Галицький фаховий коледж імені В’ячеслава Чорновола

відділення комп’ютерних технологій

циклова комісія інформатики та комп’ютерних дисциплін

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач відділенням
комп’ютерних технологій
Наталія СТЕФУРАК _____
підпис
«___» _____ 2022 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломного проєкту
освітньо-кваліфікаційного рівня «молодший спеціаліст»
зі спеціальності 122 «Комп’ютерні науки»

на тему: «Програма шифрування та синхронізації даних в комп’ютерній мережі»

Студент групи КН-41 Касіян М.І. _____
(підпис)

Керівник проєкту Сиротюк Н.С. _____
(підпис)

Консультанти:

з техніко-економічного обґрунтування Меленчук Л.І. _____
(підпис)

нормоконтролер Кульчинська Н.З. _____
(підпис)

Тернопіль – 2023

Галицький фаховий коледж імені В’ячеслава Чорновола
відділення комп’ютерних технологій
циклова комісія інформатики та комп’ютерних дисциплін

ЗАТВЕРДЖУЮ
Завідувач відділенням
комп’ютерних технологій
Наталія СТЕФУРАК _____
підпис
«___» _____ 2022 р.

ЗАВДАННЯ

на дипломне проєктування
на здобуття освітньо-кваліфікаційного рівня «молодший спеціаліст»
студенту Касіяну Максиму Ігоровичу
(прізвище, ім’я та по-батькові студента)

1. Тема проєкту _____

затверджена наказом по коледжу від “__” _____ 2022 р., № _____

2. Термін здачі студентом завершеного проєкту “___” _____2023 р.
3. Вихідні дані до проєкту _____

4. Перелік питань, які повинні бути розроблені в проєкті:
а) основна частина _____

б) техніко-економічне обґрунтування _____

5. Перелік графічного матеріалу _____

6. Консультанти проєкту:_____

Розділ	Консультанти	Підпис, дата	
		Завдання видано	Завдання прийнято
з техніко-економічного обґрунтування	_____ (вчена ступень, звання П.І.Б. _____ консультанта)		

КАЛЕНДАРНИЙ ПЛАН
дипломного проєктування

№ п/п	Найменування етапу	Терміни	
		початку	завершення
1	Вибір та затвердження теми	24.11.2022р	15.12.2022р
2	Огляд існуючих рішень, написання розділу ПЗ	15.01.2023р	02.02.2023р
3	Аналіз проблем безпеки, рішення та написання розділу ПЗ	02.02.2023р	09.02.2023р
4	Розробка вимог до проєкту, написання розділу ПЗ	10.02.2023р	16.02.2023р
5	Проєктування процесів продукту, написання розділу ПЗ	16.02.2023р	04.03.2023р
6	Встановлення середовища реалізації, написання розділу ПЗ	04.03.2023р	05.03.2023р
7	Реалізація програмного засобу	05.03.2023р	30.03.2023р
8	Розробка тестових процедур для продукту, написання розділу ПЗ	31.03.2023р	10.04.2023р
9	Проведення технічно-економічного обґрунтування проєкту, написання розділу ПЗ	11.04.2023р	07.05.2023р
10	Робота над оформленням та виправкою змісту ПЗ	07.05.2023р	14.06.2023р
11	Попередній захист проєкту, проходження нормо контролю	15.06.2023р	
12	Підготовка до захисту дипломного проєкту	17.06.2023р	21.06.2023р
13	Захист дипломного проєкту	27.06.2023р	
14			
15			

7. Дата видачі завдання “___” _____ 2022 р.
Керівник _____

Завдання прийняв до виконання _____

Реферат

Тема дипломного проєкту: «Програма шифрування та синхронізації даних в комп’ютерній мережі». Звіт містить 67 сторінок, 18 рисунків, 7 таблиць, 8 джерел.

Об’єкт дослідження – методи синхронізації даних, шифрування даних. Мета дипломного проєкту є ефективне перенесення даних на мережі, використовуючи мінімальні ресурси мережі та дотримання безпеки інформації шляхом шифрування даних. Було розглянуто алгоритми шифрування, режими операцій шифрів, мережеві протоколи. Результатом є програмний продукт, з функціоналом шифрування в спокої та синхронізації даних на локальній мережі з можливістю автоматизації.

Створення продукту було виконано у середовищі Visual Studio Code. Середовище є крос-платформним з підтримкою різних мов. Програма реалізована на мовах програмування на Rust та Python для основного та графічного компоненту відповідно, для більшої підтримки зовнішніх компонентів, високі вимоги до безпеки коду та зростаючої популярності на ринку.

Програмний засіб можливо використовувати для синхронізації даних на локальній мережі, шифрувати та дешифрувати дані використовуючи алгоритм AES.

ШИФРУВАННЯ ДАНИХ, СИНХРОНІЗАЦІЯ ФАЙЛІВ, RUST, СИМЕТРИЧНЕ ШИФРУВАННЯ.

Abstract

The topic of diploma project: “Data encryption and synchronization on a computer network software”. Project contains of 67 pages, 18 images, 7 tables and 8 sources.

Object of study – Data synchronization methods, encryption. The goal is efficient data transfer across a network with minimal resources while preserving confidentiality of data using encryption. Encryption algorithms, their mode of operation and network protocols were researched during development. As a result of this project there is a software solution for efficient file synchronization with encryption capabilities at rest.

Development of this software was done in Visual Studio Code. Environment is cross-platform and supports multiple programming languages. The program is implemented in Rust and Python languages, for main and graphical component respectively; for larger support for external components, stricter safety during implementation and an increasing demand on the market.

Software solution can be used for transferring files over a local network, and encryption or decryption of data using the AES algorithm.

DATA ENCRYPTION, FILE SYNCHRONIZATION, RUST, SYMMETRIC ENCRYPTION.

ЗМІСТ

ВСТУП

З зростаючою важливістю інтернету та персональних пристроїв в нашому житті, також зростає необхідність забезпечення безпеки під час роботи в мережі. В «цифрову епоху» люди зберігають все – від простих нотаток, до фото сім'ї та родичів, яких вони не бачили десятки років, до важливих робочих та власних документів – суто на хмарних сервісах або на одному пристрої.

Це зручно, але і небезпечно – в поточних умовах, ми можемо втратити наші пристрої в момент, а сервіси можуть відмовити користувачу у доступі до його даних; і тоді що? Навіть якщо відмова сервісу часом буває для кращого добра, скільки з нас залишило сліди на, наприклад, російських соціальних мережах, поштових та інших сервісах, які тепер не можливо ні завантажити, ні видалити без використання сервісів VPN і надіятися та шукати інший сервіс, який надасть можливість доступу до заблокованих додатків.

Як рішення для раніше згаданих проблем щодо дотримання наших даних є розробка додатку, що забезпечуватиме доступність та захист даних на різних пристроях, під'єднаних до робочої станції. Це може бути як резервні ПК, телефони та інші переносні носії (USB та SD накопичувачі).

Попри доступність, користувачам необхідна безпека – зростаюча кількість кібер-нападів на українські фірми та бази даних, найвідомішим прикладом є напад та витік даних з компанії розробки ігор GSC Game World, привертає увагу до того, щоб зловмисники не мали можливість доступу до інформації, яка є в користувача. При цьому варто згадати, що про безпеку треба дбати як при роботі з глобальними, так і з локальними сервісами, оскільки при роботі з даними в приватній Wi-Fi мережі, де один з пристроїв скомпрометований – всі дані, які проходили мережею, є під ризиком несанкціонованого доступу.

Метою дипломного проєкту є дослідження механізмів безпеки даних, проєктування та розробка програмного забезпечення для синхронізації

файлів з іншими пристроями з високим рівнем безпеки передачі даних по мережі.

Завданнями дипломного проєктування є:

1) дослідження та аналіз методів синхронізації даних при передачі їх мережею;

2) аналіз методів безпеки, вжитими подібними рішеннями та їх вразливості;

3) визначення механізмів безпеки, згідно аналізу вразливостей;

4) проєктування алгоритмів;

5) реалізація програмного продукту;

6) розробка тестових функцій для забезпечення стабільності функціонування продукту;

7) техніко-економічне обґрунтування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАНЬ

Аналіз предметної області є першим етапом у дослідженнях, розробці проєктів або вирішенні проблем, пов'язаних з певним предметом або сферою діяльності. Цей процес спрямований на отримання глибокого розуміння предметної області, його структури, функціонування та основних проблем.

Перший крок у процесі аналізу предметної області - це зібрати необхідні дані та інформацію. Це можуть бути існуючі рішення, статистичні дані та інші джерела, які відображають стан предметної області. Збір даних може проводитися шляхом вивчення попередніх досліджень та аналізу статистичних даних.

Після збору інформації слід провести її аналіз і систематизацію. Систематизація даних передбачає їх групування, класифікацію та організацію у логічну структуру, що дозволить краще розуміти взаємозв'язки та властивості.

Наступним етапом є ідентифікація основних проблем та викликів, пов'язаних з предметною областю. Це вимагає аналізу даних та виявлення ключових причин, які спричиняють виникнення цих проблем. Зазвичай цей етап включає в себе вивчення різних підходів, теорій та концепцій, пов'язаних з предметною областю, з метою зрозуміти їх застосування та ефективність у вирішенні проблем.

Після ідентифікації проблем слід провести глибинний аналіз причин, що лежать в їх основі. Метою цього аналізу є з'ясування кореневих причин проблем і визначення можливих шляхів вирішення.

На останньому етапі аналізу виводяться висновки та тези на основі отриманих знань через аналіз. Це може бути від загальних цілей до специфічних методів рішення певної проблеми, яка виявилася в аналізі.

1.1 Аналіз існуючих рішень синхронізації

Найлегший метод аналізу – огляд існуючих рішень. Цим методом можливо дізнатися про можливі проблеми з даною областю та як різні

програмні продукти або сервіси вирішують або обминають їх. Для цього, було обрано «звичний» метод через хмару для простих користувачів, в цьому випадку Mega; та два програмних рішення які не залежать від хмари – SyncThing та rsync. В кінці кожного аналізу буде короткий підсумок особливостей, недоліків та іншого.

1.1.1 Аналіз хмарного сервісу Mega

Mega (рисунок 1.1) є хмарним сервісом зберігання та синхронізації файлів, який надає користувачам можливість зберігати, передавати та доступатися до своїх даних через Інтернет. Заснований у 2013 році Кімом Дотком, одним зі співзасновників популярного файло-обмінника Megaupload, Mega був створений з метою надання безпечного та приватного хмарного сховища.

Рисунок 1.1 - Логотип Mega

Однією з основних особливостей Mega є його високий рівень шифрування та безпеки даних. Всі файли, що завантажуються на сервери Mega, автоматично шифруються на стороні клієнта, що означає, що тільки сам користувач має доступ до розшифрованого вмісту. Ключі шифрування також зберігаються на стороні користувача, що забезпечує повну контроль над даними. Це робить Mega одним з найбезпечніших хмарних сервісів, забезпечуючи конфіденційність даних.

Для синхронізації, користувачі можуть встановити спеціальне програмне забезпечення MegaSync на свої комп'ютери або мобільні пристрої, щоб автоматично синхронізувати файли між ними. Це дозволяє користувачам мати оновлені дані на всіх своїх пристроях і зручно керувати своїм контентом.

Одним з інших функціональних елементів Mega є можливість надання спільного доступу до файлів. Користувачі можуть створювати спеціальні посилання або надавати окремі права доступу до своїх файлів і папок іншим користувачам. Це дозволяє спільно працювати над проєктами, обмінюватися даними та спільно використовувати ресурси з колегами або співробітниками.

Крім цього, Mega надає користувачам обмежену кількість безкоштовного простору для зберігання даних, а також преміальні пакети починаючи від 200 грн, які дозволяють збільшити обсяг доступного простору та отримати додаткові функції та переваги.

Щодо інтерфейсу, як і всі веб-сервіси Mega має зрозумілий інтерфейс – на рисунку 1.2 є поданий приклад панелі користувача, де можливо знайти дані про використане місце, трафік, можливість завантаження ключа відновлення та інше.

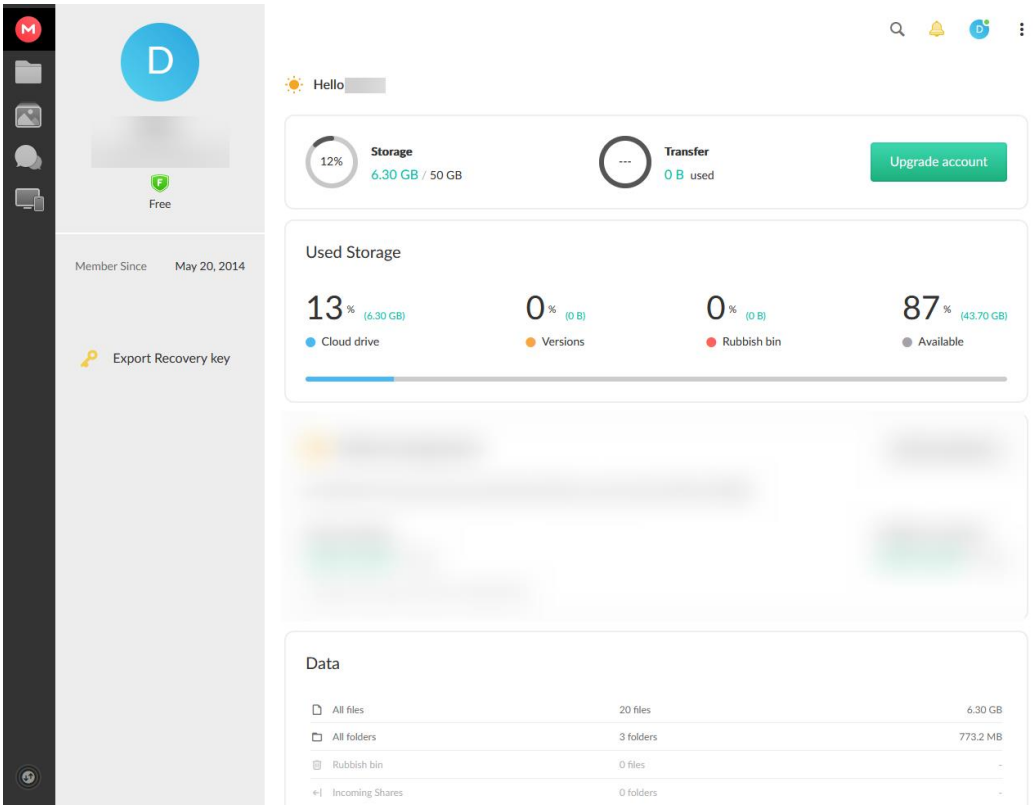

Рисунок 1.2 – Панель користувача в сервісі Mega

Основні елементи меню знаходяться на лівій панелі в порядку панелі користувача, файлів, зображення, чат та резервні копії пристроїв, з поточними завантаженнями та вивантаженням в нижньому лівому кутку.

Mega має функцію чату та відео розмови. Ми вважаємо це недоліком, оскільки це ресурси затрачені на функціонал, який не причетний до основної сфери де цей сервіс працює.

З налаштувань, максимум що можливо зробити що пов’язане з безпекою або даними це видалення облікового запису, вмикання багато-факторної аутентифікації, та зміна паралельності обмінів (до 6 файлів водночас).

