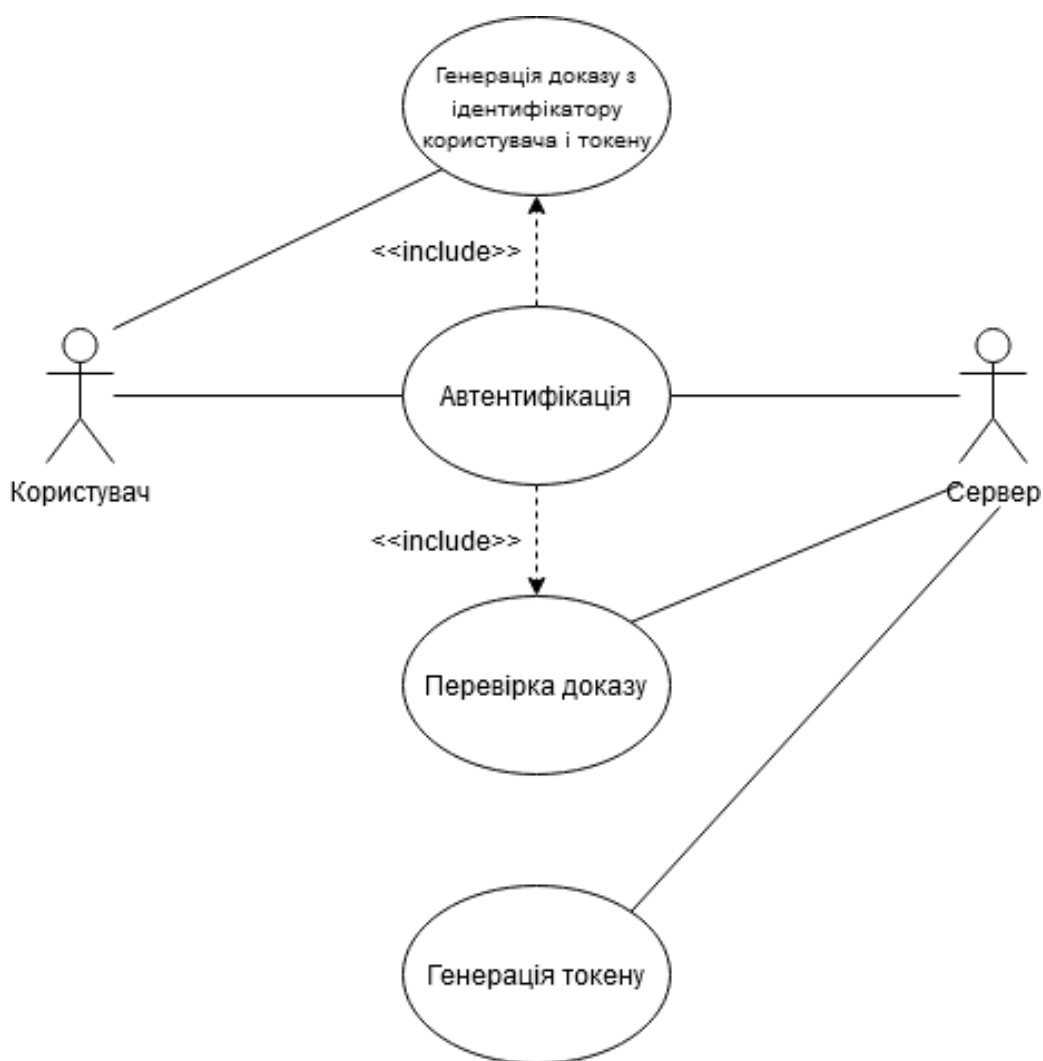


Рисунок 2.1 – Діаграма варіантів використання

З цієї діаграми можна побачити, що наявні два актора – користувач і сервер. Для здійснення процесу автентифікації зі сторони користувача необхідна генерація доведення з його ідентифікатора та токєну сервера, тоді як зі сторони сервера необхідна генерація токєну і перевірка згенерованого користувачем доведення.

2.1.2 Діаграма кооперації

Поняття кооперації (collaboration) є одним з фундаментальних понять у мові

UML. Воно служить для позначення безлічі взаємодіючих з певною метою об'єктів в загальному контексті модельованої системи.

Мета самої кооперації полягає в тому, щоб специфікувати особливості реалізації окремих найбільш значущих операцій в системі. Кооперація визначає структуру поведінки системи в термінах взаємодії учасників цієї кооперації [28].

На рис.2.2 зображено діаграму кооперації для схеми автентифікації.

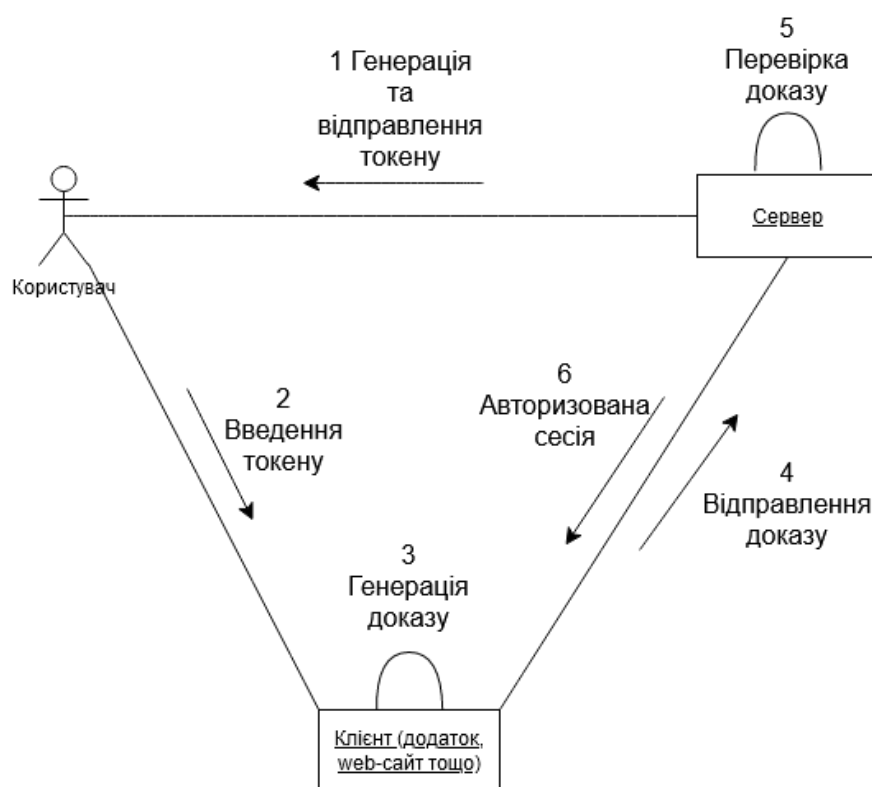


Рисунок 2.2 – Діаграма кооперації

На цій діаграмі видно, що існують три сутності: користувач, сервер та клієнт (додаток, web-сайт тощо). Також можна спостерігати порядок дій при автентифікації, а саме: генерація сервером токєну та його відправлення користувачу, введення токєну у клієнт, генерація клієнтом доведення і його відправлення серверу, перевірка сервером доведення і за істинності доведення побудова авторизованої сесії.

