НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою

Виконав : студент __4__ курсу, групи ІП-83
(шифр групи)
_____Шпильовий Роман Петрович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент Іваніщев Б.В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) професор, д.т.н., Сімоненко В. П._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2022 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Шпильового Романа Петровича

1. Тема проєкту *Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою*
керівник проєкту Іваніщев Богдан Вячеславович, асистент.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від

2. Термін здачі студентом закінченого проєкту *11 червня 2022 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд існуючих методів авторизації та аутентифікації.

Розділ 2. Огляд алгоритмів та технологій для розробки системи.

Розділ 3. Деталі розробки системи.

Розділ 4. Дослідження та аналіз розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), принципова схема (алгоритм дій програмного забезпечення).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В. П.		

7. Дата видачі завдання «30» серпня 2021 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2021-15.12.2021*	
2.	*Вивчення та аналіз завдання*	*15.12.2021-15.03.2022*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2022-25.03.2022А*	
4.	*Розробка структур окремих підсистем*	*25.03.2022-5.04.2022*	
5.	*Програмна реалізація системи*	*5.04.2022-15.04.2022*	
6.	*Оформлення пояснювальної записки*	*15.04.2022-20.05.2022*	
7.	*Захист програмного продукту*	*25.04.2022*	
8.	*Передзахист*	*11.06.2022*	
9.	*Захист*	*24.06.2022*	

Студент-дипломник _____ Роман ШПИЛЬОВИЙ
(підпис)

Керівник проєкту _____ Богдан ІВАНІЩЕВ
(підпис)

АНОТАЦІЯ

У даній бакалаврській дипломній роботі було детально розглянуто найпопулярніші на сьогоднішній день методи аутентифікації та авторизації, а також принцип їх роботи. Було проаналізовано сильні та слабкі сторони цих методів. На основі існуючих рішень було створено метод аутентифікації та авторизації з використанням еліптичної кривої Ed25519, а також розроблено програмне забезпечення для реалізації Zero Knowladge Proof алгоритму мовою програмування Rust. Проведено експерименти з використанням алгоритмів хешування SHA-256 та BLAKE2. Програма розроблена у вигляді бібліотеки, що дозволяє використовувати її іншим розробникам у своїх проєктах.

Ключові слова: авторизація, аутентифікація, Zero Knowladge, Ed25519, BLAKE2, SHA-256, Rust, Cargo.

ANNOTATION

In this project for a Bachelor's Degree the most popular methods of authentication and authorization, as well as the principle of their work were considered in detail. The strengths and weaknesses of these methods were analyzed. Based on existing solutions, a method of authentication and authorization using the elliptic curve Ed25519 was created, as well as software was developed to implement the Zero Knowladge Proof algorithm in the Rust programming language. SHA-256 and BLAKE2 hashing algorithms were used in experiments and compared with each other. The program is designed as a library that allows other developers to use it in their projects.

Key words: authorization, authentication, Zero Knowladge, Ed25519, BLAKE2, SHA-256, Rust, Cargo.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Сервіс аутентифікації та	3		
			авторизації для систем з			
			розподіленою архітектурою			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Сервіс аутентифікації та	88		
			авторизації для систем з			
			розподіленою архітектурою			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Сервіс аутентифікації та	1		
			авторизації для систем з			
			розподіленою архітектурою			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Сервіс аутентифікації та	1		
			авторизації для систем з			
			розподіленою архітектурою			
			Функціональна схема			
			(діаграма класів)			
	А4	***ІАЛЦ.467200.006 Д3***	Сервіс аутентифікації та	1		
			авторизації для систем з			
			розподіленою архітектурою			
			Алгоритм дій програмного			
			забезпечення			
	А4	***ІАЛЦ.467200.007 Д4***	Сервіс аутентифікації та	17		
			авторизації для систем з			
			розподіленою архітектурою			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА	
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***		
Розроб		Шпильовий Р. П.			*Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою* Опис альбому	Літ. / Аркуш 1 / Аркушів 1
Перев		Іваніщев Б. В.				***КПІ ім. Ігоря Сікорського ФІОТ ІП-83***

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою*»

Київ – 2022 р

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 2

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ 2

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ 2

4. ДЖЕРЕЛА РОЗРОБКИ 2

5. ТЕХНІЧНІ ВИМОГИ 2

5.1 Вимоги до розробленого продукту 2

5.2 Вимоги до програмного забезпечення 3

5.3 Вимоги до апаратної частини 3

6. ЕТАПИ РОЗРОБКИ 3

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи авторизації та аутентифікації на основі Zero Knowladge Proof алгоритму, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є сфера інформаційних технологій, а саме програми та компоненти, для яких поставлена задача безпечної авторизації та аутентифікації.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Київського Політехнічного Інституту імені Ігоря Сікорського.

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка системи для безпечної авторизації та аутентифікації на основі Zero Knowladge Proof алгоритму, що зробить ці процеси безпечними.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проєкту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, а також науково-технічна література.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має відповідати таким вимоги:

- Реалізований Zero Knowladge Proof алгоритм.
- Програма побудована у вигляді бібліотеки.
- Актуальна інформація.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Cargo.
- Visual Studio C++ Build tools.

5.3. Вимоги до апаратної частини

- Комп’ютер на базі процесора Intel® Core i3-2100T і вище.
- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

6. ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.12.2021-15.12.2021
Вивчення та аналіз завдання	15.12.2021-15.03.2022
Розробка архітектури та загальної структури системи	15.03.2022-25.03.2022
Розробка структур окремих частин системи	25.03.2022-5.04.2022
Програмна реалізація системи	05.04.2022-15.04.2022
Виправлення помилок	15.04.2022-20.05.2022
Оформлення пояснювальної записки	25.04.2022

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою*»

Київ – 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 4

ВСТУП 5

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ МЕТОДІВ АВТОРИЗАЦІЇ ТА АУТЕНТИФІКАЦІЇ 7

1.1 Визначення понять 7

1.2 Види аутентифікації 7

1.2.1 Аутентифікація за паролем 7

1.2.2 Аутентифікація за сертифікатами 10

1.2.3 Аутентифікація одноразовими паролями 12

1.2.4 Аутентифікація ключами доступу 13

1.2.5 Аутентифікація токенами 14

1.3 Альтернативи паролів 21

1.3.1 Недоліки паролів 21

1.3.2 Візуальний пароль 22

1.3.3 Менеджер паролів 24

1.3.4 Біометричні паролі 25

1.3.5 Об’єднаний єдиний вхід 25

1.4 Порівняння альтернатив 26

ВИСНОВОК ДО РОЗДІЛУ 1 29

РОЗДІЛ 2. ОГЛЯД АЛГОРИТМІВ ТА ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ 31

2.1 Поширені вразливості та помилки реалізації паролів 31

2.2 Детальний огляд основного алгоритму 32

2.2.1 Існуюча проблема 32

2.2.2 Zero Knowledge Proof 33

2.2.3 Основні властивості ZKP 34

2.2.4 Socialist Millionaire’s Problem (SMP Protocol) 34

2.2.5 Off-the-Record Protocol 35
2.3 Удосконалення SMP 36
2.3.1 Безпека SMP 36
2.3.2 ed25519 38
2.3.3 Розширення ed25519 39
2.4 Протокол 41
2.4.1 Робота алгоритму 41
2.4.2 Специфіка ECC 42
2.4.3 Хешування 42
2.4.4 SHA-256 43
2.4.5 BLAKE2 44
2.5 Вибір мови програмування 45
ВИСНОВОК ДО РОЗДІЛУ 2 47
РОЗДІЛ 3. ДЕТАЛІ РОЗРОБКИ СИСТЕМИ 48
3.1 Розробка програмних компонентів алгоритму 48
3.1.1 Реалізація стану 48
3.1.2 Відображення стану 51
3.1.3 Authenticator 51
3.1.5 Допоміжні функції 55
3.1.6 Write 60
3.1.7 Read 61
3.1.8 Функція step1_client_to_server 62
3.1.9 Функція step2_server_to_client 64
3.1.10 Функція step3_client_to_server 65
3.1.11 Функція step4_server_to_client 66
3.1.12 Функція step5_client_is_done 67
3.2 Розробка програмних компонентів тестування 69
3.2.1 Тести 69
3.2.2 Бенчмарки 71

ВИСНОВОК ДО РОЗДІЛУ 3 74
РОЗДІЛ 4. ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ..... 75
4.1 Результати роботи алгоритму SHA-256..... 75
4.2 Результати роботи алгоритму BLAKE2..... 77
4.3 Порівняння результатів 79
4.3.1 Порівняння функцій хешування..... 79
4.3.2 Порівняння із звичайними паролями..... 80
ВИСНОВОК ДО РОЗДІЛУ 4 82
ВИСНОВКИ..... 83
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 85
ДОДАТОК 1..... 3
ДОДАТОК 2..... 5
ДОДАТОК 3..... 7
ДОДАТОК 4..... 8

ПЕРЕЛІК СКОРОЧЕНЬ

HTTP	(HyperText Transfer Protocol) Проток передачі даних
WWW	(World Wide Web) Всесвітня мережа інтернет
URL	(Uniform Resource Locators) Електронна адреса
NTLM	(NT LAN Manager) Протокол мережевої аутентифікації
SSO	(Single Sign-On) Метод аутентифікації
CA	(certificate authority) Центр видачі сертифікатів
2FA	(two-factor authentication) Двофакторна аутентифікація
API	(Application Programming Interface) Програмний інтерфейс
SMS	(Short Message Service) Коротке текстове повідомлення
RSA	(Rivest, Shamir, Adleman) Криптографічний алгоритм шифрування
SSL	(Secure Sockets Layer) Криптографічний протокол
TLS	(Transport Layer Security) Протокол захисту транспортного рівня
IP	(Identity Provider)
SP	(service provider)
JSON	(JavaScript Object Notation)
SWT	(Simple Web Token)
SAML	(Security Assertion Markup Language)
JWT	(JSON Web Token)
ZKP	(Zero Knowladge Prood)
SMP	(Socialist Millionaire’s Problem)
OTR	(Off-the-Record)
DLP	(discrete logarithm problem)
ECC	(Elliptic Curve Cryptography)
DH	(Diffie-Hellman)
OWASP	(Open Web Application Security Project)

ВСТУП

Важко уявити сучасний світ без мережі Інтернет. Щохвилини величезні об’єми даних подорожують світом. Майже в кожній сфері життя так чи інакше використовується інтернет. Ні для кого не секрет, що про кожного з нас у мережі зберігається певна інформація, тому питання безпеки є дуже актуальним у контексті захисту особистих даних.

Щодня кожен з нас по декілька разів аутентифікується у різних застосунках, на різних сайтах. Наразі будь-де потрібно підтверджувати свою особистість для того щоб люди (або сервіси) розуміли з ким вони мають справу та як саме потрібно спілкуватися з конкретним співрозмовником (користувачем). У реальному житті ми можемо підтвердити свою особистість за допомогою документів, що засвідчують особу таких як паспорт, водійське посвідчення та інші. Але у цифровому світі потрібно використовувати інші методи (хоча часом в цьому процесі можуть брати участь реальні документи). Найпростішим способом є введення свого імені (логіну) та паролю. Звірившись зі своєю базою застосунок може розпізнати користувача і продовжити його минулу сесію, надати йому певні права доступу, які властиві саме цьому користувачу, або якщо дані не вірні, то заборонити подальше користування для запобігання несанкціонованого використання сервісу.

Варіантів реалізації цього процесу є велика кількість. Вони відрізняють скрадністю імплементацію, ціною впровадження в конкретний проект, ступенем захисту, швидкістю і так далі. Уже довгі роки вчені не полишають спроб знайти ідеальний метод для цієї задачі, але вони ніяк не можуть дійти консенсусу через різноманітність розроблюваних систем та різнотипність завдань, які ставляться перед розробниками.

Так як останні роки потужність систем значно виросла і час обробки запитів знизився, то все частіше розробники поступаються швидкодією методу

на користь безпеки. А який найбільш надійний спосіб зберегти свої дані? Не поширювати їх взагалі. Завдяки Zero Knowladge Proof алгоритмам ми можемо довести, що знаємо якийсь секрет, при цьому не видаючи цей секрет іншій стороні.

Отже, актуальною є задача розроблення зручного програмного забезпечення, легкого для використання, та реалізація безпечного способу авторизації та аутентифікації на основі Zero Knowladge Proof алгоритму для уникнення перехоплення даних.

РОЗДІЛ 1.
ОГЛЯД ІСНУЮЧИХ МЕТОДІВ АВТОРИЗАЦІЇ ТА АУТЕНТИФІКАЦІЇ

1.1 Визначення понять

Основні поняття які використовуються для розпізнавання користувачів у мережі – це ідентифікація, аутентифікація та авторізація. Звучать ці терміни дуже схоже, але кожен з них має свою функцію.

Ідентифікація – заява про те ким є користувач. Це може бути ім’я користувача, логін, електронна пошта, номер облікового запису і таке інше.

Аутентифікація – надання доказів того, що користувач є тим ким він ідентифікувався. Для прикладу це може бути надання паролю від облікового запису.

Авторизація – перевірка ролі користувача в системі та надання користувачу відповідного рівня доступу. Наприклад, при спробі видалити якийсь запис користувачу буде авторизовано право на це тальки якщо він є власником запису або в кього є права адміністратора.

Але в сучасних системах існують і більш складні схеми аутентифікації та авторизації.

1.2 Види аутентифікації

1.2.1 Аутентифікація за паролем

Цей метод ґрунтується на тому, що користувач повинен надати username (логін) та password (пароль) для успішної ідентифікації та аутентифікації в

системі. Пара логіну та паролю задається користувачем під час початкової реєстрації в системі, при цьому як username може бути ім’я, адреса електронної пошти користувача тощо.

Стосовно веб-застосунків, існує кілька стандартних протоколів для аутентифікації за паролем, які ми розглянемо нижче.

HTTP authentication

Цей протокол, описаний у стандартах HTTP 1.0/1.1[1], існує дуже давно і досі активно застосовується у корпоративному середовищі. Застосовується до веб-сайтів так:

Сервер, при зверненні неавторизованого клієнта до захищеного ресурсу, надсилає HTTP статус «401 Unauthorized» та додає заголовок «WWW-Authenticate» із зазначенням схеми та параметрів аутентифікації.

Браузер коли отримує таку відповідь, автоматично показує діалогове вікно введення username та password. Далі в ці поля користувач вводить деталі свого облікового запису.

Далі, для всіх наступних запитів на цей веб-сайт браузер автоматично сам буде додавати HTTP заголовок «Authorization», в якому вже будуть передаватися дані користувача для аутентифікації сервером.

Зі свого боку сервер аутентифікує користувача за даними цього заголовка. Рішення про надання доступу (авторизація) здійснюється окремо на підставі ролі користувача або інших даних облікового запису.

Весь процес стандартизований та добре підтримується всіма браузерами та веб-серверами. Існує кілька схем аутентифікації, що відрізняються за рівнем безпеки:

1. **Basic** - найпростіша схема, при якій username і password користувача передаються в заголовку Authorization у незашифрованому вигляді (base64-encoded). Однак при використанні HTTPS (HTTP over SSL) протоколу є відносно безпечним. Принцип роботи можна подивитися на рисунку 1.1. [2]

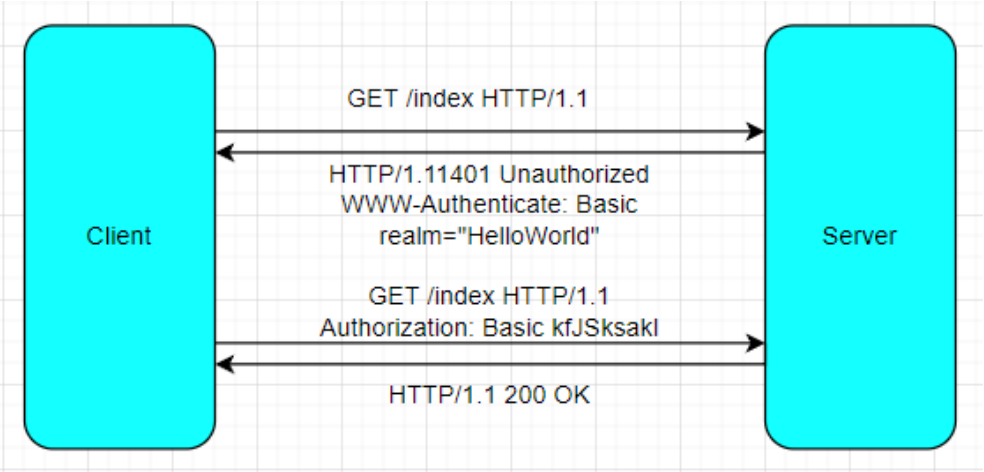


Рисунок 1.1 – Принцип роботи схеми Basic

2. **Digest** — challenge-response-схема, за якої сервер відправляє унікальне значення nonce, а браузер, у свою чергу, передає MD5 хеш пароля користувача, обчислений за допомогою зазначеного nonce. Це більш безпечна альтернатива схеми Basic при незахищених з'єднаннях, але вона схильна до man-in-the-middle (Атака посередника – коли перехоплюються і змінюються дані якими обмінюються клієнт і сервер [3]) атак (із заміною схеми на basic). Окрім цього, використання такої схеми не дозволяє використовувати сучасні хеш-функції для зберігання паролів користувачів на сервері.
3. **NTLM** (відома як Windows Authentication) - також заснована на challenge-response підході, при якому пароль не передається в чистому вигляді. Ця схема не є стандартом HTTP, але підтримується більшістю браузерів та веб-серверів. Переважно використовується для аутентифікації користувачів Windows Active Directory у веб-застосунках. Вразлива до pass-the-hash атак [4].
4. **Negotiate** — ще одна схема із сімейства Windows authentication, яка дозволяє клієнту вибрати між NTLM та Kerberos аутентифікацією. Kerberos — безпечніший протокол, заснований на принципі SSO. Однак він може функціонувати тільки якщо клієнт і сервер знаходяться в зоні intranet і вони є частиною домену Windows.

Варто зазначити, що при використанні HTTP-аутентифікації користувач не має звичної можливості вийти з веб-програми, окрім як закрити всі вікна браузера.

1.2.2 Аутентифікація за сертифікатами

Сертифікат – це набір атрибутів, які ідентифікують власника, підписаний certificate authority (CA). Certificate authority виступає у ролі посередника, що гарантує достовірність сертифікатів. Також сертифікат криптографічно пов'язаний із секретним ключем, що збережений у власника сертифіката і дозволяє беззаперечно підтвердити факт володіння сертифікатом.

На стороні клієнта закритий ключ разом із сертифікатом можуть зберігатися в операційній системі, браузері, файлі, чи на спеціальному фізичному пристрої (USB token, smart card). Зазвичай секретний ключ додатково захищений паролем або PIN-кодом.

У веб-застосунках традиційно використовують сертифікати стандарту X.509 [5]. Аутентифікація за допомогою сертифіката X.509 (приклад сертифікату X.509 можна побачити на рисунку 1.2) відбувається в момент з’єднання з сервером і є частиною протоколу SSL/TLS. Такий механізм також добре підтримується браузерами, які дозволяють користувачеві вибрати та застосувати сертифікат, якщо веб-сайт припускає такий спосіб аутентифікації.

Під час аутентифікації сервер перевіряє сертифікат на підставі наступних правил:

1. Сертифікат має бути підписаний довіреним certification authority (перевірка ланцюжка сертифікатів).
2. Сертифікат має бути дійсним на поточну дату (перевірка терміну дії).
3. Сертифікат не має бути відкликаним відповідним certificate authority (перевірка списків виключення).

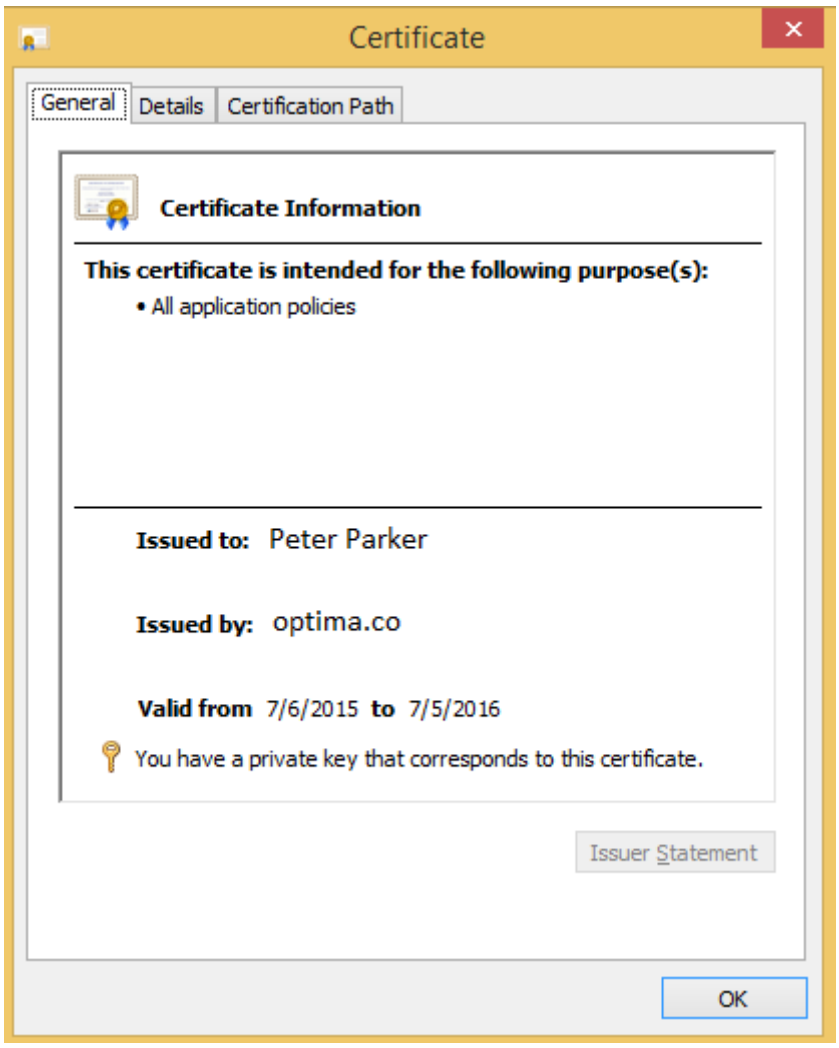


Рисунок 1.2 – Приклад сертифікату X.509

Після успішної аутентифікації, користуючись такими даними як: subject (ім'я власника), serial number (серійний номер сертифіката) або thumbprint (відбиток відкритого ключа сертифіката), а також issuer (емітент) веб-програма може виконати авторизацію запиту.

Використання сертифікатів для аутентифікації – набагато надійніший спосіб, ніж аутентифікація за допомогою паролів. Це досягається шляхом створенням у процесі аутентифікації цифрового підпису, наявність якого доводить факт застосування закритого ключа у конкретній ситуації (non-repudiation). Однак труднощі з поширенням та підтримкою сертифікатів роблять такий спосіб аутентифікації малодоступним для широкого використання.