Узагальнюючи, Mega є напевно кращим хмарним сервісом, який надає безпечне та приватне зберігання даних. Високий рівень шифрування, можливість синхронізації файлів та спільного доступу роблять Mega популярним серед користувачів, які цінують захист своїх даних та зручну роботу з файлами у хмарному середовищі.

У використанні цього хмарного сервісу є наступні переваги:

– шифрування з боку клієнта,
– доступний, безкоштовний доступ до 20ГБ місця;
– код доступний для огляду клієнтами.

Недоліки:

– можливо втратити доступ різними методами (забув пароль та ключ, заблокування від доступу провайдером/сайтом),
– платний для більших файлів,
– щоденне обмеження на завантаження у розмірі 5ГБ,
– зайвий функціонал чатів.

З цього, ми можемо взяти висновки щодо шифрування, доступного коду та оминати розробку зайвих функцій, непричетних до мети.

1.1.2 Аналіз децентралізованого рішення Syncthing

Syncthing (рисунок 1.3) є програмним забезпеченням написаним на Go для синхронізації файлів між різними пристроями через локальну мережу або Інтернет. Воно забезпечує безпечний обмін даними і дозволяє користувачам зберігати файли в синхронізованих папках на різних пристроях, забезпечуючи їх автоматичну актуалізацію та взаємодію.

Рисунок 1.3 – Логотип Syncthing

Архітектура Syncthing базується на принципі децентралізованої мережі, де кожен пристрій виступає як рівноправний вузол. Використовуючи протокол P2P (Peer-to-Peer), програма дозволяє безпосередньо обмінюватися даними між пристроями, уникнувши проміжних серверів (якщо обидва пристрої можуть з’єднатися напряму, в іншому випадку реляційні сервери вживаються) або хмарних систем. Це забезпечує конфіденційність, безпеку та повний контроль користувача над своїми даними.

Однією з головних особливостей Syncthing є його гнучкість та налаштування. Користувачі можуть налаштувати специфічні папки для синхронізації, вказати правила фільтрації файлів, налаштувати режими спільної роботи та контролю доступу. Також є можливість налаштування пропускної здатності та розкладу синхронізації, що дає користувачам повний контроль над процесом обміну даними.

Syncthing використовує шифрування для забезпечення безпеки даних. Кожен файл, який передається по мережі, шифрується за допомогою протоколу TLS (Transport Layer Security), що забезпечує захищену передачу інформації. Крім того, кожен вузол має унікальний ідентифікатор, що

дозволяє перевіряти автентичність пристроїв та запобігати несанкціонованому доступу до даних.

Syncthing є кросплатформним програмним забезпеченням, що підтримує різні операційні системи, включаючи Windows, macOS, Linux та Android. Це робить його доступним для використання на різних типах пристроїв, від персональних комп'ютерів до мобільних пристроїв.

Простого інтерфейсу у Syncthing поза межами мобільних додатків немає; в основному, всі налаштування відбуваються через веб-сторінку (рисунок 1.4)

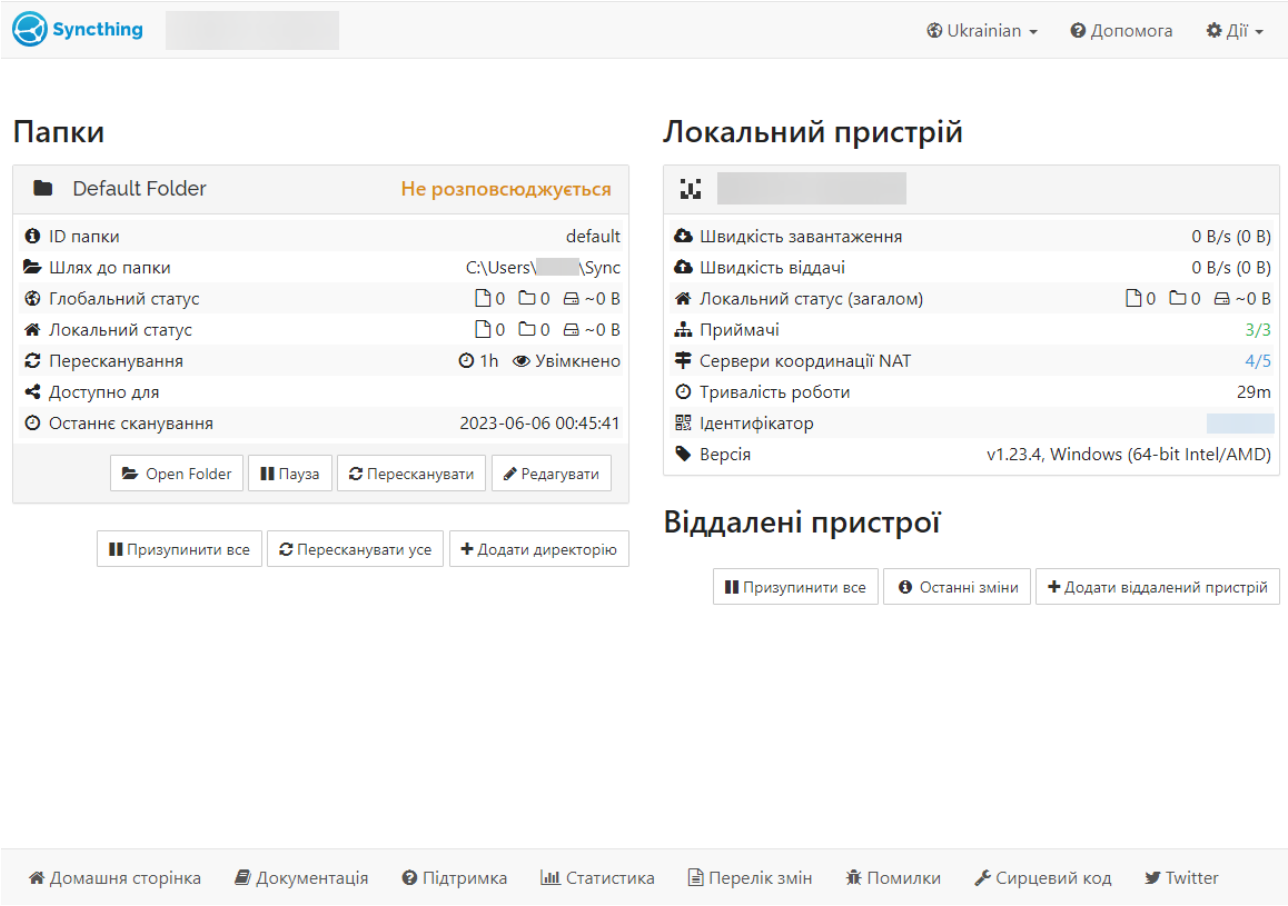

Рисунок 1.4 – Головна панель Syncthing

Перше налаштування для нового користувача може бути складним. Процес додавання віддалених пристроїв є простим за допомогою 6-значних ідентифікаторів; але під час додавання необхідно підтверджувати з’єднання, яке в великих масштабах може зайняти багато часу. Але оскільки це пов’язано з безпекою це не вважається недоліком. Але налаштування нової

папки відразу проявляє недолік гнучкості; надаючи новим користувачам легкий доступ до набору налаштувань, поданому на рисунку 1.5 частіше всього відразу заплутає їх від подальшого користування.

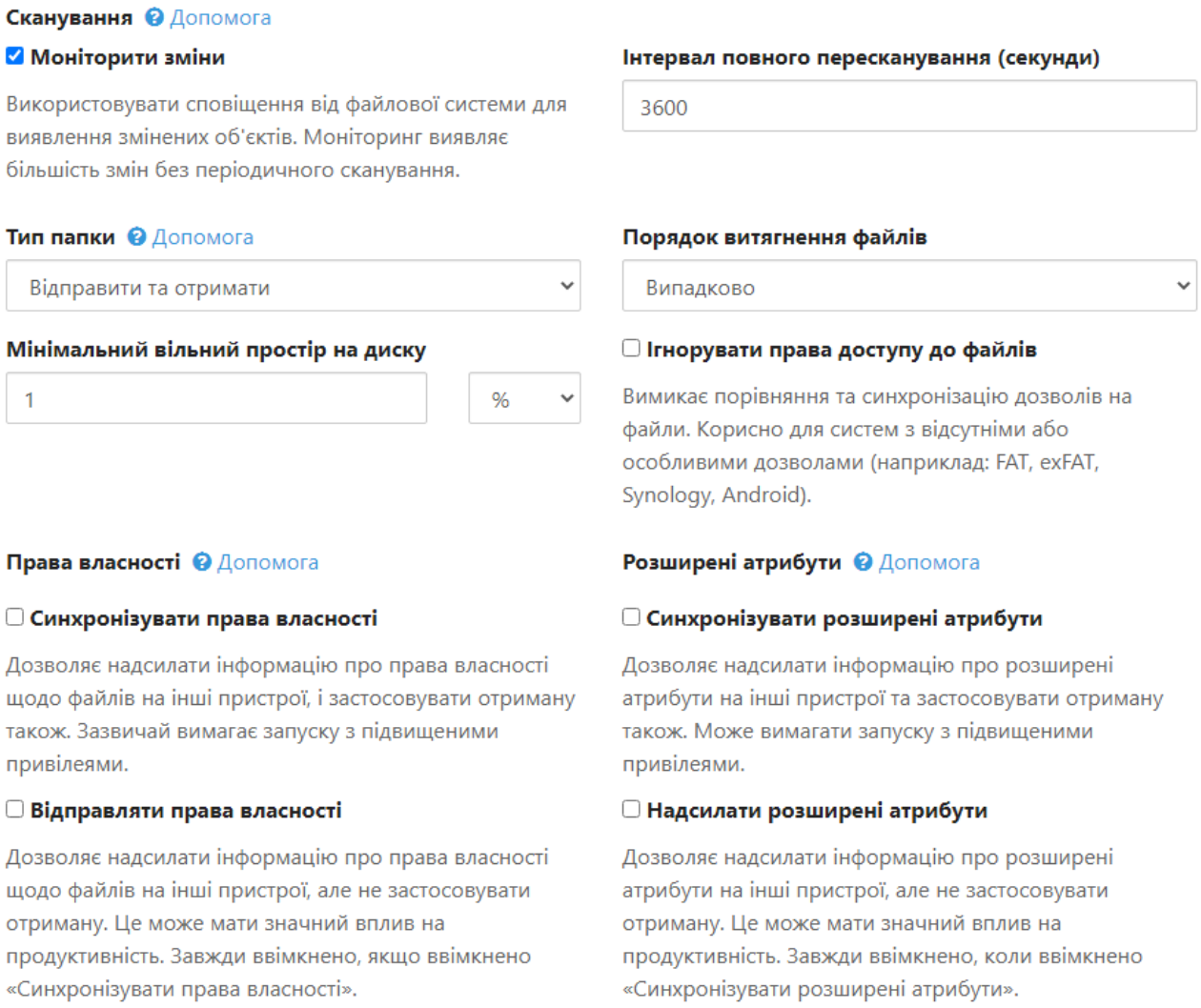

Рисунок 1.5 – Набір можливих налаштувань папок в Syncthing

Узагальнюючи, Syncthing є потужним інструментом для синхронізації файлів, який надає децентралізований підхід, захищену передачу даних та гнучкості. Його використання сприяє безпечному обміну файлами між різними пристроями, що робить його популярним серед користувачів, які цінують конфіденційність і контроль над своїми даними.

Переваги цього продукту:

– повністю відкритий код,

– захищений через TLS,

– просте додавання пристроїв,

– кросплатформний.

Недоліками є наступні проблеми -

– веб-інтерфейс вимагає браузер, зайві ресурси

– складний для нового користувача

За результатами аналізу визначено наступні особливості даного програмного продукту:

1) Децентралізований – при зв’язанні по мережі інтернет, пристрої використовують один з багатьох реляційних серверів для з’єднання. По бажанню, можливо вживати власний сервер для додаткової ізоляції.

2) Дійсно гнучкий – керування версіями, передача прав власності та інших атрибутів, регулювання папки на лише відправляти або отримувати на пристроях може створити повноцінну мережу з різними правами доступу.

З вище переліченого, для продукту виноситься необхідність бути кросплатформною та не вживати веб інтерфейс, з увагою на простоту використання.

1.1.3 Аналіз Linux-утиліти rsync

Утиліта rsync на мові С є більше інструментом ніж рішенням для простих користувачів; для ефективної синхронізації і передачі файлів між різними комп'ютерами чи пристроями. Воно використовує алгоритм, що забезпечує інкрементальну передачу даних, що означає, що лише змінені частини файлів передаються, зменшуючи трафік мережі та зберігаючи час.

Rsync працює на власному протоколі передачі файлів через мережу, який відрізняється своїми унікальними особливостями; утиліта використовує алгоритм блочного переносу даних, де кожен файл розбивається на блоки, а передача відбувається лише для змінених або нових блоків. Це дозволяє значно зменшити обсяг передачі даних і прискорити процес синхронізації.

Rsync також відомий своїми можливостями з компресії та шифрування даних. Він підтримує компресію даних за допомогою zlib, що дозволяє

зменшити розмір переданих файлів і зменшити навантаження на мережу. Крім того, rsync може працювати через захищені канали за допомогою протоколу Secure Shell (SSH), що забезпечує шифрування передачі даних та захист від несанкціонованого доступу.

Іншою важливою особливістю rsync є його здатність відновити передачу даних. Якщо передача файлів була перервана через втрату з'єднання або інші проблеми, rsync здатний продовжити передачу з того місця, де вона була припинена, замість повного повторного копіювання файлу.

Інтерфейсу як такого не існує – rsync працює лише на командній стрічці. Це є не зручним для простого користувача, але дане рішення не було заплановано для них, а більше на системних адміністраторів, які автоматизують синхронізацію.

Підсумовуючи, rsync є потужним інструментом для ефективної синхронізації файлів між різними комп'ютерами. Він використовує власний алгоритм для блочної передачі даних, підтримує компресію та шифрування, забезпечує резюмовану передачу та працює на різних платформах. Rsync є популярним інструментом серед адміністраторів систем, розробників програмного забезпечення та користувачів, які потребують ефективної та надійної синхронізації файлів.

Переваги утиліти:

- відкритий код,
- підтримка SSH та zlib,
- відновлення передачі,
- повна автоматизація,

Недоліки:

- тільки під Linux, але порти на інші системи існують,
- лише текстовий інтерфейс.

На відмінно від попередніх рішень, у даного продукту є лише одна, але важлива, особливість – Блоковий метод синхронізації є унікальним, та не існує в простих існуючих рішеннях як хмарні сервіси.

Підсумовуючи це, в нас є важливі наступні проблеми:

1) Метод синхронізації – продукт rsync має унікальний метод синхронізації, який позволяє зменшити навантаження на мережу в рази у випадках де зміни до файлів малі

2) Інтерфейс – Syncthing має доступний інтерфейс, хоча часом він є надто доступним; розробка легшого варіанту є важливим

3) Безпека – Всі три рішення мають свої рішення – Шифрування, TLS та використовуючи SSH. Кожен має свій набір проблем, і явно піддається для подальшого аналізу.