2.1.3 Діаграма послідовності

Хоча діаграми кооперації і використовуються для специфікації динаміки поведінки систем, але час у явному вигляді в них відсутній. Однак часовий аспект поведінки може мати суттєве значення під час моделювання синхронних процесів, що описують взаємодії об'єктів. Саме для цієї мети в мові UML використовуються діаграми послідовності (sequence).

За допомогою діаграми послідовності можна описати повний контекст взаємодій як своєрідний часовий графік «життя» всієї сукупності об'єктів, які взаємодіють між собою для реалізації варіанта використання програмної системи, досягнення бізнес-мети або виконання якогось завдання [28].

На рис. 2.3 зображено діаграму послідовності для схеми автентифікації.

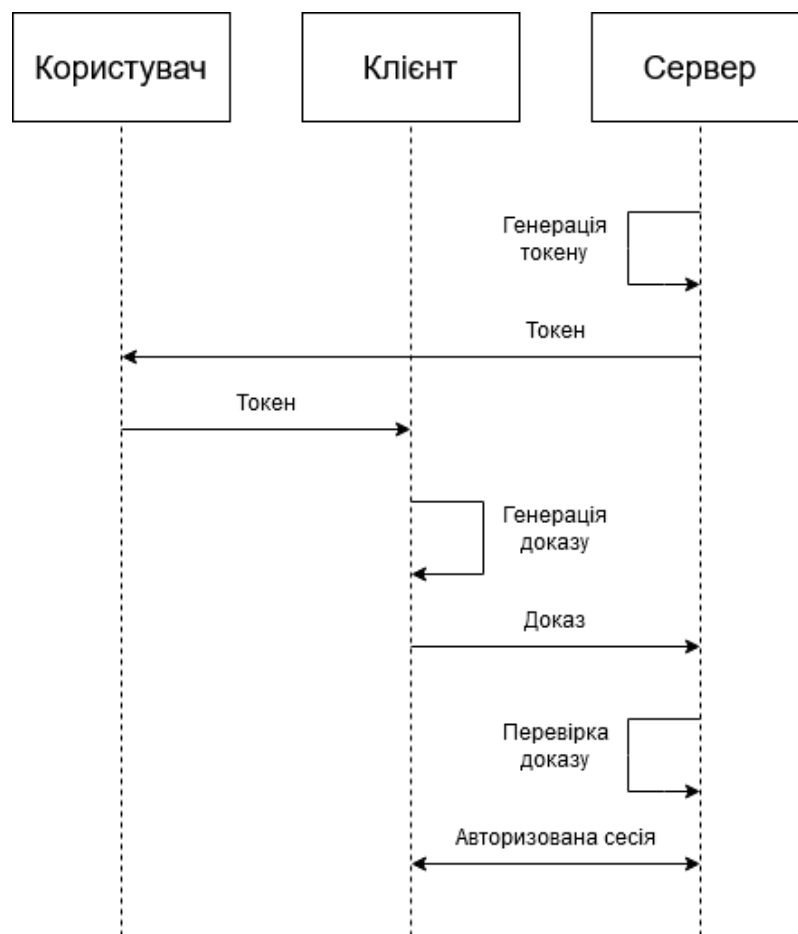


Рисунок 2.3 – Діаграма послідовності

Тут можна спостерігати подібну картину, як і в діаграмі кооперації. Проте варто зазначити, що тут описанні взаємодії у часовому просторі.

2.1.4 Діаграма класів

Діаграма класів (class diagram) слугує для представлення статичної структури моделі системи в термінології класів об'єктно-орієнтованого програмування. Діаграма класів може відображати, зокрема, різні взаємозв'язки між окремими сутностями предметної області, такими як об'єкти і підсистеми, а також описує їхню внутрішню структуру і типи відносин. На цій діаграмі не вказується інформація про часові аспекти функціонування системи. З цього погляду діаграма класів може слугувати подальшим розвитком концептуальної моделі проектованої системи [28].

На рис. 2.4 зображено діаграму класів для схеми автентифікації.

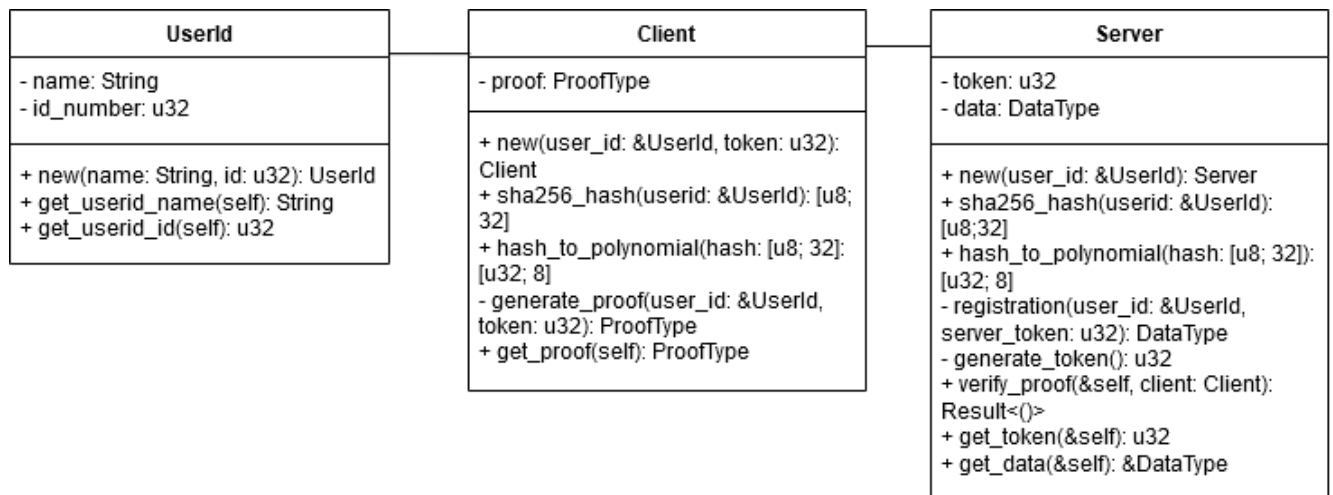


Рисунок 2.4 – Діаграма класів

2.1.5 Діаграма розгортання

Однією з форм фізичного представлення програмної системи є діаграма розгортання. Вона застосовується для представлення загальної конфігурації та топології розподіленої програмної системи та містить зображення розміщення компонентів за окремими вузлами системи. З іншого боку, діаграма розгортання показує наявність фізичних з'єднань маршрутів передачі між апаратними пристроями, задіяними у реалізації системи [28].

На рис. 2.5 зображено діаграму розгортання для схеми автентифікації.