1.2.3 Аутентифікація одноразовими паролями

Аутентифікація одноразовими паролями зазвичай застосовується додатково до аутентифікації за паролями для реалізації two-factor authentication[6] (2FA). Для двофакторної аутентифікації користувачеві необхідно надати підтверджені дані двох типів для успішного входу в систему: щось, що він знає (наприклад, пароль від облікового запису) і щось, чим він володіє фізично (наприклад, пристрій для генерації одноразових паролів). Наявність цих двох факторів відразу дозволяє значною мірою збільшити рівень безпеки, що може бути потрібна для певних видів веб-додатків.

Також одноразові паролі доволі часто використовуються для додаткової аутентифікації перед виконанням якихось важливих дій, наприклад, зміна паролю, переказ коштів тощо.

Існують різні джерела створення одноразових паролів. Найбільш популярні:

1. Апаратні або програмні токени, що застосовуються для генерування одноразових паролів на підставі секретного ключа, зафіксованого в них, і поточного часу. Секретні ключі користувачів, також зберігаються на сервері, що дозволяє щоразу перевіряти введені паролі. Приклад апаратної реалізації токенів - RSA SecurID[7]; програмної програми Google Authenticator [8].
2. Випадково генеровані коди, які передаються користувачеві через SMS чи будь-який інший канал зв'язку. У такому випадку фактор володіння – телефон користувача (точніше – SIM-картка, прив'язана до певного номера).
3. Роздруківка або scratch card із заздалегідь створеним списком одноразових паролів. Для кожного наступного входу в систему потрібно вводити новий одноразовий пароль із зазначеним номером.

У веб-застосунках такий механізм аутентифікації можу бути реалізованим шляхом розширення forms authentication: після первинної аутентифікації за паролем, створюється сесія користувача, однак у контексті цієї сесії користувач не має доступу до додатку, поки він не виконає додаткову аутентифікацію за одноразовим паролем.

1.2.4 Аутентифікація ключами доступу

Такий метод часто використовується для аутентифікації пристроїв, сервісів або інших програм при зверненні до веб-сервісів. Тут секретом виступають ключі доступу (access key, API key) – це доволі довгі та унікальні рядки, які містять рандомний набір символів, що по суті може замінити комбінацію імені користувача та паролю.

Частіше за все сервер генерує ключі доступу у відповідь на запит користувачів, які далі зберігають ці ключі в клієнтських програмах. При створенні ключа також можна обмежити термін дії та рівень доступу, який отримає клієнтська програма під час аутентифікації за допомогою цього ключа.

Використання ключів дозволяє уникнути передачі пароля користувача стороннім програмам. Ключі мають значно більшу унікальність якщо порівнювати їх з паролями, тому підібрати їх практично неможливо. До того ж, навіть якщо ключ був розкритий – це не призводить до компрометації основного облікового запису користувача – достатньо лише скасувати цей ключ і створити новий.

Не існує єдиного протоколу з технічної точки зору: ключі можуть передаватися в різних частинах запиту HTTP: URL query, request body або HTTP header. Як і у випадку аутентифікації за паролем, найоптимальніший варіант – використання HTTP хедера. Також іноді використовують HTTP-схему Bearer [2] щоб передавати токен у заголовку (Authorization: Bearer [token]). Для

уникнення перехоплення ключів, з'єднання з сервером має бути обов'язково захищене протоколом SSL/TLS.

Крім того, існують складніші схеми аутентифікації ключів для незахищених з'єднань. У цьому випадку ключ зазвичай складається з двох частин: публічної та секретної. Публічна частина використовується для ідентифікації клієнта, а секретна частина дозволяє згенерувати підпис. Наприклад, за аналогією з digest authentication схемою, сервер може надіслати клієнту унікальне значення nonce або timestamp, а клієнт повернути хеш або HMAC цього значення, обчислений з використанням секретної частини ключа. Це дозволяє уникнути передачі всього ключа в оригінальному вигляді та захищає від replay attacks [9].

1.2.5 Аутентифікація токенами

Такий спосіб аутентифікації найчастіше застосовується при побудові розподілених систем SSO, де одна програма (service provider або relying party) делегує функцію аутентифікації користувачів іншому додатку (identity provider або authentication service) [10]. Типовий приклад цього способу – вхід до програми через обліковий запис у соціальних мережах. Тут соціальні мережі є сервісами аутентифікації, а програма довіряє функцію аутентифікації користувачів соціальним мережам.

Реалізація цього способу полягає в тому, що identity provider (IP) надає достовірні відомості про користувача у вигляді токена, а service provider (SP) додаток використовує цей токен для ідентифікації, аутентифікації та авторизації користувача.

На загальному рівні весь процес виглядає наступним чином:

1. Клієнт аутентифікується в identity provider одним із способів, зручним для нього (пароль, ключ доступу, сертифікат, Kerberos або інші).

2. Клієнт просить identity provider надати йому токен для конкретного SP-програми. Identity provider генерує токен та відправляє його клієнту.
3. Клієнт аутентифікується в SP-програмі за допомогою цього токена.

Процес, який ми щойно розглянули, відображає механізм аутентифікації активного клієнта, тобто клієнта, що може виконувати запрограмовану послідовність дій (наприклад, додаток iOS/Android). Браузер же — пасивний клієнт у тому сенсі, що він може відображати сторінки, у відповідь на запит користувачів. У цьому випадку аутентифікація досягається за допомогою автоматичного перенаправлення браузера між веб-додатками identity provider та service provider.

Існує декілька стандартів, що точно визначають протокол взаємодії між клієнтами (активними та пасивними) та IP/SP-додатками і формат підтримуваних токенів. Серед найбільш популярних стандартів - OAuth, OpenID Connect, SAML та WS-Federation. Розглянемо їх докладніше нижче.

Сам токен зазвичай є структурою даних, яка містить інформацію про те хто згенерував токен, хто є одержувачем токена, термін його дії, набір відомостей про користувача (claims). Також, токен додатково підписується щоб запобігти непердбачуваним змін та гарантувати справжність.

При аутентифікації за допомогою токена програма повинна виконати такі перевірки:

1. Токен був виданий довіреним identity provider додатком (перевірка поля issuer).
2. Токен призначається поточному додатку SP (перевірка поля audience).
3. Термін дії токена ще не минув (перевірка поля expiration date).
4. Токен справжній і не змінено (перевірка підпису).

В разі успішної перевірки програма авторизує запит на основі даних про користувача, які містяться в токені.

Формати токенів

Існує кілька поширених форматів токенів для веб-застосунків:

1. Simple Web Token (SWT) — найбільш простий формат, який являє собою набір довільних пар ім'я/значення у форматі кодування HTML form. Стандарт передбачає декілька зарезервованих імен: Issuer, Audience, ExpiresOn та HMACSHA256 [11]. Токен підписується за допомогою симетричного ключа, таким чином, що обидва: identity provider та service provider додатки повинні мати цей ключ для можливості створення/перевірки токена. Приклад SWT можна побачити на рисунку 1.3.

```
Issuer=http://auth.myservice.com&
Audience=http://myservice.com&
ExpiresOn=1435937883&
UserName=John Smith&
UserRole=Admin&
HMACSHA256=KOUQRPSpy64rvT2KnYyQKtFFXUlggnesSpE7ADA4o9w
```

Рисунок 1.3 – приклад Simple Web Token (після декодування)

2. JSON Web Token (JWT). Приклад того, як виглядає токен можна побачити на рисунку 1.4.

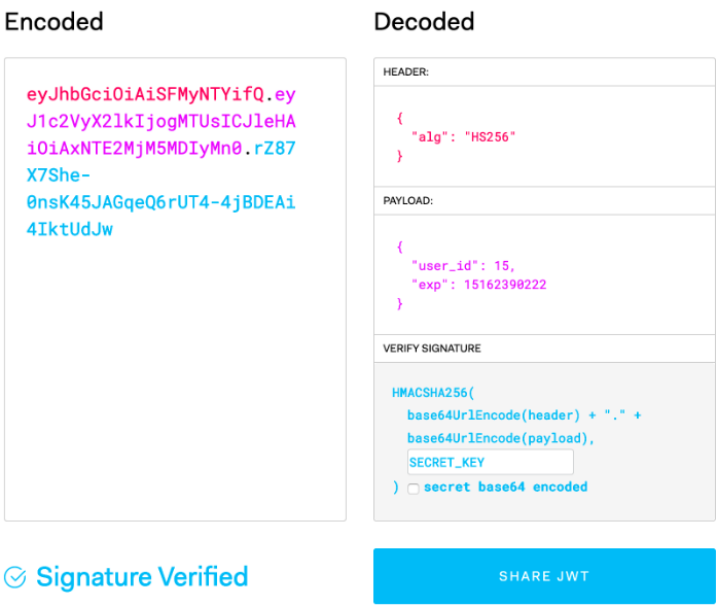


Рисунок 1.4 – Приклад JSON Web Token

Токен містить три блоки, розділених крапками: заголовок, набір полів (claims) та підпис. Перші два блоки представлені в JSON-форматі та додатково закодовані у формат base64. Набір полів містить довільні пари ім'я/значення, JWT передбачає декілька зарезервованих імен (iss, aud, exp та інші) [12]. Підпис може генеруватися за допомогою симетричних алгоритмів шифрування, або асиметричних.

3. Security Assertion Markup Language (SAML) – описує токени в XML-форматі, які включають інформацію про емітент та суб’єкт, а також умови для перевірки достовірності токена і набір додаткових тверджень про користувача. Підпис SAML-токенів здійснюється на основі асиметричної криптографії. Окрім цього, на відміну від інших форматів, SAML-токени містять механізм за допомогою якого можна підтвердити володіння токеном, це в свою чергу, дозволяє запобігти перехопленню токенів через man-in-the-middle атаки при використанні незахищених з'єднань [13].

Стандарти WS-Trust та WS-Federation

WS-Trust і WS-Federation входять до групи стандартів WS-*, які описують SOAP/XML веб-сервіси. Ці стандарти розробляються групою компаній, куди входять Microsoft, IBM, VeriSign та інші. Поряд із SAML, ці стандарти досить складні, використовуються переважно у корпоративних сценаріях.

Стандарт WS-Trust визначає інтерфейс сервісу авторизації, що називається Secure Token Service (STS). Цей сервіс працює за протоколом SOAP та підтримує створення, оновлення та анулювання токенів. При цьому стандарт допускає використання токенів різного формату, проте на практиці переважно використовуються SAML-токени [14].

Стандарт WS-Federation описує механізм взаємодії сервісів між компаніями, зокрема протоколів обміну токенів [15]. У цей же час WS-Federation розширює інтерфейс та функції сервісу STS, що був описаний у стандарті WS-Trust. Також стандарт WS-Federation визначає:

- Формат та способи обміну метаданими про послуги.
- Функцію єдиного виходу із усіх систем (single sign-out).
- Сервіс атрибутів, що надає додаткову інформацію користувача.
- Сервіс псевдонімів дозволяє створювати альтернативні імена користувачів.
- Підтримка пасивних клієнтів (браузерів) за допомогою переадресації.

Можна сказати, що WS-Federation дозволяє вирішити ті самі завдання, що й SAML, проте їх підходи та реалізація певною мірою відрізняються.

Стандарти OAuth та OpenID Connect

На відміну від SAML та WS-Federation, стандарт Open Authorization (OAuth) не описує протокол аутентифікації користувача. Натомість він визначає механізм отримання доступу однієї програми до іншої від імені користувача. Хоча також існують і схеми, які дозволяють виконати аутентифікацію користувача на базі цього стандарту.

Перша версія стандарту розроблялася у 2007 – 2010 роках, а поточна версія 2.0 опублікована у 2012 році. Версія 2.0 значно розширює і водночас спрощує стандарт, але несумісна з версією 1.0. Зараз OAuth 2.0 дуже популярний і використовується всюди для надання делегованого доступу та аутентифікації користувачів на третій стороні.

І так, як працює OAuth? Він описує, як програма яка робить запит на інформацію (client) може отримати доступ до потрібної інформації користувача (resource server) з дозволу самого користувача (resource owner). Схему взаємодії компонентів у стандарті OAuth можна побачити на рисунку 1.5. [16] У загальному вигляді весь процес складається з кількох кроків:

1. Користувач (resource owner) дає дозвіл програмі (client) на доступ до певного ресурсу у вигляді гранту.

2. Програма звертається до сервера авторизації та отримує токен доступу до ресурсу в обмін на свій грант. Під час виклику програма додатково аутентифікується за допомогою ключа доступу, виданого йому під час попередньої реєстрації.
3. Програма використовує цей токен для отримання необхідних даних від сервера ресурсів.

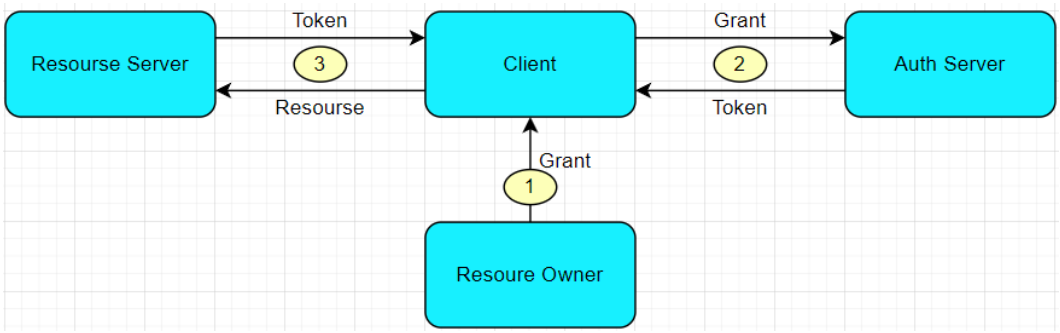


Рисунок 1.5 – Схему взаємодії компонентів у стандарті OAuth

Стандарт має чотири види грантів, які визначають можливі сценарії застосування:

Authorization Code — цей грант користувач може отримати від авторизованого сервера після успішної аутентифікації та підтвердження згоди на надання доступу. Такий спосіб найчастіше використовується у веб-додатках. Процес отримання гранту дуже схожий на механізм аутентифікації пасивних клієнтів SAML і WS-Federation.

Implicit — застосовується, коли програма не має можливості отримати токен від сервера авторизації без загрози його викрадення. У цьому випадку грант – це токен, який було отримано від сервера авторизації, а крок № 2 зникає із наведеного вище сценарію.

Resource Owner Password Credentials – грант є парою username/password користувача. Може застосовуватися, якщо програма є «інтерфейсом» для сервера ресурсів

Client Credentials – у цьому випадку не існує жодного користувача, а програма отримує доступ до усіх ресурсів за допомогою своїх ключів доступу (зникає перший крок).

Стандарт не визначає формат токена: у сценаріях, адресованих стандартом, додатку немає необхідності аналізувати токен тому, що він служить лише для отримання доступу до різних ресурсів. Саме тому ні грант, ні токен не можуть бути використані для аутентифікації користувача. Однак якщо програмі потрібно отримати стовідсотково точну інформацію про користувача, то використовуються такі способи:

1. Найчастіше API сервери ресурсів включають операцію, що надає інформацію про користувача (наприклад, /me в Facebook API). Програма може виконувати цю операцію щоразу після отримання токена для ідентифікації клієнта. Такий метод іноді називають псевдоаутентифікацією.
2. Використовується стандарт OpenID Connect, розроблений як обгортка облікових даних навколо OAuth (опублікований у 2014 році). Згідно із цим стандартом, сервер авторизації надає додатковий identity token на кроці №2. Цей токен у форматі JWT містимитиме набір певних полів (claims) з інформацією про користувача.

Варто також зауважити, що OpenID Connect, який замінив попередні версії стандарту OpenID 1.0 та 2.0, також містить набір необов'язкових додатків для пошуку серверів авторизації, динамічної реєстрації клієнтів та керування сесією користувача.

1.3 Альтернативи паролів

1.3.1 Недоліки паролів

Ми розглянули основні способи аутентифікації та авторизації для сучасних застосунків та веб-програм, але який з них найкращий? Відповідь на це питання не можуть однозначно дати найкращі вчені всього світу. Основна проблема в тому, що для різних ситуацій краще використовувати різні рішення. Але хоч і не існує «найкращого» способу аутентифікації є найпопулярніший і це – аутентифікація по паролю. Це зумовлено як історично (це один з перших методів аутентифікації), так і простотою імплементації цього способу. Та не можна забувати про головний недолік цього способу – незахищеність перед будь-якими видами атак. Тому така популярність паролів над усіма іншими методами аутентифікації є серйозною вадою для безпеки застосунків. До того ж розробники часто нехтують базовими кроками для збільшення рівня безпеки, наприклад, такими як salting і hashing [17].

Ще один недолік паролів – те, що вони засновані лише на елементі пам’яті людей. Це потребує зусиль від користувачів для того щоб їх не забути. Рідко користувачі підходять з повним усвідомленням відповідальності до підбору паролів тому, що складно для кожного сайту придумати окремий пароль та ще й постійно його пам’ятати, тому часто використовуються однакові паролі.

Одне дослідження [18] показало, що більшість користувачів мають багато облікових записів, до яких вони забули свої паролі, а також облікові записи які вони взагалі не пам’ятають що зареєстрували. Інше дослідження [19] використовувало розширення для браузера для спостереження за тисячами звичок користувачів стосовно паролів, знаходячи в середньому 25 облікових записів і 6 унікальних паролів на користувача. Тобто користувачі масово використовують однакові паролі для своїх облікових записів.

Чим же можна замінити безпосереднє введення паролю?

1.3.2 Візуальний пароль

До візуальних паролів можна віднести малювання різних рисунків і подальша перевірка схожості положень точок з початковим патерном. Дехто вважає запам’ятовування рисунків легшим ніж звичайних паролів, тому підносять такий спосіб аутентифікації на рівень вище за звичайні паролі. Але на мою думку набагато ефективнішими є системи які використовують модель «спільного секрету» (shared-secret[20]). Це віддалено нагадує Zero Knowledge Protocol, але насправді модель спільного секрету набагато простіший спосіб і, що варто зазначити, менш безпечний.

Найкраще суть такого методу демонструє аутентифікація на основі методу GrIDsure[21].

Розглянемо приклад. При реєстрації користувачеві надається сітка (наприклад, 5×5) і він обирає шаблон тобто послідовність клітинок. Наприклад, існує 254 можливих візерунка довжиною 4. Кожного разу, коли користувач бажає аутентифікуватися на захищеному ресурсі, йому зображується велика сітка, що містить випадкові символи. Потім користувач вводить символи з клітинок, які відповідають його унікальному візерунку (шаблону). Оскільки асоціація цифр із клітинками є рандомізованою, то рядок введений користувачем, відрізняється кожного разу. Таким чином він розкриває знання про свою таємницю, не вводячи сам секрет. Наглядніше можна побачити на рисунку 1.6.

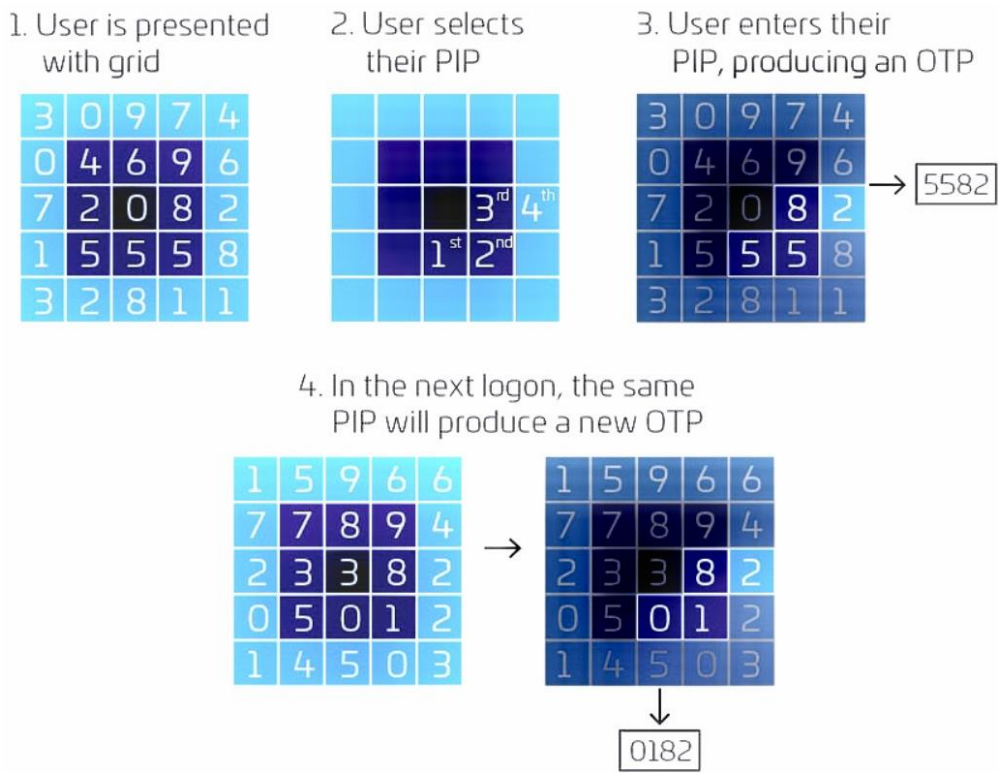


Рисунок 1.6 – Приклад роботи GrIDsure

Ця схема зручніша за паролі адже можна візуально запам’ятати шаблон і це спростить введення кожного паролю. Але у цього способу також є недоліки: на відміну від паролів, які часто можна вводити за допомогою пам’яті м’язів, транскрибування цифр із клітинок сітки вимагає зусиль і уваги і, ймовірно, буде повільніше. Також такий спосіб повність недоступний для незрячих користувачів.

Збоку безпеки є невеликий недолік відносно звичайних паролів. Зловмисникам доволі легко запам’ятати шаблон який вводить жертва у випадку коли є зоровий контакт з клітинками та полем вводу. До того ж обмежана кількість клітинок робить можливим злом шляхом перебору (нагадаю що для поля 5 на 5 кількість 4-значних кодів - всього 254), тому необхідно реалізовувати додатковий захист від такого виду атак.

1.3.3 Менеджер паролів

Останнім часом набирають популярності програми які можуть зберігати всі ваші паролі в одному місці. Для користувачів iOS така функція взагалі вбудована в їх телефони. Перевагою такого способу є те, що користувачу потрібно пам’ятати один пароль (у випадку з телефоном можна використати відбиток пальця або обличчя) для того, щоб отримати доступ до всіх ресурсів відразу. Це дозволяє створити унікальний пароль для кожного ресурсу і таким чином, це збільшує безпеку користувача загалом. Але до вибора менеджера паролів потрібно підійти з відповідальністю.