1.2 Аналіз проблем безпеки і методи їх вирішення

Перше за все, нам необхідно визначити критичну інформацію, яка буде доступною; у таблиці 1.1 описано два елементи, які скомпрометують всю безпеку якщо наданий неправильний доступ:

Таблиця 1.1 - Перелік критичної інформації програмного забезпечення

Назва інформації	Назва процесу, в якому використовується	Обґрунтування
Конфігурація	Запуск, шифрування, синхронізація	Може вимкнути велику кількість мір заходів та підвищення ефективності
Приватні ключі	Шифрування даних	Легкий доступ анульовує захист у випадку атаки з доступом до ПК

Опрацьовані файли не відносяться до критичних ресурсів або інформації, оскільки їх конфіденційність залежить від клієнта.

До того-ж, програмне забезпечення матиме можливість синхронізувати файли з їхніми правами та іншими метаданими збереженими. Тим самим, у

таблиці щодо прав користувачів поле файлів не можливо підтверджувати або спростувати будь-які права.

Таблиця 1.2 - Перелік користувачів з правами доступу до конфіденційної інформації і критично важливих ресурсів

Група користувачів	Критична інформація	Права доступу					
		Читання	Запис	Оновлення	Видалення	Архівація	Відновлення
Адміністратор	Конфігурація	+	-	-	-	+	+
	Ключі	+	+	+	+	+	+
Простий користувач	Конфігурація	-	-	-	-	-	-
	Ключі	-	-	-	-	-	-

Обґрунтування: Для користувачів, обидва компоненти не мають бути доступними, оскільки як вище згадано, можливі наслідки втручання користувача є порушенням безпеки через навмисне вимкнення шифрування. Адміністраторам доступ до ключів має бути мінімізованим, оскільки програмне рішення самостійно виконує необхідні операції. Читання, архівація та відновлення є доступним на випадок що необхідне відновлення без генерування нових; це не є рекомендованим, створюючи слабке місце (взлом адміністратора), але необхідне для сумісності з системами резервних копій, робота з лише одним кінцем, тощо.

Таблиця 1.3 - Перелік загроз безпеки

Назва загрози	Де виникає	Вразливий ресурс	Порушена послуга безпеки
Підміна посередині (без захисту)	Передача даних	Дані	Цілісність, конфіденційність
Спотворення	Передача даних	Дані	Цілісність
Прослуховування	Передача даних	Дані, ключі	Конфіденційність
Вгадання ключа	Створення ключа	Ключі	Конфіденційність

Ці всі загрози зіткаються практично усі продукти, які передають дані. Через це, з часом було розроблено багато різних механізмів щодо вирішення, або принаймні зменшенні рівня загрозу від них. Деякі з них є перелічені у таблиці 1.4:

Таблиця 1.4 - Перелік механізмів безпеки

Механізм безпеки	Засіб захисту	Загроза безпеці
Зміна алгоритму передачі	Алгоритм фрагментів	Підміна посередині (НЗ), Прослуховування
Використання зовнішньої ентропії	Ввід користувача, пристроїв	Вгадання ключа
Перевірка хеш-функціями	Алгоритм SHA-256	Спотворення
Використання інших методів обміну ключів	Метод Діффі - Геллмана	Прослуховування (Ключі)

Детальніше про дані механізми безпеки:

1) Зміна алгоритму на rsync-подібний визначає, що після першої передачі усі майбутні зміни змісту файлів передаються лише у ділянках, які містять зміну. Це нічого не змінює при першому обміні, але після першого обміну будь які спроби прослуховування є не ефективними, а у випадку підміни – спотворює дані у не використаний формат, замість шкідливого коду.

2) Зовнішня ентропія є особливим елементом який використовується багатьма більшими корпораціями, включаючи сервіс захисту від DDoS Cloudflare, які використовують камери як випадкові дані для ключів. Оскільки не кожен клієнт матиме периферію, необхідної для цього, інші методи

введення користувача (наприклад, рух миші та ввід з клавіатури) можуть використовуватися для випадковості.

3) Хеш функції є прості, певна сума отримана на базі змісту файлів. Будь які зміни змінюють цю хеш-суму, повідомляючи що файл спотворений. Інші алгоритми можуть використовуватися, але другий найвідоміший варіант MD5 має проблеми з колізією хеш сум від різних файлів.

4) Метод обмінів ключів Д-Г суперечить випадковій генерації ключів, але позволяє обминати проблему прослуховування взагалі – ключі ніколи не виходять на мережу, а лише фрагменти секрету, які після отримання обома сторонами створює спільний ключ.

Це є найефективніші на нашу думку механізми захисту. Але це не значить, що всі будуть вжиті – деякі з них є лише небезпечні при незахищеній передачі, інші є занадто складними або вимагають певні пристрої для роботи (наприклад, для ентропії – пристрою необхідно буде або мати якісь пристрої підключені, або модуль TPM, що в деяких середовищах може не бути).

Загалом на цьому етапі виконано наступне:

1) розглянуто існуючі рішення для аналізу їхніх методів розв’язування певних проблем, пов’язаних з синхронізацією та шифруванням;

2) висвітили особливості, недоліки та переваги кожного з них;

3) обрали певні ключові елементи з них для затвердження вимог до програмного продукту;

4) визначили критичну інформацію та можливі вразливості до захисту даних, перелічили механізми безпеки, які можуть зменшити рівень ризику цих вразливостей або анулювати їх цілком.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

Наступним етапом в розробці програмного продукту є його проєктування, яке відіграє важливу роль у створенні ефективного та надійного забезпечення. Необхідність у проєктуванні програмного забезпечення випливає зі складності завдань, які повинні бути вирішені, а також з багатостороннього процесу розробки програмного забезпечення.

По-перше, проєктування програмного забезпечення необхідне для формалізації вимог до системи. Під час цього етапу відбувається аналіз, специфікація та моделювання методів задовільнення вимог. Проєктування допомагає перекласти потреби користувачів в технічні вимоги, що визначають поведінку системи і її можливості.

По-друге, проєктування програмного забезпечення забезпечує архітектурний підхід до розробки програмних систем. Воно включає в себе проєктування архітектури програми, що визначає структуру системи, розподіл функцій між компонентами, їх взаємодію та внутрішню організацію. Архітектурний підхід дозволяє під час проєктування вирішити проблеми масштабованості, надійності, безпеки та ефективності системи.

По-третє, проєктування програмного забезпечення дозволяє визначити оптимальні рішення щодо використання технологій та інструментів розробки. Це означає вибір підходів та інших компонентів, які найкраще відповідають вимогам системи.

Отже, проєктування програмного забезпечення є необхідною складовою процесу розробки програмних продуктів. Воно вирішує завдання формалізації вимог, вибору технологій та забезпечує покращення якості та підтримку програмного продукту. Цей процес дозволяє забезпечити високу ефективність, надійність та функціональність програмного забезпечення, що задовольняє потреби користувачів.

2.1 Проєктування процесу шифрування даних

Шифрування даних - це процес перетворення інформації зрозумілої для людини у форму, незрозумілу без спеціального ключа чи пароля. Це важлива техніка, що забезпечує конфіденційність та безпеку переданої інформації. Основною мета є достовірність, що тільки авторизовані користувачі мають доступ до даних, а хакери, зловмисники чи треті сторони не можуть прочитати цю інформацію.

Шифрування базується на використанні алгоритмів, які застосовуються до початкових даних з використанням ключа, який перетворює ці дані у незчитувані. Тільки особа, яка знає правильний ключ або пароль, може розшифрувати їх і перетворити назад у оригінальну форму

Існує багато різних алгоритмів шифрування, таких як AES, RSA та інші, які використовуються в залежності від потреб і вимог безпеки. Застосування шифрування дозволяє захистити дані від несанкціонованого доступу, забезпечити конфіденційність під час передавання даних через мережу та зберігання інформації в зашифрованому вигляді на пристроях зберігання.

Шифрування може бути симетричним або асиметричним. У симетричному шифруванні використовується один і той самий ключ для як шифрування, так і розшифрування даних. Це означає, що якщо хтось отримає доступ до ключа, він зможе розшифрувати дані. У асиметричному шифруванні використовується пара ключів: приватний і публічний. Лише приватний ключ може розшифрувати дані та є лише в автора ключа, тоді як публічний ключ розповсюджується серед користувачів і використовується для шифрування перед їх передачею.

Але асиметричне шифрування має свої недоліки; найвідоміший варіант асиметричного шифрування, RSA, є алгоритмом з високою обчислювальною складністю порівняно з іншими алгоритмами, особливо при операціях шифрування та розшифрування великих обсягів даних. Це може призвести до

значного зниження продуктивності і швидкодії системи при роботі з великими файлами або низької потужності пристроїв.

Через цей недолік, асиметричне шифрування в основному використовують для шифрування даних малого розміру, наприклад інших ключів, для їх безпечної передачі.

Було вирішено вжити алгоритм шифрування Advanced Encryption Standard з режимом, доречними для дотримання підтримки з системою різниць файлів, спроєктованих у розділі 1.1. Цей стандарт, встановлений у ISO/IEC 18033-3 – блокових шифрів, є асиметричним методом шифрування у блоках, використовуючи техніки замін-перестановок.

Advanced Encryption System (AES) - це симетричний алгоритм шифрування, що використовується для захисту даних. Вона була розроблена з метою заміни старішого стандарту шифрування, DES (Data Encryption Standard), який на сьогоднішній день є легко-зламним через атаку грубою силою через малий розмір ключа (56 бітів, порівняно з 128 мінімум). Будучи стандартом, що вживається урядами для збереження секретної інформації (наприклад США), AES використовується у різних сферах, включаючи комунікації в Інтернеті, збереження даних, захист мобільних пристроїв та інше.

Функціювання шифру AES складається з ключа розміром з 128, 192 або 256 бітів, та даних, розміщених у форматі масиву 4x4, в порядку колон (тобто доступ відбувається в порядку a11, a21, a31, т. д.)

Сам алгоритм оперує в режимі циклів, де багаторазово виконуються один набір команд. Залежно від розміру ключа, алгоритм проходить від десяти до чотирнадцяти циклів. Перед початком, початковий ключ розширюється для створення серії менших ключів, які використовуються в кожному циклі шифрування; цей процес називається розширенням ключів, використовуючи розпорядок ключів. Також, до першого циклу, зміст ключа додається до даних використовуючи операцію АБО-НІ (XOR) на рівні бітів,

де істинна – лише коли один з бітів є істинною. Після цього, цикл може розпочинатися.[8]

Кожен цикл, окрім останнього, складається з наступних операцій:

1) SubBytes – нелінійна заміна, де кожен байт є замінений згідно восьми-бітній таблиці заміни.

2) ShiftRows – пересування байтів в кожному ряді за певним відступом. Кожен ряд має більший відступ, роблячи кінцевий результат подібним на лінію, оглядаючи останні елементи. Приклад функціонування є поданим на рисунку

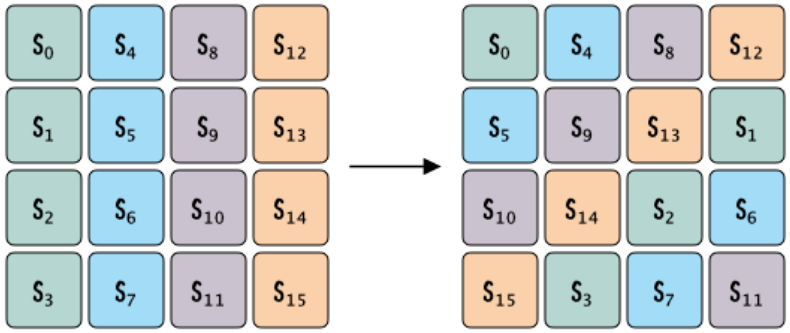

Рисунок 2.2 – Приклад роботи ShiftRows, де S – байти даних.

3) MixColumns – Лінійна трансформація, яка застосовується до кожного стовпця байтів. Кожен байт множиться на певні константи та обертається у межах 8-бітного без-знакового поля. Це означає, що кожен байт буде замінений на нове значення, що залежить від його вихідного значення та констант.

Результатом цього, алгоритм набуває дифузії – дані є розповсюдженими по всьому блоці, тим самим ускладнюючи процес можливого взлому через шифровані дані.

4) AddRoundKey – Аналогічно до початку першого циклу, на кінці кожного знову додається унікальний ключ, який залежить від розширених ключів для даного циклу.

На останньому циклі процес має малу зміну, оскільки процесу MixColumns немає. Після цього, отриманий результат вважається незламним

поза атаки грубою силою та методами, які неможливо виконати на сьогоднішній день.

Але це лише 16 байти даних; в сьогоденні це максимум текстовий файл розміром з 16 літер. Для наступних даних необхідний інший початковий стан. Для цього, усі блокові алгоритми мають різні режими операції. Для нашого програмного забезпечення був обраний режим CBC (Cipher Block Chaining, “ланцюг шифр-блоків”).

Основний принцип роботи режиму CBC полягає в тому, що перед шифруванням кожен блок даних комбінується з зашифрованим попереднім блоком, що додає додаткову складність і стійкість до шифрування. Конкретно, перед шифруванням кожного блоку його вміст по-бітово об'єднується з зашифрованим попереднім блоком шляхом операції XOR. Це дозволяє враховувати взаємозв'язок між блоками даних під час шифрування, забезпечуючи дифузію даних та унеможливлюючи відновлення оригінальної інформації без ключа. Візуальний приклад роботи даного режиму є поданим на рисунку x та x, шифрування та розшифрування відповідно:

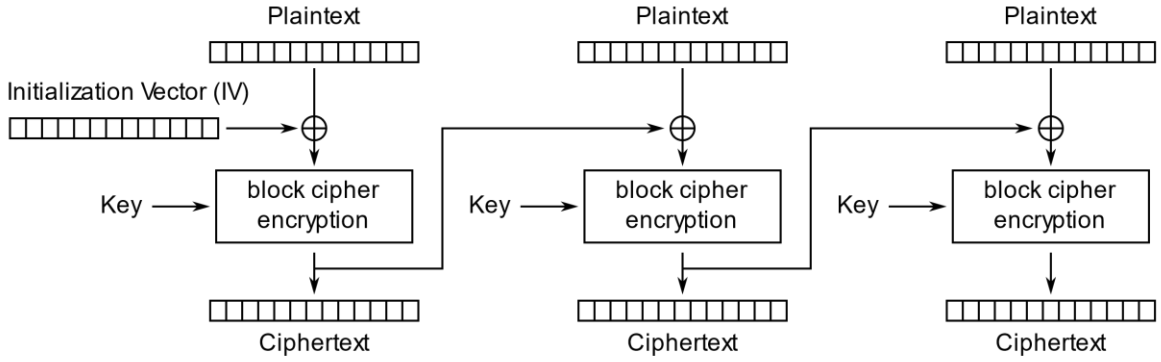

Рисунок 2.3 – Схема шифрування в режимі CBC

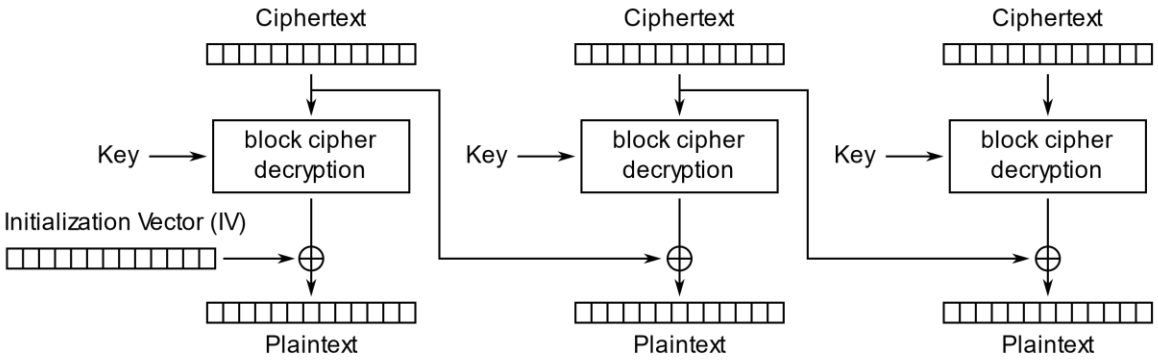

Рисунок 2.4 – Схема розшифрування в режимі CBC

Один з особливостей режиму CBC є тим, що дешифрування з неправильним початковим значенням лише порушує цілісність першого блоку. Процес розшифрування використовує шифрований блок даних для наступних блоків. Через це, зміна навіть одного біту є критичнішою - будь-яка зміна буде впливати на подальший процес розшифрування і призводити до повної некоректності результату. Це дозволяє виявляти спроби зміни даних або впровадження помилок зловмисниками або проблеми продукту і протоколу, але лише після виявлення даної точки зміни.