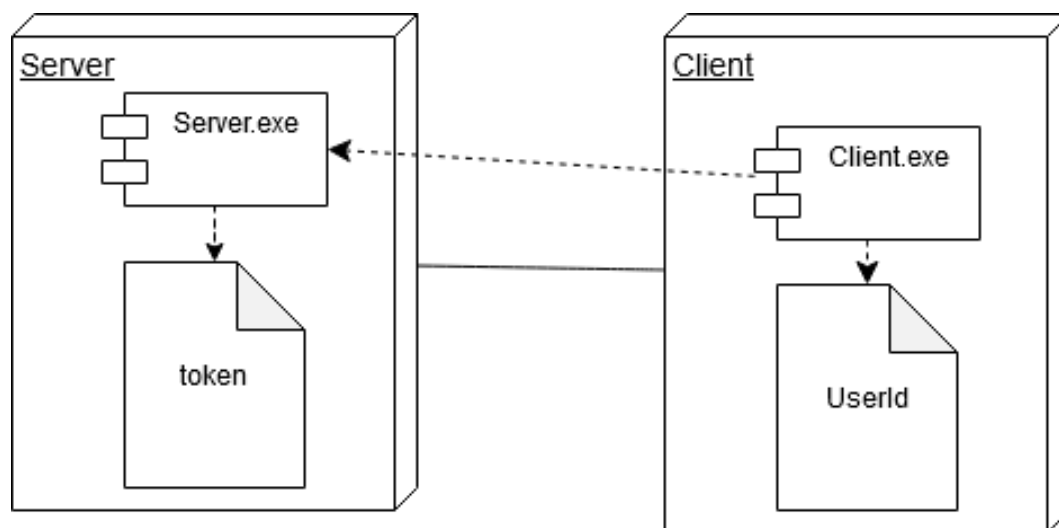


Рисунок 2.5 – Діаграма розгортання

2.2 Вибір мови програмування

Для виконання реалізації необхідно визначитись із мовою програмування. Було визначено такі критерії:

- висока продуктивність і швидкодія;
- безпечна робота з пам'яттю, надійність коду.

Серед популярних мов програмування можна виділити наступні:

- C/C++;
- Python;
- Java;
- C#;
- Go;
- JavaScript;
- Rust.

C/C++ - компільована, статично типізована мова програмування [29]. З моменту свого створення C++ став однією з найбільш широко використовуваних мов програмування у світі. Мова може працювати на найвищих рівнях абстракції і на апаратному рівні. C++ надає стандартні бібліотеки із високим рівнем оптимізації. Він забезпечує доступ до апаратних функцій низького рівня, щоб максимально збільшити швидкість та скоротити споживання пам'яті [30].

Проте у цій мові є один серйозний недолік. Він являє собою відсутність автоматичного управління пам'яттю, що може призводити до вразливостей (наприклад, переповнення буферу чи витік пам'яті).

Python — інтерпретована, динамічно типізована мова програмування. Має ефективні високорівневі структури даних і простий, але ефективний підхід до об'єктно-орієнтованого програмування. Завдяки простому синтаксису і динамічній типізації Python є зручною мовою програмування для створення сценаріїв і швидкої розробки програм у багатьох областях на більшості платформ. Python має автоматичне управління пам'яттю, що забезпечує безпеку пам'яті і даних [31].

Не зважаючи на ці переваги, Python має невисоку швидкість роботи та неефективне витрачання пам'яті, що є досить критичним для системи автентифікації [32].

Java – це статично типізована об'єктно-орієнтована мова програмування загального призначення. Має простий зрозумілий синтаксис і семантику. Його великою перевагою є портативність і кроссплатформеність завдяки Java Virtual Machine (JVM). Має високий рівень безпеки даних. Проте головним його

недоліком є низька швидкодія і споживання пам'яті [33].

C# - це кросплатформна мова загального призначення. На основі об'єктно-орієнтованих принципів він включає безліч функцій з інших парадигм, а не найменш функціонального програмування. Низькорівневі функції підтримують сценарії високої ефективності без небезпечного написання коду. Більшість виконання і бібліотек .NET написана на C#, і прогрес у C# часто використовується всім розробників .NET [34]. C#-розробники часто використовують для розробки додатків Windows, веб-сервісів і розробки ігор за допомогою ігрового двигуна Unity. Ця мова також широко використовується розробниками при розробці корпоративного програмного забезпечення та серверних систем [35].

Продуктивність: хоча C# забезпечує хорошу продуктивність, він не такий ефективний, як мови нижчого рівня, такі як C або C++. В основному це пов'язано з додатковими абстракціями та накладними витратами на час виконання, введеними платформою .NET [35].

Go - компільована і багатопотокова мова [36]. Вона поєднує високу продуктивність компільованих мов з легкістю написання коду. Має свій збиральник сміття (англ. Garbage collector) [37].

Основні переваги мови програмування Go [37]:

- простий та лаконічний синтаксис;
- статична типізація;
- швидкість компіляції;
- паралелізм;
- дуже потужна стандартна бібліотека;
- можливість писати у функціональному стилі.

Недоліків, які могли бути критичними для створення системи автентифікації, не знайдено.

JavaScript – це високорівнева скриптова мова програмування, яка широко використовується для створення інтерактивних застосунків і сайтів [38].

Переваги JavaScript [38]:

- легкий для вивчення синтаксис;
- кросплатформенність;
- широкий спектр бібліотек та фреймворків.

Недоліки JavaScript [38]:

- більш низька швидкодія;
- вразливість до кібератак, наприклад міжсайтовий скриптинг.

Rust - це сучасна багатопарадигмальна мова, яка має строгу типізацію і сфокусована на безпечній роботі з пам'яттю. Переваги мови програмування Rust [39]:

- продуктивність: швидкий і ефективний з використанням пам'яті, підходить для критично важливих служб, вбудованих пристроїв і легко інтегрується з іншими мовами.

- надійність: система типів і модель власності забезпечують безпеку пам'яті та потоків, зменшуючи помилки під час компіляції.

- продуктивність: повна документація, компілятор, який забезпечує чудову діагностику, і розширений інструментарій, включаючи менеджер пакунків і інструмент збірки (cargo), автоматичне форматування (rustfmt), лінтер (clippy) і підтримку редактора (rust-analyzer).

У табл. 2.1 приведено загальне порівняння показаних мов програмування.

Таблиця 2.1 – Порівняння мов програмування

Мова програмування	Висока швидкодія	Безпека пам'яті і даних
C/C++	+	-
Python	-	+
Java	-	+
C#	-	+
Go	+	+
JavaScript	-	-
Rust	+	+

Для програмної реалізації схеми автентифікації підходять мови Go і Rust. Вони обидві мають високу швидкодію та безпеку пам'яті. Але окремою перевагою Rust є наявність готової бібліотеки `plonky2` для Rust, яка реалізує внутрішні механізми протоколу, що значно спрощує створення програмної реалізації системи автентифікації. Тому для програмної реалізації було обрано мову програмування Rust.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Огляд бібліотеки plonky2

Для Rust існує окрема бібліотека plonky2, яка реалізовує внутрішній механізм роботи протоколу Plonky2 і дозволяє зручно використовувати і вбудовувати цей протокол у проєкти. Ця бібліотека створена авторами протоколу [17].

Перелік усіх основних модулів, які наявні у бібліотеці [40]:

- field - реекспорт plonky2_field;
- fri – реалізація FRI;
- gadgets - допоміжні гаджети, які надають додаткові методи для CircuitBuilder, щоб полегшити створення схеми;
- gates - користувацькі вентиля plonky2;
- hash - хеш-логіка plonky2 для хешування в схемі та перевірки доведень структури Меркла, а також реалізації спеціальних хеш-функцій;
- ior – логіка інтерактивних доведень оракула;
- plonk – система доведення plonky2;
- recursion - логіка рекурсії для перевірки рекурсивних схем plonky2;
- util - допоміжний модуль для допоміжних методів і логіки серіалізації plonky2.