Існують варіанти з відкритим та закритим кодом. У першому випадку користувач може сам переглянути код і точно дізнатися, які методи використовуються, як саме паролі зберігаються. Це надає впевненості. Але з іншого боку ми маємо вільний доступ зловмисників до вихідного коду програми. Тобто переглянувши код вони можуть знайти вразливість і після цього легко атакувати користувачів.

Якщо обирати рішення з закритим кодом, то треба з розумом обирати компанію, що підтримує програму, щоб вона бережно зберігала ваші паролі, ніколи їх не поширювала, а також вчасно випускала оновлення безпеки.

З точки зору безпеки – рішення неоднозначне. З одного боку ми маємо завжди різні паролі, які окремо дуже важко зламати. Але з іншого боку існує один пароль, який відкриває доступ відразу до всього. На мою думку, з боку користувачів, при використанні менеджера паролів потрібно також використовувати двофакторну аутентифікацію для важливих дій (переказ коштів, зміна паролю) для того, щоб додатково вберегти свій обліковий запис.

1.3.4 Біометричні паролі

Кожному з нас відомо, що у кожної людини унікальні відбитки пальців, форма обличчя, сітчатка ока, свій тембр голосу та інше. Ці особливості можна використовувати для аутентифікації в різних програмах. Останнє десятиліття датчик відбитку пальця поширився по світу і використовується майже у кожному смартфоні. Точність таких датчиків доволі висока і хибні спрацьовування або помилки скоріше виняток ніж правило. Розпізнавання обличчя дуже поширене в Китаї для проведення банківських операцій, останнім часом виробники телефонів почали додавати у свої смартфони датчики для розпізнавання обличчя, хоча технологія ще розвивається.

Головна перевага такого способу аутентифікації – це зручність для користувача. Не потрібно придумувати та пам’ятати паролі, адже у будь-який момент можна показати своє обличчя і легко довести хто ти є насправді (як з людьми в реальному житті). Також швидкість розпізнавання біометричних даних на багато швидша ніж введення паролю.

Із недоліків такий спосіб має важкість фізичного втілення та поширення серед користувачів. Це пов’язано з тим, що він потребує додаткових зовнішніх датчиків для зчитування біометричної інформації. Також існує ймовірність того, що датчик буде неякісним, що збільшить кількість помилок при аутентифікації та понизить рівень безпеки. Також є способи для обману таких датчиків, наприклад, зробити підробку відбитка за допомогою відбитку зі скла та гелевого тіла.

1.3.5 Об’єднаний єдиний вхід

SSO [22] дозволяє виконати аутентифікацію користувача, перенаправивши його на надійний сервер, який зможе підтвердити особистість користувача.

Про те як це працює вже згадувалося вище, на прикладі OAuth.

Найбільшою перевагою такого способу є зручність для користувача, адже треба пам’ятати лише один пароль від ресурсу, що зможе аутентифікувати користувача на інших. Також цей спосіб показує себе більш швидким за звичайні паролі, адже може використовуватись поточна активна сесія з cookies[23], та й вводити один і той самий пароль (нам потрібно аутентифікуватися лише в одному акаунті, який потім вже нас аутентифікує на усіх інших сервісах) завжди швидше завдяки пам’яті м’язів.

Основний недолік – незахищеність від зловмисників. Адже отримавши доступ до одного акаунту шахраї автоматично мають доступ до всіх акаунтів сервісів, що використовують об’єднаний єдиний вхід.

1.4 Порівняння альтернатив

Ми розглянули 4 альтернативи звичайним паролям: візуальний пароль, менеджер паролів, біометричні паролі, об’єднаний єдиний вхід. Порівняймо їх у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика

Властивості	**Назва методу**				
	Звичайні паролі	Візуальні паролі	Менеджер паролів	Біометричні паролі	Об’єднаний єдиний вхід
Легкий для запам’ятовування	3	4	5	5	5
Незалежність від сторонніх пристроїв/програм	5	5	3	1	4
Час користувача для аутентифікації	3	4	5	5	5
Ціна для розробника	5	3	2	1	5
Час спрацювання алгоритму	5	3	4	5	2
Безпека	2	2	2	3	3

Для оцінювання використаємо шкалу від 1 до 5, де 5 – найкраще задовольняє обрану властивість, тобто найкращий варіант для застосування, а 1 – взагалі суперечить властивості, найгірший в цьому плані варіант, 3 – щось середнє. За відправну точку візьмемо звичайні паролі.

Як бачимо паролі є лідером в швидкості спрацювання алгоритму, не потребують сторонніх програм для реалізації та є дуже легкими для імплементації розробниками.

Візуальні паролі в свою чергу є більш зручними для користувача та легшими для запам’ятання, але натомість їх важче реалізувати розробникам.

Менеджер паролів максимально полегшує для користувачів роботу з паролями так як сам «запам’ятовує» і використовує потрібні паролі. Проблема лише в тому, що потрібно використовувати сторонні програми, які в свою чергу передають та зберігають паролі на третій стороні, це збільшує ризики витоку інформації та злому, але завдяки тому що кожен сервіс має свій унікальний пароль – це збільшує захищеність користувача в цілому, тому менеджер паролів отримує такий самий бал безпеки як і звичайні паролі. Також реалізація самого менеджера паролів потребує чималих зусиль з боку розробників.

Біометричні методи аутентифікації дозволяють користувачу взагалі не придумувати пароль, що є дуже зручно. Натомість для такого методу необхідні додаткові датчики для розпізнання біометричних даних, а для розробників реалізація цієї схеми потребує дуже багато зусиль, адже потрібно використовувати machine learning і штучний інтелект. Але коли все налаштовано, то це забезпечує вищий рівень безпеки ніж звичайні паролі, адже біометрію важче підробити.

Метод об’єднаного єдиного входу пропонує найбільше переваг, на мою думку. Це дуже зручно для користувача – потрібно пам’ятати лише один пароль, і зазвичай він зберігається в кукі, тому на його введення взагалі не потрібно витрачати час. Також це легко для розробників, адже існує купа бібліотек, які реалізують цей метод і є захищеними. Майже зберігається

незалежність від сторонніх програм, хоча аутентифікація повністю залежить від довіреного сервісу аутентифікації, ймовірність виходу його з ладу дуже мала (якби траплялися часті падіння сервісу, його б не обрали «довіреним). Але потрібно відносно довго чекати аутентифікації, адже відбувається звернення до стороннього сервісу, що в свою чергу зумовлює затримку. Стосовно безпеки цей спосіб можна назвати відносно безпечним тому, що аутентифікація здійснюється на сторонньому сервісі який є перевіреним і довіреним. Та не забуваймо про те, що потрібно берегти пароль від цього сервісу, щоб він не потрапив до зловмисників.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі були розглянуті різні методи аутентифікації та авторизації. Існує декілька основних методів для того щоб аутентифікати користувача. Аутентифікація за паролем дозволяє підтвердити достовірність даних за допомогою введення секретного набору символів, який повинен бути відомий лише одному користувачу. Такий спосіб є доволі швидким, але не дуже надійним, адже доволі легко піддається злому і потребує більше уваги розробників до безпеки.

Аутентифікація сертифікатами частіше за все здійснюється за допомогою сертифіката X.509. Цей метод надійніший завдяки трьом перевіркам сертифікату під час авторизації, але через труднощі з поширенням та підтримкою сертифікатів він не дуже поширений.

Одноразові паролі використовуються частіше за все для забезпечення 2FA аутентифікації. Тобто як допоміжні до інших методів.

Аутентифікація ключами доступу дозволяє уникнути передачі пароля користувача стороннім програмам. Це забезпечує високий рівень безпеки, але потрібно подбати про те, щоб ключі не буде перехоплено.

Токени використовуються в сучасному світі дуже широко. Було розглянуто найпопулярніші формати токенів: SWT, JWT, SAML.

На прикладі стандарту OAuth було розглянуто можливість аутентифікуватися в різних сервісах за допомогою одного облікового запису, шляхом звернення до стороннього сервісу, який є перевіреним та довіреним.

Було запропоновано декілька альтернатив звичайному введенню паролю, а саме: візуальні паролі (на основі GrIDsure), менеджери паролів, біометрична аутентифікація, об’єднаний єдиний вхід. Порівнявши ці методи з паролями можна зробити висновок, що жоден із цих способів не може забезпечити максимальної захищеності для даних користувача.

Таким чином актуальною є задача розроблення повністю безпечного методу аутентифікації. Це можливо завдяки методу криптографії zero-knowledge proof. Перевагою використання такого методу є те, що він сумісний як зі звичайними паролями, так і з візуальними або біометричними. Завдяки йому ми можемо приховати початкові дані користувачів від сторонніх ресурсів (навіть від самого сервісу де відбувається аутентифікація) і, як результат, рівень безпеки наближається до максимального.

РОЗДІЛ 2.
ОГЛЯД АЛГОРИТМІВ ТА ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Поширені вразливості та помилки реалізації паролів

Аутентифікація по паролю вважається не дуже надійним способом, оскільки пароль часто можна підібрати, а користувачі схильні використовувати прості та однакові паролі в різних системах, або записувати їх на клаптиках паперу. Якщо зловмисник зміг дізнатися пароль, користувач часто про це не дізнається. Крім того, розробники додатків можуть припуститися низки концептуальних помилок, що спрощують злом облікових записів.

Нижче наведено список найбільш поширених помилок при використанні аутентифікації по паролю:

- Веб-програма дозволяє користувачам створювати прості паролі;
- Веб-додаток не захищений від можливості перебору паролів (brute-force attacks[24]);
- Веб-програма сама генерує та розповсюджує паролі користувачам, однак не вимагає зміни пароля після першого входу (тобто початковий пароль десь записаний);
- Веб-програма допускає передачу паролів незахищеним HTTP-з'єднанням або в рядку URL;
- Веб-програма не використовує безпечні хеш-функції для зберігання паролів користувачів;
- Веб-програма не надає користувачам можливість зміни пароля або не нотифікує користувачів про зміну паролів;

- Веб-програма використовує вразливу функцію відновлення пароля, яку можна використовувати для отримання несанкціонованого доступу до інших облікових записів;
- Веб-програма не вимагає повторної аутентифікації користувача для важливих дій: зміна пароля, зміни електронної пошти, тощо;
- Веб-програма створює session tokens таким чином, що вони можуть бути підібрані або передбачені для інших користувачів;
- Веб-додаток допускає передачу session tokens незахищеним HTTP-з'єднанням, або в рядку URL;
- Веб-додаток вразливий для session fixation-атак[25] (тобто не замінює session token при переході анонімної сесії користувача до аутентифікованої);
- Веб-застосунок не встановлює прапори HttpOnly та Secure для browser cookies, що містять session tokens;
- Веб-програма не знищує сесії користувача після короткого періоду неактивності або не надає функції виходу з аутентифікованої сесії.

2.2 Детальний огляд алгоритму який увійшов у основу об’єкту розробки

2.2.1 Існуюча проблема

Поки що більшість систем безпеки використовували лише три типи криптографічних примітивів: шифрування, узгодження ключа та цифрові підписи. Високий рівень безпеки аутентифікації досягається шляхом поєднання деяких з цих примітивів у протоколі.

З часом, кожен метод аутентифікації в Інтернет-системах зламувався певним чином і вимагав нових методів для забезпечення кращих гарантій безпеки та пом’якшення атак, наприклад:

- для паролів із відкритим текстом використовувалися всі види пасивного перехоплення трафіку;
- як тільки сторони почали обмінюватися хешами, почалися атаки на словники та активні атаки на браузер (з використанням хешу для фальшивої аутентифікації рукостискання);
- як тільки сторони почали використовувати захищену аутентифікацію, яка була заснована на ключах доступу, зловмисники почали використовувати атаки man-in-the-middle для деградації шифрів;

У міру того, як історія розвивається, все більше і більше сучасних методів захисту даних виходять з ладу проти складних нападів, адже зловмисники не сидять склавши руки. Чи не було б чудово взагалі уникати передачі паролів?

2.2.2 Zero Knowledge Proof

Zero Knowledge протокол (або Zero Knowledge Proof скорочено ZKP) — це спосіб проведення аутентифікації, коли паролями не обмінюються, що означає, що їх неможливо вкрасти. Це круто, оскільки це робить спілкування настільки безпечним і захищеним, що ніхто інший не зможе дізнатися взагалі ніякі дані..

ZKP дає змогу довести, що ви знаєте якусь таємницю (або багато секретів) комусь на іншому «кінці дроту», фактично не розкриваючи саму інформацію. Сам термін «нульове знання» походить від того того факту, що ніякої інформації про таємницю не розкривається, але друга сторона (так званий «Verifier») може бути упевнена, що перша сторона (так званий «Prover») знає таємницю, про яку йдеться. Коли вам потрібно доводити, що ви знаєте секрет, не розповідаючи його? Тоді коли ви не довіряєте іншій людині, але вам

потрібно переконати її, що ця інформація вам відома (наприклад, при аутентифікації) [26].

2.2.3 Основні властивості ZKP

1. Повнота: якщо твердження істинне, чесний Verifier (тобто той, хто належним чином дотримується протоколу) буде переконаний у цьому факті чесним Prover`ом.

2. Обґрунтованість: якщо твердження є хибним, жоден фальшивий Prover не зможе переконати чесного Verifier, що воно істинне, за винятком дуже маленької ймовірності.

3. Нульове знання: якщо твердження істинне, жоден верифікатор не дізнається нічого, крім цього факту. Це формалізується, показуючи, що кожен шахрайський верифікатор має певний симулятор, який, враховуючи лише твердження, що підлягає доказу (і не має доступу до доказувача), може створити розшифровку, яка «виглядає як» взаємодія між чесним доповідачем і шахрайським верифікатором.

Перші дві властивості становлять основу аутентифікації і можуть бути досягнуті шляхом правильного поєднання інших криптографічних примітивів. Остання властивість гарантує, що потік протоколу та результат мають значення лише для чесних учасників протоколу і нікого іншого. Іншими словами, такий протокол не передає жодної допоміжної інформації стороннім спостерігачам або навіть самому перевіряючому.

2.2.4 Socialist Millionaire’s Problem (SMP Protocol)

Завдання мільйонерів-соціалістів (англійською Socialist Millionaires Problem, SMP) — криптографічна задача, в якій два мільйонери хочуть з’ясувати, чи рівні їхні статки, не розголошуючи точні суми. Вирішення цієї

задачі використовується як криптографічний протокол, який дозволяє двом сторонам перевірити чи можна довіряти віддаленому учаснику за допомогою загального секрету, не передаючи свій секрет іншій стороні.

Переваги SMP:

- новий механізм аутентифікації пароля HTTP (гарантує, що пароль або хеші файлу ніколи не надсилаються іншій стороні);
- новий спосіб підтвердження встановленого секретного ключа після узгодження ключа;
- механізм взаємної аутентифікації;
- механізм дистанційної атестації;

Була обрана проблема соціаліста-мільйонера (SMP) за її простоту та хороші результати в протоколі OTR, де вона використовується для аутентифікації віддалених сторін.

2.2.5 Off-the-Record Protocol

OTR – це криптографічний протокол, який забезпечує шифрування для розмов миттєвих повідомлень. OTR використовує комбінацію алгоритму симетричного ключа AES з довжиною ключа 128 біт, обміну ключами Діффі-Хеллмана з розміром групи 1536 біт і хеш-функцією SHA-1. На додаток до аутентифікації та шифрування, OTR забезпечує пряму таємницю та піддатливе шифрування. [27]

Основна задача протоколу полягала в тому, що учасникам розмови можна було б аутентифікуватися, зберігаючи при цьому конфіденційність розмов, на кшталт приватної розмови в реальному житті.

Протокол OTR був розроблений криптографами Іаном Голдбергом і Микитою Борисовим і випущений 26 жовтня 2004 року. Вони надають клієнтську бібліотеку для полегшення підтримки розробників клієнта миттєвих

повідомлень, які хочуть впровадити протокол. Існують плагін, який дозволяє використовувати OTR через будь-який протокол миттєвого обміну повідомленнями, який підтримується Pidgin або Kopete, пропонуючи функцію автоматичного визначення, яка запускає сеанс OTR з друзями, у яких його ввімкнено, не заважаючи звичайним незашифрованим розмовам.

Через обмеження протоколу OTR не підтримує багатокористувацький груповий чат з 2009 року, але він може бути запроваджений у майбутньому. Починаючи з третьої версії протоколу, під час аутентифікованого обміну ключами отримується додатковий симетричний ключ, який можна використовувати для безпечного зв'язку (наприклад, передачі зашифрованих файлів) через інший канал. Підтримка зашифрованого аудіо чи відео не планується (для цього існують SRTP із ZRTP).

2.3 Удосконалення SMP

2.3.1 Безпека SMP

Щоб побудувати схему аутентифікації навколо SMP, я використав протокол OTR як довідник. Однак OTR використовує 1536-бітну групову алгебру як основу своїх обчислень. Оскільки SMP за своїми операціями дуже схожий на обмін ключами Діффі-Хеллмана, він майже такий самий в аспектах безпеки. Нещодавнє розкриття атаки LogJam на протокол TLS [28], спрямованої на обмін ключами Діффі-Хеллмана, є доказом того, що 512-бітові групи можуть бути зламані за допомогою field sieve алгоритму. У цій же статті наводяться оцінки, що 1024-бітні групи вже можуть бути вразливими зараз або можуть стати вразливими у найближчому майбутньому.

Виходячи з вищесказаного, я вважаю, що 1536-бітна версія може не забезпечити належного рівня безпеки. Крім того, я вважаю що краще мати

більш фундаментальне рішення для забезпечення безпеки, ніж просто збільшення розміру бітів поля, що створює додатковий тиск як на швидкість алгоритму, так і на вимоги до пам’яті. Тому я вирішив посилити SMP, шляхом повторної реалізації на основі криптографії еліптичної кривої.

Безпека як SMP, так і DH здебільшого базується на задачі дискретного логарифма (DLP): пошук експоненти, що використовується для обчислення потужності деякої бази в циклічній групі, не має алгоритмів із поліноміальною складністю часу. Проте в ECC ця проблема вирішена ще більш безпечно, оскільки, на відміну від алгебри простих полів, де такі алгоритми існують із субекспоненціальною часовою складністю, для ECC існують лише алгоритми експоненціального часу.

Щоб зробити SMP (а також більшість алгоритмів на основі DLP) сумісними з ECC, у потоці алгоритму потрібно змінити наступне:

- вибрати цільові параметри домену ECC;
- запровадити детерміновані відображення секретних і випадкових протокольних значень у великі цілі числа, більші за 0 і менші за порядок базової точки поля ECC;
- замінити всі операції множення груп простих полів на операції додавання точки ECC;
- замінити всі операції ділення групи простих полів на операції віднімання точки ECC;
- замінити всі зведення до степеню на множення точки ECC на великі цілі числа.

Розглянувши всі можливі варіанти хороших основ ECC, я вирішив використовувати ed25519, тому що:

- нам потрібно щось краще захищене від бічних атак, ніж звичайні вигини NIST;
- швидкість і продуктивність;

- знаходиться у відкритому доступі, і перевірено експертами (Daniel J. Bernstein);
- має мало слабких приватних ключів або зовсім не має слабких приватних ключів, тому вищезгадані заміни можуть бути легко розроблені (простою хеш-функцією зі скороченим виводом).

Однак оригінальній реалізації ed25519 бракувало деяких базових примітивів ECC, тому її довелося розширити для SMP.

2.3.2 ed25519

Ed25519 – це одна з реалізацій еліптичних кривих, створена спеціально для алгоритму цифрового підпису ECDSA.

Ed25519 був представлений OpenSSH у версії 6.5. Це реалізація EdDSA з використанням кривої Twisted Edwards (Крива Едвардса). Він використовує криптографію з еліптичними кривими, яка забезпечує кращу безпеку та більш високу продуктивність у порівнянні з DSA або ECDSA.

В алгебраїчній геометрії скручені криві Едвардса є плоскими моделями еліптичних кривих. Еліптичні криві є важливими в криптографії з відкритим ключем, а криві Едвардса є основою схеми електронного підпису під назвою EdDSA, яка забезпечує високу продуктивність, уникаючи проблем із безпекою, які виникали в інших схемах цифрового підпису.

Сьогодні RSA є найбільш широко використовуваним алгоритмом відкритого ключа для ключів SSH. Але в порівнянні з ed25519 він повільніший і вважається небезпечним, якщо він генерується з ключем довжиною менше 2048 біт.

Відкритий ключ ed25519 компактний. Він містить лише 68 символів у порівнянні з RSA 3072, який має 544 символи. Генерування ключа майже таке само швидке, як процес підписання. Також швидко виконується перевірка

підпису пакета з ed25519. Він створений для захисту від зіткнень. Зіткнення хеш-функції не зламає систему.

2.3.3 Розширення ed25519

Незважаючи на те, що ed25519 є хорошим кандидатом для нашої роботи, це алгоритм цифрового підпису. У SMP нам потрібна інша комбінація примітивних операцій ECC і, як наслідок, інша обробка параметрів алгоритму.

Крім того, базовий ed25519 може похвалитися операціями з постійним часом, захищаючи секрети користувачів від атак з боку каналу (переважно часу). Оскільки код був написаний з урахуванням високої оптимізації, ed25519 мав обмежену підмножину операцій для початку:

• Q = d * G – базове множення базової точки ECC;

• R = s * G + r * Q – сума двох точок, де одна кратна базовій точці.

З іншого боку, SMP вимагає T = u * P - простого множення точки, але де P - деяка випадкова (заздалегідь не відома) точка. Перші два очевидні варіанти не є правильними:

1) Щоб реалізувати цю операцію на основі наявних функцій ed25519, очевидним способом було повторне використання другої функції (суми двох точок), передаючи n - порядок базової точки ECC - як параметр s. Фактично, оскільки n * G = O (нульова точка ECC), то

n * G + r * Q = O + r * Q = r * Q

Однак перша операція ed25519 — це постійний час, а друга — ні. Для використання цифрового підпису це не створює серйозної загрози – друга операція використовується лише для перевірки підписів відкритим ключем (не використовується секретна інформація). У нашому ж випадку наша адаптована функція може бути використана для множення якогось секретного випадкового значення на довільну точку.

2) Підхід повторного використання, який було використано під час першої операції ed25519 (скалярне множення, замінене додаванням точок із таблиці попередньо обчислених точок), також мало сенсу, оскільки його властивості постійного часу засновані на великій попередньо обчисленій таблиці (~30 KB) спеціально для G (який є постійним і відомим заздалегідь). Ми не можемо дозволити собі робити ті самі обчислення в процесі для довільної точки через міркування продуктивності: оскільки ми можемо отримати випадкову точку від «співрозмовника», нам доведеться щоразу перераховувати таблицю.

