**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

**НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**

**КАФЕДРА КОМП'ЮТЕРНИХ НАУК**

# Кваліфікаційна робота

на тему: «**РОЗРОБКА ВІДДАЛЕНОЇ БЕЗПЕЧНОЇ ДЕЦЕНТРАЛІЗОВАНОЇ СИСТЕМИ ЗБЕРІГАННЯ ТА ОБРОБКИ ДАНИХ**»

на здобуття освітнього ступеня бакалавра
зі спеціальності 122 Комп'ютерні науки
*(код, найменування спеціальності)*
освітньо-професійної програми Комп'ютерні науки
*(назва)*

*Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело*

_____ Микита ТЕРЕМКО
*(підпис)* *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав: здобувач вищої освіти гр. КНД-41

Микита ТЕРЕМКО
*(Ім'я, ПРІЗВИЩЕ)*

Керівник: Микола ГНІДЕНКО
*Науковий ступінь, вчене звання* *(Ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
*Науковий ступінь, вчене звання* *(Ім'я, ПРІЗВИЩЕ)*

**Київ 2024**

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

**Навчально–науковий інститут інформаційних технологій**

Кафедра Комп'ютерних наук
Ступінь вищої освіти − «Бакалавр»
Спеціальність підготовки − «122 Комп'ютерні науки»
Освітньо-професійна програма − «Комп'ютерні науки»

**ЗАТВЕРДЖУЮ**
Завідувач кафедри Комп'ютерних наук

_____Віктор ВИШНІВСЬКИЙ
«_____» _____ 2024 р.

# ЗАВДАННЯ
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

Теремко Микита Олександрович
(прізвище, ім'я, по батькові)

1.Тема кваліфікаційної роботи: «Розробка віддаленої безпечної децентралізованої системи зберігання та обробки даних»

Керівник кваліфікаційної роботи Микола ГНІДЕНКО к.т.н, доцент.
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від 23.02.2024 №36

2. Строк подання студентом роботи «31» травня 2024 р.

3. Вихідні дані до роботи: науково-технічна література з питань реалізації хмарних технологій, Intrusion-Tolerance Using Fine-Grain Fragmentation-Scattering: M. Fray, Y. Deswarte, and D. Powell; A Distributed Architecture for Secure Database Services: G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu; Dependable and Secure Storage in a Cloud-of-Clouds.: A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa; Performance study of selective encryption in comparison to full encryption for still visual images: O. A. Khashan, A. M. Zin, & E. A. Sundararajan.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Рішення питання, реалізації методів фрагментування, та де фрагментування;
2. Створення абстрактної інформаційної карти даних для користувача;

3. Реалізації метода перевірки цілісності фрагмента, та файлу;
4. Реалізація алгоритму шифрування, та дешифрування фрагменті;
5. Реалізації метода перевірки цілісності фрагмента, та файлу;
6. Рішення для безпечної, дистрибутивної системи зберігання обробки даних.

5. Перелік ілюстративного матеріалу: презентація
1. Мета та актуальність роботи
2. Перехід до дистрибутивнооого хмарного середовища
3. Персональна інформація користувача
4. Алгоритм Фрагментації
5. Найважливіші властивості фрагментації
6. Врахування залишкових біт та цілісності файлу
7. Алгоритм дефрагментації
8. Порівняння кінцевих файлів
9. Шифрування даних
10. Використання пас-фраз
6. Дата видачі завдання «23» лютого 2024 р

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Підбір науково-технічної літератури | 24.02-10.03.24 | Виконано |
| 2 | Аналіз перспектив хмарних технологій з використанням безпечного фрагментованого, шифрованого сховища | 11.03-30.03.24 | Виконано |
| 3 | Створення методів фрагментації, та дефрагментації | 01.04-10.04.24 | Виконано |
| 4 | Створення методів шифрування, та дешифрування, криптографії | 11.04-30.94.24 | Виконано |
| 5 | Створення абстрактної карти інформації користувача | 01.05-16.05.24 | Виконано |
| 6 | Вступ, висновки, реферат | 17.05-18.05.24 | Виконано |
| 7 | Розробка обов'язкових демонстраційних креслень | 19.05-20.05.24 | Виконано |

Здобувач вищої освіти _____ Микита ТЕРЕМКО
*(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Микола ГНІДЕНКО
*(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

# РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 69 стор., 17 рис., 8джерел.

*Мета роботи* – розробка безпечних методів обробки, зберігання інформації, як даних, використовуючи технології децентралізованого сховища, також шифрування фрагментів інформації, та знаходження різних методів маніпулювання даними. Для покращення безпеки, та взаємодії роботи хмарних платформ. Реалізація цих методів на мові програмування Java.