Модулі, які використанні у програмній реалізації [40]:

- GoldilocksField - поле, вибране для швидкого скорочення. Його порядок $2^{64} - 2^{32} + 1$. Знаходиться в основному модулі field;
- Field – тип скінченного поля. Знаходиться в основному модулі field;
- PartialWitness - використовується для поступового створення даних свідоцтва, зберігаючи часткову інформацію, яка може бути доповнена пізніше. Знаходиться в основному модулі ior;
- WitnessWrite - використовується для серіалізації даних свідоцтва у зручний формат. Знаходиться в основному модулі ior;

- `CircuitBuilder` - структура, яка використовується для побудови схеми `plonky2`. Він надає весь необхідний інструментарій, який, з початкової конфігурації схеми, дозволить розробити схему та пов'язану з нею схему дані перевірки/верифікатора. Знаходиться в основному модулі `plonk`;

- `CircuitConfig` - конфігурація під час побудови схеми. Це визначає форму контуру а також цільовий рівень безпеки та параметри підпротоколу (наприклад, `FRI`). Знаходиться в основному модулі `plonk`;

- `CircuitData` - дані схеми, які вимагаються доказуючим та перевіряючим. Знаходиться в основному модулі `plonk`;

- `GenericConfig` - узагальнена конфігураційна структура, яка дозволяє налаштовувати різні параметри системи, такі як розміри полів, циклічні залежності та інші важливі параметри. Знаходиться в основному модулі `plonk`;

- `PoseidonGoldilocksConfig` - конкретна конфігурація, яка використовує хеш-функцію `Poseidon` та поле `Goldilocks`. Знаходиться в основному модулі `plonk`;

- `ProofWithPublicInputs` - структура, яка поєднує доведення з його відкритими входами. Знаходиться в основному модулі `plonk`.

Таким чином, бібліотека `plonky2` має усі необхідні модулі для реалізації системи автентифікації.

3.2 Розробка програмного коду модулів системи автентифікації

Програмна реалізація схеми автентифікації являє собою бібліотеку `auth_lib`, яку можна вбудовувати у різні проєкти. У бібліотеці наявні наступні модулі:

- `user_id` – ідентифікатор користувача, який містить у собі його ім'я та ідентифікаційний номер;
- `client` – клієнтський додаток, який генерує доведення;
- `server` – серверна частина, яка перевіряє доведення.

Поля модулю `user_id`:

- name (строка String) – ім'я користувача;
- id_number (беззнакове ціле число u32) – ідентифікаційний номер користувача.

Поле модулю client є proof (ProofWithPublicInputs<GoldilocksField, PoseidonGoldilocksConfig, 2>, далі ProofType) - згенероване доведення.

Поля модулю server:

- token (беззнакове ціле число u32) – токен, який сервер відправляє користувачу;
- data (CircuitData<GoldilocksField, PoseidonGoldilocksConfig, 2>, далі DataType) – арифметична схема, на якій перевіряється доведення.

Методи модулю user_id:

- new(name: String, id: u32) – функція-конструктор, який створює екземпляр структури UserId;
- get_userid_name(self) – повертає поле name;
- get_userid_id(self) – повертає поле id.

Методи модулю client:

- new(user_id: &UserId, token: u32) – функція-конструктор, який створює екземпляр Client;
- sha256_hash(userid: &UserId) – хешування sha256. Повертає хеш типу u32;
- hash_to_polynomial(hash: [u8; 32]) – перетворює хеш у коефіцієнти поліному, є частиною арифметизації хешу. Повертає масив з восьми елементів u32;
- generate_proof(user_id: &UserId, token: u32) – генерація доведення. Повертає доведення типу ProofType;
- get_proof(self) – повертає поле proof.

Методи модулю server:

- new(user_id: &UserId) – функція-конструктор, який створює екземпляр структури Server;
- sha256_hash(userid: &UserId) – хешування sha256. Повертає хеш типу u32;
- hash_to_polynomial(hash: [u8; 32]) – перетворює хеш у коефіцієнти

поліному, є частиною арифметизації хешу. Повертає масив з восьми елементів u32;

- registration(user_id: &UserId, server_token: u32) – при реєстрації користувача у мережі будує арифметичну схему з його даних та згенерованого сервером токена. Повертає арифметичну схему типу DataType;

- generate_token() – генерує токен u32;

- verify_proof(&self, client: Client) – перевіряє доведення. Повертає результат Result<>: якщо доведення істинне, то повертає Ok, якщо ні, то повертає Error;

- get_token(&self) – повертає поле token;

- get_data(&self) – повертає поле &data.

3.3 Опис роботи програмної реалізації

Програмна реалізація являє собою бібліотеку auth_lib. Для демонстрації її функціоналу створено тестову консольну програму AuthPlonky2. Функціонал тестової програми складається з реєстрації в мережі та самої автентифікації.

Під час реєстрації формується ідентифікатор користувача, який складається з його ім'я та його ідентифікаційного номеру.

На рис. 3.1 приведено приклад реєстрації користувача з іменем “John” та його ідентифікаційним номером 167825312.

```
=====Sign up=====
Input your name:
John
Input your ID:
167825312
```

Рисунок 3.1 – Приклад реєстрації користувача

Ці дані відправляються на сервер, де генерується схема `CircuitData` для користувача.

Під час автентифікації користувач вводить у клієнт свої дані, які вводив при реєстрації. Також він від сервера отримує токен, який необхідно ввести у клієнті. Цей токен для арифметичної схеми є входом. Після вводу даних клієнт генерує доведення.

На рис. 3.2 приведено приклад входу користувача з іменем “John” та його ідентифікаційним номером 167825312 у систему.

```
=====Log in=====
Input your name:
John
Input your ID:
167825312
Given token: 563944021
Input token:
563944021
```

Рисунок 3.2 – Приклад входу користувача у систему

Після генерації доведення передається серверу на перевірку. Якщо ідентифікатор є неправильним (тобто не співпадає з тим, який був під час реєстрації), або токен не співпадає із тим, що згенеровано сервером, то сервер поверне помилку і користувач автентифікацію не пройде.

На рис. 3.3 приведено успішно пройдену автентифікацію. На рис. 3.4 і 3.5 приведено приклади безуспішної автентифікації: при неправильному ідентифікаторові та неправильному токені.

```
=====Log in=====
Input your name:
John
Input your ID:
167825312
Given token: 563944021
Input token:
563944021
Authentication was successful. You are authorized in the system
```

Рисунок 3.3 – Приклад успішної автентифікації

```
=====Sign up=====
Input your name:
John
Input your ID:
167825312
=====Log in=====
Input your name:
Adam
Input your ID:
01223542
Given token: 1260721296
Input token:
1260721296
Authentication failed
```

Рисунок 3.4 - Приклад безуспішної автентифікації при неправильному ідентифікаторові

```
=====Sign up=====
Input your name:
John
Input your ID:
167825312
=====Log in=====
Input your name:
John
Input your ID:
167825312
Given token: 259103559
Input token:
11772221
Authentication failed
```

Рисунок 3.5 - Приклад безуспішної автентифікації при неправильному токени

Сама програмна реалізація приведена у додатку А, а також у GitHub-репозиторії [41].