Замість того, щоб намагатися зробити операцію постійною, я вирішив використати інший підхід. Припустимо, нам потрібно обчислити R = d * Q:

- генерувати випадкове ціле число rnd;
- обчислити R = rnd * G + d * Q – змінний час, де зловмисник може отримати лише деяку інформацію про rnd + d, але оскільки він не знає ні d, ні rnd, це схоже на те, ніби d було зашифровано rnd;
- обчислити P = rnd * G – операція з постійним часом, доступна в оригінальному ed25519, тому зловмисник нічого не отримує;
- обчислити R = R - P = rnd * G + d * Q - rnd * G = d * Q – операція з постійним часом (просте віднімання точки, не передбачає обробки секретної інформації).

2.4 Протокол

2.4.1 Робота алгоритму

Перейдемо до роботи самого алгоритму.

Припустимо, у нас є дві сторони з секретами x і y відповідно, і вони хочуть знати, чи є x == y (чи секрети однакові). Вони використовують криву ed25519 з G як базовою точкою. Аліса запускає протокол:

Аліса:

- Обирає два випадкові числа: a2 і a3;
- Обчислює G2a = a2 * G, а також G3a = a3 * G;
- Відправляє G2a та G3a Бобу;

Боб:

- Обирає два випадкові числа: b2 і b3;
- Обчислює G2b = b2 * G, а також G3b = b3 * G;
- Обчислює G2 = b2 * G2a, а також G3 = b3 * G3a;
- Обирає випадкове число r;
- Обчислює Pb = r * G3, а також Qb = r * G + y * G2;
- Відправляє G2b, G3b, Pb та Qb Алісі;

Аліса:

- Обчислює G2 = a2 * G2b, а також G3 = a3 * G3b;
- Обирає випадкове число s;
- Обчислює Pa = s * G3, а також Qa = s * G + x * G2;
- Обчислює Ra = a3 * (Qa - Qb);
- Відправляє Pa, Qa та Ra Бобу;

Боб:

- Обчислює Rb = b3 * (Qa - Qb);
- Обчислює Rab = b3 * Ra;

- Перевіряє чи рівні Rab == Pa – Pb;
- Відправляє Rb Алісі;

Аліса:

- Обчислює Rab = a3 * Rb;
- Перевіряє чи рівні Rab == Pa – Pb.

Якщо перевірка Rab == Pa - Pb успішна, то кожна сторона переконана, що x == y. Оскільки Rab = (Pa - Pb) + (a3 * b3 * (x - y)) * G2, тоді і тільки тоді, коли x == y, Rab = (Pa - Pb) + 0 * G2 = Pa - Pb. Якщо x≠ y, то a3 * b3 * (x - y) * G2 — це випадкова точка ECC, невідома жодній стороні, тому інформація не розкривається.

Наглядно як працює алгоритм можна побачити у додатку 1.

2.4.2 Специфіка ECC

Усі числа, які використовуються в обчисленнях ECC для найкращої безпеки, мають бути меншими за порядок базової точки ECC. Для ed25519 відповідним числом є будь-який 32-байтовий масив, але з трьома останніми бітами, очищеними в першому байті, очищеним першим і другим бітом, встановленим в останньому байті. Це стосується будь-яких використаних випадкових чисел, а також самих секретів.

2.4.3 Хешування

Хешування — це процес перетворення будь-якого заданого ключа або рядка символів в інше значення. Зазвичай це представлено коротшим значенням або ключем фіксованої довжини, що представляє та полегшує пошук або використання вихідного рядка.

Найпопулярнішим використанням хешування є реалізація хеш-таблиць. Хеш-таблиця зберігає пари ключів і значень у списку, доступному через її індекс. Оскільки пари ключів і значень необмежені, хеш-функція відобразить ключі до розміру таблиці. Тоді хеш-значення стає індексом для певного елемента.

Хеш-функція генерує нові значення відповідно до математичного алгоритму хешування, відомого як хеш-значення або просто хеш. Щоб запобігти перетворенню хешу назад у вихідний ключ, хороший хеш завжди використовує односторонній алгоритм хешування.

Згідно нашого алгоритму секретна інформація не використовується беспосередньо в розрахунках SMP. Щоб дозволити порівняння рядка довільної довжини, робиться хешування всієї інформації та після цього порівнюються хеші. У своїй програмі я використовував алгоритми хешування SHA-256 та blake2.

2.4.4 SHA-256

SHA-256 – криптографічна хеш-функція, розроблена Агенством національної безпеки США. SHA розшифровується як Secure Hash Algorithm. Сама по собі хеш-функція – це математична операція, що виконується з цифровими даними. Порівнюючи обчислений "хеш" (результат виконання алгоритму) з відомим і очікуваним хеш-значенням, можна визначити цілісність даних. Односторонній хеш може бути згенерований з будь-якого фрагмента даних, але дані не можуть бути згенеровані з хешу.

Алгоритм SHA-256 заснований на методі побудови Меркля-Дамгарда, згідно з яким початковий індекс ділиться на блоки відразу після внесення змін, а ті, в свою чергу, на 16 слів.

Основні характеристики алгоритму:

- індикатор розміру блоку: 64 байти;
- максимально можлива довжина повідомлення: 33 байти;
- характеристика розміру дайджесту повідомлення: 32 байти;
- стандартний розмір слова: 4 байти;
- параметр довжини внутрішньої позиції: 32 байти;
- кількість ітерацій в одному циклі: 64;
- швидкість, досягнута за протоколом: приблизно 400 Мб/с.

2.4.5 BLAKE2

BLAKE2 — це криптографічна хеш-функція швидша, ніж MD5, SHA-1, SHA-2 і SHA-3, але принаймні настільки ж безпечна, як і останній стандарт SHA-3. BLAKE2 зазначено в RFC 7693. [29]

BLAKE2 представлений у двох варіантах:

- BLAKE2b (або просто BLAKE2) оптимізовано для 64-розрядних платформ, включаючи ARM з підтримкою NEON, він створює дайджести будь-якого розміру від 1 до 64 байт;
- BLAKE2s оптимізовано для 8-32-розрядних платформ і він створює дайджести будь-якого розміру від 1 до 32 байт.

BLAKE2 включає 4-сторонній паралельний BLAKE2bp і 8-сторонній паралельний BLAKE2sp, розроблений для підвищення продуктивності на багатоядерних або SIMD-процесорах. BLAKE2 пропонує алгоритми, налаштовані на ваші конкретні вимоги, такі як хешування з ключем (тобто MAC або PRF), хешування за допомогою salting, оновлюване або інкрементальне хешування дерева або будь-яка їх комбінація. Ці версії вказані в документі BLAKE2.

BLAKE2 також включає варіанти BLAKE2x, які можуть створювати дайджести довільної довжини.

BLAKE2 працює на 64-розрядних процесорах: на Intel Core i5-6600 (мікроархітектура Skylake, 3310 МГц) BLAKE2b може обробляти 1 гібібайт в секунду або зі швидкістю 3,08 циклів на байт.

На рисунку 2.1 показано, як BLAKE2 перевершує MD5, SHA-1, SHA-2 і SHA-3 на процесорі Intel Skylake (швидкості для хешування з використанням одного ядра; за допомогою кількох ядер BLAKE2 може бути навіть швидшим):

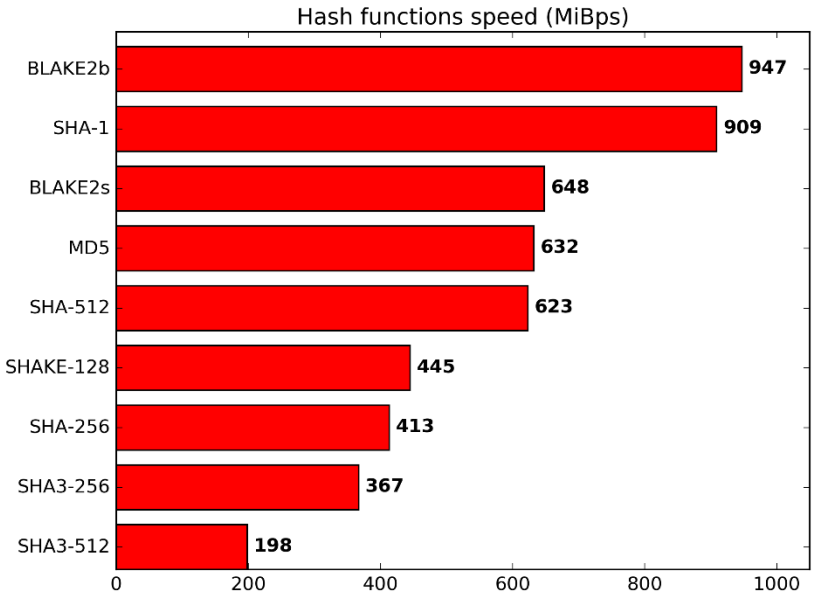


Рисунок 2.1 – Порівняння швидкості хеш функцій

В теорії BLAKE2 повинен показати приріст в швидкості на 50%, у порівнянні з SHA-256. Пізніше ми перевіримо це на практиці.

2.5 Вибір мови програмування

Розроблений алгоритм можна реалізувати майже у кожній мові програмування. Я обрав мову програмування Rust для виконання цього завдання. Rust позбавлений багатьох больових точок, які є в сучасних мовах програмування. Завдяки цьому є цілий список переваг Расту перед іншими мовами, хоче не можна не відмітити і деякі недоліки.

Переваги:

- без проблем працює на Unix та Mac;

- є абстракції, які значно полегшують регулювання пам'яті вручну;
- надійна система взаємодії з пам'яттю, яка виключає помилки сегментації;
- автоматичне представлення способів виправити помилки під час компіляції;
- компілятор містить збирач та менеджер пакетів, інструменти для тестування та створення документації;
- у безпечному коді немає можливості застосовувати покажчики (тільки посилання на 100% реальні об'єкти);
- доступний опис помилок у шаблонах.

Недоліки:

- Компілятор занадто жорстко фіксує виклики до пам'яті;
- Немає типових для ООП-стилю успадкування та класів.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі було розглянуто найбільш поширені помилки при реалізації аутентифікації звичайними паролями. Також було детально розглянуто принцип роботи протоколу доказу нульового знання (Zero Knowladge Proof), метою якого є доведення того, що значення однакові, без розголошення значень іншій стороні. Було розглянуто основні властивості ZKP, такі як повнота, обґрунтованість, нульове знання. Розглянуто принцип роботи OTR протоколу.

Для вирішення задачі було обрано SMP Protocol, який дозволяє доволі ефективно вирішувати задачку про мільйонерів та показав себе як доволі швидке рішення. Але SMP не ідеальний, тому було запропоновано ряд поліпшень для удосконалення безпеки протоколу, одним з них є введення еліптичної кривої ed25519, яку також потрібно було дещо вдосконалити.

Потім було розглянуто алгоритм роботи кінцевої програми покроково, а також наведено структурну схему системи у додатку 1. Далі було описано всі особливості використаних еліптиних кривих, а також методів хешування. Також було надано короткий опис хеш функцій SHA-256 та BLAKE2.

Було розглянуто мову програмування котра буде використовуватися для реалізації розробленого алгоритму. Це мова програмування – Rust.

РОЗДІЛ 3.
ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

Відповідно до досліджених матеріалів та розробленого алгоритму, ми можемо перейти до фази розробки програмного забезпечення. Система буде реалізована у вигляді бібліотеки з відповідною реалізацією алгоритму, а також додатково розроблених тестів для перевірки швидкодії програми, і для порівняння результатів роботи різних хеш-функцій.

3.1 Розробка програмних компонентів алгоритму

Далі буде описано основні частини коду для реалізації алгоритму основної програми. Повний код програми прикріплено у додатку 4.

3.1.1 Реалізація стану

Так як у нашому алгоритмі аутентифікація відбувається в декілька кроків, нам необхідно відстежувати стан протягом всього процесу аутентифікації. Це допоможе нам відслідковувати на якому саме кроці ми зараз знаходимось. Для цього я використав enum або іншими словами - перерахування. В загальному enum представляє собою набір констант або станів, в яких може бути змінна перерахування, і залежно від поточного стану, ми можемо виконати ті чи інші дії.

У нашому випадку ми маємо 6 станів, а саме: New, ClientStep3, ServerStep4, ClientStep5, Complete, Poisoned. На кожному стані зберігається різна інформація, яка залежить від етапу алгоритма на якому ми знаходимося зараз. Давайте розглянемо детальніше, що саме зберігає кожен стан, які дані він містить і для чого вони використовуються.

Стан New зберігає дані для порівняння – hasher, а також має прапорець (булеве значення) poisoned. Цей прапорець спільний майже для всіх станів, і він служить нам для того щоб гарантувати доведення алгоритму до кінця, навіть коли аутентифікація буде невдалою. Тобто якщо у нас з'являється помилка і аутентифікація неуспішна, то ми змінюємо цей прапорець на true і продовжуємо виконання алгоритму далі. Це забезпечує нам сталий час виконання алгоритму, що в свою чергу підвищує рівень захисту від timing attack.

Стан ClientStep3 зберігає дані, які потрібні для порівняння – hasher, два скалярні значення a2, a3 (числа по 256 біт) і прапорець poisoned.

Стан ServerStep4 зберігає одне скалярне значення b3, прапорець poisoned, а також 5 точок на еліптичній кривій: G3a, G2, G3, Pb, Qb. Ці точки є результатами різних операцій минулого кроку і в подальшому будуть відправлені на на ступний крок.

Стан ClientStep5 в собі зберігає один скаляр a3, прапорець poisoned і три точки: G3b, Pa_Pb, Qa_Qb, які отримані з минулого кроку, а також використовуються для подальших обчислень.

Complete стан має лише одне булеве значення result. Воно використовується для того щоб отримати кінцевий результат, а саме був наш алгоритм успішним чи ні.

У стан Poisoned система переходить коли аутентифікація була неуспішною.

Реалізацію стану можна побачити на рисунку 3.1.

```
enum State<D: Digest> {
    New {
        hasher: D,
        poisoned: bool,
    },
    ClientStep3 {
        hasher: D,
        a2: Scalar,
        a3: Scalar,
        poisoned: bool,
    },
    ServerStep4 {
        b3: Scalar,
        G3a: EdwardsPoint,
        G2: EdwardsPoint,
        G3: EdwardsPoint,
        Pb: EdwardsPoint,
        Qb: EdwardsPoint,
        poisoned: bool,
    },
    ClientStep5 {
        a3: Scalar,
        G3b: EdwardsPoint,
        Pa_Pb: EdwardsPoint,
        Qa_Qb: EdwardsPoint,
        poisoned: bool,
    },
    Complete {
        result: bool,
    },
    Poisoned,
}
```

Рисунок 3.1 – Реалізація стану

Реалізований стан дозволяє нам точно контролювати поточний етап алгоритму, а також дані які використовуються на кожному кроці.

3.1.2 Відображення стану

Для того щоб ми могли бачити який стан у програми саме зараз нам потрібно написати функцію виводу на екран. Реалізація функції приведена на рисунку 3.2.

```
impl<D: Digest> fmt::Debug for State<D> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            State::New { .. } => f.debug_struct("New").finish(),
            State::ClientStep3 { .. } => f.debug_struct("ClientStep3").finish(),
            State::ServerStep4 { .. } => f.debug_struct("ServerStep4").finish(),
            State::ClientStep5 { .. } => f.debug_struct("ClientStep5").finish(),
            State::Complete { .. } => f.debug_struct("Complete").finish(),
            State::Poisoned => f.debug_struct("Poisoned").finish(),
        }
    }
}
```

Рисунок 3.2 – Реалізація функції відображення стану

Дана функція, в залежності від поточного стану програми виводить відповідну назву на екран. Варто зазначити, що на екран не виводяться усі поля стану, лише його назва.

3.1.3 Authenticator

Authenticator використовується для того, щоб підготувати дані для порівняння, почати порівняння даних, прейти на наступний крок (продовжити порівняння), перевірки чи порівняння завершено, а також перевірки результату порівняння. За кожен етап відповідає своя функцію.

Функція створення

Для створення Authenticator використовується проста функція new, реалізація якої приведене на рисунку 3.3.

```
pub fn new() -> Self {
    Self {
        state: State::New {
            hasher: D::new(),
            poisoned: false,
        },
    }
}
```

Рисунок 3.3 – Реалізація функції new

Ця функція змінює стан таким чином, що поле hasher тепер рівне самій функції, а прапорець poisoned має значенно false. Це дозволяє почати процес нового порівняння.

Функція push_secret

За допомогою Authenticator можна порівнювати секрети будь-якого розміру. Це реалізовано завдяки push_secret. Ця функція додає щоразу нові дані, якщо його було викликано декілька разів. Тобто ця функція дозволяє підготувати дані для перевірки. Реалізацію можна побачити на рисунку 3.4.

```
pub fn push_secret(&mut self, secret: &[u8]) -> anyhow::Result<()> {
    match &mut self.state {
        State::New { hasher, .. } => {
            hasher.update(secret);
            Ok(())
        }
        _ => anyhow::bail!("comparison is already in progress"),
    }
}
```

Рисунок 3.4 – Реалізація функції push_secret

Ця функція також відслідковує чи вже почався процес порівняння, адже додавати нові дані можливо лише до початку порівняння. У разі якщо процес порівняння вже почато – користувач отримує помилку і нові дані додано не буде.

Функція begin

В першу чергу функція перевіряє стан, і якщо він встановлений New, то вона алокує буфер великого розміру, визиває функцію step1_client_to_server і повертає ОК. Наглядно можемо побачити на рисунку 3.5.

```
pub fn begin(&mut self) -> anyhow::Result<Vec<u8>> {
    match &self.state {
        State::New { .. } => {
            let mut output = vec![0; STEP1_SIZE];
            let len = step1_client_to_server(&mut self.state, &mut output)?;
            debug_assert_eq!(output.len(), len);
            Ok(output)
        }
        _ => anyhow::bail!("inconsistent state"),
    }
}
```

Рисунок 3.5 – Реалізація функції begin

Також, якщо стан не дорівнює New, як заплановано функцією, то це призводить до того, що на екран буде виведено помилку про неправильний стан «inconsistent state».

Функція proceed

Ця функція працює як диспетчер – передає управління потрібній частині коду. Її реалізація на рисунку 3.6. Основна ідея така ж як і у begin, ми перевіряємо стан і в залежності від стану обираємо яку дію потрібно виконати:

- Якщо стан New, то визиваємо функцію step2_server_to_client;
- Якщо стан ClientStep3, то визиваємо функцію step3_client_to_server;
- Якщо стан ServerStep4, то визиваємо функцію step4_server_to_client;
- Якщо стан ClientStep5, то визиваємо функцію step5_client_is_done.

```rust
pub fn proceed(&mut self, message: &[u8]) -> anyhow::Result<Option<Vec<u8>>> {
    match &self.state {
        State::New { .. } => {
            let mut output = vec![0; STEP2_SIZE];
            let len = step2_server_to_client(&mut self.state, message, &mut output)?;
            debug_assert_eq!(output.len(), len);
            Ok(Some(output))
        }
        State::ClientStep3 { .. } => {
            let mut output = vec![0; STEP3_SIZE];
            let len = step3_client_to_server(&mut self.state, message, &mut output)?;
            debug_assert_eq!(output.len(), len);
            Ok(Some(output))
        }
        State::ServerStep4 { .. } => {
            let mut output = vec![0; STEP4_SIZE];
            let len = step4_server_to_client(&mut self.state, message, &mut output)?;
            debug_assert_eq!(output.len(), len);
            Ok(Some(output))
        }
        State::ClientStep5 { .. } => {
            step5_client_is_done(&mut self.state, message)?;
            Ok(None)
        }
        _ => anyhow::bail!("inconsistent state"),
    }
}
```

Рисунок 3.6 – Реалізація функції proceed

Якщо функція не виявить жоден відомий їй стан, то користувач отримає помилку про неправильний стан «inconsistent state».

is_complete

Функція для перевірки завершеності порівняння. ЇЇ реалізаація зображена на рисунку 3.7.

```rust
pub fn is_complete(&self) -> bool {
    matches!(&self.state, State::Complete { .. } | State::Poisoned)
}
```

Рисунок 3.7 – Реалізація функції is_complete

Функція перевіряє чи переведено стан у Complete або Poisoned, і повертає true або false.

Функція result

Повертає результат: порівняння було успішним чи ні. Її реалізація на рисунку 3.8.

```rust
pub fn result(&self) -> anyhow::Result<bool> {
    match &self.state {
        State::Complete { result } => Ok(*result),
        _ => anyhow::bail!("inconsistent state"),
    }
}
```

Рисунок 3.8 – Реалізація result

Це булева функція тому повертає вона одне з двох значень: або true, або false.

3.1.5 Допоміжні функції

У коді є чимало невеликих функцій, які використовуються як допоміжні і дозволяють перетворити одні дані в інші, зробити певні маніпуляції з точками тощо. Розглянемо основні з них.

clamp_scalar

Функція невелика, побачити її реалізацію можна на рисунку 3.9.

```rust
fn clamp_scalar(scalar: &mut [u8; SCALAR_SIZE]) {
    scalar[0x00] &= 0b_1111_1000;
    scalar[0x1F] &= 0b_0111_1111;
    scalar[0x1F] |= 0b_0100_0000;
}
```

Рисунок 3.9 – Реалізація функції clamp_scalar

Функція робить «затискання» точок кривої. Це необхідно щоб підготувати точки до подальших обчислеть.

random_scalar

Функція обчислює випадковий скаляр. Побачити як вона виглядає можна на рисунку 3.10.

```rust
fn random_scalar() -> Result<Scalar, Error> {
    let mut scalar = [0; SCALAR_SIZE];
    soter::random(&mut scalar)?;
    clamp_scalar(&mut scalar);
    Ok(Scalar::from_bits(scalar))
}
```

Рисунок 3.10 – Реалізація функції random_scalar

Як бачимо навіть випадкові скаляри проходять процедуру «закріплення» функцією clamp_scalar, тобто всі дані які обчислюються є попередньо закріпленими.

reduce_*_point_hash_into_scalar

На рисунку 3.11 можна побачити функцію reduce_1_point_hash_into_scalar.

```rust
fn reduce_1_point_hash_into_scalar(W: &EdwardsPoint, i: &[u8]) -> anyhow::Result<Scalar> {
    let mut hasher = Blake2b512::new();
    hasher.update(W.compress().as_bytes());
    hasher.update(i);
    let bytes = hasher.finalize();
    Ok(Scalar::from_bytes_mod_order_wide(bytes.as_ref()))
}
```

Рисунок 3.11 – Реалізація функції reduce_1_point_hash_into_scalar

Трохи нижче, на рисунку 3.12 приведено приклад реалізації функції reduce_2_point_hash_into_scalar.

```rust
fn reduce_2_point_hash_into_scalar(
    W1: &EdwardsPoint,
    W2: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<Scalar> {
    let mut hasher = Blake2b512::new();
    hasher.update(W1.compress().as_bytes());
    hasher.update(W2.compress().as_bytes());
    hasher.update(i);
    let bytes = hasher.finalize();
    Ok(Scalar::from_bytes_mod_order_wide(bytes.as_ref()))
}
```

Рисунок 3.12 – Реалізація функції reduce_2_point_hash_into_scalar

Ці дві функції дуже схожі за своїм призначенням – вони дозволяють нам перетворити точку у скалярне значення. Відрізняються вони лише тим, що перша функція робить це для однієї точки, а друга – для двох точок.

sign_scalar

Таких функцій ми маємо також дві sign_scalar1 і sign_scalar2. З деталями реалізації можна ознайомитись на рисунку 3.13 та рисунку 3.14 відповідно.

```
fn sign_scalar1(s: &Scalar, i: &[u8]) -> anyhow::Result<Signature1> {
    let r = &random_scalar()?;
    let W = r * G;
    let c = reduce_1_point_hash_into_scalar(&W, i)?;
    let d = (c * s + r).reduce();
    Ok(Signature1 { c, d })
}
```

Рисунок 3.13 – Реалізація функції sign_scalar1

sign_scalar1 підписує скаляр за допомогою арифметичних операцій, а також використання функції reduce_1_point_hash_into_scalar.

```
fn sign_scalar2(
    s1: &Scalar,
    s2: &Scalar,
    B1: &EdwardsPoint,
    B2: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<Signature2> {
    let r1 = &random_scalar()?;
    let r2 = &random_scalar()?;
    let W1 = r1 * B2;
    let W2 = r2 * B1 + r1 * G;
    let c = reduce_2_point_hash_into_scalar(&W1, &W2, i)?;
    let d1 = (c * s1 + r1).reduce();
    let d2 = (c * s2 + r2).reduce();
    Ok(Signature2 { c, d1, d2 })
}
```

Рисунок 3.14 – Реалізація функції sign_scalar2

sign_scalar2 дозволяє підписати нам 2 скаляри відразу. Для цього також використовуються алгербаїчні операції, а також функція reduce_2_point_hash_into_scalar.

verify_scalar

Ця функція дозволяє нам перевірити правильність даних для того щоб бути впевненим, що їх не було замінено. Така перевірка дозволяє уникнути поломок системи при сторонніх атаках.