Рішення другої проблеми є присутнім на інших режимах, наприклад GCM (Лічильник Галуа), що є другим найбільш вживаним режимом; у цьому режимі є тег автентичності; цілісність якого гарантує цілісність всього файлу. То чому не використовувати його? Як згадано в попередньому розділі, синхронізація файлів буде виконуватися основному в фрагментах, що відрізняються; але водночас, шифрування часто змінює весь файл замість одної ділянки. Це пов’язано з GCM, де початковий вектор повинен бути випадково новим кожного разу, інакше вся безпека є під ризиком порушення. На відміну від цього, режим CBC є менш критичним – позволено вживати однаковий вектор, та зміни змісту шифрованих даних лише змінюватиметься починаючи з точки де дані були змінені. Це не краще рішення, оскільки можливо далі змінити принцип роботи (наприклад, ввести функцію де

змінені біти використовують оригінальний вектор, але це погіршує безпеку далі у випадку що нападнику відомо про це).

Як щодо перенесення ключів? Після огляду методу щодо створення ключів Діффі-Геллмана, було прийнято рішення не виконувати цього і наполягати на те, що користувач власноруч перенесе ключ на необхідні пристрої. Це пов’язано з тим, що програма має виконуватися лише з одного боку; цей метод вимагає роботу даного продукту на іншому пристрої. До того-ж, даний метод є більше призначеним на асиметричні ключі, які не використовуватимуться в продукті. А при цьому, проста передача ключа по мережі також знищує будь яку міру безпеки даних. Додаткове шифрування даного ключа під RSA також не має змісту - передати ключі на інший бік без роботи спеціального забезпечення на обох сторонах не можливо.

2.2 Проєктування системи синхронізації даних

Синхронізація даних є важливим процесом, який дозволяє забезпечити взаємну узгодженість даних між різними пристроями або системами. Це означає, що дані, які зберігаються на одному пристрої, будуть оновлюватись і відображатись на інших пристроях у відповідності до останніх змін.

Одним з основних методів синхронізації це є повний перезапис файлів. Він є пов’язаним з хмарними сервісами, серверами FTP та простим вставленням. В даному методі, нова версія файлу відправляється на всі пристрої, які пов'язані з цим файлом, і стара версія повністю замінюється.

Повна перезапис файлу є найчастіше вживаним методом синхронізації, та його недоліки можливо вважати незначними, якщо дані є досить невеликими або якщо внесені зміни стосуються більшості або всіх даних. Однак, цей метод може бути неефективним, якщо файл дуже великий, оскільки весь файл повинен бути переданий повністю, навіть якщо зміни стосуються лише невеликої частини файлу. Крім того, цей метод може викликати перебої в роботі системи, оскільки клієнти можуть бути тимчасово

недоступні під час передачі і обробки файлу через навантаження на I/O або центрального процесору.

Окрім повного перезапису файлу, існують й інші методи синхронізації даних, такі як синхронізація на основі змін. Цей метод використовується для передачі лише змінених або нових даних з одного пристрою на інший. Наприклад, замість передачі всього файлу, тільки змінені рядки або блоки даних можуть бути передані для оновлення файлу на інших пристроях. Це зменшує час і обсяг передачі даних, знижує навантаження на мережу і дозволяє швидше оновлення і синхронізацію даних.

Було прийнято рішення використовувати синхронізацію файлів на основі поділу на фрагменти. Також до функціоналу буде включено синхронізація прав та інших метаданих, пов’язаних з файлами.

Процес синхронізації буде відбуватися наступним чином (рисунок 2.5):

1) Порівняння структури кінцевої директорії - процес починається з порівняння рівності існування папок, файлів та прав між джерелом та кінцевою директорією. При відсутності папки, продукт відразу створюватиме її.

2) Аналіз дерева структури директорії - також, для виявлення зайвих папок, продукт проводитиме аналіз дерева; у випадку, що один або більше папок необхідно видалити, продукт буде виконувати це в певному порядку для уникань можливих проблем видалення вже-неіснуючої папки.

3) Копіювання відсутніх файлів - всі файли, які є відсутні на в кінцевій директорії, копіюються з вихідного джерела на призначене місце без порівнянь.

4) Копіювання змінених даних - файли, які існують на обох боках, розділяють на блоки певного розміру байтів. Після цього, ці блоки хешуються алгоритмом та порівнюються; у випадку, що хеш не співпадає, даний блок даних є переписаний джерелом.

5) Копіювання прав доступу - окрім самого файлу, також копіюються і відповідні права доступу, які пов'язані з ним. Це включає права власника,

групи та дозволи на читання, запис та виконання файлу. Це дозволяє зберегти однакові права доступу до файлів на вихідному та призначеному місці.

Рисунок 2.5 – Схема алгоритму синхронізації даних

Згідно цього плану функціонал також вживатиме порівняння хеш-сум. Для цього було обрано менш-відомий алгоритм SeaHash.

SeaHash – хеш-алгоритм, розроблений користувачем Ticki з метою бути одним з найшвидших не-шифрованих хеш-функцій. Цю мету алгоритм виконує зчитуючи 8 байтів на раз та використовуючи паралельність на рівнів інструкцій центрального процесора.

Чому не SHA? Швидкість - основною перевагою даного алгоритму є незалежність всіх станів. Точніше, при розробці продукту з ціллю паралельності дана функція може використовувати до 4 потоків водночас, надаючи надзвичайну високу пропускну здатність (6.7 ГБ/с на 2.5 ГГц ЦП). До того-ж, оскільки ми не зберігаємо це та не використовуємо його для підтвердження цілісності, безпека хешів не потрібна – лише те, чи змінений файл та фрагмент чи ні.

Процес хешування відбувається наступним чином:

1) починаючи з 4 станів, функція виконує операцію XOR над ними разом з початковими векторами,

2) стани проходить дифузію; в останньому блоці, значення підставляє 0 де недостатньо.

3) коли всі стани є оброблені дифузією, вони всі знову проходять XOR до числа байтів, які були записані в процесі,

4) результат проходить останню дифузію, в результаті якого є готова хеш-сума для порівняння

Візуалізація алгоритму є доступною на рисунку 2.6:

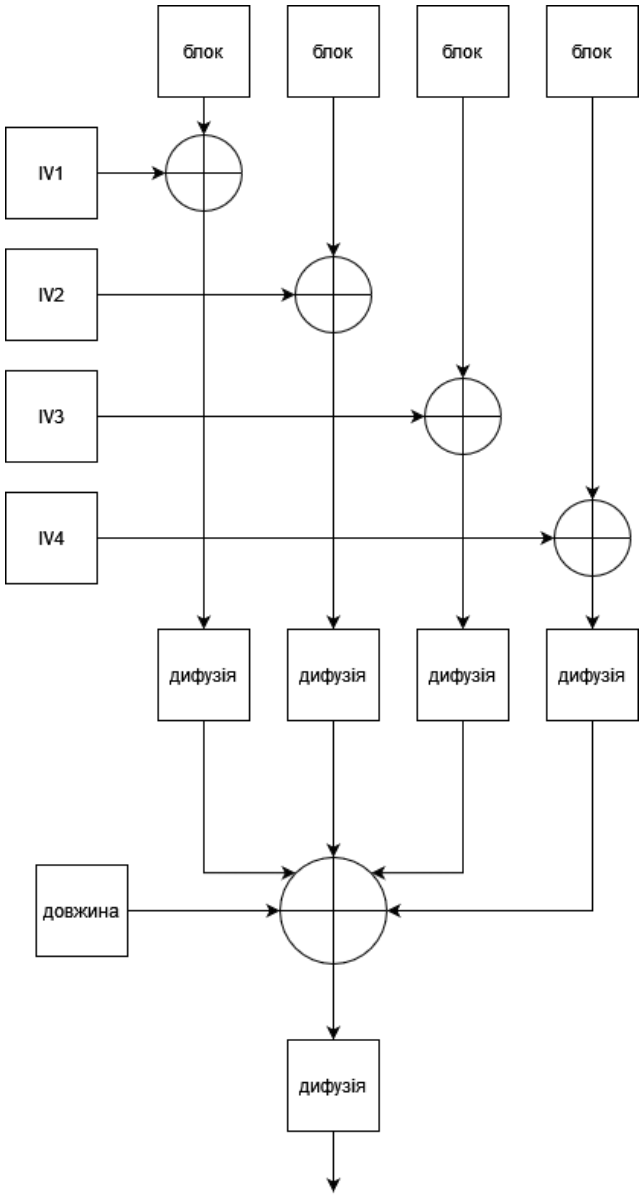

Рисунок 2.6 – Робота алгоритму SeaHash

2.3 Проєктування процесу передачі файлів по мережі

Найчастіше, відповідь до такого питання скоріше всього буде використання власного методу та протоколу. Для великої розробки цей метод є доцільним, але й у ній є ряд недоліків:

Для безпеки, розробка власного протоколу для передачі даних може мати вразливості. Неправильна конфігурація, недоліки безпеки або помилки в реалізації можуть призвести до вразливості, які можуть бути використані зловмисниками для несанкціонованого доступу до системи або пошкодження даних, включаючи цілу файлову систему. Використання вже існуючих та перевірених протоколів може зменшити ризик виникнення таких проблем, оскільки ці протоколи були піддані багатоетапним процесам перевірки та вдосконалення.

Щодо витрат, розробка власного протоколу є дорогим та довгим процесом. Це вимагає більшого набору розробників, тестових проб, та експертів з безпеки, що виходить поза межі даного дипломного проєктування. Використання наявних протоколів варіантів дозволяє значно знизити витрати, оскільки необхідно лише інтегрувати їх протоколи у наш продукт.

На глобальному масштабі є проблема вичерпання IP-адрес: Через швидкий зріст у пристроях, IP адреси формату v4 вже вичерпалися у 2020 році на всіх континентах. Через це, багато пристроїв тепер є згрупованими та фільтрованими під однією адресою провайдера інтернету. Тому, якщо ваш пристрій не може виконати пряме підключення до іншого пристрою за його IP-адресою, скоріше всього він знаходиться поза мережею NAT їхнього провайдера, до якого не можливо з’єднатися. Це в основному стосується простих користувачів, які не мають серверне обладнання та тарифів що надають справжню власну адресу. На локальній мережі цієї проблеми не існує, та для великих або децентралізованих продуктів можливо зробити реляційний сервер та методом “пробивання” порта UDP встановлювати з’єднання крізь нього, але в межах даної дипломної немає достатньо ресурсів і щоб почати випробування реалізації цього.

Сумісність та відповідність стандартам: Використання наявних протоколів передачі файлів забезпечує легшу розробку крізь платформи, які вже підтримують ці стандарти. Це дозволяє спілкуватися з різними системами без необхідності розробки та налагодження спеціальних рішень для кожного окремого клієнта.

Ресурсозбереження: Використання наявних протоколів передачі файлів дозволяє скористатися більш оптимізованими механізмами, які працюють ефективно та використовують мінімум ресурсів мережі та пристроїв Це може бути особливо важливо для великих обсягів даних та високої продуктивності.

Враховуючи це та межі нашого продукту, де процеси відбуватимуться, вживання та інтеграція існуючих протоколів є швидшим, безпечнішим та ефективнішим рішенням.

Наступні протоколи для передачі файлів можна використовувати за допомогою мережі:

FTP (File Transfer Protocol) - один з найпоширеніших протоколів для передачі файлів. Він дозволяє клієнтам встановлювати з'єднання з FTP-сервером та передавати файли в обидва напрямки. Цей протокол підтримує аутентифікацію для забезпечення безпеки передачі файлів.

SFTP (SSH File Transfer Protocol) - протокол передачі файлів, який використовує SSH (Secure Shell) для забезпечення безпечного з'єднання і передачі даних. Він надає шифрування та аутентифікацію, що робить його більш безпечним в порівнянні з FTP.

SCP (Secure Copy Protocol) - протокол, який також використовує SSH для безпечної передачі файлів між вузлами мережі. Він пропонує безпечну аутентифікацію та шифрування даних, що дозволяє безпечно передавати файли.

BitTorrent - це протокол передачі файлів, який базується на підході "розподіленого завантаження". Він дозволяє користувачам одночасно завантажувати та вивантажувати файли з різних джерел, що забезпечує

швидку передачу даних. Він може бути використаний для завантаження великих файлів.

SMB (Server Message Block) - протокол мережевого рівня, який використовується для обміну даними між комп'ютерами в локальних мережах. Він забезпечує зручний доступ до файлів, принтерів, порталів та інших ресурсів на вузлах мережі.

З перелічених протоколів, багато мають функціонал, недоречний для нашого продукту – BitTorrent працює лише в умовах, де можлива децентралізація та існує єдиний стан даних; SFTP, FTP та SCP оперують лише в режимі повного перепису. Хоча можливо створити «наглядач» над змінами, це вимагатиме великі зміни щодо як синхронізація виконується; передаючи лише змінені фрагменти, та надіючись що інша сторона зможе ввести зміни правильно.

Після огляду рішень вирішено вжити протокол SMB, який вирішує дані питання:

– Розробка крізь платформи: SMB є широко підтримуваним протоколом, який працює на різних операційних системах, таких як Windows, macOS і Linux. Це означає, що ви можете передавати файли між різними платформами без необхідності в розробці окремого методу обробки даних для кожної платформи.

– Додаткова безпека: протокол має додаткові методи захисту, такі як шифрування трафіку, доступ лише до певних директорій та аутентифікацію користувачів. Це дозволяє забезпечити конфіденційність та захист даних (особливо якщо продукт вживати без шифрування) під час передачі через мережу.

– Стабільність: SMB має розроблену і перевірену архітектуру, яка дозволяє ефективно та надійно передавати файли між комп'ютерами. Він підтримує механізми відновлення після збоїв, які забезпечують цілісність даних та забезпечують доставку без втрат та зменшуючи необхідні процедури з боку продукту.

– Простота: Оскільки всі основні процеси є вбудованими в операційну системи, користувачу не треба встановлювати будь які додаткові ПЗ. Це особливо є перевага над FTP, який не має вбудованих видів та SSH, який в основному використовується лише на Linux.