3.4 Результати та оцінка виконання програми

Програмна реалізація системи автентифікації з використанням системи доведень з нульовим розголошенням успішно реалізовує автентифікацію користувача. Для успішної автентифікації необхідно і правильність даних користувача, і правильність введеного токена. Необхідно зазначити, що дана програмна реалізація не передбачає способи та методи передачі інформації між сторонами. Також надійне зберігання ідентифікаторів і токена залежить від самого клієнтського застосунку та серверного програмного забезпечення. Також необхідно зазначити, що у програмній реалізації використовувались дещо спрощений спосіб арифметизації вихідних даних, але навіть у такому вигляді реалізація добре виконує задачу автентифікації.

Проведено також тестування швидкості процесу автентифікації і генерації доведення. Тестування проводилось на процесорі AMD Athlon 200GE 3.2 ГГц, ОЗУ – 4 Гб DDR4. Алгоритм не є багатопоточним, тому розрахунки проводились на одному ядрі. Тестування швидкості являє собою середній час виконання процесу автентифікації і генерації й перевірки доведення. Виявлено, що середній час генерації доведення дорівнює приблизно 2.1 секунди, тоді як мінімальний час генерації доведення дорівнює 192 мілісекунди. Середній час перевірки доведення дорівнює 131 мілісекунда, тоді як мінімальне значення дорівнює 115 мілісекунд. Сумарно, середній час автентифікації дорівнює 2.2 секунди. Варто зазначити, що це гарні результати у порівнянні з минулими результатами від авторів Plonky2, а саме у 2020 році швидкість генерації доведення сягала 60 секунд [42].

Автори протоколу говорять, що при оптимізації швидкості перевірки ми використовуємо швидкість кодового слова $1/8$. Потім, при налаштуванні на 100 біт припущеної безпеки, Plonky2 має поріг 212 вентилів. Створення цих рекурсивних доказів на Macbook Air 2021 займає близько 300 мс [17].

При оптимізації розміру автори використовують більші швидкості кодових слів. Використовуючи коефіцієнт $1/256$, Plonky2 здатний стиснути будь-який

доказ приблизно до 43 кілобайт. Створення цих доказів займає приблизно 11,6 секунд, знову ж таки на Macbook Air 2021 року [17].

Таким чином, вивчення та впровадження Plonky2 як протоколу з нульовим розголошенням у схемах автентифікації є виправданим і корисним завданням, що поєднує в собі високу швидкодію та надійність, задовольняючи сучасні вимоги до безпечних і ефективних систем автентифікації.

Програма, яка виконувала вимірювання, приведена у додатку Б.

ВИСНОВКИ

Системи доведення з нульовим розголошенням є потужним криптографічним інструментом, що знайшли своє застосування у багатьох сферах, включаючи автентифікацію та захист конфіденційності.

Було розглянуто протокол Plonky2, як приклад неінтерактивної системи доведення з нульовим розголошенням. Завдяки FRI, рекурсії та обраному полю цей протокол демонструє високу ефективність, і завдяки цьому цей протокол можна використовувати у проектах, де швидкодія грає важливу роль.

У даній роботі було розглянуто приклади схем автентифікації, які використовують системи доведення з нульовим розголошенням, а саме SRP та автентифікацію в Iden3.

Було зроблено постановку задачі. Схему автентифікації в Iden3 було обрано за основу для програмної реалізації. Для більш детального розуміння було створено UML-діаграми, що описують варіанти використання, кооперацію та послідовність операцій у системі.

Розробка програмного забезпечення на основі Plonky2 здійснювалася за допомогою мови програмування Rust і бібліотеки plonky2, яка надала необхідний інструментарій для реалізації. Практична реалізація являє собою бібліотеку `auth_lib`, яка має необхідний функціонал для успішної реалізації схеми автентифікації.

Програмна реалізація успішно реалізує процес генерації та верифікації доведень. Верифікація проходить успішно лише за умови істинності доведень. Програма демонструє високі показники продуктивності та точності, що свідчить про ефективність використання Plonky2 у реальних застосуваннях.

Проведене дослідження та розробка демонструють високу ефективність неінтерактивних систем доведення з нульовим розголошенням і їх значний потенціал для забезпечення безпеки і конфіденційності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Клопотовський Д., Писаренко Л. Класифікація механізмів аутентифікації користувачів і їх огляд. URL: <https://ed.kpi.ua/wp-content/uploads/conferences/2017/2017-178-185.pdf> (дата звернення: 15.04.2024).
2. Литвин В., Мандрона М. Аналіз методів автентифікації в інформаційних системах. URL: https://virt.ldubgd.edu.ua/pluginfile.php/14209/mod_folder/content/0/Кафедра%20УІБ/Мандрона%20М.М/2013/1.pdf (дата звернення: 15.04.2024).
3. Goldwasser S., Micali S., Rackoff C. The Knowledge Complexity of Interactive Proof Systems. SIAM Journal on Computing. 1989. Т. 18, № 1. С. 186–208. URL: <https://doi.org/10.1137/0218012> (дата звернення: 15.04.2024).
4. CO6GC: Introduction to Zero-Knowledge Proofs (Part 1) - COSIC. COSIC. URL: <https://www.esat.kuleuven.be/cosic/blog/co6gc-introduction-to-zero-knowledge-proofs-1/> (дата звернення: 15.04.2024).
5. How to Explain Zero-Knowledge Protocols to Your Children / J. Quisquater та ін. Advances in Cryptology – CRYPTO '89: 9th Annual International Cryptology Conference. 1989. Vol. 435. С. 628–631. URL: https://doi.org/doi:10.1007/0-387-34805-0_60 (дата звернення: 15.04.2024).
6. Schnorr C. P. Efficient signature generation by smart cards. Journal of Cryptology. 1991. Т. 4, № 3. С. 161–174. URL: <https://doi.org/10.1007/bf00196725> (дата звернення: 16.04.2024).
7. Feige U., Fiat A., Shamir A. Zero-knowledge proofs of identity. Journal of Cryptology. 1988. Т. 1, № 2. С. 77–94. URL: <https://doi.org/10.1007/bf02351717> (дата звернення: 16.04.2024).
8. Guillou L., Quisquater J.-J. A Practical Zero-Knowledge Protocol Fitted to Security Microprocessor Minimizing Both Transmission and Memory. Advances in Cryptology — EUROCRYPT '88. 1988. LNCS 330. С. 123-128. URL: https://doi.org/10.1007/3-540-45961-8_11 (дата звернення: 16.04.2024).
9. Fiat A., Shamir A. How to Prove Yourself: Practical Solutions to Identification

and Signature Problems. *Advances in Cryptology — CRYPTO' 86: Proceedings. Lecture Notes in Computer Science*. 1986. Vol. 263. С. 186–194. URL: https://doi.org/10.1007/3-540-47721-7_12

10. What are zk-SNARKs?. Z.Cash. URL: <https://z.cash/technology/zksnarks/> (дата звернення: 17.04.2024).

11. Pinocchio: Nearly Practical Verifiable Computation / B. Parno та ін. *Cryptology ePrint Archive*. 2013. № 2013/279. URL: <https://ia.cr/2013/279> (дата звернення: 17.04.2024).

12. Groth J. On the Size of Pairing-based Non-interactive Arguments. *Cryptology ePrint Archive*. 2016. № 2016/260. URL: <https://ia.cr/2016/260> (дата звернення: 18.04.2024).

13. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings / M. Maller et al. *CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019. P. 2111–2128. URL: <https://doi.org/10.1145/3319535.3339817> (дата звернення: 18.04.2024).

14. Bowe S., Grigg J., Hopwood D. Recursive Proof Composition without a Trusted Setup. *Cryptology ePrint Archive*. 2019. № 2019/1021. URL: <https://ia.cr/2019/1021> (дата звернення: 18.04.2024).

15. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *Cryptology ePrint Archive*. 2019. № 2019/953. URL: <https://ia.cr/2019/953> (дата звернення: 19.04.2024).

16. Scalable, transparent, and post-quantum secure computational integrity / E. Ben-Sasson та ін. *Cryptology ePrint Archive*. 2018. № 2018/046. URL: <https://ia.cr/2018/046> (дата звернення: 20.04.2024).

17. GitHub - 0xPolygonZero/plonky2. GitHub. URL: <https://github.com/0xPolygonZero/plonky2> (дата звернення: 20.04.2024).

18. Fast Reed-Solomon Interactive Oracle Proofs of Proximity / E. Ben-Sasson та ін. *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. 2018. № 14. С. 14:1–14:17. URL: <https://doi.org/10.4230/LIPIcs.ICALP.2018.14> (дата звернення: 21.04.2024).

19. A Tutorial on Writing proofs with Plonky2 (Part I). Medium. URL: <https://polymerlabs.medium.com/a-tutorial-on-writing-zk-proofs-with-plonky2-part-i-be5812f6b798> (дата звернення: 21.04.2024).

20. DEEP-FRI: Sampling outside the box improves soundness. / E. Ben-Sasson та ін. Cryptology ePrint Archive. 2019. № 2019/336. URL: <https://ia.cr/2019/336> (дата звернення: 22.04.2024).

21. Butler B. Improved Authentication for Email Encryption and Security. Proton. URL: <https://proton.me/blog/encrypted-email-authentication> (дата звернення: 01.05.2024).

22. The Secure Remote Password Protocol. SRP: Industry-Standard Strong Password Security. URL: <http://srp.stanford.edu/ndss.html> (дата звернення: 02.05.2024).

23. Zero Knowledge Authentication. Sedicii. URL: <https://sedicii.com/news/Zero-Knowledge-Authentication/> (дата звернення: 02.05.2024).

24. Що таке NuCypher (NU)?. Kraken. URL: <https://www.kraken.com/uk-ua/learn/what-is-nucypher-nu> (дата звернення: 02.05.2024)

25. Making a more secure, accessible medical system with zero-knowledge. Aleo | Fully Private Applications. URL: <https://aleo.org/post/the-future-of-zero-knowledge-with-aleo/> (дата звернення: 02.05.2024).

26. Login protocol. Iden3 Documentation. URL: <https://docs.iden3.io/protocol/zklogin/> (дата звернення: 03.05.2024).

27. ISO/IEC 19501:2005. ISO. URL: <https://www.iso.org/standard/32620.html> (дата звернення: 06.05.2024).

28. Леоненков О. В. Самовчитель UML. 2-ге вид. 2004. 432 с.

29. Stroustrup B. C++ Programming Language. Addison Wesley, 2013. 1368 с.

30. Welcome back to C++ - Modern C++. Microsoft Learn: Build skills that open doors in your career. URL: <https://learn.microsoft.com/en-us/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=msvc-170> (дата звернення: 10.05.2024).

31. The Python Tutorial – Python 3.8.19 documentation. 3.12.3 Documentation.

URL: <https://docs.python.org/3.8/tutorial/index.html> (дата звернення: 10.05.2024).

32. Python: плюси і мінуси мови, які завдання вирішує і чи варто вивчати. Avada Media. URL: <https://avada-media.ua/ua/services/python-plyusy-i-minusy-yazyka-kakiye-zadachi-reshayet-i-stoit-li-izuchat/> (дата звернення: 10.05.2024).

33. Що таке Java і де вона використовується – GoIT Global. GoIT Global. URL: <https://goit.global/ua/articles/shcho-take-java-i-de-vona-vykorystovuietsia/> (дата звернення: 10.05.2024).

34. Overview - A tour of C#. Microsoft Learn: Build skills that open doors in your career. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview> (дата звернення: 10.05.2024).

35. 6 сучасних мов програмування та їх недоліки - Ласкаво просимо IT-професіонал. Welcome IT professional. URL: <https://uk.itpedia.nl/2023/07/13/6-moderne-programmeertalen-en-hun-downsides/> (дата звернення: 10.05.2024).

36. Мова програмування Go: перспективи, плюси та мінуси. senior.ua. URL: <https://senior.ua/articles/mova-programuvannya-go-perspektivi-plyusi-ta-mnusi> (дата звернення: 10.05.2024).

37. Як і для чого вивчати Golang. Переваги і недоліки мови. DOU. URL: <https://dou.ua/forums/topic/41933/> (дата звернення: 10.05.2024).

38. Що таке JavaScript і для чого він потрібен – GoIT Global. GoIT Global. URL: <https://goit.global/ua/articles/shcho-take-javascript-i-dlia-choho-vin-potriben/> (дата звернення: 10.05.2024).

39. GitHub - rust-lang/rust: Empowering everyone to build reliable and efficient software. GitHub. URL: <https://github.com/rust-lang/rust> (дата звернення: 10.05.2024).

40. plonky2 - Rust. Docs.rs. URL: <https://docs.rs/plonky2/0.2.2/plonky2/> (дата звернення: 11.05.2024).

41. GitHub - justKolya/auth_lib_plonky2. GitHub. URL: https://github.com/justKolya/auth_lib_plonky2 (дата звернення: 27.05.2024).

42. Introducing Plonky2. Polygon Technology. URL: <https://polygon.technology/blog/introducing-plonky2> (дата звернення: 28.05.2024).

ДОДАТОК А

КОД ПРОГРАМНОЇ РЕАЛІЗАЦІЇ НА RUST

А.1 Код бібліотеки auth_lib

Код user_id.rs

```
#[derive(Clone)]
```

```
pub struct UserId {  
    name: String,  
    id_number: u32,  
}
```

```
impl UserId{  
    pub fn new(name: String, id: u32) -> UserId{  
        UserId{  
            name: name,  
            id_number: id,  
        }  
    }  
    pub fn get_userid_name(self) -> String{  
        self.name  
    }  
    pub fn get_userid_id(self) -> u32{  
        self.id_number  
    }  
}
```

Код client.rs

```

use plonky2::field::goldilocks_field::GoldilocksField;
use plonky2::field::types::Field;
use plonky2::iop::witness::{PartialWitness, WitnessWrite};
use plonky2::plonk::circuit_builder::CircuitBuilder;
use plonky2::plonk::circuit_data::CircuitConfig;
use plonky2::plonk::config::{GenericConfig, PoseidonGoldilocksConfig};
use plonky2::plonk::proof::ProofWithPublicInputs;
use sha2::{Digest, Sha256};
use crate::user_id::UserId;

type ProofType = ProofWithPublicInputs<GoldilocksField,
PoseidonGoldilocksConfig, 2>;

pub struct Client {
    proof: ProofType,
}

impl Client {

    pub fn new(user_id: &UserId, token: u32) -> Client {
        Client {
            proof: Self::generate_proof(user_id, token),
        }
    }

    pub fn sha256_hash(userid: &UserId) -> [u8; 32] {