Для одного скаляру перевірку робить функція verify_scalar1 наведена на рисунку 3.15.

```rust
fn verify_scalar1(S: &Signature1, P: &EdwardsPoint, i: &[u8]) -> anyhow::Result<bool> {
    let Signature1 { c, d } = S;
    if !c.is_canonical() || !d.is_canonical() {
        return Ok(false);
    }
    let W_prime = d * G - c * P;
    let c_prime = reduce_1_point_hash_into_scalar(&W_prime, i)?;
    if *c != c_prime {
        return Ok(false);
    }
    Ok(true)
}
```

Рисунок 3.15 – Реалізація функції verify_scalar1

Для перевірки декількох скалярів нам потрібно використовувати функцію verify_scalar2, реалізація якої представлена на рисунку 3.16.

```rust
fn verify_scalar2(
    S: &Signature2,
    P1: &EdwardsPoint,
    P2: &EdwardsPoint,
    B1: &EdwardsPoint,
    B2: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<bool> {
    let Signature2 { c, d1, d2 } = S;
    if !c.is_canonical() || !d1.is_canonical() || !d2.is_canonical() {
        return Ok(false);
    }
    let W1_prime = d1 * B2 - c * P1;
    let W2_prime = d2 * B1 + d1 * G - c * P2;
    let c_prime = reduce_2_point_hash_into_scalar(&W1_prime, &W2_prime, i)?;
    if *c != c_prime {
        return Ok(false);
    }
    Ok(true)
}
```

Рисунок 3.16 – Реалізація функції verify_scalar2

Обидві функції роблять ряд обчислянь для того щоб упевнитись у тому, що дані мають вірний формат та їх не було пошкоджено в процесі передачі між кроками.

sign_point і verify_point

Для точок у нас також використовуються окремі функції для підпису і верифікації. Реалізацію sign_point можна побачити на рисунку 3.17.

```
fn sign_point1(s: &Scalar, B: &EdwardsPoint, i: &[u8]) -> anyhow::Result<Signature1> {
    let r = &random_scalar()?;
    let W1 = r * G;
    let W2 = r * B;
    let c = reduce_2_point_hash_into_scalar(&W1, &W2, i)?;
    let d = (c * s + r).reduce();
    Ok(Signature1 { c, d })
}
```

Рисунок 3.17 – Реалізація функції sign_point

Її принцип дуже схожий з функцією підпису для скаляру.

Вигляд функції verify_point зображено на рисунку 3.18.

```
fn verify_point1(
    S: &Signature1,
    P1: &EdwardsPoint,
    P2: &EdwardsPoint,
    B: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<bool> {
    let Signature1 { c, d } = S;
    if !c.is_canonical() || !d.is_canonical() {
        return Ok(false);
    }
    let W1_prime = d * G - c * P1;
    let W2_prime = d * B - c * P2;
    let c_prime = reduce_2_point_hash_into_scalar(&W1_prime, &W2_prime, i)?;
    if *c != c_prime {
        return Ok(false);
    }
    Ok(true)
}
```

Рисунок 3.18 – Реалізація функції verify_point

Подібно до функцій verify_scalar, дана функція за допомогою обчислень перевіряє чи задовольняють передані дані умову. Якщо з ними все в порядку, то їх можна використовувати в подальших обчисленнях. Якщо щось не так, то ці дані не буде використано у обчисленнях. Таким чином зловмисники не можуть зламати хід виконання програми тому, що дані завжди будуть верифіковані.

3.1.6 Write

У нашому алгоритмі немає порівняння даних напряму, ми порівнюємо хеші. Тому всі значення перетворюються в хеш та записуються в буфер. Функцію запису в буфер я виніс в окремий trait, для того щоб об'єднати запис як скалярних значень, так і точок і підписів. Реалізовано це так як зображено на рисунку 3.19.

```
trait Write: io::Write {
    fn write_scalar(&mut self, s: &Scalar) -> anyhow::Result<()> {
        self.write_all(s.as_bytes())
            .context("failed to write scalar")?;
        Ok(())
    }

    fn write_point(&mut self, P: &EdwardsPoint) -> anyhow::Result<()> {
        self.write_all(P.compress().as_bytes())
            .context("failed to write point")?;
        Ok(())
    }

    fn write_signature1(&mut self, S: &Signature1) -> anyhow::Result<()> {
        self.write_scalar(&S.c)
            .context("failed to write signature")?;
        self.write_scalar(&S.d)
            .context("failed to write signature")?;
        Ok(())
    }

    fn write_signature2(&mut self, S: &Signature2) -> anyhow::Result<()> {
        self.write_scalar(&S.c)
            .context("failed to write signature")?;
        self.write_scalar(&S.d1)
            .context("failed to write signature")?;
        self.write_scalar(&S.d2)
            .context("failed to write signature")?;
        Ok(())
    }
}
```

Рисунок 3.19 – Реалізація trait Write

В залежності від даних які нам треба записати в буфер, програма обирає алгоритм дій.

Для скаляру (функція write_scalar) ми використовуємо .as_bytes(), щоб перетворити його в байти і за допомогою .write_all() записуємо в буфер.

Для точки (функція write_point) ми робимо ті ж самі дії, але попередньо використавши .compress(), щоб перевести точку до скалярного значення та перетворити потім в байти.

Для підписів ми використовуємо вище реалізовану write_scalar, для кожного поля підпису.

Також передбачено появу помилок типу «failed to write scalar/point/signature» у разі якщо записати дані не вдалося.

3.1.7 Read

Для того щоб потім дістати значення з буферу та мати можливість їх використовувати було створено trait Read. Реалізацію наведено на рисунку 3.20.

Так само як Write, Read обирає яку дію потрібно виконати в залежності від того з якою структурою має справу. Загалом дії протилежні від дій Write.

Якщо ми маємо діло зі скаляром, то використовується функція read_scalar. За допомогую .read_exact() зчитується значення та записується в змінну.

Для точки процес трішки інакший тому, що після .read_exact() нам потрібно ще виконати операцію .decompress() для того щоб отримати початкові дані. Далі вони записуються у змінну.

Для підписів використовується функція read_scalar для кожного значення.

Так само як і у Write, у Read передбачено перевірку на те чи пройшла операція успішно, і якщо ні, то виводиться помилка типу «failed to read scalar/point/signature».

```rust
trait Read: io::Read {
    fn read_scalar(&mut self) -> anyhow::Result<Scalar> {
        let mut scalar = [0; SCALAR_SIZE];
        self.read_exact(&mut scalar)
            .context("failed to read scalar")?;
        Ok(Scalar::from_bits(scalar))
    }

    fn read_point(&mut self) -> anyhow::Result<EdwardsPoint> {
        let mut point = [0; POINT_SIZE];
        self.read_exact(&mut point)
            .context("failed to read point")?;
        let point = CompressedEdwardsY(point)
            .decompress()
            .context("invalid Curve25519 point")?;
        Ok(point)
    }

    fn read_signature1(&mut self) -> anyhow::Result<Signature1> {
        let c = self.read_scalar().context("failed to read signature")?;
        let d = self.read_scalar().context("failed to read signature")?;
        Ok(Signature1 { c, d })
    }

    fn read_signature2(&mut self) -> anyhow::Result<Signature2> {
        let c = self.read_scalar().context("failed to read signature")?;
        let d1 = self.read_scalar().context("failed to read signature")?;
        let d2 = self.read_scalar().context("failed to read signature")?;
        Ok(Signature2 { c, d1, d2 })
    }
}
```

Рисунок 3.20 – Реалізація trait Read

Як бачимо реалізація trait`ів Read та Write є дуже схожою, але воно і не дивно, адже вони роблять обернені дії з даними.

3.1.8 Функція step1_client_to_server

Першою функцією яка реалізовує технічну частину алгоритму є step1_client_to_server. Її реалізацію можна побачити на рисунку 3.21.

В першу чергу функція перевіряє чи знаходимося ми зараз у правильному стані – це стан New. Якщо так, тоді починаємо реалізовувати сам алгоритм.

```rust
fn step1_client_to_server<D: Digest>(
    state: &mut State<D>,
    mut output: &mut [u8],
) -> anyhow::Result<usize> {
    if output.len() < STEP1_SIZE {
        anyhow::bail!("Buffer too small: {STEP1_SIZE}")
    }
    if let State::New { hasher, poisoned } = mem::replace(state, State::Poisoned) {
        // Nothing interesting is happening on step 1. Client generates
        // a couple of ephemeral scalars and sends public points to Server.
        let a2 = random_scalar()?;
        let a3 = random_scalar()?;
        let G2a = &a2 * G;
        let G3a = &a3 * G;

        let S2a = sign_scalar1(&a2, &[0x01])?;
        let S3a = sign_scalar1(&a3, &[0x02])?;

        output.write_point(&G2a)?;
        output.write_signature1(&S2a)?;
        output.write_point(&G3a)?;
        output.write_signature1(&S3a)?;

        *state = State::ClientStep3 {
            hasher,
            a2,
            a3,
            poisoned,
        };
        return Ok(STEP1_SIZE);
    }
    unreachable!()
}
```

Рисунок 3.21 – Реалізація функції step1_client_to_server

Створюються два рандомні числа a2 і a3. Після цього вони множаться на G – довільну точку на еліптичній кривій, і таким чином ми отримуємо дві точки G2a і G3a.

Далі скаляри підписуються за допомогою функції sign_scalar1. Після підпису отримаємо S2a та S3a.

Потім за допомогою функцій write_point і write_signature1 ми перетворюємо наші дані в байти та записуємо в буфер output.

Після цього ми змінюємо наш стан на ClientStep3 і повертаємо OK, тобто закінчуємо виконання функції. Якщо щось пішло не так то стан змінюється на Poisoned.

3.1.9 Функція step2_server_to_client

Наступна функція, яка продовжує наш алгоритм – це step2_server_to_client. Реалізація функції зображена на рисунку 3.22. Код на рисунку зображений у два стовпці для компактності.

```rust
fn step2_server_to_client<D: Digest>(
    state: &mut State<D>,
    mut input: &[u8],
    mut output: &mut [u8],
) -> anyhow::Result<usize> {
    if output.len() < STEP2_SIZE {
        anyhow::bail!("Buffer too small: {STEP2_SIZE}")
    }
    if input.len() != STEP1_SIZE {
        *state = State::Poisoned;
        anyhow::bail!("invalid step 1 message")
    }
    if let State::New {
        hasher,
        mut poisoned,
    } = mem::replace(state, State::Poisoned)
    {
        let G2a = input.read_point()?;
        let S2a = input.read_signature1()?;
        let G3a = input.read_point()?;
        let S3a = input.read_signature1()?;

        if !verify_scalar1(&S2a, &G2a, &[0x01])? {
            poisoned = true;
        }
        if !verify_scalar1(&S3a, &G3a, &[0x02])? {
            poisoned = true;
        }

        let b2 = random_scalar()?;
        let b3 = random_scalar()?;
        let G2b = &b2 * G;
        let G3b = &b3 * G;

        let G2 = b2 * G2a;
        let G3 = b3 * G3a;

        if G2.is_identity() || G3.is_identity() {
            poisoned = true;
        }

        let y = secret_scalar(hasher)?;
        let r = random_scalar()?;
        let Pb = r * G3;
        let Qb = y * G2 + &r * G;

        let S2b = sign_scalar1(&b2, &[0x03])?;
        let S3b = sign_scalar1(&b3, &[0x04])?;
        let SQb = sign_scalar2(&r, &y, &G2, &G3, &[0x05])?;

        output.write_point(&G2b)?;
        output.write_signature1(&S2b)?;
        output.write_point(&G3b)?;
        output.write_signature1(&S3b)?;
        output.write_point(&Pb)?;
        output.write_point(&Qb)?;
        output.write_signature2(&SQb)?;

        *state = State::ServerStep4 {
            b3,
            G3a,
            G2,
            G3,
            Pb,
            Qb,
            poisoned,
        };
        return Ok(STEP2_SIZE);
    }
    unreachable!()
}
```

Рисунок 3.22 – Реалізація функції step2_server_to_client

В першу чергу функція перевіряє всі вхідні дані на відповідність. Буфер перевіряється на відповідність заданій довжині, якщо довжина інша, то повертається помилка з відповідним повідомленням

Також йде дві перевірки стану. Перша перевірка – це перевірка чи стан Poisoned, якщо так, то щось пішло не так на минулому кроці. Друга перевірка – якщо ми бачимо що наш стан New, значить щось іде не так і ми переводимо стан в Poisoned.

Після цього нам потрібно прочитати отримані дані, для цього ми використовуємо read функції. Також нам потрібно верифікувати прочитані дані для того, щоб уникнути збоїв у програмі.

Після верифікації йде продовження виконання алгоритму: беруться два рандомні скаляри, потім вони множаться на точку G еліптичної кривої, після цього обчислюються G2 та G3, а потім ще й Pb і Qb. Тобто всі дії виконуються за алгоритмом наведеним у додатку 1.

Коли всі обчислення будуть виконані ми підписуємо отримані скаляри за допомогою функцій sign() і отримуємо S2b, S3b, SQb.

Далі всі наші дані за допомогою функцій write записуються у буфер, стан програми змінюється на ServerStep4 і повертається OK.

3.1.10 Функція step3_client_to_server

Після успішного другого кроку ми переходимо до третього. Код реалізації можна побачити на рисунку 3.23.

Спочатку у нас проходять такі ж перевірки, як і на другому кроці, тобто на довжину буфера та на стан у якому ми зараз знаходимось, нам потрібен стан ClientStep3.

Якщо перевірки були успішними, то ми перетворюємо вхідні дані за допомогою функцій read та перевіряємо їх на валідність функціями verify. Далі продовжуємо реалізацію основного алгоритму.

В результаті арифметичних дій ми отримуємо Pa, Qa, Pa_Pb, Qa_Qb, Ra, а після відповідних підписів також SQa та SRa.

В кінці ми всі дані перетворюємо в байти та записуємо в буфер функціями write.

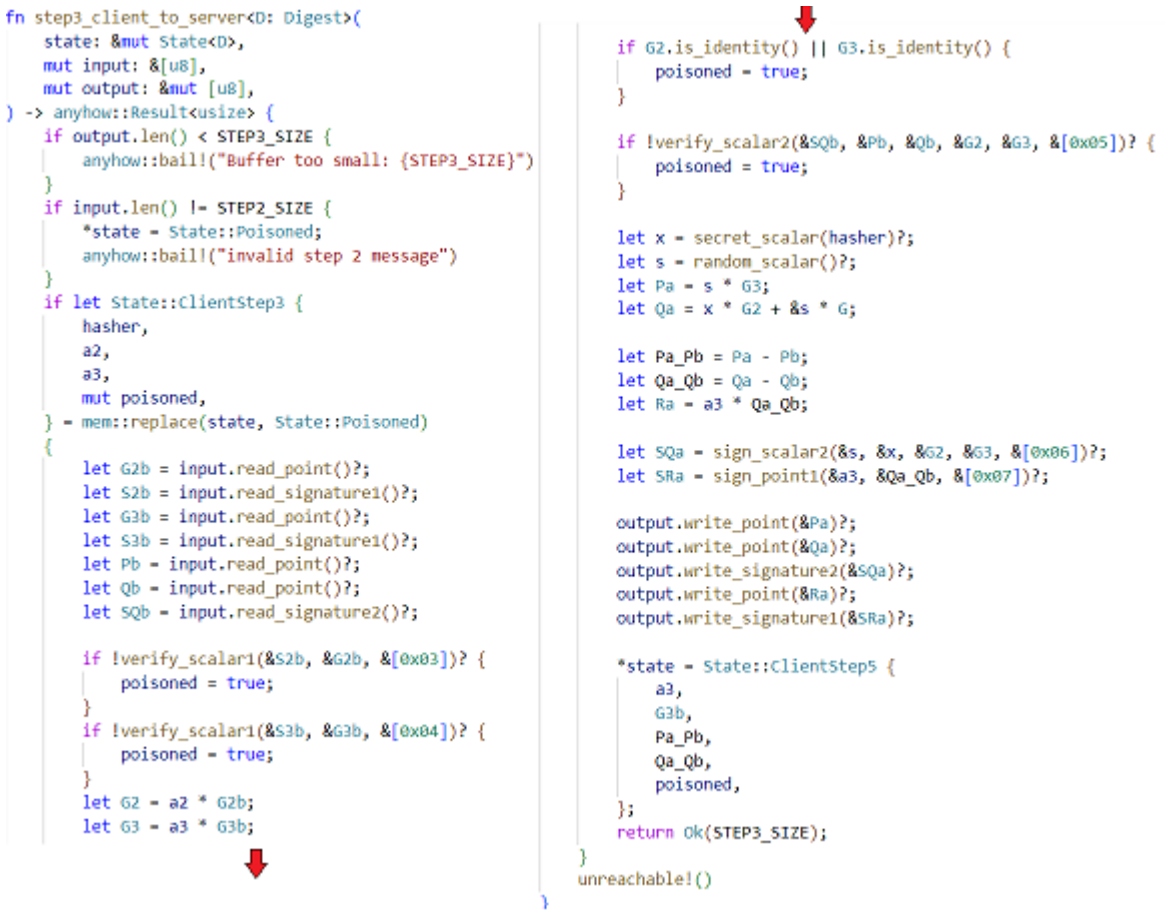


Рисунок 3.23 – Реалізація функції step3_client_to_server

Після того як дані записані в буфер, ми змінюємо стан на ClientStep5 та повертаємо ОК. На цьому третій крок завершено.

3.1.11 Функція step4_server_to_client

Наступний крок реалізовано функцією step4_server_to_client. Її реалізація приведена на рисунку 3.24.

Спочатку йде перевірка довжини буфера, потім перевірка на стан. На цьому кроці стан повинен бути ServerStep4.

Якщо перевірки успішні, то читаємо дані функцією read з буфера та верифікуємо їх функцією verify_scalar.

Після цього проводимо арифметичні операції відповідно до нашого алгоритму. В результаті отримаємо Pa_Pb, Qa_Qb, Rb, Rab, а після підписання ще й SRb.

```rust
fn step4_server_to_client<D: Digest>(
    state: &mut State<D>,
    mut input: &[u8],
    mut output: &mut [u8],
) -> anyhow::Result<usize> {
    if output.len() < STEP4_SIZE {
        anyhow::bail!("Buffer too small: {STEP4_SIZE}")
    }
    if input.len() != STEP3_SIZE {
        *state = State::Poisoned;
        anyhow::bail!("invalid step 3 message")
    }
    if let State::ServerStep4 {
        b3,
        G3a,
        G2,
        G3,
        Pb,
        Qb,
        mut poisoned,
    } = mem::replace(state, State::Poisoned)
    {
        let Pa = input.read_point()?;
        let Qa = input.read_point()?;
        let SQa = input.read_signature2()?;
        let Ra = input.read_point()?;
        let SRa = input.read_signature1()?;

        if !verify_scalar2(&SQa, &Pa, &Qa, &G2, &G3, &[0x06])? {
            poisoned = true;
        }

        let Pa_Pb = Pa - Pb;
        let Qa_Qb = Qa - Qb;

        if !verify_point1(&SRa, &G3a, &Ra, &Qa_Qb, &[0x07])? {
            poisoned = true;
        }

        let Rb = b3 * Qa_Qb;
        let Rab = b3 * Ra;

        let result = Rab == Pa_Pb;

        let SRb = sign_point1(&b3, &Qa_Qb, &[0x08])?;

        output.write_point(&Rb)?;
        output.write_signature1(&SRb)?;

        *state = if poisoned {
            State::Poisoned
        } else {
            State::Complete { result }
        };
        return Ok(STEP4_SIZE);
    }
    unreachable!()
}
```

Рисунок 3.24 – Реалізація функції step4_server_to_client

У result ми записуємо Rab == Pa_Pb.

Після цього всього ми перевіряємо: якщо прапорець poisoned у нас true, то ми переводимо стан у Poisoned. Якщо ж ніяких помилок не було, тоді переводимо стан в Complete.

3.1.12 Функція step5_client_is_done

Останній крок алгоритму – це функція step5_client_is_done.

На початку, як і в інших кроках, проводяться перевірки стану і довжини буфера. Для цього кроку нам потрібно перебувати в стані ClientStep5.

Потім за допомогою функцій read дані читаються з буфера. Далі за допомогою функції verify_point1 ми перевіряємо валідність точки. Якщо все добре – переходимо далі.

Обчислюємо Rab = a3 * Rb.

В result присвоюємо Rab == Pa_Pb.