Через це, використання даного протоколу збігається найбільше з завданнями проєкту, попри низький рівень безпеки. Якщо користувачі бажають, SMB підтримує тунелі через SSH та VPN, тому можливо обрати ще один рівень безпеки понад шифруванням самих файлів, ціною швидкості.

2.4 Проєктування інтерфейсу продукту

Інтерфейс для програмного забезпечення є ключовим аспектом, що визначає взаємодію між користувачем та продуктом. Коректне проєктування забезпечує ефективність та доступність використання програмного продукту, дозволяючи користувачам взаємодіяти з функціями та можливостями системи.

Але в даному випадку, продукт не матиме повноцінного графічного інтерфейсу. Є багато причин для цього:

– Складність розробки: Розробка графічного інтерфейсу може бути складним завданням, особливо для одного розробника. Це вимагає знання спеціалізованих інструментів, дизайну інтерфейсів, роботи з графікою та взаємодії з користувачем.

– Можлива несумісність крізь платформи: Враховуючи що продукт розраховується на роботу крізь платформи, реалізація інтерфейсу для них вимагатиме набагато більше ресурсів. Кожна платформа має свої власні стандарти та специфікації для графічного інтерфейсу, що може призводити до необхідності створювати та підтримувати окремі версії продукту для кожної платформи.

– Зайвість: В деяких випадках, особливо для простих програм або утиліт командного рядка подібних до даного проєкту, графічний інтерфейс може бути зайвим або надмірним. Інтерфейс командного рядка в даному

випадку є простішим та ефективнішим для користувачів; особливо для досвідчених, які знає команди або має певний рівень технічної експертизи.

В додаток до цього, продукт є призначений для виконання на фоні через методи запланованих запусків, що зменшує потребу для інтерфейсу. В майбутньому, якщо продукт буде випущеним для загального доступу, інші розробники зможуть реалізувати зовнішню оболонку для цього.

Виходячи з цього, інтерфейс не підлягає детальному проєктуванню – програма прийматиме аргументи як параметри синхронізації, джерело та кінцева точка. Якщо синхронізація запускатиметься через термінал або його еквівалент на відповідній операційній системі, продукт містить виведення стрічки прогресу або перелік даних, які пройшли синхронізацію; що даватиме знак, що продукт працює та не припинив роботу. У випадку що програмне забезпечення має помилку, причина також буде виведена в термінал.

Але увага не командний рядок та відсутність повноцінного графічного інтерфейсу не значить, що розробка прототипу не виконуватиметься. Для цього буде створена ще одна оболонка, призначена для систем Windows яка міститиме діалог з наступними пунктами:

- вибір джерела даних,
- вибір кінцевого пункту для даних,
- чи шифрувати ці дані,
- чи вмикати детальний режим.

Після відповіді на всі ці питання, програма відкриє стрічку з командою для запуску основного продукту, який буде виконувати синхронізацію та надавати подальшу інформацію. Оскільки це лише для Windows, є можливість застосувати спеціальні діалогові вікна для вибору джерела та кінцевого пункту, зразок якого є поданим на рисунку 2.7:

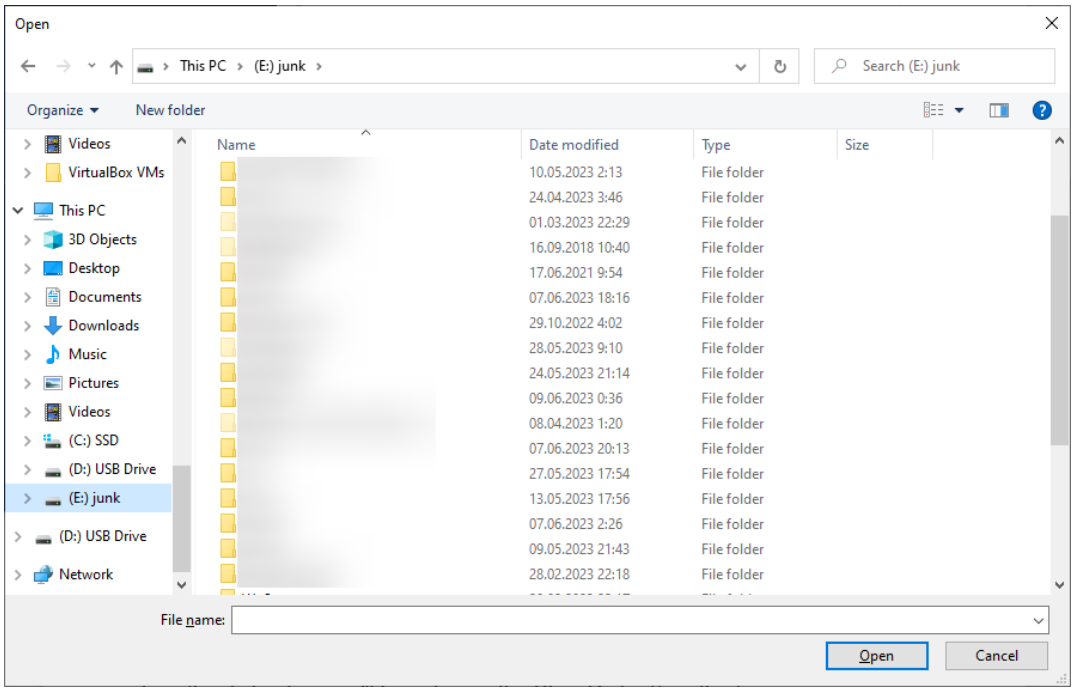

Рисунок 2.7 – Файловий діалог Windows

Також до цього – продукт буде розподілений на три програми: синхронізатор, шифрувальник, та графічна оболонка. Таким чином, користувач може обирати чи необхідно їм шифрувати дані або навпаки, шифрувати їх для передачі іншим шляхом. В загальному, типове використання продукту, та шлях даних до кінцевого призначення є поданим на рисунку 2.8:

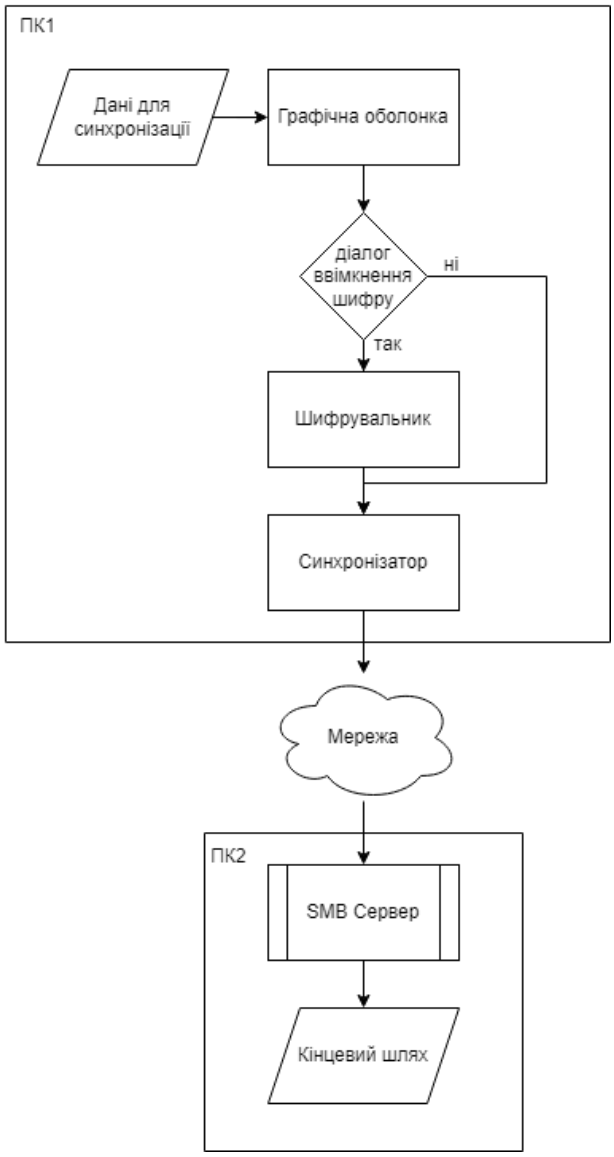

Рисунок 2.8 – Схема функціонування програмного продукту

Підсумовуючи цей розділ було оглянуто наступні ключові розробки в нашому продукті та дійшли висновку:

– Для процесу синхронізації обрано різниці фрагментів, використовуючи результати хешів для порівняння змісту обох точок. Порівняно з повними копіями, це є набагато швидшим навіть враховуючи затрати на шифрування, які є оптимізовані на високу пропускну здатність.

– Шифрування використовує AES, який вважається стандартом по причині; та розробка власного шифру створить більше проблем ніж їх вирішить. Ми виявили проблему, що деякі режими шифрування є несумісними з нашою синхронізацією, та вирішили мінімізувати через режим

СВС – тим чином, розмір змін лише залежить від точки, де були створені зміни.

– Враховуючи поточні проблеми з глобальною мережею як виснаження адрес та тим чином не можливість з’єднати пристрої без використання та розробки власного протоколу для клієнта і сервера, що в свою чергу приносить свій ряд проблем безпеки та ціни; ми вирішили звернути увагу на локальні мережі використовуючи протокол SMB, який є сумісний з багатьма платформами та не потребує жодних зайвих рухів.

– Не кожна програма потребує графічний інтерфейс. Через простоту роботи продукту, надається увага на роботу через автозапуск та командну стрічку, що позволить користувачам «поставити і забути» про неї, поки вона періодично власноруч синхронізуватиме. Але для Windows є проста серія діалогів для нових користувачів, які не зрозуміють як правильно використовувати даний продукт.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Вибір програмного забезпечення для реалізації продукту

Для розробки даного програмного продукту було використане середовище розробки Microsoft Visual Studio Code та програмні мови Rust і Python.

Rust - це сучасна, мульти парадигмова мова програмування, яка була розроблена з урахуванням проблем безпеки, швидкості та багатопотокового програмування. Rust поєднує в собі потужність системного програмування з виразністю та безпечністю мов програмування вищих рівнів.

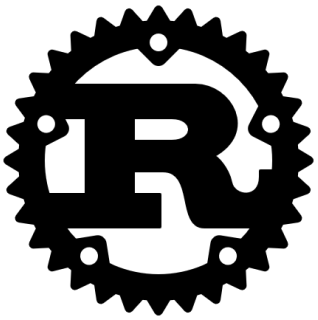

Рисунок 3.1 – Логотип Rust

Першочерговою особливістю мови Rust є її фокус на безпеку програмування. Rust надає вбудовані механізми для уникнення поширених помилок, таких як сегментація пам'яті, гонка даних та недійсні посилання. Це досягається за допомогою системи власництва та виключень, яка дозволяє статично перевіряти наявність неправильних посилань та розробляти безпечні програми.

Не менш важливим для мови Rust є її швидкість та ефективність. Rust надає по необхідності прямий доступ до системних ресурсів, таких як пам'ять та низькорівневі операції, що дозволяє писати високоефективні програми.

Rust також відзначається своєю потужною системою типів, яка допомагає виявляти помилки на етапі компіляції та полегшує роботу зі складними структурами даних. Мова підтримує явне визначення типів, поліморфізм, збирання сміття та широкий набір вбудованих типів даних.

Крім того, Rust пропонує потужні механізми шаблонів та макросів для забезпечення перевикористання коду та створення загальних абстракцій.

Нарешті, Rust має активну та зростаючу спільноту розробників, що сприяє обміну знаннями, підтримці та поширенню кращих практик у використанні мови. Існує велика кількість сторонніх бібліотек та інструментів, які підтримуються спільнотою та сприяють швидкому розвитку програмного забезпечення на мові Rust.

Загалом, мова програмування Rust вирізняється своїм фокусом на безпеку, швидкість та багатопотокове програмування. Вона надає розробникам потужні інструменти для розробки безпечного та ефективного програмного забезпечення. Її потужна система типів, механізми власництва та компіляції роблять її ідеальним вибором для системного програмування, вбудованих систем та інших критичних застосунків, де безпека та продуктивність є важливими.

Python - це високорівнева інтерпретована програмна мова загального призначення. Python володіє чистим і зрозумілим синтаксисом, що сприяє легкому читанню і розумінню коду.

Рисунок 3.2 – Логотип Python

Однією з основних філософій Python є принцип "читабельність коду", який підкреслює важливість зрозумілого і елегантного вигляду програми. Python ставить на перше місце читабельність коду перед його компактністю або скороченням. Це дозволяє науковим дослідникам і спеціалістам з області даних швидко розробляти та експериментувати зі складними алгоритмами і моделями.

Python також відомий своєю простотою використання і широкою спільнотою розробників. Існує безліч ресурсів, документаційних матеріалів, курсів та підручників, що допомагають новачкам швидко освоїти мову та використовувати її для наукових досліджень.

В цілому, Python - це потужна і гнучка мова програмування, яка дозволяє доповнювати існуючі продукти простими і зрозумілими оболонками, додатками та іншими елементами.

Microsoft Visual Studio Code (рис 3.2), також відомий як VS Code, є безкоштовним і відкритим середовищем розробки, розробленим компанією Microsoft. Воно стало одним з найпопулярніших редакторів коду серед програмістів та розробників завдяки своїй потужності, розширюваності та зручному інтерфейсу.

Однією з ключових особливостей Visual Studio Code є його підтримка різних платформ, таких як Windows, macOS та Linux. Це дозволяє розробникам працювати з різними операційними системами без втрати функціональності або продуктивності.

Рисунок 3.2 – Логотип Visual Studio

VS Code має потужні можливості підсвічування синтаксису для багатьох мов програмування, включаючи популярні мови, такі як Python, C++, Java, Rust та багато інших за допомогою розширень. До цього, дане середовище розробки надає інтегроване керування версіями за допомогою системи контролю версій Git, що спрощує роботу зі змінами коду.

Окрім того, VS Code пропонує багато корисних функцій, таких як автодоповнення коду, вбудоване налагодження, розширення та плагіни. Розширення дозволяють розробникам налаштовувати та розширювати функціональність редактора під їхні потреби, додаючи нові функції,

підтримку мов та інше. Це дозволяє створювати персоналізоване середовище розробки, що відповідає конкретним вимогам проєкту або розробника.

Visual Studio Code має інтуїтивно зрозумілий та добре організований інтерфейс користувача, що дозволяє швидко орієнтуватися по проєктам, відкривати файли, керувати розширеннями та використовувати інструменти розробки. Його легкість використання та широкий набір функцій роблять його привабливим вибором для розробників усіх рівнів навичок.

Найважливіше є тим, що VS Code є одним з трьох середовищ розробки, які повністю підтримують програмну мову Rust. Іншими альтернативами є Eclipse та IntelliJ, один з яких є платним а інший не дуже зручним до використання.

3.2 Реалізація системи синхронізації

Враховуючи, що продукт вживатиме протокол SMB[2][4], не треба створювати будь-які додаткові методи для роботи з нею; в операційних системах, для яких цей продукт підтримуватиметься, вони інтегровані як проста директорія в файловій системі.

Перш за все, оскільки було вирішено, що необхідно видавати інформацію про що виконується, необхідно створити власний клас для операцій файлів, замість використання типу що вбудований в стандартну бібліотеку[3], структура якого є подана на лістингу 3.1.