```

```

        let message = format!("{}", <UserId as
Clone>::clone(&userid).get_userid_name(), <UserId as
Clone>::clone(&userid).get_userid_id());
        let mut hasher = Sha256::new();
        hasher.update(message);
        let result = hasher.finalize();
        let mut hash = [0u8; 32];
        hash.copy_from_slice(&result);
        hash
    }

```

```

pub fn hash_to_polynomial(hash: [u8; 32]) -> [u32; 8] {
    let mut polynomial = [0u32; 8];

    for i in 0..8 {
        let offset = i * 4;
        polynomial[i] = u32::from_be_bytes([
            hash[offset],
            hash[offset + 1],
            hash[offset + 2],
            hash[offset + 3],
        ]);
    }
}

```

```

    polynomial
}

fn generate_proof(user_id: &UserId, token: u32) -> ProofType {
    let hash = Self::sha256_hash(&user_id);
    let arith = Self::hash_to_polynomial(hash);
    const D: usize = 2;

```

```

type C = PoseidonGoldilocksConfig;
type F = <C as GenericConfig<D>>::F;
let config = CircuitConfig::standard_recursion_config();
let mut builder = CircuitBuilder::<F, D>::new(config);

let x = builder.add_virtual_target();
let t = builder.constant(F::from_canonical_u32(token));
let k = builder.constant(F::from_canonical_u32(arith[0]));
let k0 = builder.add(t, k);
let k1 = builder.mul_const(F::from_canonical_u32(arith[1]), x);
let s1 = builder.add(k0, k1);
let x2 = builder.mul(x, x);
let k2 = builder.mul_const(F::from_canonical_u32(arith[2]), x2);
let s2 = builder.add(s1, k2);
let x3 = builder.mul(x2, x);
let k3 = builder.mul_const(F::from_canonical_u32(arith[3]), x3);
let s3 = builder.add(s2, k3);
let x4 = builder.mul(x3, x);
let k4 = builder.mul_const(F::from_canonical_u32(arith[4]), x4);
let s4 = builder.add(s3, k4);
let x5 = builder.mul(x4, x);
let k5 = builder.mul_const(F::from_canonical_u32(arith[5]), x5);
let s5 = builder.add(s4, k5);
let x6 = builder.mul(x5, x);
let k6 = builder.mul_const(F::from_canonical_u32(arith[6]), x6);
let s6 = builder.add(s5, k6);
let x7 = builder.mul(x6, x);
let k7 = builder.mul_const(F::from_canonical_u32(arith[7]), x7);
let s7 = builder.add(s6, k7);

```

```

        builder.register_public_input(x);
        builder.register_public_input(s7);
        let mut pw = PartialWitness::new();
        pw.set_target(x, F::from_canonical_u32(token));
        let data = builder.build::<C>();
        let proof = data.prove(pw).unwrap();
        proof
    }
    pub fn get_proof(self) -> ProofType{
        self.proof
    }
}

```

Код server.rs

```

use anyhow::Result;
use plonky2::field::goldilocks_field::GoldilocksField;
use plonky2::field::types::Field;
use plonky2::plonk::circuit_builder::CircuitBuilder;
use plonky2::plonk::circuit_data::{CircuitConfig, CircuitData};
use plonky2::plonk::config::{GenericConfig, PoseidonGoldilocksConfig};
use plonky2::iop::witness::{PartialWitness, WitnessWrite};
use crate::client::Client;
use crate::user_id::UserId;
use sha2::{Digest, Sha256};

type DataType = CircuitData<GoldilocksField, PoseidonGoldilocksConfig, 2>;

```

```

pub struct Server {
    token: u32,
    data: DataType,
}

impl Server {
    pub fn new(user_id: &UserId) -> Server {
        let token = Self::generate_token();
        Server {
            token,
            data: Self::registration(user_id, token),
        }
    }

    pub fn sha256_hash(userid: &UserId) -> [u8; 32] {
        let message = format!("{}", <UserId as
Clone>::clone(&userid).get_userid_name(), <UserId as
Clone>::clone(&userid).get_userid_id());
        let mut hasher = Sha256::new();
        hasher.update(message);
        let result = hasher.finalize();
        let mut hash = [0u8; 32];
        hash.copy_from_slice(&result);
        hash
    }

    pub fn hash_to_polynomial(hash: [u8; 32]) -> [u32; 8] {
        let mut polynomial = [0u32; 8];

        for i in 0..8 {
            let offset = i * 4;

```

```

    polynomial[i] = u32::from_be_bytes([
        hash[offset],
        hash[offset + 1],
        hash[offset + 2],
        hash[offset + 3],
    ]);
}

```

```

polynomial
}

fn registration(user_id: &UserId, server_token: u32) -> DataType{
    let hash = Self::sha256_hash(&user_id);
    let arith = Self::hash_to_polynomial(hash);

    const D: usize = 2;
    type C = PoseidonGoldilocksConfig;
    type F = <C as GenericConfig<D>>::F;

    let config = CircuitConfig::standard_recursion_config();
    let mut builder = CircuitBuilder::<F, D>::new(config);

    let x = builder.add_virtual_target();
    let t = builder.constant(F::from_canonical_u32(server_token));
    let k = builder.constant(F::from_canonical_u32(arith[0]));
    let k0 = builder.add(t, k);
    let k1 = builder.mul_const(F::from_canonical_u32(arith[1]), x);
    let s1 = builder.add(k0, k1);
    let x2 = builder.mul(x, x);
    let k2 = builder.mul_const(F::from_canonical_u32(arith[2]), x2);
    let s2 = builder.add(s1, k2);

```

```

let x3 = builder.mul(x2, x);
let k3 = builder.mul_const(F::from_canonical_u32(arith[3]), x3);
let s3 = builder.add(s2, k3);
let x4 = builder.mul(x3, x);
let k4 = builder.mul_const(F::from_canonical_u32(arith[4]), x4);
let s4 = builder.add(s3, k4);
let x5 = builder.mul(x4, x);
let k5 = builder.mul_const(F::from_canonical_u32(arith[5]), x5);
let s5 = builder.add(s4, k5);
let x6 = builder.mul(x5, x);
let k6 = builder.mul_const(F::from_canonical_u32(arith[6]), x6);
let s6 = builder.add(s5, k6);
let x7 = builder.mul(x6, x);
let k7 = builder.mul_const(F::from_canonical_u32(arith[7]), x7);
let s7 = builder.add(s6, k7);

builder.register_public_input(x);
builder.register_public_input(s7);
let mut pw = PartialWitness::new();
pw.set_target(x, F::from_canonical_u32(server_token));
let data = builder.build::<C>();
data
}

fn generate_token() -> u32 {
    let token = rand::random::<u32>();
    token
}

pub fn verify_proof(&self, client: Client) -> Result<> {
    let data = Self::get_data(&self);

```

```

    let proof = client.get_proof();
    data.verify(proof)
}
pub fn get_token(&self) -> u32{
    self.token
}
pub fn get_data(&self) -> &DataType{
    &self.data
}
}