Більш детально можна розглянути на рисунку 3.25.

```
fn step5_client_is_done<D: Digest>(state: &mut State<D>, mut input: &[u8]) -> anyhow::Result<()> {
    if input.len() != STEP4_SIZE {
        *state = State::Poisoned;
        anyhow::bail!("invalid step 4 message")
    }
    if let State::ClientStep5 {
        a3,
        G3b,
        Pa_Pb,
        Qa_Qb,
        mut poisoned,
    } = mem::replace(state, State::Poisoned)
    {
        let Rb = input.read_point()?;
        let SRb = input.read_signature1()?;

        if !verify_point1(&SRb, &G3b, &Rb, &Qa_Qb, &[0x08])? {
            poisoned = true;
        }

        let Rab = a3 * Rb;

        let result = Rab == Pa_Pb;

        *state = if poisoned {
            State::Poisoned
        } else {
            State::Complete { result }
        };
        return Ok(());
    }
    unreachable!()
}
```

Рисунок 3.25 – Реалізація функції step5_client_is_done

Якщо ніяких помилок немає, переводимо в стан Complete, якщо ж були помилки, то нам потрібно перейти в стан Poisoned. Після цього крок завершено.

На цьому реалізацію нашого алгоритму завершено і можна переходити до тестувать та бенчмарків.

3.2 Розробка програмних компонентів тестування

Далі буде описано код для реалізації тестів та бенчмарків.

3.2.1 Тести

У програмі реалізовано 4 тести, які своєю структурою дуже схожі один на одного: test_sha256_ok, test_sha256_different, test_blake2_ok, test_blake2_different. Як можна здогадатися з назв функцій, ми маємо по дві функції на кожен алгоритм хешування. 2 функції для алгоритму SHA-256, та 2 для BLAKE2. Для кожного з них реалізована функція, яка проводить перевірку даних які співпадають, а також функція, яка порівнює дані, які не збігаються. Пропоную розглянути роботу функцій детальніше на прикладі функції test_sha256_ok, реалізація якої приведена на рисунку 3.26.

Тест починається з того, що створюється Authenticator на клієнті, а також на сервері.

Далі ми беремо якийсь секрет, наприклад, це може бути пароль, та присвоюємо у змінну SECRET.

І клієнт і сервер за допомогою .push_secret() пушають секрет кожен у свій аутентифікатор.

Коли всі потрібні дані запушені викликається функція .begin()для клієнта, яка починає перший крок і увесь процес порівняння паролів.

Далі ми маємо цикл, у якому відбуваються переходи до наступних кроків. На першій ітерації поточний крок передається серверу server.proceed(), сервер в свою чергу повертає також крок (наступний), якщо сервер не повернув крок, тоді відбувається break. Але якщо сервер повернув крок, тоді цей крок йде клієнтові за допомогою client.proceed(). Клієнт в свою чергу повертає також крок, який переходить до серверу і так далі у циклі. Коли цикл буде завершено,

то у поля client.result та server.result буде записано true – тобто це буде означати, що перевірка пройшла успішно і секрети однакові. Також для відстеження помилок використовуються unwrap функції.

```rust
#[test]
fn test_sha256_ok() {
    let mut client = Authenticator::<Sha256>::new();
    let mut server = Authenticator::<Sha256>::new();

    const SECRET: &[u8] = b"SECRET";

    client.push_secret(SECRET).unwrap();
    server.push_secret(SECRET).unwrap();

    let mut step = client.begin().unwrap();

    let mut i = 0;
    loop {
        println!(">> {}", i);
        i += 1;
        step = if let Some(step) = server.proceed(&step).unwrap() {
            step
        } else {
            break;
        };
        step = if let Some(step) = client.proceed(&step).unwrap() {
            step
        } else {
            break;
        };
    }

    assert!(client.result().unwrap());
    assert!(server.result().unwrap());
}
```

Рисунок 3.26 – Реалізація функції test_sha256_ok

Для функції test_sha256_different виконуються всі ті ж самі операції, за винятком того, що секрети будуть різні у клієнта і сервера. Тоді в результаті ми отримаємо false замість true у полі server.result.

Якщо порівнювати ці дві функції з test_blake2_ok та test_blake2_different, то вони відрізняються лише тим, що на етапі створення аутентифікатора ми

передаємо у аутентифікатор метод хешування не Sha256, а Blake2b256. Вся інша логіка роботи тестів залишається та ж сама.

3.2.2 Бенчмарки

Для того щоб зробити якісь висновки стосовно швидкості роботи реалізованого алгоритму, нам потрібно виміряти час, потрібний на його виконання при різнах умовах. Таку функцію виконують бенчмарки. Мною було розроблено два бенчмарки: перший проводить вимір часу необхідного для виконання алгоритму вцілому, другий вимірює час затрачений на кожен конкретний крок.

Функція run_full дозволяє виміряти час для всього алгоритму. Реалізація цієї функція така ж сама як і для тестів, які були описані вище, тому ми її вже бачили (див. рис. 3.26).

Основна ж функція – це criterion_benchmarkаме вона запускає всі бенчмарки. Її початок можна побачити на рисунку 3.27. Для самого вимірювання часу використовується інструмент расту під назвою bench_function. Ця функція запускає обрані функції (чи методи) велику кількість разів доти, поки результати не стабільні, а також сама прибирає аномалії. Для нашого випадку декілька тисяч запусків.

```
fn criterion_benchmark(c: &mut Criterion) {
    {
        let mut sha256 = c.benchmark_group("sha256");
        sha256.bench_function("all-steps-ok", |b| {
            b.iter(|| run_full::<Sha256>(b"HelloWorld", b"HelloWorld"))
        });

        sha256.bench_function("all-steps-err", |b| {
            b.iter(|| run_full::<Sha256>(b"HELLOWORLD", b"HelloWorld"))
        });
```

Рисунок 3.27 – Реалізація бенчмарку для повного алгоритму

На цьому фрагменті ми бачимо запуск бенчмарку для алгоритму SHA-256 для варіанту коли ми вимірюємо час роботи всього алгоритму одразу.

У першій частині ми вимірюємо час потрібний для виконання алгоритму коли порівнювані дані збігаються та алгоритм виконується успішно.

Далі ми повторюємо вимір часу для випадку коли вхідні дані різні і алгоритм не є успішним. Також для зручності на екран виводяться дані про те, який саме бенчмарк було запущено.

Після цього ми робимо те ж саме, але для кожного кроку окремо. Наприклад перший крок зображено на рисунку 3.28.

```
sha256.bench_function("step-1", |b| {
    b.iter(|| {
        let mut client = Authenticator::<Sha256>::new();
        client.begin().unwrap()
    })
});
```

Рисунок 3.28 – Реалізація бенчмарку для першого кроку

Для того щоб виміряти час затрачений на один крок ми виконуємо bench_function лише для client.begin() та вимірюємо час.

Інші кроки виконуються подібно до цього, їх можна розглянути на рисунку 3.29.

Для другого кроку ми присвоюємо в якусь змінну результат роботи першого кроку (step1 = client.begin), далі ми проводимо bench_function для другого кроку, передавши його за допомогою client.proceed.

На третьому кроці нам потрібно спочатку в першу змінну присвоїти результат виконання першого кроку, далі пройти другий крок, використовуючи результат першого. А потім результат другого кроку передати за допомогою client.proceed і виміряти затрачений час, за допомогою функції bench_function.

Четвертий крок повторяє алгоритм третього: ми шукаємо результат першого кроку, з його допомогою знаходимо результат другого кроку, а з ним у свою чергу ми можемо знайти результат третього. Отримавши результати третього кроку ми можемо виміряти час необхідний для виконання четвертого кроку за допомогою bench_function.

```rust
sha256.bench_function("step-2", |b| {
    let mut client = Authenticator::<Sha256>::new();
    let step1 = client.begin().unwrap();

    b.iter(|| {
        let mut server = Authenticator::<Sha256>::new();
        server.proceed(&step1).unwrap().unwrap()
    })
});

sha256.bench_function("step-3", |b| {
    let mut client = Authenticator::<Sha256>::new();
    let mut server = Authenticator::<Sha256>::new();
    let step1 = client.begin().unwrap();
    let step2 = server.proceed(&step1).unwrap().unwrap();

    b.iter(|| {
        let mut client = client.clone();
        client.proceed(&step2).unwrap().unwrap()
    })
});

sha256.bench_function("step-4", |b| {
    let mut client = Authenticator::<Sha256>::new();
    let mut server = Authenticator::<Sha256>::new();
    let step1 = client.begin().unwrap();
    let step2 = server.proceed(&step1).unwrap().unwrap();
    let step3 = client.proceed(&step2).unwrap().unwrap();

    b.iter(|| {
        let mut server = server.clone();
        server.proceed(&step3).unwrap().unwrap()
    })
});
```

Рисунок 3.29 – реалізація бенчмарків для всіх кроків

На цьому функція бенчмарку для алгоритму хешування SHA-256 завершена. Але так як ми перевіряємо декілька алгоритмів, то нам треба реалізувати такі ж самі функції і для алгоритму BLAKE2. Реалізація майже не відрізняється, потрібно лише вписувати Blake2b256 замість sha-256 при створенні аутентифікатору.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було розглянуто деталі реалізації програмного забезпечення для алгоритму розробленого у другому розділі. Для вирішення поставленої задачі було використано мову програмування Rust.

Було описано роботу основних функцій програми, а також приведено приклади їх реалізації мовою Rust. В деталях описано як працює алгоритм та кожен крок цього алгоритму. Окрім цього було розглянуто допоміжні функції, які дозволяють підписувати скаляри та точки, верифікувати їх, порівнювати, перетворювати в байти. Реалізовано методи для запису та зчитування байтів з буферів.

Також наведено приклади тестування, які дозволяють запустити алгоритм та перевірити його виконання за умови, що вхідні дані однакові, а також за умови різних вхідних даних. Тобто коли алгоритм виконано успішно, або з помилкою. Реалізовані бенчмарки дозволяють вимірювати час витрачений для кожного кроку окремо та для алгоритму вцілому. Це дасть нам змогу порівняти отримані результати.

Варто відмітити, що тестування та бенчмарки проведені для двох алгоритмів хешування SHA-256 та BLAKE2.

Загалом програма написана як бібліотека для того, щоб розробник міг її інтегрувати у свій проект та з легкістю використовувати.

РОЗДІЛ 4.
ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

У цьому розділі буде наведно результати роботи програмного забезпечення, розробленого на основі нашого варіанту протоколу про нульове знання. Було досліджено роботу двох алгоритмів хешування SHA-256 та BLAKE2. Детальніше те як кожен з них впорався із завданням розглянемо нижче.

4.1 Результати роботи алгоритму SHA-256

У результаті виконання бенчмарку із використанням алгоритму хешування SHA-256 ми отримали результат який наведено на рисунку 4.1.

Функція bench_function проводить багато запусків обраної частини коду і, що найголовніше, показує нам середній час виконання. Як бонус ми також маємо час найшвидшого виконання і час найповільнішого виконання. Розглянемо час виконання повного успішного та неуспішного алгоритму у таблиці 4.1.

Таблиця 4.1 – Результат бенчмарку для повного алгоритму SHA-256

Назва	Мінімальний час	Середній час	Максимальний час
sha256/all-steps-ok	2.5579 ms	2.5604 ms	2.5648 ms
sha256/all-steps-err	2.5618 ms	2.5661 ms	2.5712 ms

Основне що ми можене спостерігати з таблиці – це те, що мінімальний і максимальний час відрізняється лише на 0.2 відсотка. Якщо ж порівняти успішний варіант з варіантом де стається помилка, то вони також відрізняються у середньому на 0.2 відсотки, що можна вважати похибкою, адже мінімальний

і максимальний час мають таку ж різницю. Це дозволяє нам бути упевненим, що система є захищеною від timing attack і наш алгоритм доходить до кінця не зважаючи на те, що у ньому сталася помилка і результат є неуспішним.

```
sha256/all-steps-ok     time:   [2.5579 ms 2.5604 ms 2.5648 ms]
Found 12 outliers among 100 measurements (12.00%)
  1 (1.00%) low mild
  4 (4.00%) high mild
  7 (7.00%) high severe
sha256/all-steps-err    time:   [2.5618 ms 2.5661 ms 2.5712 ms]
Found 16 outliers among 100 measurements (16.00%)
  2 (2.00%) high mild
  14 (14.00%) high severe
sha256/step-1           time:   [78.508 us 78.542 us 78.586 us]
Found 14 outliers among 100 measurements (14.00%)
  3 (3.00%) high mild
  11 (11.00%) high severe
sha256/step-2           time:   [628.33 us 628.55 us 628.83 us]
Found 15 outliers among 100 measurements (15.00%)
  1 (1.00%) high mild
  14 (14.00%) high severe
sha256/step-3           time:   [953.96 us 954.14 us 954.34 us]
Found 16 outliers among 100 measurements (16.00%)
  3 (3.00%) low severe
  1 (1.00%) low mild
  3 (3.00%) high mild
  9 (9.00%) high severe
sha256/step-4           time:   [656.91 us 657.15 us 657.51 us]
Found 14 outliers among 100 measurements (14.00%)
  1 (1.00%) low mild
  2 (2.00%) high mild
  11 (11.00%) high severe
```

Рисунок 4.1 – Результат роботи алгоритму SHA-256

Час необхідний для виконання кожного конкретного кроку можна побачити у таблиці 4.2.

Тут ми також можемо побачити що для кожного кроку відмінність мінімального та максимального часу дуже маленька, що в свою чергу ще раз нам доводить те, що алгоритм виконується за постійний час, який не залежить

від того був він успішним, або помилка виникла на першому, другому чи інших кроках.

Таблиця 4.2 – Бенчмарк покрокового алгоритму SHA-256

Назва	Мінімальний час	Середній час	Максимальний час
sha256/step-1	78.508 us	78.542 us	78.586 us
sha256/step-2	628.33 us	628.55 us	628.83 us
sha256/step-3	953.96 us	954.14 us	954.34 us
sha256/step-4	656.91 us	657.15 us	657.51 us

Варто відмітити те, що якщо підсумувати час необхідний для всіх кроків, то ми отримаємо значення менше ніж при вимірюванні часу для всього алгоритму. Це повязано з тим, що для переходу від одного кроку до іншого нам також потрібен деякий час для передачі даних, тому доцільніше розглядати загальний час.

4.2 Результати роботи алгоритму BLAKE2

Другим алгоритмом, який було випробувано, був BLAKE2. В теорії він повинен відпрацьовувати швидше за SHA-256, давайте подивимося, що ж виходить на практиці. Результат бенчмарку можна побачити на рисунку 4.2.

Так само як і для SHA-256 порівняємо час повного успішного виконання та повного виконання з помилкою у таблиці 4.3.

Таблиця 4.3 – Результат бенчмарку для повного алгоритму BLAKE2

Назва	Мінімальний час	Середній час	Максимальний час
blake2/all-steps-ok	2.5578 ms	2.5599 ms	2.5633 ms
blake2/all-steps-err	2.5643 ms	2.5675 ms	2.5715 ms

Як бачимо з використанням іншого алгоритму нічого глобально не змінолося, тенденція залишилася та ж сама: незалежно від того чи було

виконано алгоритм успішно, чи з помилкою, ми все одно отримуємо сталий час із мінімальною похибкою.

```
blake2/all-steps-ok     time:   [2.5578 ms 2.5599 ms 2.5633 ms]
Found 15 outliers among 100 measurements (15.00%)
  8 (8.00%) high mild
  7 (7.00%) high severe
blake2/all-steps-err    time:   [2.5643 ms 2.5675 ms 2.5715 ms]
Found 5 outliers among 100 measurements (5.00%)
  1 (1.00%) high mild
  4 (4.00%) high severe
blake2/step-1           time:   [78.476 us 78.501 us 78.530 us]
Found 11 outliers among 100 measurements (11.00%)
  2 (2.00%) high mild
  9 (9.00%) high severe
blake2/step-2           time:   [628.53 us 629.35 us 630.25 us]
Found 24 outliers among 100 measurements (24.00%)
  2 (2.00%) low mild
  3 (3.00%) high mild
  19 (19.00%) high severe
blake2/step-3           time:   [950.66 us 950.94 us 951.26 us]
Found 21 outliers among 100 measurements (21.00%)
  3 (3.00%) low severe
  2 (2.00%) low mild
  2 (2.00%) high mild
  14 (14.00%) high severe
blake2/step-4           time:   [654.83 us 657.75 us 662.05 us]
Found 16 outliers among 100 measurements (16.00%)
  3 (3.00%) high mild
  13 (13.00%) high severe
```

Рисунок 4.2 – Результат бенчмарку алгоритму BLAKE2

Порівняти покрокові результати можна в таблиці 4.4.

Таблиця 4.4 – Бенчмарк покрокового алгоритму SHA-256

Назва	Мінімальний час	Середній час	Максимальний час
blake2/step-1	78.476 us	78.501 us	78.530 us
blake2/step-2	628.53 us	629.35 us	630.25 us
blake2/step-3	950.66 us	950.94 us	951.26 us
blake2/step-4	654.83 us	657.75 us	662.05 us

Зі зміною алгоритму хешування з SHA-256 на BLAKE2 видно, що досі зберігається сталість часу для виконання кожного кроку. Різниці між мінімальним та максимальним часом майже немає, вона також всередньому становить близько 0.2 відсотків, що можна списати на похибку у вимірюванні.

4.3 Порівняння результатів

Тепер ми можемо порівняти отримані результати.

4.3.1 Порівняння функцій хешування

Для порівняння результатів ми будемо використовувати середній час всіх вимірювать. Усі результати зібрані в таблиці 4.5.

Таблиця 4.5 – Порівняння результатів алгоритму SHA-256 та BLAKE2

Вимір	SHA-256	BLAKE2
all-steps-ok	2.5604 ms	2.5599 ms
all-steps-err	2.5661 ms	2.5675 ms
step-1	78.542 us	78.501 us
step-2	628.55 us	629.35 us
step-3	954.14 us	950.94 us
step-4	657.15 us	657.75 us

З таблиці видно, що середній час виконання алгоритму майже однаковий для обох алгоритмів. Різниця між алгоритмами навіть менша ніж похибка (різниця між максимальним та мінімальним часом виконання). Тому можна сказати, що для нашого алгоритму функції хешування SHA-256 та BLAKE2 є однаковими з точки зору витрат часу.

4.3.2 Порівняння із звичайними паролями

Що ж ми отримаємо якщо повернемося до початку і порівняємо наш алгоритм із звичайними паролями? Для чистоти експерименту потрібно врахувати те, що у наших бенчмарках програма виконувався на одній машині, яка виступала як в ролі клієнта, так і в ролі сервера. У реальному житті ці ролі буде розділено між двома машинами: реальним клієнтом та реальним сервером. Тому нам потрібно врахувати затримки, які будуть спричинені передачею інформації через інтернет. Зараз майже всюди є можливість підключитися до високошвидкісного інтернет з'єднання і середньою затримкою можна вважати 70-100 мілісекунд. У своїх розрахунках я буду округляти все в більший бік для того, щоб зімітувати найгіршу ситуацію.

Основний недолік нашого алгоритму перед перевіркою звичайного пароля – це кількість звернень до сервера. Як можна бачити в додатку 1 передача даних у нас відбувається 5 разів, тобто це еквівалентно 500-ам мілісекундам затримки. Додамо до цього 2.6 мілісекунди на виконання самого алгоритму та отримаємо 502.6 мілісекунди, тобто трохи більше ніж пів секунди.

Час для перевірки звичайного паролю можна взяти нульовий, адже сучасні системи володіють такими технічними ресурсами, що порівняння відбувається майже миттєво. Але у нас все ще залишається затримка передачі даних. Для паролю дані передаються лише двічі: коли клієнт відправляє дані на сервер, та коли сервер відповідає після перевірки паролю. Тому для перевірки паролю ми маємо затримку в 200 мілісекунд. Це в 2.5 рази швидше ніж наша варіація, але не будемо забувати, що у даному випадку ми розглядаємо звичайний, явний пароль без хешування чи інших методів, які допомогають захистити дані. Тобто у такому випадку пароль є зовсім незахищеним.

Для того, щоб помістити паролі в реальне середовище, нам потрібно згадати деякі рекомендації для використання паролів. А саме те, що нам потрібна доволі надійна функція хешування для того, щоб не передавати їх у

відкритому вигляді. Для цього може підійти найрозповсюдженіший алгоритм SHA-256, або як рекомендують у OWASP (Open Web Application Security Project) краще використовувати для паролів функцію Argon2, а ця функція не найшвидша у своєму роді.

Ще однією гарною практикою для паролів вважається введення обмеження на мінімальний час перевірки паролю в 0.5 секунд, а зазвичай в 1 секунду. Це дозволяє захистити систему від brute force attack, адже якщо кожна перевірка паролю займає довгий час, то перебір паролів може зайняти майже цілу вічність. [30]

Тому, підсумовуючи, можемо додати до часу перевірки паролю ще 1 секунду. Ми досі не враховуємо час потрібний для хешування, тому що важко знайти точні дані, але навіть у такому випадку ми отримуємо 1.2 секунди для перевірки паролю. А це вже більш ніж вдвічі повільніше за нашу реалізацію.

Звісно, в реальному світі існує безліч інших чинників, які впривають на час роботи того чи іншого алгоритму, і ми не можемо однозначно сказати, що одне рішення вдвічі краще іншого по швидкості. Та ми можемо зробити висновок, що при дотриманні рекомендацій пов’язаних з реалізацією паролів, Zero Knowladge Protocol, частіше за все, буде показувати швидкість таку саму або навіть кращу, і це при забезпеченні максимального рівня безпеки, адже ніякі дані не передаються від клієнта до сервера та навпаки, а у випадку неуспішної аутентифікації сторони не дізнаються ніякої нової інформації, окрім того факту, що процес аутентифікації був неуспішним.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі ми розглянули результати роботи наших бенчмарків та порівняли результати виконання програми при різних початкових умовах. Ми побачили за який час виконується алгоритм повністю та кожен його крок окремо. У порівняльних таблицях ми наглядно побачили, що час виконання для алгоритмів SHA-256 та BLAKE2 є майже однаковим.

Також ми порівняли нашу реалізацію ZKP із звичайними паролями у процесі аутентифікації. В аспекті безпеки наша реалізація є беззаперечним лідером, а стосовно часу виконання ми побачили, що вона є не гіршою, а частіше за все навіть кращою (швидшою).

ВИСНОВКИ

Під час виконання дипломної роботи було розроблено алгоритм на основі Zero Knowladge Proof протоколу, а також реалізовано програмне забезпечення у вигляді бібліотеки, що робить можливим використання цієї розробки у багатьох інших проектах. Також було зроблено тестування та проведено перевірку на час роботи алгоритму.

У першому розділі було розглянуто різні види аутентифікації та авторизації. Зокрема можливості використання для цих задач паролів, сертифікатів, одноразових паролів, ключів доступу та токенів. Було наведено декілька альтернатив звичайних паролів: візуальні паролі, менеджери паролів, біометричні паролі, об’єднаний єдиний вхід. Були визначено їх недоліки та переваги. Головним недоліком була недосконалість цих варіантів з боку безпеки.