Лістинг 3.1 – Загальний клас для файлових операції:

```
pub trait FileOps {
    fn path(&self) -> &PathBuf;
    fn remove(&self, path: &PathBuf);
    fn copy(&self, src: &PathBuf, dest: &PathBuf);
}
```

Для файлів, команда match виконується над відповідною операцією копіювання/видалення оскільки вони завжди можуть повертати або Ok() або Err(), для успіху та помилки відповідно.

Лістинг 3.2 – Обгортка функцій операцій над файлами

```
pub struct File {
    path: PathBuf,
```

```
    size: u64,
}
impl FileOps for File {
    fn path(&self) -> &PathBuf {
        &self.path
    }
    fn remove(&self, path: &PathBuf) {
        match fs::remove_file(&path) {
            Ok(_) => info!("Deleting file {:?}", path),
            Err(e) => error!("Error -- Deleting file {:?}: {}",
path, e),
        }
    }
    fn copy(&self, src: &PathBuf, dest: &PathBuf) {
        match fs::copy(&src, &dest) {
            Ok(_) => info!("Copying file {:?} -> {:?}", src,
dest),
            Err(e) => error!("Error -- Copying file {:?}: {}",
src, e),
        }
    }
}
```

Це є одна з багатьох різниць порівняно з C та C++, де необхідно вирішувати, чи необхідно охоплювати код в блок перевірки на помилки; компілятор Rust відразу видає помилку якщо у коді немає метода на опрацювання обох ситуацій.

Один тип файлів, які користувачі часом забувають, є символічні зв'язки. Вони не передаються по мережі, але враховуючи що програмний продукт можливо використовувати для синхронізації між двома локальними точками, нам необхідно врахувати їхню реалізацію також. Тут знову з'являється перевага над C та іншими мовами; розробка в Rust позволяє розробити блоки коду, унікальні для даної платформи, лише за знаком #[cfg], який даватиме компілятору знати на що цей розділ націлений. Можливими варіантами є архітектура, система, бітність.

Лістинг 3.3 – Реалізація операцій над символічними зв'язками

```
pub struct Symlink {
    path: PathBuf,
    target: PathBuf,
}
impl FileOps for Symlink {
    fn path(&self) -> &PathBuf {
        &self.path
```

```
    }
    fn remove(&self, path: &PathBuf) {
        match fs::remove_file(&path) {
            Ok(_) => info!("Deleting symlink {:?}", path),
            Err(e) => error!("Error -- Deleting symlink {:?}:
{}", path, e),
        }
    }
    #[cfg(target_family = "unix")]
    fn copy(&self, _src: &PathBuf, dest: &PathBuf) {
        use std::os::unix::fs;

        match fs::symlink(&self.target, &dest) {
            Ok(_) => info!("Creating symlink {:?} -> {:?}",
dest, self.target),
            Err(e) => error!("Error -- Creating symlink {:?}:
{}", dest, e),
        }
    }
    #[cfg(target_family = "windows")]
    fn copy(&self, _src: &PathBuf, dest: &PathBuf) {
        use std::os::windows::fs;
        if self.target.is_file() {
            match fs::symlink_file(&self.target, &dest) {
                Ok(_) => info!("Creating symlink file {:?} ->
{:?}", dest, self.target),
                Err(e) => error!("Error -- Creating symlink
file{:?}: {}", dest, e),
            }
        }
        if self.target.is_dir() {
            match fs::symlink_dir(&self.target, &dest) {
                Ok(_) => info!("Creating symlink dir {:?} ->
{:?}", dest, self.target),
                Err(e) => error!("Error -- Creating symlink dir
{:?}: {}", dest, e),
            }
        }
    }
}
```

Також можливо замітити, що мова позволяє виконувати виклик бібліотеки всередині коду. Це корисно, оскільки в даному випадку при компіляції буде включена в програму лише необхідний елемент з бібліотеки, замість всіх варіантів.

Залишилось розробити тип, який об’єднуватиме всі ці типи в один список, та методи отримання файлів з папок.

Лістинг 3.4 – Структура даних про всі дані в папці

```
pub struct FileSets {
    files: HashSet<File>,
    dirs: HashSet<Dir>,
    symlinks: HashSet<Symlink>,
}
```

Продукт використовує HashSet по причині – колізія може виникнути на системах сімейства Unix (та Windows у певних випадках) та створити 2 файли з однаковою назвою. Оскільки хеші мають бути унікальними, це запобігає проблемам.

Узагальнюючи, в нас є три структури: папки, файли, та сим. зв'язки; вони з'єднанні по класу FileOps, який надає однакову специфікацію щодо роботи при зміні даних, які реалізовані як оболонки операціям зі стандартної бібліотеки з виведенням інформації, про їх виконання. Схема зв'язків є поданою на рисунку 3.3:

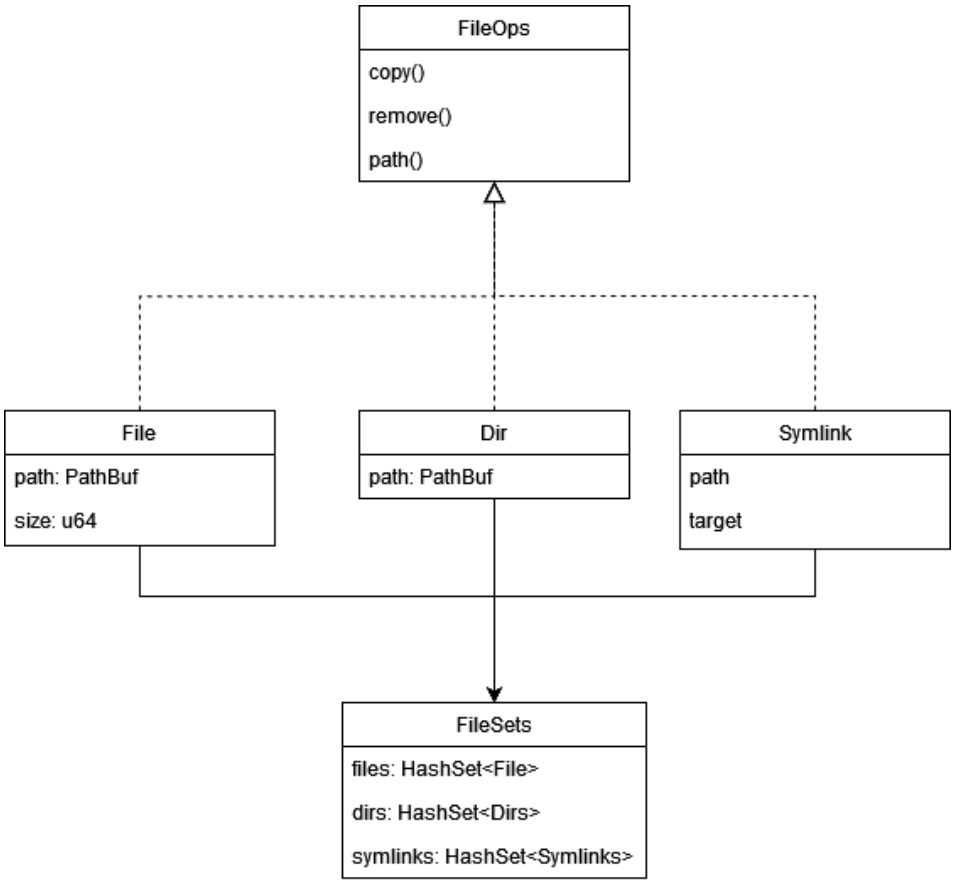

Рисунок 3.3 – Схема зв'язків типів даних

Лістинг 3.5 – Обгортка функції отримання даних в папці

```
pub fn get_all_files(src: &str) -> Result<FileSets, io::Error> {
    get_all_files_helper(&PathBuf::from(&src), &src)
}
```

Чому необхідно створювати функцію, яка лише викликає іншу функцію? Для спрощення виклику функції в інших частинах, та в справжній функції; оскільки перетворення абсолютного шляху для файлових операцій на відносний для зберігання на кожному кроці ускладнюватиме код без користі.

Якщо виявляються якісь проблеми з отриманням метаданих або файлом, продукт видаєм помилку про неї та продовжує роботу. Для визначення папок для рекурсивного сканування є код, наведений у лістингу:

Лістинг 3.6 – Реалізація сканування папки

```
if metadata.is_dir() {
     dirs.insert(Dir {
          path: relative_path.to_path_buf(),
     });
     match get_all_files_helper(&file.path(), base) {
          Ok(file_sets) => {
               files.extend(file_sets.files);
               dirs.extend(file_sets.dirs);
               symlinks.extend(file_sets.symlinks);
     }
     Err(e) => {
     error!("Error - Retrieving files: {}", e);
     continue;
     }
     }
}
```

Виклик функції знову зсередини для знаходження всіх даних незалежно від глибини дерева папок.

Тепер, після ініціалізації всіх типів даних, можливо використовувати їх для реалізації основного функціоналу - синхронізації.

Лістинг 3.7 – Опис функції синхронізації

```
pub fn synchronize(src: &str, dest: &str, flags: Flag) ->
Result<(), io::Error> {
    let src_file_sets = file_ops::get_all_files(&src)?;
    let src_files = src_file_sets.files();
```

```
    let src_dirs = src_file_sets.dirs();
    let src_symlinks = src_file_sets.symlinks();
    let dest_file_sets = file_ops::get_all_files(&dest)?;
    let dest_files = dest_file_sets.files();
    let dest_dirs = dest_file_sets.dirs();
    let dest_symlinks = dest_file_sets.symlinks();
    progress::progress_init(
        (src_files.len()
            + src_dirs.len()
            + src_symlinks.len()
            + dest_files.len()
            + dest_dirs.len()
            + dest_symlinks.len()) as u64,
    );
    let dirs_to_copy = src_dirs.par_difference(&dest_dirs);
    let symlinks_to_copy =
src_symlinks.par_difference(&dest_symlinks);
    let files_to_copy = src_files.par_difference(&dest_files);
    let files_to_compare =
src_files.par_intersection(&dest_files);

    file_ops::copy_files(dirs_to_copy, &src, &dest);
    file_ops::copy_files(symlinks_to_copy, &src, &dest);
    file_ops::copy_files(files_to_copy, &src, &dest);
    file_ops::compare_and_copy_files(files_to_compare, &src,
&dest, flags);
    if delete {
        let dirs_to_delete =
dest_dirs.par_difference(&src_dirs);
        let dirs_to_delete: Vec<&file_ops::Dir> =
file_ops::sort_files(dirs_to_delete);
        file_ops::delete_files_sequential(dirs_to_delete,
&dest);
    }

    Ok(())
}
```

Даним лістингом виконуються наступні процеси:

1) отримання даних про всі файли в джерелі та кінцевому пункті,

2) ініціалізація інформативної стрічки прогресу (якщо виконується в командному рядку)

3) визначення відсутніх файлів, папок, зв’язків,

4) та визначення існуючих файлів на обох боках.

5) копіювання файлів,

6) якщо флаг видалення кінцевих папок присутній, то знайти і видалити їх в порядку (для уникання проблем щодо видалення неіснуючих папок).

З першого погляду здається що копіювання файлів відбувається цілком, але ні; ці операції copy_files також є оболонками, спеціально розроблені для виконання над цілими наборами даних.

Лістинг 3.8 – Розробка копіювання фрагментів

```
fn diff_copy(src: &PathBuf, dest: &PathBuf) -> Result<(),
io::Error> {
        if !Path::new(&dest).exists() {
            fs::copy(&src, &dest)?;
        }

        const CHUNK_SIZE: usize = 10000;

        let src_file = fs::File::open(&src)?;
        let mut src_reader =
BufReader::with_capacity(CHUNK_SIZE, &src_file);
        let dest_file = OpenOptions::new()
            .write(true)
            .read(true)
            .create(true)
            .open(&dest)?;
        dest_file.set_len(src_file.metadata()?.len())?;
        let mut dest_reader =
BufReader::with_capacity(CHUNK_SIZE, &dest_file);
        let mut dest_writer =
BufWriter::with_capacity(CHUNK_SIZE, &dest_file);

        loop {
            let mut src_buffer = [0; CHUNK_SIZE];
            let mut dest_buffer = [0; CHUNK_SIZE];

            if src_reader.read(&mut src_buffer)? == 0 {
                break;
            }
            dest_reader.read(&mut dest_buffer)?;

            if seahash::hash(&src_buffer) !=
seahash::hash(&dest_buffer) {
                dest_writer.write(&src_buffer)?;
            } else {
```

```
                dest_writer.seek(SeekFrom::Current(CHUNK_SIZE as
i64));
            }
        }

        Ok(())
    }
```

Ця функція виконує наступні дії в порядку:

1) у випадку, що файл на кінцевому шляху не існує (через перевірку, чи .exists() видає false) переходим на копіювання цілком,

2) встановлюємо розмір фрагменту 10кб та відкриваєм файли з параметрами перезапису,

3) знаючи розмір через метадані, розширяємо або зменшуємо розмір кінцевого файлу до розміру джерела,

4) створюємо буфер зчитування та запису для кінцевого файлу.

Після цього, проводиться цикл наступних процесів поки копіювання не завершене:

1) створюємо та очищуємо підбуфери (оскільки дії з BufReader/Writer відразу зчитує наступні дані у файлі),

2) якщо при зчитувані з джерела нові дані не надходять, виходим з циклу, інакше також зчитуємо блок з кінцевого файлу,

3) хешуєм обидва файли та порівнюєм їх; якщо вони не ідентичні вписуємо цей фрагмент, в іншому випадку - пропускаєм цей фрагмент в буфері запису для уникання зсуву даних.

Схематична візуалізація роботи цього коду є поданим на рисунку 3.4.

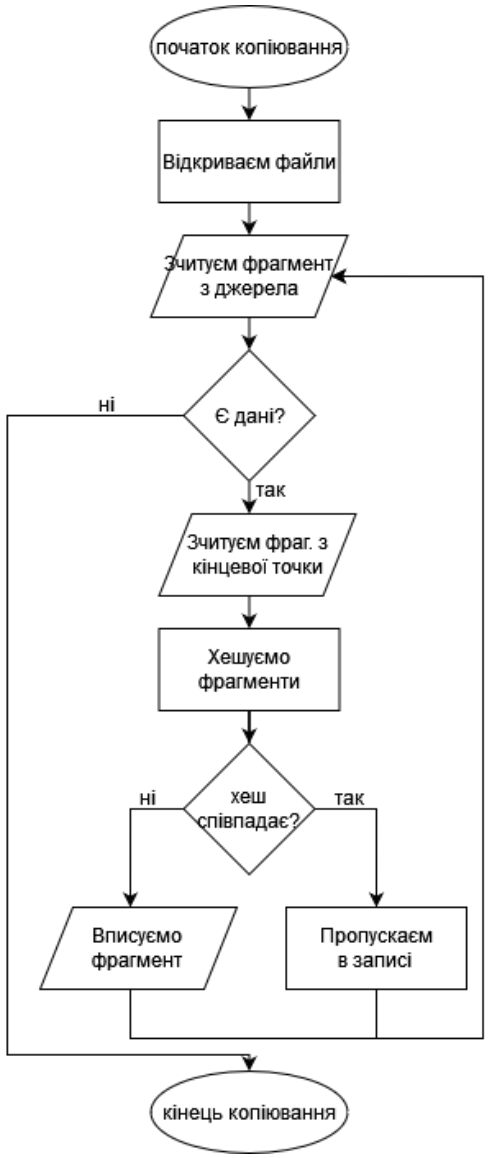

Рисунок 3.4 – Схема копіювання фрагментів даних

3.3 Реалізація системи шифрування даних

Перше правило розробки шифру створити безпечний шифр з нуля не можливо, особливо для AES. Але через бібліотеку нижчого рівня, можливо створити інтерфейс та шифратор [1].