```

A.2 Код тестової програми AuthPlonky2

```

use std::io;
use auth_lib::user_id::UserId;
use auth_lib::client::Client;
use auth_lib::server::Server;

fn main(){
    let mut su_name = String::new();
    let mut su_id = String::new();

    // реєстрація в мережі
    println!("====Sign up====");
    println!("Input your name:");
    io::stdin().read_line(&mut su_name).expect("");
    println!("Input your ID:");
    io::stdin().read_line(&mut su_id).expect("");
}

```

```

let su_id = su_id.trim().parse().unwrap();

let su_john = UserId::new(su_name, su_id);
let server = Server::new(&su_john);

// автентифікація
println!("====Log in====");
let mut name = String::new();
let mut id = String::new();
println!("Input your name:");
io::stdin().read_line(&mut name).expect("");
println!("Input your ID:");
io::stdin().read_line(&mut id).expect("");
let id = id.trim().parse().unwrap();
let given_token = server.get_token();
println!("Given token: {}", given_token);
println!("Input token: ");
let mut token = String::new();
io::stdin().read_line(&mut token).expect("");
let token = token.trim().parse().unwrap();
let john = UserId::new(name, id);

let client = Client::new(&john, token);
let result = server.verify_proof(client);
match result {
    Ok(_) => println!("Authentication was successful. You are authorized in
the system"),
    Err(_) => println!("Authentication failed"),
}
}

```

A.3 Код програми, яка робила вимірювання швидкості генерації і перевірки доведення

```

use std::time::{Instant, Duration};
use auth_lib::user_id::UserId;
use auth_lib::client::Client;
use auth_lib::server::Server;

fn main(){
    let names = ["Daniel", "Brown", "Blair", "Rivera", "Ewan", "Sanchez", "Emil",
        "Henry", "Emma", "Baker",
            "Fabio", "Matthew", "Perez", "Aiden", "Sophia", "Lee", "Nguyen",
        "Bence", "Bertie", "Clark",
            "Arvid", "Cormac", "Baptiste", "Flores", "Thompson", "Moore",
        "Penelope", "Angel", "Frankie",
            "Carter", "Dylan", "Miller", "William", "Lewis", "Levi", "Cassidy",
        "Williams", "Harris",
            "Magnolia", "Cian", "Samuel", "Hill", "Devon", "Elijah",
        "Campbell", "Martin", "Robinson", "Scarlett",
            "Dara", "Lucas", "Ethan", "Gijs", "Eden", "Antoni", "Andy",
        "Bonny", "James", "Luna", "Green",
            "Ashton", "Mitchell", "Mercedes", "Emerson", "Cecily", "Bram",
        "Rodriguez", "Victoria", "Mimi",
            "Isabella", "Evelyn", "Johnson", "Winter", "Sofia", "Joseph",
        "Lopez", "Cristobal", "Constantine",
            "Jackson", "Ramirez", "Noah", "Jack", "White", "Adrian",
        "Grace", "Walker", "Avery", "Aston", "Dana",
            "Jackson", "Edith", "Torres", "Harper", "Dallas", "Garcia",
        "Carmine", "Adams", "Johannes", "Aria",

```

"Minnie", "Avery", "Madison", "Martinez", "Brett", "Eleanor",
 "Liam", "Eleanor", "Charlotte", "King",
 "Blake", "Darragh", "Cameron", "Sage", "Archie", "Alex",
 "Carson", "Wright", "Anton", "Wilson", "Marlowe",
 "Logan", "Aspen", "Abigail", "Allen", "August", "Dakota",
 "Mason", "Amelia", "Sebastian", "Chloe", "Olivia",
 "Thomas", "Owen", "Oliver", "Hernandez", "Benjamin", "Scott",
 "Jones", "Emmy", "Ariel", "Anderson", "Nelson",
 "Frankie", "Smith", "Ava", "Charlie", "Evan", "Fay", "Belle",
 "Elizabeth", "Aria"];

let ids:[u32; 150] = [1942129257, 1828396027, 1508260882, 623203067,
 1480297250, 1978964114, 274970328, 1934552778,
 337237075, 13270987, 1659329258, 204098051, 198497110,
 1216460020, 1746535209, 1215582514, 354956438,
 1251618601, 1031868767, 724433947, 1324678695, 365796463,
 153347558, 20001462, 420492590, 385731516,
 1938276773, 1857642339, 656046994, 841563322, 1540509000,
 1437490707, 1450803961, 728459960, 114432155,
 119309957, 1706640098, 478888935, 2000344722, 862409508,
 110281464, 1188404752, 1775779134, 777593951,
 1131842160, 201428733, 1296296660, 1109715227, 130063627,
 2055281171, 1943611396, 1194830595, 714494906,
 1802855389, 586503164, 479444531, 1169143838, 2090162496,
 720623705, 1311650887, 518231863, 381616236,
 1848768805, 379447295, 965558180, 1852005385, 593574530,
 2117133462, 630315335, 1052427859, 1332057759, 643180225,
 415252528, 1143592921, 1673950388, 1974074141, 1875377728,
 202641124, 736116032, 1153208317, 2057095270, 393385620,
 1741928632, 956962945, 935589528, 622218576, 201637826,
 531707407, 2140001997, 1842844943, 337271932, 1360145895,

```

116527434, 996800722, 924268120, 553567588, 238073596,
877287336, 2123642395, 1997449571, 922453885, 1814052306,
1972900766, 384740935, 1048634835, 1686657342, 317669818,
415069363, 1681462421, 36240885, 1591345983, 677884366,
772225254, 410725002, 989824005, 1708657597, 1551899341,
1911661788, 1278624219, 2085821881, 43786130, 1349280364,
1286451786, 1606930131, 1625543355, 458760396, 465367794,
1413945434, 1611186, 1071192911, 1925200110, 191653883,
1182412967, 106317987, 1906333956, 1026168270, 412184632,
1329584670, 184910454, 130990019, 1093216417, 298684903,
415909625, 257302636, 1011888365, 449218899, 492912001,
525999467, 569316547, 1536879319];

```

```
let mut sum_proof: Duration = Duration::new(0, 0);
```

```
let mut sum_ver: Duration = Duration::new(0, 0);
```

```
let mut sum_all: Duration = Duration::new(0, 0);
```

```
let mut temp: Duration = Duration::new(0, 0);
```

```
for i in 0..151{
```

```
    let su_john = UserId::new(names[i].to_string(), ids[i]);
```

```
    let server = Server::new(&su_john);
```

```
    let given_token = server.get_token();
```

```
    let token = given_token;
```

```
    let john = UserId::new(names[i].to_string(), ids[i]);
```

```
    let gen_proof_time = Instant::now();
```

```
    let client = Client::new(&john, token);
```

```
    let gen_proof_stop = gen_proof_time.elapsed();
```

```
    println!("=====");
```

```
    println!("Generation proof duration: {:?}", gen_proof_stop);
```

```
sum_proof += gen_proof_stop/150;
temp += gen_proof_stop;
let ver_time = Instant::now();
let result = server.verify_proof(client);
let ver_stop = ver_time.elapsed();
println!("Verification proof duration: {:?}", ver_stop);
sum_ver += ver_stop/150;
temp += ver_stop;
sum_all += temp/150;
temp = Duration::new(0, 0);
}
println!("Proof: {:?}; Ver: {:?}; All: {:?}", sum_proof, sum_ver, sum_all)
}
```