У другому розділі детально було описано роботу Zero Knowladge протоколу, а також його властивості. Було обрано SMP протокол як основу для реалізації Zero Knowladge, а також озвучено певні зміни в протоколі для його поліпшення. Розглянуто еліптичку криву ed25519, а також зміни які довелось зробити для її розширення. Розроблено кінцевий алгоритм дії програми та визначено технології, які будуть використні для реалізації програмного забезпечення.

У третьому розділі було реалізовано програмне забезпечення для раніше розробленого (у Розділі II) алгоритму. Було продемонстровано скріншоти з фрагментами коду розробленої системи, а також описано принцип роботи приведеного на рисунках коду. Показано приклади тестування та бенчмарків для вимірювання часу.

У четвертому розділі було проведено детальний аналіз отриманих результатів. Приведено скріншоти результатів бенчмарків, а також, за

допомогою порівняльних таблиць було порівняно швидкість роботи програми при використанні алгоритму SHA-256 та BLAKE2. У результаті, відмінностей у швидкодії цих двох алгоритмів не виявлено. Було теоретично порівняно нашу реалізацію із звичайними паролями і виявлено, що наш алгоритм не поступається у швидкодії гарній реалізації паролю, але наш варіант є безпечнішим, адже дані взагалі не передаються між клієнтом і сервером.

Отже, в результаті ми отримуємо бібліотеку, яку інші розробники можуть легко інтегрувати у свої проекти та доволі просто використовувати для забезпечення найвищого рівня безпеки процесу аутентифікації. При цьому втрати в часі будуть не суттєвими.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hypertext Transfer Protocol -- HTTP/1.0 [Електронний ресурс] // www.w3.org. W3C. – 1996. – Режим доступу до ресурсу: https://www.w3.org/Protocols/HTTP/1.0/spec.html#BasicAA.
2. MDN contributors. HTTP authentication [Електронний ресурс] / MDN contributors. – 2022. – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication.
3. Attack74: Name:man-in-the-middle [Електронний ресурс] // The All.Net Security Database. – 2003. – Режим доступу до ресурсу: http://all.net/CID/Attack/Attack74.html.
4. NTLM user authentication [Електронний ресурс] // Microsoft – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/troubleshoot/windows-server/windows-security/ntlm-user-authentication.
5. What Is an X.509 Certificate & How Does It Work? [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://sectigo.com/resource-library/what-is-x509-certificate.
6. What Is Two-Factor Authentication (2FA)? [Електронний ресурс] // Twilio – Режим доступу до ресурсу: https://authy.com/what-is-2fa/.
7. Multi-Factor Authentication [Електронний ресурс] // SecurID. – 2022. – Режим доступу до ресурсу: https://www.securid.com/products/multi-factor-authentication/.
8. Баранова Н. Google Authenticator — приложение для двухэтапной аутентификации [Електронний ресурс] / Наталья Баранова // Теплица социальных технологий. – 2018. – Режим доступу до ресурсу: https://te-st.ru/2018/02/08/google-authenticator/
9. Настройка аутентификации на базе ключей SSH на сервере Linux [Електронний ресурс]. – 2021. – Режим доступу до ресурсу:

https://www.digitalocean.com/community/tutorials/how-to-configure-ssh-key-based-authentication-on-a-linux-server-ru.

10. What Is Token-Based Authentication? [Електронний ресурс] // Okta. – 2020. – Режим доступу до ресурсу: https://www.okta.com/identity-101/what-is-token-based-authentication/
11. HMACSHA256 Class [Електронний ресурс] // Microsoft. – 2022. – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha256?view=net-6.0
12. Introduction to JSON Web Tokens [Електронний ресурс] // jwt.io – Режим доступу до ресурсу: https://jwt.io/introduction
13. What is Security Assertion Markup Language (SAML)? [Електронний ресурс] // Oracle – Режим доступу до ресурсу: https://www.oracle.com/security/cloud-security/what-is-saml/
14. WS-Trust 1.4 [Електронний ресурс] // OASIS. – 2012. – Режим доступу до ресурсу: http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html
15. Authenticate users with WS-Federation in ASP.NET Core [Електронний ресурс] // Microsoft – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/ws-federation?view=aspnetcore-6.0
16. dmitrybitman. OAuth 2.0 простым и понятным языком [Електронний ресурс] / dmitrybitman. – 2011. – Режим доступу до ресурсу: https://habr.com/ru/company/vk/blog/115163/
17. J. Bonneau and S. Preibusch, “The password thicket: technical and market failures in human authentication on the web,” in Proc. WEIS 2010, 2010.
18. S. Gaw and E. W. Felten, “Password Management Strategies for Online Accounts,” in ACM SOUPS 2006: Proc. 2nd Symp. on Usable Privacy and Security, pp. 44–55.

19. D. Florêncio and C. Herley, “A large-scale study of web password habits,” in WWW ’07: Proc. 16th International Conf. on the World Wide Web. ACM, 2007, pp. 657–666.
20. Menezes A. Discrete Mathematics and Its Applications / A. Menezes, P. van Oorschot, S. Vanstone // Handbook of Applied Cryptography / A. Menezes, P. van Oorschot, S. Vanstone. – https://cacr.uwaterloo.ca/hac/. – C. chapter 10 and 12.
21. Grid Authentication [Електронний ресурс] // Thales – Режим доступу до ресурсу: https://cpl.thalesgroup.com/access-management/authenticators/grid-authentication
22. How Does Single Sign-On Work? [Електронний ресурс] // onelogin – Режим доступу до ресурсу: https://www.onelogin.com/learn/how-single-sign-on-works
23. Что такое файлы cookies и зачем они нужны [Електронний ресурс] // ssl.com. – 2019. – Режим доступу до ресурсу: https://ssl.com.ua/blog/what-are-cookies/
24. Brute Force Attack: Definition and Examples [Електронний ресурс] // kaspersky. – 2021. – Режим доступу до ресурсу: https://www.kaspersky.com/resource-center/definitions/brute-force-attack.
25. Understanding Session Fixation Attacks [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://secureteam.co.uk/articles/web-application-security-articles/understanding-session-fixation-attacks/.
26. Dilmegani C. Zero-Knowledge Proof: How it Works & Applications in 2022 [Електронний ресурс] / Cem Dilmegani // aimultiple. – 2022. – Режим доступу до ресурсу: https://research.aimultiple.com/zero-knowledge-proofs/
27. Off-the-Record Messaging Protocol version 2 [Електронний ресурс] – Режим доступу до ресурсу: https://otr.cypherpunks.ca/Protocol-v2-3.1.0.html

28. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice [Електронний ресурс] // WeakDH. – 2015. – Режим доступу до ресурсу: https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf
29. BLAKE2 — fast secure hashing [Електронний ресурс] // BLAKE2. – 2017. – Режим доступу до ресурсу: https://www.blake2.net/
30. S. Whited. Best practices for password hashing and storage [Електронний ресурс] / S. Whited. – 2020. – Режим доступу до ресурсу: https://tools.ietf.org/id/draft-whited-kitten-password-storage-00.html.

ДОДАТОК 1

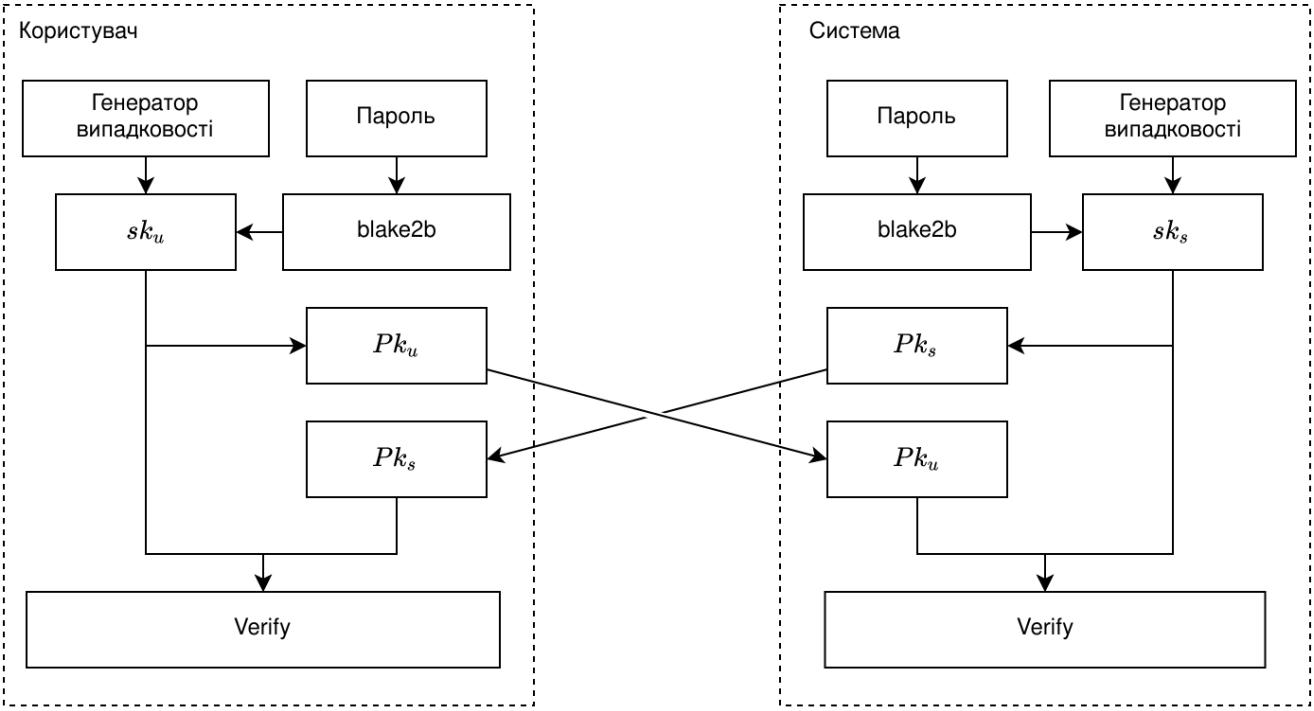
Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Шпильовий Р. П.			**Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою** Структурна схема системи	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Іваніщев Б. В.					
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83	
Н. Контр.		Сімоненко В. П.					
Затвердив							

ДОДАТОК 2

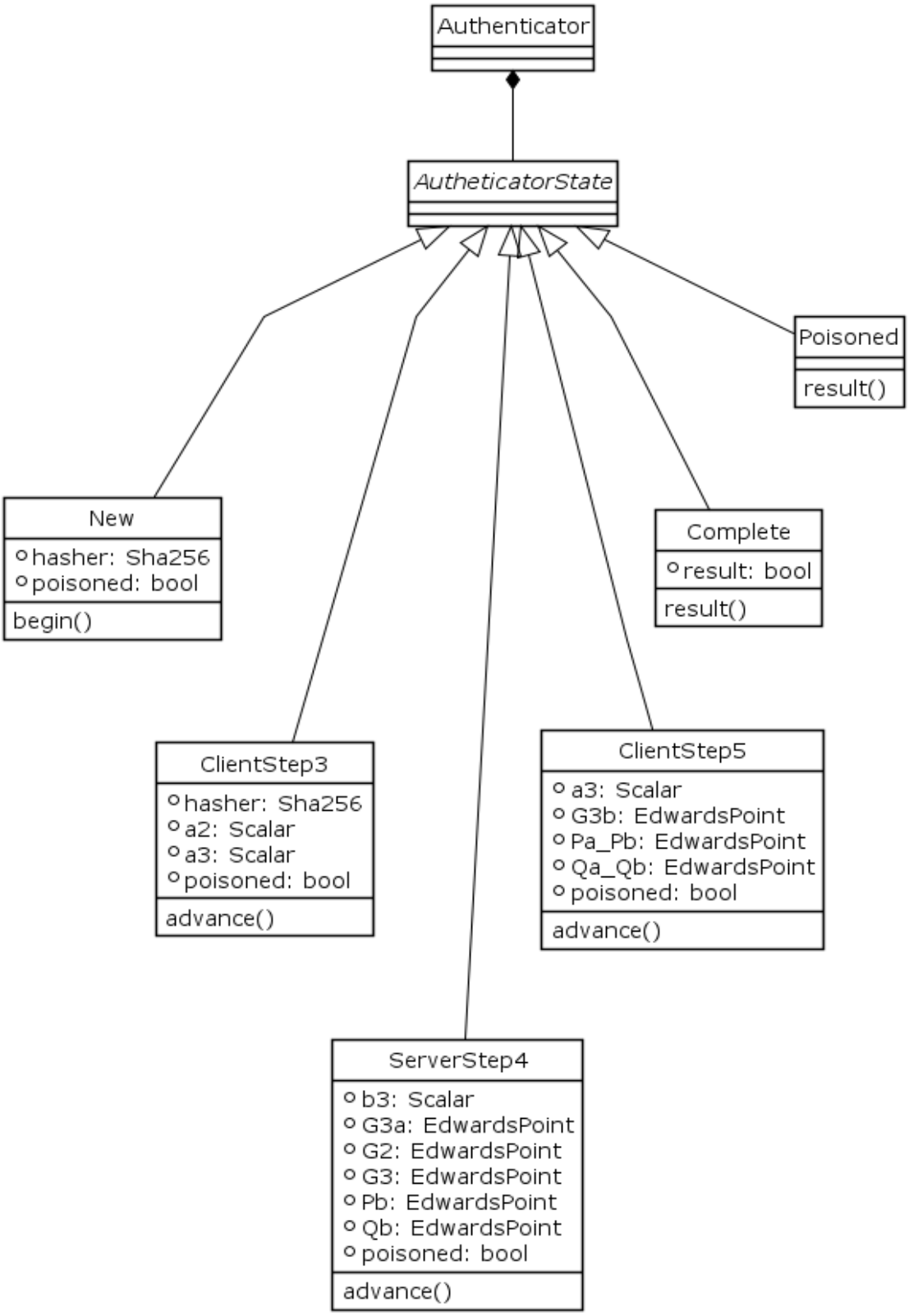
Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Шпильовий Р. П.			**Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою** Функціональна схема (діаграма класів)	Літ. / Аркуш: 1 / Аркушів: 1
Перевірив		Іваніщев Б. В.				
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83
Н. Контр.		Сімоненко В. П.				
Затвердив						

ДОДАТОК З

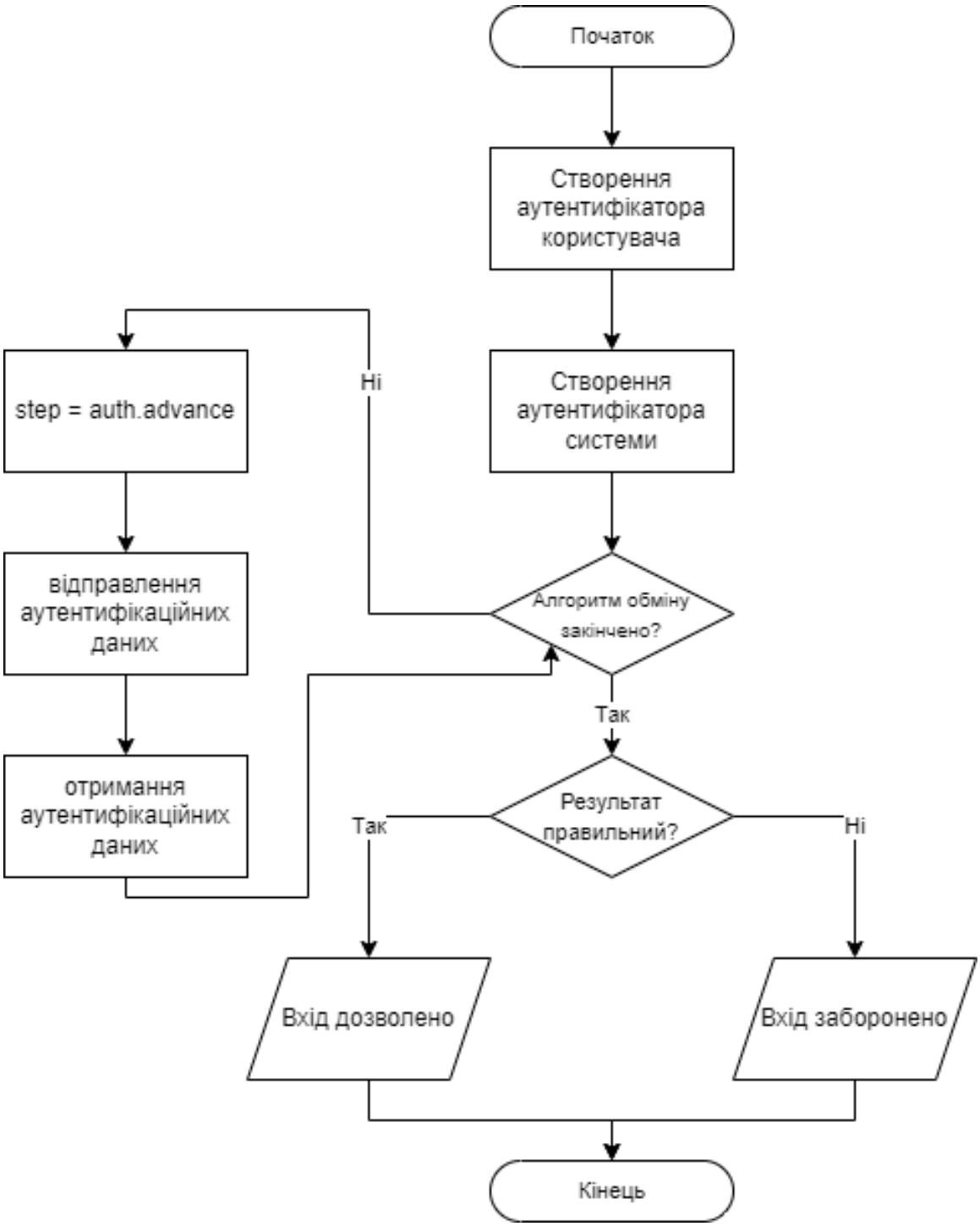
Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Шпильовий Р. П.			**Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою** Алгоритм дій програмного забезпечення	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Іваніщев Б. В.					
Н. Контр.		Сімоненко В. П.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-83	
Затвердив							

ДОДАТОК 4

Сервіс аутентифікації та авторизації для систем з розподіленою архітектурою

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 17

Київ 2022 р

```
//Benchmarks
use blake2::Digest;
use criterion::{criterion_group, criterion_main, Criterion};
use sha2::Sha256;
use zk_auth::{Authenticator, Blake2b256};

fn run_full<D: Digest>(secret1: &[u8], secret2: &[u8]) {
    let mut client = Authenticator::<D>::new();
    let mut server = Authenticator::<D>::new();

    client.push_secret(secret1).unwrap();
    server.push_secret(secret2).unwrap();

    let mut step = client.begin().unwrap();

    loop {
        step = if let Some(step) = server.proceed(&step).unwrap() {
            step
        } else {
            break;
        };
        step = if let Some(step) = client.proceed(&step).unwrap() {
            step
        } else {
            break;
        };
    }
}

fn criterion_benchmark(c: &mut Criterion) {
    {
        let mut sha256 = c.benchmark_group("sha256");
        sha256.bench_function("all-steps-ok", |b| {
            b.iter(|| run_full::<Sha256>(b"HelloWorld", b"HelloWorld"))
        });

        sha256.bench_function("all-steps-err", |b| {
            b.iter(|| run_full::<Sha256>(b"HELLOWORLD", b"HelloWorld"))
        });

        sha256.bench_function("step-1", |b| {
            b.iter(|| {
                let mut client = Authenticator::<Sha256>::new();
                client.begin().unwrap()
            })
        });
```

```
    sha256.bench_function("step-2", |b| {
        let mut client = Authenticator::<Sha256>::new();
        let step1 = client.begin().unwrap();

        b.iter(|| {
            let mut server = Authenticator::<Sha256>::new();
            server.proceed(&step1).unwrap().unwrap()
        })
    });


    sha256.bench_function("step-3", |b| {
        let mut client = Authenticator::<Sha256>::new();
        let mut server = Authenticator::<Sha256>::new();
        let step1 = client.begin().unwrap();
        let step2 = server.proceed(&step1).unwrap().unwrap();

        b.iter(|| {
            let mut client = client.clone();
            client.proceed(&step2).unwrap().unwrap()
        })
    });

    sha256.bench_function("step-4", |b| {
        let mut client = Authenticator::<Sha256>::new();
        let mut server = Authenticator::<Sha256>::new();
        let step1 = client.begin().unwrap();
        let step2 = server.proceed(&step1).unwrap().unwrap();
        let step3 = client.proceed(&step2).unwrap().unwrap();

        b.iter(|| {
            let mut server = server.clone();
            server.proceed(&step3).unwrap().unwrap()
        })
    });
}

{
    let mut blake2 = c.benchmark_group("blake2");

    blake2.bench_function("all-steps-ok", |b| {
        b.iter(|| run_full::<Blake2b256>(b"HelloWorld", b"HelloWorld"))
    });

    blake2.bench_function("all-steps-err", |b| {
        b.iter(|| run_full::<Blake2b256>(b"HELLOWORLD", b"HelloWorld"))
    });

    blake2.bench_function("step-1", |b| {
        b.iter(|| {
            let mut client = Authenticator::<Blake2b256>::new();
            client.begin().unwrap()
        })
    });

    blake2.bench_function("step-2", |b| {
```

```
            let mut client = Authenticator::<Blake2b256>::new();
            let step1 = client.begin().unwrap();

            b.iter(|| {
                let mut server = Authenticator::<Blake2b256>::new();
                server.proceed(&step1).unwrap().unwrap()
            })
        });

        blake2.bench_function("step-3", |b| {
            let mut client = Authenticator::<Blake2b256>::new();
            let mut server = Authenticator::<Blake2b256>::new();
            let step1 = client.begin().unwrap();
            let step2 = server.proceed(&step1).unwrap().unwrap();

            b.iter(|| {
                let mut client = client.clone();
                client.proceed(&step2).unwrap().unwrap()
            })
        });

        blake2.bench_function("step-4", |b| {
            let mut client = Authenticator::<Blake2b256>::new();
            let mut server = Authenticator::<Blake2b256>::new();
            let step1 = client.begin().unwrap();
            let step2 = server.proceed(&step1).unwrap().unwrap();
            let step3 = client.proceed(&step2).unwrap().unwrap();

            b.iter(|| {
                let mut server = server.clone();
                server.proceed(&step3).unwrap().unwrap()
            })
        });
    }
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

```
// Algorithm
#![allow(non_snake_case)]

use anyhow::Context;
use blake2::{Blake2b, Blake2b512};
use curve25519_dalek::{
    constants::ED25519_BASEPOINT_TABLE,
    edwards::{CompressedEdwardsY, EdwardsBasepointTable, EdwardsPoint},
    scalar::Scalar,
    traits::IsIdentity,
};
use rand::Rng;
use sha2::Digest;
use std::{fmt, io, mem};

pub type Blake2b256 = Blake2b<typenum::consts::U32>;