Для початку, алгоритми шифрування потребують методи генерації ключів та початкових значень:

Лістинг 3.9 – Функції генерації ключів та початкових значень

```
pub fn generate_key() -> [u8; KEY_SIZE] {
    let mut rng = rand::thread_rng();
    let mut key: [u8; KEY_SIZE] = [0; KEY_SIZE];
    rng.fill_bytes(&mut key);
    key
}
```

```
pub fn generate_iv() -> [u8; IV_SIZE] {
    let mut rng = rand::thread_rng();
    let mut iv: [u8; IV_SIZE] = [0; IV_SIZE];
    rng.fill_bytes(&mut iv);
    iv
}
```

Через відсутність доступу до зовнішньої ентропії через підтримку більшої кількості пристроїв, використовуємо випадковість.

Для того, щоб програма знала що ключ не є для даного файла використовується HMAC з SHA-256:

Лістинг 3.10 – Реалізація перевірки відповідності ключа до файла та створення HMAC

```
pub fn digest(&self) -> [u8; DIGEST_SIZE] {
        let mac = self.mac_bytes();
        let iv = self.iv_bytes();
        hmac_256_digest(&mac, &iv)
    }
pub fn owns_file(&self, filename: &str) -> bool {
        let mut fd =
            File::open(filename).expect(format!("failed to open
file {}", filename).as_str());
        let mut buffer = [0; DIGEST_SIZE];
        fd.read(&mut buffer)
            .expect(format!("failed to read the first bytes of
{}", filename).as_str());

        self.check_digest(&buffer)
}
```

Ці функції виконують створення та зчитування даного HMAC, відповідно. Використовуючи булеан з owns_file можливо відразу знати чи ключ підходитиме до даного файлу чи ні.

Лістинг 3.11 – Реалізація процесу шифрування даних

```
pub fn encrypt(&self, data: &[u8]) -> Result<Vec<u8>,
symmetriccipher::SymmetricCipherError> {
        let enc_key = self.key_bytes();
        let iv = self.iv_bytes();
        let mut encryptor = aes::cbc_encryptor(
            aes::KeySize::KeySize256,
            &enc_key,
            &iv,
            blockmodes::PkcsPadding,
        );
        let mut ciphertext = Vec::<u8>::new();
        let mut read_buffer = buffer::RefReadBuffer::new(data);
        let mut buffer = [0; BUF_SIZE];
```

```
        let mut write_buffer = buffer::RefWriteBuffer::new(&mut
buffer);
        let digest = self.digest();
        ciphertext.extend_from_slice(&digest);
        loop {
            let result = encryptor.encrypt(&mut read_buffer,
&mut write_buffer, true)?;

            ciphertext.extend(
                write_buffer
                    .take_read_buffer()
                    .take_remaining()
                    .iter()
                    .map(|&i| i),
            );
            match result {
                BufferResult::BufferUnderflow => break,
                BufferResult::BufferOverflow => {}
            }
        }
        Ok(ciphertext)
}
```

На вигляд здається, що багато чого виконується, але це не так - одна половина це ініціалізація всіх даних, а друга – цикл шифрування всіх даних. Спрощена схема роботи цього є подана на рисунку 3.5:

Рисунок 3.5 - Схема роботи алгоритму шифрування CBC

Для дешифрування в нас аналогічно відбувається даний процес, з одною зміною щодо необхідності зчитування HMAC даних на зашифрованому файлі, є поданим у лістингу 3.12:

Лістинг 3.12 – Процес дешифрування даних

```
pub fn decrypt(
        &self,
        cyphertext: &[u8],
    ) -> Result<Vec<u8>, symmetriccipher::SymmetricCipherError>
{
        let mut decryptor = aes::cbc_decryptor(
            aes::KeySize::KeySize256,
            &self.key_bytes(),
            &self.iv_bytes(),
            blockmodes::PkcsPadding,
        );

        let mut plaintext = Vec::<u8>::new();
        let hmac_bytes: [u8; DIGEST_SIZE] =
ciphertext[..DIGEST_SIZE].try_into().unwrap();
        if !self.check_digest(&hmac_bytes) {
            eprintln!("Cannot decrypt: data was not encrypted by
this key");
            return Ok((*ciphertext).to_vec());
        }

        let ciphertext = &ciphertext[DIGEST_SIZE..];
        let mut read_buffer =
buffer::RefReadBuffer::new(&cyphertext);
        let mut buffer = [0; BUF_SIZE];
        let mut write_buffer = buffer::RefWriteBuffer::new(&mut
buffer);

        loop {
            let result = decryptor.decrypt(&mut read_buffer,
&mut write_buffer, true)?;

            plaintext.extend(
                write_buffer
                    .take_read_buffer()
                    .take_remaining()
                    .iter()
                    .map(|&i| i),
            );
            match result {
                BufferResult::BufferUnderflow => break,
                BufferResult::BufferOverflow => {}
            }
        }
```

```
            Ok(plaintext)
        }
    }
```

Для цього коду є поданим схема на рисунку 3.6. Також, оскільки використання HMAC не було згадано в проєктуванні – це є хеш функція що забезпечує цілісність та автентичність. В основному, він вживається для швидкого зчитування та підтвердження що ключ підходить для цього файлу.

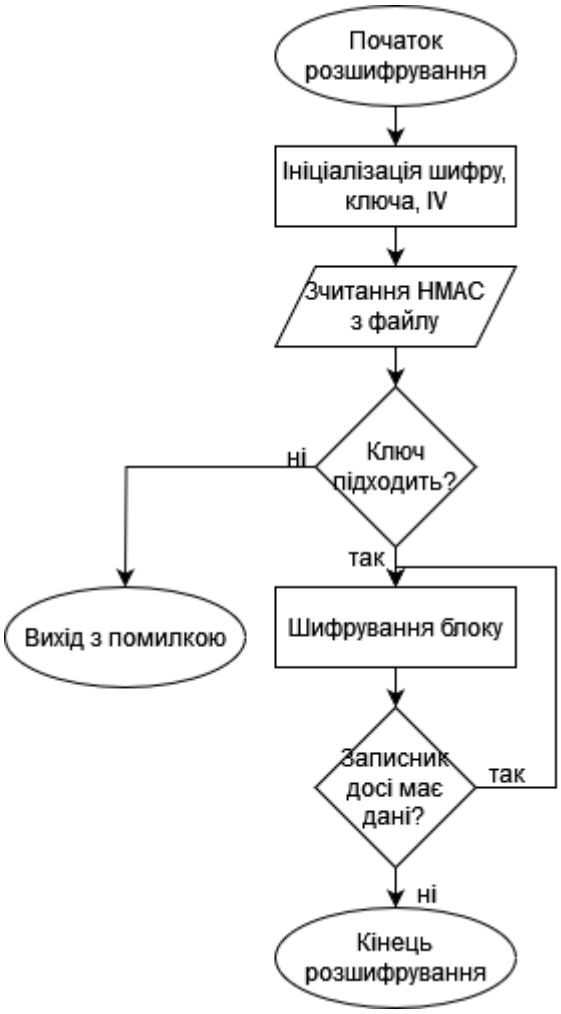

Рисунок 3.6 – схема процесу розшифрування

Поза межами цього, особливої роботи з шифруванням немає, оскільки зміни самого алгоритму AES або SHA не рекомендовані будь-ким.

3.4 Реалізація інтерфейсу

Для створення інтерфейсу для текстового режиму у нашому продукті в нас є наступні два варіанти: розробка власного інтерпретатора, або використання бібліотеки clap[7].

При використанні даної бібліотеки розробка аргументів є лише процесом створення структури для параметрів та Enum для підкоманд. У лістингу 3.13 є поданий приклад для шифрувальної частини продукту:

Лістинг 3.13 – Структура аргументів для шифрувальної частини продукту

```
#[derive(Parser)]
#[command(about)]
struct Cli {
    #[command(subcommand)]
    mode: CMDs,

}
#[derive(Subcommand)]
enum CMDs{
    Encrypt {
        #[arg(short,long,value_name = "FILE")]
        src: PathBuf,
        #[arg(short,long,value_name = "KEYFILE")]
        key: Option<PathBuf>,
    },
    Decrypt {
        #[arg(short,long,value_name = "FILE")]
        src: PathBuf,
        #[arg(short,long,value_name = "KEYFILE")]
        key: Option<PathBuf>,
    }
}
```

Використовування структуру Cli для опису параметрів, та CMDs для опису параметрів для підкоманд, в даному випадку шифрування та розшифрування. Параметр Option визначає що його може не бути, і нам слід буде перевіряти під час розробки процедур для цих підкоманд.

Лістинг 3.14 – Приклад реалізації інтерпретації аргументів

```
fn main() {
    let args = Cli::parse();
    match &args.mode {
        CMDs::Encrypt { src, key } => {
     …
        }
        CMDs::Decrypt { src, key } => {
     …
        }
    }
}
```

Код, не причетний до інтерфейсу, був видалений у даному блоці. У випадку, що жоден з підкоманд був вжитий, програма виводитиме «допомогу» по використанні її, використовуючи дані в файлі проєкту.

Для реалізації графічної оболонки використовується мова Python, оскільки більшість з функцій Rust для цього не потрібні, та виклик програм всерівно відбуватиметься на командній стрічці, де Python також підтримується для виправлення помилок та іншого.

Лістинг 3.15 – Реалізація графічного діалогу у мові Python

```
import tkinter as tk
from tkinter import Tk
from tkinter import filedialog
from tkinter.filedialog import askdirectory
from tkinter import messagebox
import subprocess
crypt_box = tk.messagebox.askquestion('Encrypt files?', 'Do you
want to encrypt your files before synchronizing? Your current
files will not be affected.')
key_exist = "no"
crypt_box = "no"
key_path = ""
if crypt_box == "yes": key_exist =
tk.messagebox.askquestion('Existing key?', 'Do you have an
existing key for encryption? If not, one will be generated.')
if key_exist == "yes": key_path = filedialog.askopenfilename()
src_path = askdirectory(title='Source path')
dest_path = askdirectory(title='Destination path')
if crypt_box == "yes":
    cryptproc = "velocrypt.exe" + src_path
    if (key_exist): cryptproc += "--key" + key_path
    subprocess.call(cryptproc)
    src_path += "_vcrypt"
    subprocess.call(["velosync.exe", src_path, dest_path])
else: subprocess.call(["velosync.exe", src_path, dest_path])
```

В результаті, даний код виводитиме наступні діалоги:

– чи зашифровувати дані,

– якщо так, то чи має користувач певний ключ,

– якщо так, його місцезнаходження,

– місце папки, яка буде джерелом,

– місце пункту призначення даних.

3.5 Тестування програмного продукту

Тестування програмного забезпечення є невід'ємною складовою процесу розробки програмних систем і відіграє важливу роль у забезпеченні якості та надійності програмних продуктів. Воно спрямоване на виявлення помилок, недоліків у функціонуванні програмного забезпечення, забезпечення його відповідності вимогам, а також на підтвердження його працездатності та стабільності.

Однією з головних причин проведення тестування програмного забезпечення є забезпечення високої якості продукту. Помилки та дефекти, які залишаються невиявленими в процесі розробки, можуть мати серйозні наслідки для користувачів програмного забезпечення. Перевірка та тестування дозволяють виявити такі проблеми та недоліки, що дозволяє їх виправити до випуску програмного продукту на ринок. Це допомагає уникнути негативних наслідків для користувачів, забезпечуючи їм надійне та функціональне програмне забезпечення.

Крім того, тестування дозволяє встановити відповідність програмного забезпечення вимогам, встановленим для нього. Це включає в себе перевірку правильності реалізації функціональності, відповідність нормативним документам, а також вимоги до продуктивності, безпеки та стабільності програмного забезпечення. Виявлення невідповідностей дозволяє нам та іншим розробникам вносити необхідні зміни та удосконалення з метою досягнення заданих критеріїв якості.

До цього, тестування є важливим етапом для виявлення потенційних проблем у взаємодії програмного забезпечення з різними операційними системами, пристроями та середовищами. Це дозволяє попередити можливі несумісності та проблеми, які можуть виникнути при використанні програмного забезпечення на різних платформах.

Тестування продукту в мові Rust є простою – компілятор виконуватиме всі функції, перед якими стоїть знак #[test]. При цьому, варто зауважити що

тест вважатиметься успішним при поверненні будь-якого стану ОК на кінці; через це, перевірка даних має виконуватися через assert_eq!:

Лістинг 3.16 – Тестовий блок успішного шифрування та дешифрування даних

```
#[cfg(test)]
mod tests {
    use crate::aescbc::Key;
    use crate::aescbc::Config;
    #[test]
    fn test_encrypt_and_decrypt() {
        let key = Key::generate(&;

        let plaintext = b"This is a secret";
        let cyphertext = key.encrypt(plaintext).unwrap();
        let decrypted = key.decrypt(&cyphertext).unwrap();
        assert_eq!(decrypted, b"This is a secret");
    }
}
```

Код в лістингу 3.17 є для тестування правильності шифрування. Оскільки це також тестує відповідну роботу створення ключів, тим чином повністю тестуючи роботу шифрувального компоненту.

Лістинг 3.17 – Тестування копіювання та створення папок

```
    #[test]
    fn multi_level() {
        const TEST_DIR: &str = "test_get_all_files_multi_level";
        const SUB_DIRS: [&str; 2] = ["dir1", "dir1/dir2"];
        const TEST_FILES: [&str; 3] = ["file.txt",
"dir1/file.txt", "dir1/dir2/file2.txt"];
        const TEST_DATA: [&[u8]; 3] = [b"1", b"",
b"1234567890"];

        fs::create_dir_all([TEST_DIR,
SUB_DIRS[1]].join("/")).unwrap();

        for i in 0..TEST_FILES.len() {
            let path = [TEST_DIR, TEST_FILES[i]].join("/");
            fs::File::create(&path).unwrap();
            fs::write(&path, TEST_DATA[i]).unwrap();
        }

        let file_sets = get_all_files(TEST_DIR).unwrap();
        let mut file_set = HashSet::new();
        let mut dir_set = HashSet::new();

        for i in 0..TEST_FILES.len() {
            file_set.insert(File {
```

```
                path: PathBuf::from(TEST_FILES[i]),
                size: TEST_DATA[i].len() as u64,
            });
        }

        for i in 0..SUB_DIRS.len() {
            dir_set.insert(Dir {
                path: PathBuf::from(SUB_DIRS[i]),
            });
        }

        assert_eq!(file_sets.files(), &file_set);
        assert_eq!(file_sets.dirs(), &dir_set);

        fs::remove_dir_all(TEST_DIR).unwrap();
    }
```

Цей блок фокусується на здатності створити пусті, багаторівневі папки, що є критичним у синхронізації. В порядку, вона створює тестові значення, створює файли відповідно до кількості тестових файлів, та порівнює зміст згенерованих даних про структуру з даними, з яких вони були створені в першу чергу.

Тестування інтерфейсу в основному випробовує випадки нестачі аргументів, і перевіряється по успішності запуску (status.success(), який має бути false).

Лістинг 3.18 – Тестування компоненту інтерфейсу

```
#[test]
fn test_no_dest() {
        Command::new("cargo")
            .args(&["build", "--release"])
            .output()
            .unwrap();

        let output = Command::new("target/release/velosync")
            .args(&["sync", "src"])
            .output()
            .unwrap();

        assert_eq!(output.status.success(), false);
}

#[test]
fn test_too_many_args() {
        Command::new("cargo")
            .args(&["build", "--release"])
            .output()
```

```
            .unwrap();

        let output = Command::new("target/release/velosync")
            .args(&["sync", "src", "dest", "dest"])
            .output()
            .unwrap();

        assert_eq!(output.status.success(), false);
}
```

Через лінійність роботи скрипту Python, його тестування не вважається за потрібним; проблеми які можуть статися з ним полягають на помилки у передачі параметрів до інших компонентів програмного продукту, які містять власні перевірки на коректність. Аналогічно, проведення десятків тестувань та документація кожного з них не вважається доречною завдяки нашим тестовим функціям, які автоматично випробовуються при кожній компіляції нашого продукту, що підхоплює більшість можливостей цей продукт позволяє виконувати.

Підсумовуючи це все було виконано реалізацію нашого програмного продукту за наступними етапами:

– вибір програмного забезпечення для реалізації, та обґрунтування вибору їх,

– реалізації системи синхронізації, підкреслюючи унікальність мови Rust порівняно з іншими та влаштування основного коду,

– опис системи шифрування, переважно огляд навколо самим шифром та створення ключів, їх збереження, шифрування та дешифрування даних та спрощення їх у схематичну форму,

– реалізація інтерфейсу використовуючи бібліотеку, призначену для підтримки простішого методу створення та підтримки аргументів та налаштувань, а також створення Python скрипту, для графічного еквіваленту для нових користувачів даного продукту.

Після завершення реалізації, проводиться обчислення та обґрунтування економічної цінності даного програмного продукту, якщо продовжувати розробку в повному характері у команді розробників.

4 ТЕХНІЧНО-ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ

Техніко-економічне обґрунтування є ключовою складовою для будь-яких проєктів. Мета цього розділу є визначення витрати та прибуток проведення даного продукту за методами аналізу ринку, розрахунку ефективності. На кінці розділу приймається рішення, чи доцільно продовжувати розвиток даного продукту чи ні.

4.1 Аналіз ринку забезпечення синхронізації та шифрування даних

Перш за все, нам необхідно висвітлити наступні тенденції, які є пов’язаними з продуктами та сервісами які надають синхронізацію даних. Враховуючи раніше згадані існуючі рішення, додаткові продукти як Resilio Sync, FreeFileSync та всі інші хмарні сервіси, як Google Drive, Mediafire та Dropbox мають багато спільних характеристик:

1) Популярність хмарних рішень: Це є очевидним, але синхронізаційне програмне забезпечення, яке працює на основі хмарних технологій, має набагато більшу силу на ринку. Це пов’язано з тим, що користувачі в першу чергу приділяють увагу зручності способи синхронізувати дані між різними пристроями, і хмарні рішення та їхня доступність 99.9999% будь де та будь коли надають широкий спектр можливостей для цього.

2) Мобільна синхронізація: З поширенням смартфонів та планшетів, зберігання мобільних даних на іншому сховищі стає важливою функцією для багатьох користувачів. Продукти, які можуть синхронізувати дані між або з мобільних платформ при забезпеченні швидкості передачі має великий потенціал на ринку серед простих користувачів.

3) Автоматизація та інтелектуальність: Серед компаній та мереж подібного масштабу, спеціалізовані продукти, що можуть розпізнавати та автоматично синхронізувати важливі дані завжди було в потребі.

4) Збільшена увага до безпеки: З огляду на зростаючі загрози кібербезпеці, користувачі дедалі стають більш свідомими щодо захисту своїх даних. Продукти для синхронізації, яке пропонує високий рівень шифрування

та заходи безпеки, має малу перевагу на ринку, оскільки це лише мала частина користувачів, яка надаватиме перевагу цьому.

З вище переліченого, програмний продукт відповідає двом з тенденцій. Оскільки продукт має один з кращих методів захисту, люди з пріоритетом до захисту можуть мати інтерес до використання його. У випадку виходу на ринок, ми вважаємо що цей продукт буде достатньо прибутковим для його окупності серед ентузіастів по захисту даних.

4.2 Розрахунок витрат на проєктування та реалізації продукту

Перед обчисленням, необхідно розділити завдання на етапи розробки, виділяючи необхідний час та виконавців кожного з них. Ці частини є поданими у таблиці 4.1:

Таблиця 4.1 – Етапи проведення розробки продукту

№ п/п	Назва етапу	Виконавець	Час виконання, год
1	Формування завдання	Керівник проєкту	6
2	Проєктування заходів безпеки	Аналітик з кібербезпеки	13
3	Розробка продукту	Програміст	60
4	Розробка заходів тестування продукту	QA-інженер	4
5	Проведення тестування	QA-інженер	6
6	Оформлення документації продукту	QA-інженер	10
7	Супровід продукту на випуск	Програміст	4
		Керівник проєкту	4
	Загалом	-	107

При розрахуванні заробітної плати, не можливо обчислювати плату, меншу за оклад 1 розряду, який станом на 2023 рік становить 2893 грн. У погодинному форматі, вважаючи восьми годинний робочий день, це становить 2893/176 = 16,43 грн на годину. Розряди та тарифні коефіцієнти робітників є поданою у таблиці 4.2:

Таблиця 4.2 – Тарифні коефіцієнти посад

Посада	Розряд	Тарифний коефіцієнт
Керівник проєкту	8	4,46
Аналітик з кібер безпеки	12	7,44
Програміст	6	4,12
QA-Інженер	4	3,59

Обчислення зарплат є за формулою множення окладу 1 розряду на коефіцієнт, та на кількість відпрацьованих годин. Обчислюючи ми отримуємо наступні числа:

Керівник проєкту – 16,43 * 4,46 * 10 = 733 грн

Аналітик з кібербезкеки – 16,43 * 7,44 * 13 = 1590 грн

Програміст – 16,43 * 4,12 * 64 = 4332 грн

QA-Інженер – 16,43 * 3,59 * 20 = 1180 грн.

Враховуючи податки, всі ці дані зменшуються на 18% як податок на дохід осіб та 1,5% згідно військового збору. До цього, також необхідно 22% згідно єдиного соціального внеску.

Обчислення внеску проходить використовуючи загальну суму до оплати робітникам, у даному випадку 7835 * 22% = 1723 грн.

Згідно цих даних, розрахунок заробітної плати робітників та відрахувань є описаними у таблиці 4.3:

Таблиця 4.3 – Заробітна плата робітників

№	Посада	Оклад	Відрахування	Кількість		Сума
п/п	виконавця	Грн	Грн	Чол.	годин	з/п, грн
1	Керівник проєкту	733	143	1	10	590
2	Аналітик з к.б.	1590	310	1	13	1280
3	Програміст	4332	845	1	64	3487
4	QA-інженер	1180	230	1	20	950

4.3 Обґрунтування необхідності розробки

Перед випуском продукту необхідно довести що він має мати певний ефект для виправдання його розробки, це може бути підвищення

ефективності або безпеки даних, зниження енерговитрат або інших факторів, зменшення яких призводить до збільшення прибутку для підприємства.

З цих пунктів, наш продукт має зміст підвищення ефективності у виді автоматизації процесу синхронізації. У великих підприємствах, де можливо необхідно дотримувати певний базовий стан декількох директорій на ряді серверів або робочих станцій, це зменшує роботу системного адміністратора. У специфічних умовах де віддалена точка має використовувати мінімум ресурсів, даний продукт має додаткову перевагу над усіма іншими рішеннями, які потребують присутності програми сервера або еквіваленту на обох станціях.

Під час аналізу було виявлено що продукт матиме попит серед системних адміністраторів та інших сферах, де необхідна конфіденційність. Було обчислено деякі витрати, необхідні на розробку даного продукту з командою розробників включаючи експертів з кібербезпеки та інженера по контролю якості,

Використання даної розробки призводить до підвищення продуктивності, зменшуючи витрати та простій робочих станцій у певних умовах де синхронізація та безпека необхідна.

В загальному, продукт матиме високий попит у виході на ринок програм з функціоналом безпечної синхронізації.

ВИСНОВКИ

В результаті дипломного проєктування було розроблено програмний продукт, який призначений для безпечного та швидкого шифрування та синхронізації даних при передачі їх локальною мережею. Використовуючи протокол SMB можна легко передавати файли між робочими станціями, робота яких забезпечується різними операційними системами через локальну мережу.

Під час виконання дипломного проєктування виконані наступні завдання:

– проаналізували предметну область та існуючі рішення, включаючи хмарні сервіси для типових користувачів та Syncthing з rsync як вільні рішення для досвідчених користувачів. Аналізуючи їхні недоліки та особливості, ми виділили необхідний функціонал програми синхронізації.

– оглянули можливі проблеми, пов’язані з сферою синхронізації та шифрування по мережі в цілому. Аналізуючи основні елементи, висвітили можливі рішення для проблем, включаючи хешування для забезпечення цілісності та інші методи створення та передачі ключів для зменшення ризику втрати конфіденційності.

– при проєктуванні продукту, розглянули різні методів реалізації функціоналу завдання, такі як метод шифрування, файлові протоколи для зв’язку по мережі та шляхи забезпечення ефективного синхронування даних. Також аргументували вибір даного рішення порівняно з альтернативами.

– реалізовуючи продукт, надали короткий опис обраному програмному забезпеченню та мові для розробки нашого продукту. При описі коду, надали пояснення щодо роботи цих алгоритмів, особливості реалізації та використаної мови. В процесі тестування, розробили тестові блоки, які автоматизували процес тестування виконуючи можливі варіанти використання перед допуском до стабільного випуску.

– проаналізували ринок на попит серед потенційних клієнтів та виявили, що ринок містить ПЗ з подібним функціоналом. Розрахували ціну розробки даного продукту в умовах команди та навели приклад на користь розробки в умовах обмеженості ресурсів, де продукт може оперувати виключно з одного боку з’єднання.

Для подальшої розробки можливо створення повноцінного графічного інтерфейсу та продовження роботи над розробкою власного протоколу для процесу синхронізації. Даний продукт має попит серед системних адміністраторів та ентузіастів по кібезбезпеці, та є прибутковим.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. AES 0.8.3. *docs.rs*: вебсайт. URL: https://docs.rs/aes/0.8.3 (дата звернення: 17.03.2023)

2. Chanda A. Network Programming with Rust. Packt Publishing, 2018. P. 280.

3. Hohenheim J., Durante D. Rust Standard Library Cookbook. Packt Publishing, 2018. P. 360

4. Microsoft SMB Protocol and CIFS Protocol Overview. *Microsoft*: вебсайт. URL: https://learn.microsoft.com/en-us/windows/win32/fileio/microsoft-smb-protocol-and-cifs-protocol-overview (дата звернення: 31.04.2023)

5. Paar C., Pelzl J. Understanding Cryptography. Springer Publishing, 2009. P. 390.

6. Seahash 4.1.0. *docs.rs*: вебсайт. URL: https://docs.rs/seahash/4.1.0 (дата звернення: 05.03.2023)

7. Youens-Clark K. Command-Line Rust. O'Reilly Media, 2022. P. 396.

8. ДСТУ ISO/IEC 18033-3:2015 Інформаційні технології. Методи захисту. Алгоритми шифрування. Частина 3. Блокові шифри. ТК 20, 2015.

ВІДГУК

на дипломний проєкт
освітньо-кваліфікаційного рівня «молодший спеціаліст»
зі спеціальності 122 “Комп'ютерні науки”
Галицького фахового коледжу імені В’ячеслава Чорновола
Касіяна Максима Ігоровича

на тему **«Програма шифрування та синхронізації файлів в комп’ютерні мережі»**

Дипломний проект присвячений вирішенню практичної задачі – організації шифрування та синхронізації файлів при їх передачі комп'ютерними мережами.

В дипломному проекті визначено актуальність реалізації додатку, проаналізовано доцільність застосування додатку у визначеному форматі та практичність його застосування користувачами.

У процесі роботи над дипломним проектом автор детально вивчив та проаналізував предметну область, дослідив процеси шифрування та синхронізації даних та передачі файлів мережею, спроектував інтерфейс продукту. Студент в роботі аргументував вибір програмних засобів для реалізації програмного додатку, попередньо проаналізувавши аналоги, протестував додаток та розрахував витрати на проектування та обгрунтував необхідність розробки. Студент зарекомендував себе як відповідальний спеціаліст у галузі розробки програмного забезпечення.

Під час виконання плану дипломного проекту студент продемонстрував вміння роботи з джерелами, вести пошук інформації, ставити та вирішувати фахові завдання.

У цілому дипломний проект виконаний на хорошому рівні і заслуговує оцінки добре.

Сиротюк Н.С. — керівник дипломного проекту:
викладач ЦК інформатики та комп'ютерних дисциплін

РЕЦЕНЗІЯ

на дипломний проєкт

студента відділення комп'ютерних технологій

Галицького фахового коледжу імені В'ячеслава Чорновола

студента IV курсу групи КН-41

Касіяна Максима Ігоровича

(прізвище та ініціали)

Спеціальність 122 „Комп'ютерні науки"

Обсяг дипломного проєкту: _67__ стор.

Тема: «Програма шифрування та синхронізації даних в комп'ютерній мережі»

1 Актуальність теми: забезпечення кофіденційності та доступності інформації в комп'ютерних мережах є необхідною та актальною задачею. Розробка програмного засобу для шифруання та синхронізації файлів в компютерній мережі є розповсюдженою практичною задачею.

2 Практична або теоретична цінність опрацьованих питань: Розроблено програмний засіб, для шифрування та синхронізації даних використовуючи SMB протокол. Проведено техніко-економічне обгрунтування запропонованих рішень.

3 Недоліки роботи: в роботі доцільно привести діаграму розроблених класів.

4 Загальний висновок: робота відповідає вимогам до дипломних проектів освітньо кваліфікаційного рівня «молодший спеціаліст» зі спеціальності 122 «Компютерні науки» та заслуговує оцінку «відмінно»

Рецензент _Івасьєв Степан Володимирович_____

(прізвище та ініціали рецензента)

«_22_» _червня_ 2023. _____

UNICHECK
by Turnitin

Ім'я користувача:
Василь Кузик

ID перевірки:
1015615614

Дата перевірки:
15.06.2023 15:52:42 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
15.06.2023 16:00:11 EEST

ID користувача:
100012366

Назва документа: Касіян

Кількість сторінок: 61 **Кількість слів:** 11093 **Кількість символів:** 81644 **Розмір файлу:** 983.57 KB **ID файлу:** 1015263101

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

0.46%
Схожість

Найбільша схожість: 0.11% з Інтернет-джерелом (https://git.woozle.org/neale/Marlin/commit/c1ff38c7a0d2d57863832421

0.46% Джерела з Інтернету | 24 ..... Сторінка 63

Не знайдено джерел з Бібліотеки

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Підозріле форматування | 12 сторінок