#[derive(Clone, Debug)]
pub struct Authenticator<D: Digest> {
    state: State<D>,
}

#[allow(clippy::large_enum_variant)]
#[derive(Clone)]
enum State<D: Digest> {
    New {
        hasher: D,
        poisoned: bool,
    },
    ClientStep3 {
        hasher: D,
        a2: Scalar,
        a3: Scalar,
        poisoned: bool,
    },
    ServerStep4 {
        b3: Scalar,
        G3a: EdwardsPoint,
        G2: EdwardsPoint,
        G3: EdwardsPoint,
        Pb: EdwardsPoint,
        Qb: EdwardsPoint,
        poisoned: bool,
    },
    ClientStep5 {
        a3: Scalar,
        G3b: EdwardsPoint,
        Pa_Pb: EdwardsPoint,
        Qa_Qb: EdwardsPoint,
        poisoned: bool,
    },
    Complete {
        result: bool,
    },
    Poisoned,
}
```

```
impl<D: Digest> fmt::Debug for State<D> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            State::New { .. } => f.debug_struct("New").finish(),
            State::ClientStep3 { .. } => f.debug_struct("ClientStep3").finish(),
            State::ServerStep4 { .. } => f.debug_struct("ServerStep4").finish(),
            State::ClientStep5 { .. } => f.debug_struct("ClientStep5").finish(),
            State::Complete { .. } => f.debug_struct("Complete").finish(),
            State::Poisoned => f.debug_struct("Poisoned").finish(),
        }
    }
}

impl<D: Digest> Authenticator<D> {
    #[allow(clippy::new_without_default)]
    pub fn new() -> Self {
        Self {
            state: State::New {
                hasher: D::new(),
                poisoned: false,
            },
        }
    }

    /// Collect the data to be compared.
    pub fn push_secret(&mut self, secret: &[u8]) -> anyhow::Result<()> {
        match &mut self.state {
            State::New { hasher, .. } => {
                hasher.update(secret);
                Ok(())
            }
            _ => anyhow::bail!("comparison is already in progress"),
        }
    }

    /// Begin the comparison, producing the first protocol message.
    /// The initiator client calls `begin()` to initiate the comparison.
    pub fn begin(&mut self) -> anyhow::Result<Vec<u8>> {
        match &self.state {
            State::New { .. } => {
                let mut output = vec![0; STEP1_SIZE];
                let len = step1_client_to_server(&mut self.state, &mut output)?;
                debug_assert_eq!(output.len(), len);
                Ok(output)
            }
            _ => anyhow::bail!("inconsistent state"),
        }
    }

    /// Continue comparison with a protocol message.
    /// Call `proceed()` with the message received from your remote peer.
    /// - If you get `None`, comparison is complete.
    /// - If you get `Some(response)`, send `response` to your peer,
    pub fn proceed(&mut self, message: &[u8]) -> anyhow::Result<Option<Vec<u8>>> {
        match &self.state {
            State::New { .. } => {
                let mut output = vec![0; STEP2_SIZE];
```

```
                let len = step2_server_to_client(&mut self.state, message, &mut output)?;
                debug_assert_eq!(output.len(), len);
                Ok(Some(output))
            }
            State::ClientStep3 { .. } => {
                let mut output = vec![0; STEP3_SIZE];
                let len = step3_client_to_server(&mut self.state, message, &mut output)?;
                debug_assert_eq!(output.len(), len);
                Ok(Some(output))
            }
            State::ServerStep4 { .. } => {
                let mut output = vec![0; STEP4_SIZE];
                let len = step4_server_to_client(&mut self.state, message, &mut output)?;
                debug_assert_eq!(output.len(), len);
                Ok(Some(output))
            }
            State::ClientStep5 { .. } => {
                step5_client_is_done(&mut self.state, message)?;
                Ok(None)
            }
            _ => anyhow::bail!("inconsistent state"),
        }
    }

    /// Check if the comparison is complete.
    pub fn is_complete(&self) -> bool {
        matches!(&self.state, State::Complete { .. } | State::Poisoned)
    }

    /// Inspect the comparison result.
    pub fn result(&self) -> anyhow::Result<bool> {
        match &self.state {
            State::Complete { result } => Ok(*result),
            _ => anyhow::bail!("inconsistent state"),
        }
    }
}

const STEP1_SIZE: usize = 2 * (POINT_SIZE + SIGNATURE1_SIZE);

fn step1_client_to_server<D: Digest>(
    state: &mut State<D>,
    mut output: &mut [u8],
) -> anyhow::Result<usize> {
    if output.len() < STEP1_SIZE {
        anyhow::bail!("Buffer too small: {STEP1_SIZE}")
    }
    if let State::New { hasher, poisoned } = mem::replace(state, State::Poisoned) {
        // Nothing interesting is happening on step 1. Client generates
        // a couple of ephemeral scalars and sends public points to Server.
        let a2 = random_scalar()?;
        let a3 = random_scalar()?;
        let G2a = &a2 * G;
        let G3a = &a3 * G;

        let S2a = sign_scalar1(&a2, &[0x01])?;
        let S3a = sign_scalar1(&a3, &[0x02])?;
```

```
        output.write_point(&G2a)?;
        output.write_signature1(&S2a)?;
        output.write_point(&G3a)?;
        output.write_signature1(&S3a)?;

        *state = State::ClientStep3 {
            hasher,
            a2,
            a3,
            poisoned,
        };
        return Ok(STEP1_SIZE);
    }
    unreachable!()
}

const STEP2_SIZE: usize = 2 * (POINT_SIZE + SIGNATURE1_SIZE) + 2 * POINT_SIZE +
SIGNATURE2_SIZE;

fn step2_server_to_client<D: Digest>(
    state: &mut State<D>,
    mut input: &[u8],
    mut output: &mut [u8],
) -> anyhow::Result<usize> {
    if output.len() < STEP2_SIZE {
        anyhow::bail!("Buffer too small: {STEP2_SIZE}")
    }
    if input.len() != STEP1_SIZE {
        *state = State::Poisoned;
        anyhow::bail!("invalid step 1 message")
    }
    if let State::New {
        hasher,
        mut poisoned,
    } = mem::replace(state, State::Poisoned)
    {
        let G2a = input.read_point()?;
        let S2a = input.read_signature1()?;
        let G3a = input.read_point()?;
        let S3a = input.read_signature1()?;

        if !verify_scalar1(&S2a, &G2a, &[0x01])? {
            poisoned = true;
        }
        if !verify_scalar1(&S3a, &G3a, &[0x02])? {
            poisoned = true;
        }

        let b2 = random_scalar()?;
        let b3 = random_scalar()?;
        let G2b = &b2 * G;
        let G3b = &b3 * G;

        let G2 = b2 * G2a;
        let G3 = b3 * G3a;
```

```
        if G2.is_identity() || G3.is_identity() {
            poisoned = true;
        }

        let y = secret_scalar(hasher)?;
        let r = random_scalar()?;
        let Pb = r * G3;
        let Qb = y * G2 + &r * G;

        let S2b = sign_scalar1(&b2, &[0x03])?;
        let S3b = sign_scalar1(&b3, &[0x04])?;
        let SQb = sign_scalar2(&r, &y, &G2, &G3, &[0x05])?;

        output.write_point(&G2b)?;
        output.write_signature1(&S2b)?;
        output.write_point(&G3b)?;
        output.write_signature1(&S3b)?;
        output.write_point(&Pb)?;
        output.write_point(&Qb)?;
        output.write_signature2(&SQb)?;

        *state = State::ServerStep4 {
            b3,
            G3a,
            G2,
            G3,
            Pb,
            Qb,
            poisoned,
        };
        return Ok(STEP2_SIZE);
    }
    unreachable!()
}

const STEP3_SIZE: usize = 2 * POINT_SIZE + SIGNATURE2_SIZE + POINT_SIZE + SIGNATURE1_SIZE;

fn step3_client_to_server<D: Digest>(
    state: &mut State<D>,
    mut input: &[u8],
    mut output: &mut [u8],
) -> anyhow::Result<usize> {
    if output.len() < STEP3_SIZE {
        anyhow::bail!("Buffer too small: {STEP3_SIZE}")
    }
    if input.len() != STEP2_SIZE {
        *state = State::Poisoned;
        anyhow::bail!("invalid step 2 message")
    }
    if let State::ClientStep3 {
        hasher,
        a2,
        a3,
        mut poisoned,
    } = mem::replace(state, State::Poisoned)
    {
        let G2b = input.read_point()?;
```

```
        let S2b = input.read_signature1()?;
        let G3b = input.read_point()?;
        let S3b = input.read_signature1()?;
        let Pb = input.read_point()?;
        let Qb = input.read_point()?;
        let SQb = input.read_signature2()?;

        if !verify_scalar1(&S2b, &G2b, &[0x03])? {
            poisoned = true;
        }
        if !verify_scalar1(&S3b, &G3b, &[0x04])? {
            poisoned = true;
        }
        let G2 = a2 * G2b;
        let G3 = a3 * G3b;

        if G2.is_identity() || G3.is_identity() {
            poisoned = true;
        }

        if !verify_scalar2(&SQb, &Pb, &Qb, &G2, &G3, &[0x05])? {
            poisoned = true;
        }

        let x = secret_scalar(hasher)?;
        let s = random_scalar()?;
        let Pa = s * G3;
        let Qa = x * G2 + &s * G;

        let Pa_Pb = Pa - Pb;
        let Qa_Qb = Qa - Qb;
        let Ra = a3 * Qa_Qb;

        let SQa = sign_scalar2(&s, &x, &G2, &G3, &[0x06])?;
        let SRa = sign_point1(&a3, &Qa_Qb, &[0x07])?;

        output.write_point(&Pa)?;
        output.write_point(&Qa)?;
        output.write_signature2(&SQa)?;
        output.write_point(&Ra)?;
        output.write_signature1(&SRa)?;

        *state = State::ClientStep5 {
            a3,
            G3b,
            Pa_Pb,
            Qa_Qb,
            poisoned,
        };
        return Ok(STEP3_SIZE);
    }
    unreachable!()
}

const STEP4_SIZE: usize = POINT_SIZE + SIGNATURE1_SIZE;

fn step4_server_to_client<D: Digest>(
```

```
    state: &mut State<D>,
    mut input: &[u8],
    mut output: &mut [u8],
) -> anyhow::Result<usize> {
    if output.len() < STEP4_SIZE {
        anyhow::bail!("Buffer too small: {STEP4_SIZE}")
    }
    if input.len() != STEP3_SIZE {
        *state = State::Poisoned;
        anyhow::bail!("invalid step 3 message")
    }
    if let State::ServerStep4 {
        b3,
        G3a,
        G2,
        G3,
        Pb,
        Qb,
        mut poisoned,
    } = mem::replace(state, State::Poisoned)
    {
        let Pa = input.read_point()?;
        let Qa = input.read_point()?;
        let SQa = input.read_signature2()?;
        let Ra = input.read_point()?;
        let SRa = input.read_signature1()?;

        if !verify_scalar2(&SQa, &Pa, &Qa, &G2, &G3, &[0x06])? {
            poisoned = true;
        }

        let Pa_Pb = Pa - Pb;
        let Qa_Qb = Qa - Qb;

        if !verify_point1(&SRa, &G3a, &Ra, &Qa_Qb, &[0x07])? {
            poisoned = true;
        }

        let Rb = b3 * Qa_Qb;
        let Rab = b3 * Ra;

        let result = Rab == Pa_Pb;

        let SRb = sign_point1(&b3, &Qa_Qb, &[0x08])?;

        output.write_point(&Rb)?;
        output.write_signature1(&SRb)?;

        *state = if poisoned {
            State::Poisoned
        } else {
            State::Complete { result }
        };
        return Ok(STEP4_SIZE);
    }
    unreachable!()
}
```

```
fn step5_client_is_done<D: Digest>(state: &mut State<D>, mut input: &[u8]) ->
anyhow::Result<()> {
    if input.len() != STEP4_SIZE {
        *state = State::Poisoned;
        anyhow::bail!("invalid step 4 message")
    }
    if let State::ClientStep5 {
        a3,
        G3b,
        Pa_Pb,
        Qa_Qb,
        mut poisoned,
    } = mem::replace(state, State::Poisoned)
    {
        let Rb = input.read_point()?;
        let SRb = input.read_signature1()?;

        if !verify_point1(&SRb, &G3b, &Rb, &Qa_Qb, &[0x08])? {
            poisoned = true;
        }

        let Rab = a3 * Rb;

        let result = Rab == Pa_Pb;

        *state = if poisoned {
            State::Poisoned
        } else {
            State::Complete { result }
        };
        return Ok(());
    }
    unreachable!()
}

const SCALAR_SIZE: usize = 32;

fn clamp_scalar(scalar: &mut [u8]) {
    scalar[0x00] &= 0b_1111_1000;
    scalar[0x1F] &= 0b_0111_1111;
    scalar[0x1F] |= 0b_0100_0000;
}

fn secret_scalar<D: Digest>(hasher: D) -> anyhow::Result<Scalar> {
    let mut scalar: [u8; 32] = hasher
        .finalize()
        .as_slice()
        .try_into()
        .expect("length should be 32 bytes");
    clamp_scalar(&mut scalar);
    Ok(Scalar::from_bits(scalar))
}

fn random_scalar() -> anyhow::Result<Scalar> {
    let mut scalar: [u8; SCALAR_SIZE] = rand::thread_rng().gen();
    clamp_scalar(&mut scalar);
```

```
    Ok(Scalar::from_bits(scalar))
}

const G: &EdwardsBasepointTable = &ED25519_BASEPOINT_TABLE;

struct Signature1 {
    c: Scalar,
    d: Scalar,
}

fn sign_scalar1(s: &Scalar, i: &[u8]) -> anyhow::Result<Signature1> {
    let r = &random_scalar()?;
    let W = r * G;
    let c = reduce_1_point_hash_into_scalar(&W, i)?;
    let d = (c * s + r).reduce();
    Ok(Signature1 { c, d })
}

fn verify_scalar1(S: &Signature1, P: &EdwardsPoint, i: &[u8]) -> anyhow::Result<bool> {
    let Signature1 { c, d } = S;
    if !c.is_canonical() || !d.is_canonical() {
        return Ok(false);
    }
    let W_prime = d * G - c * P;
    let c_prime = reduce_1_point_hash_into_scalar(&W_prime, i)?;
    if *c != c_prime {
        return Ok(false);
    }
    Ok(true)
}

fn reduce_1_point_hash_into_scalar(W: &EdwardsPoint, i: &[u8]) -> anyhow::Result<Scalar> {
    let mut hasher = Blake2b512::new();
    hasher.update(W.compress().as_bytes());
    hasher.update(i);
    let bytes = hasher.finalize();
    Ok(Scalar::from_bytes_mod_order_wide(bytes.as_ref()))
}

struct Signature2 {
    c: Scalar,
    d1: Scalar,
    d2: Scalar,
}

fn sign_scalar2(
    s1: &Scalar,
    s2: &Scalar,
    B1: &EdwardsPoint,
    B2: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<Signature2> {
    let r1 = &random_scalar()?;
    let r2 = &random_scalar()?;
    let W1 = r1 * B2;
    let W2 = r2 * B1 + r1 * G;
    let c = reduce_2_point_hash_into_scalar(&W1, &W2, i)?;
```

```
    let d1 = (c * s1 + r1).reduce();
    let d2 = (c * s2 + r2).reduce();
    Ok(Signature2 { c, d1, d2 })
}

fn verify_scalar2(
    S: &Signature2,
    P1: &EdwardsPoint,
    P2: &EdwardsPoint,
    B1: &EdwardsPoint,
    B2: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<bool> {
    let Signature2 { c, d1, d2 } = S;
    if !c.is_canonical() || !d1.is_canonical() || !d2.is_canonical() {
        return Ok(false);
    }
    let W1_prime = d1 * B2 - c * P1;
    let W2_prime = d2 * B1 + d1 * G - c * P2;
    let c_prime = reduce_2_point_hash_into_scalar(&W1_prime, &W2_prime, i)?;
    if *c != c_prime {
        return Ok(false);
    }
    Ok(true)
}

fn reduce_2_point_hash_into_scalar(
    W1: &EdwardsPoint,
    W2: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<Scalar> {
    let mut hasher = Blake2b512::new();
    hasher.update(W1.compress().as_bytes());
    hasher.update(W2.compress().as_bytes());
    hasher.update(i);
    let bytes = hasher.finalize();
    Ok(Scalar::from_bytes_mod_order_wide(bytes.as_ref()))
}

fn sign_point1(s: &Scalar, B: &EdwardsPoint, i: &[u8]) -> anyhow::Result<Signature1> {
    let r = &random_scalar()?;
    let W1 = r * G;
    let W2 = r * B;
    let c = reduce_2_point_hash_into_scalar(&W1, &W2, i)?;
    let d = (c * s + r).reduce();
    Ok(Signature1 { c, d })
}

fn verify_point1(
    S: &Signature1,
    P1: &EdwardsPoint,
    P2: &EdwardsPoint,
    B: &EdwardsPoint,
    i: &[u8],
) -> anyhow::Result<bool> {
    let Signature1 { c, d } = S;
    if !c.is_canonical() || !d.is_canonical() {
```

```
        return Ok(false);
    }
    let W1_prime = d * G - c * P1;
    let W2_prime = d * B - c * P2;
    let c_prime = reduce_2_point_hash_into_scalar(&W1_prime, &W2_prime, i)?;
    if *c != c_prime {
        return Ok(false);
    }
    Ok(true)
}

const POINT_SIZE: usize = 32;
const SIGNATURE1_SIZE: usize = 2 * SCALAR_SIZE;
const SIGNATURE2_SIZE: usize = 3 * SCALAR_SIZE;

trait Write: io::Write {
    fn write_scalar(&mut self, s: &Scalar) -> anyhow::Result<()> {
        self.write_all(s.as_bytes())
            .context("failed to write scalar")?;
        Ok(())
    }

    fn write_point(&mut self, P: &EdwardsPoint) -> anyhow::Result<()> {
        self.write_all(P.compress().as_bytes())
            .context("failed to write point")?;
        Ok(())
    }

    fn write_signature1(&mut self, S: &Signature1) -> anyhow::Result<()> {
        self.write_scalar(&S.c)
            .context("failed to write signature")?;
        self.write_scalar(&S.d)
            .context("failed to write signature")?;
        Ok(())
    }

    fn write_signature2(&mut self, S: &Signature2) -> anyhow::Result<()> {
        self.write_scalar(&S.c)
            .context("failed to write signature")?;
        self.write_scalar(&S.d1)
            .context("failed to write signature")?;
        self.write_scalar(&S.d2)
            .context("failed to write signature")?;
        Ok(())
    }
}

impl Write for &mut [u8] {}

trait Read: io::Read {
    fn read_scalar(&mut self) -> anyhow::Result<Scalar> {
        let mut scalar = [0; SCALAR_SIZE];
        self.read_exact(&mut scalar)
            .context("failed to read scalar")?;
        Ok(Scalar::from_bits(scalar))
    }
```

```
    fn read_point(&mut self) -> anyhow::Result<EdwardsPoint> {
        let mut point = [0; POINT_SIZE];
        self.read_exact(&mut point)
            .context("failed to read point")?;
        let point = CompressedEdwardsY(point)
            .decompress()
            .context("invalid Curve25519 point")?;
        Ok(point)
    }

    fn read_signature1(&mut self) -> anyhow::Result<Signature1> {
        let c = self.read_scalar().context("failed to read signature")?;
        let d = self.read_scalar().context("failed to read signature")?;
        Ok(Signature1 { c, d })
    }

    fn read_signature2(&mut self) -> anyhow::Result<Signature2> {
        let c = self.read_scalar().context("failed to read signature")?;
        let d1 = self.read_scalar().context("failed to read signature")?;
        let d2 = self.read_scalar().context("failed to read signature")?;
        Ok(Signature2 { c, d1, d2 })
    }
}

impl Read for &[u8] {}

#[cfg(test)]
mod tests {
    use super::*;
    use sha2::Sha256;

    #[test]
    fn test_sha256_ok() {
        let mut client = Authenticator::<Sha256>::new();
        let mut server = Authenticator::<Sha256>::new();

        const SECRET: &[u8] = b"SECRET";

        client.push_secret(SECRET).unwrap();
        server.push_secret(SECRET).unwrap();

        let mut step = client.begin().unwrap();

        let mut i = 0;
        loop {
            println!(">> {}", i);
            i += 1;
            step = if let Some(step) = server.proceed(&step).unwrap() {
                step
            } else {
                break;
            };
            step = if let Some(step) = client.proceed(&step).unwrap() {
                step
            } else {
                break;
            };
```

```
    }

    assert!(client.result().unwrap());
    assert!(server.result().unwrap());
}

#[test]
fn test_sha256_different() {
    let mut client = Authenticator::<Sha256>::new();
    let mut server = Authenticator::<Sha256>::new();

    client.push_secret(b"secret").unwrap();
    server.push_secret(b"SECRET").unwrap();

    let mut step = client.begin().unwrap();

    let mut i = 0;
    loop {
        println!(">> {}", i);
        i += 1;
        step = if let Some(step) = server.proceed(&step).unwrap() {
            step
        } else {
            break;
        };
        step = if let Some(step) = client.proceed(&step).unwrap() {
            step
        } else {
            break;
        };
    }

    assert!(!client.result().unwrap());
    assert!(!server.result().unwrap());
}

#[test]
fn test_blake2_ok() {
    let mut client = Authenticator::<Blake2b256>::new();
    let mut server = Authenticator::<Blake2b256>::new();

    const SECRET: &[u8] = b"SECRET";

    client.push_secret(SECRET).unwrap();
    server.push_secret(SECRET).unwrap();

    let mut step = client.begin().unwrap();

    let mut i = 0;
    loop {
        println!(">> {}", i);
        i += 1;
        step = if let Some(step) = server.proceed(&step).unwrap() {
            step
        } else {
            break;
        };
```

```
            step = if let Some(step) = client.proceed(&step).unwrap() {
                step
            } else {
                break;
            };
        }

        assert!(client.result().unwrap());
        assert!(server.result().unwrap());
    }

    #[test]
    fn test_blake2_different() {
        let mut client = Authenticator::<Blake2b256>::new();
        let mut server = Authenticator::<Blake2b256>::new();

        client.push_secret(b"secret").unwrap();
        server.push_secret(b"SECRET").unwrap();

        let mut step = client.begin().unwrap();

        let mut i = 0;
        loop {
            println!(">> {}", i);
            i += 1;
            step = if let Some(step) = server.proceed(&step).unwrap() {
                step
            } else {
                break;
            };
            step = if let Some(step) = client.proceed(&step).unwrap() {
                step
            } else {
                break;
            };
        }

        assert!(!client.result().unwrap());
        assert!(!server.result().unwrap());
    }
}
```