*Об`єкт дослідження* – процес безпечного зберігання обробки даних користувача, в інфраструктурі хмарних платформ.

*Предмет дослідження* – методи запобігання несанкціонованого отримання доступу до даних користувача, котрі зберігаються в хмарі.

*Короткий зміст роботи:* у даній роботі реалізується, методи фрагментації, дефрагментації, шифрування, дешифрування, створення абстрактної карти інформації, перевірки цілісності.

Для реалізації цих методів використовуються, такі бібліотеки на мові програмування Java, як: com.fasterxml.jackson; org.bouncycastle.jce; org.bouncycastle.openpgp; org.bouncycastle.bcpg; java.security.KeyPair; java.security.KeyPairGenerator; java.security.Security; java.security.SecureRandom, а також й інші.

Під час роботи було детально розглянуту різні алгоритми впроваджуючи безпеку, що стосовно криптографії, а також і логічних можливих послідовностей взаємодії з даними.

## ABSTRACT

Text part of the qualification work for the bachelor's degree: 69 pages, 17 pictures, 8 sources.

*The purpose of the work* – developing secure methods of processing and storing information as data using decentralized storage technologies, encrypting fragments of information, and finding different methods of data manipulation. To improve the security and interaction of cloud platforms. Implementation of these methods in the Java programming language.

*Object of research* – the process of securely storing user data in the infrastructure of cloud platforms.

*Subject of research* – methods to prevent unauthorized access to user data stored in the cloud.

*Summary of the work: t*his paper implements the methods of fragmentation, defragmentation, encryption, decryption, creation of an abstract map of information, and integrity checks.

To implement these methods, the following libraries in the Java programming language are used: com.fasterxml.jackson; org.bouncycastle.jce; org.bouncycastle.openpgp; org.bouncycastle.bcpg; java.security.KeyPair; java.security.KeyPairGenerator; java.security.Security; java.security.SecureRandom, as well as others.

During the work, various algorithms for implementing security were considered in detail, both in terms of cryptography and logical possible sequences of interaction with data.

**ЗМІСТ**

# ВСТУП

Використання хмарних сховищ, вже досить давно стало повсякденністю. Любий додаток, чи сервіс зберігає особисту інформацію користувачів використовуючи хмарні технології. Це достатньо ефективно, та економічно вигідно. Це стосується не тільки можливості зберігання даних з веб додатків, чи програмних застосунків. Наразі користувач має можливість реалізовувати свої власні рішення, беручи в оренду певний, саме потрібний йому обсяг інфраструктури хмарних платформ. Тому першочерговим заходом при роботі з інформацією, потрібно подбати про безпеку системи зберігання, обробки інформації, як даних.

Що річно у всьому світі трапляються значні інциденти, пов'язані з втратою, чи отриманням несанкціонованого доступу до певної інформації. Це приносить значні збитки всім, хто потрапив в руки хакерів, та шахраїв. Наслідки можуть бути занадто вагомими, та їх розповсюдженість зможе доторкнутись майже кожної людини. Тому безпека інформації, як фактор впливу на світ, дійсно важлива.

При відсутності можливості, любого іншого користувача, навіть провайдера хмарного сховища, отримати доступ до персональної інформації, окрім власника цієї інформації. Ми можемо значно підвищити рівень захисту всього середовища хмарних послуг. Та забезпечити користувачів впевненістю безпеки в прийдешньому дню.

В допомозі реалізації цієї задачі, будуть використані таки методи, як фрагментація, шифрування, формування абстрактної карти інформації. Розсіювання інформації, як метод дистрибутивності. Маючи лише особисту власну послідовність зберігання фрагментів в сховищі, котрі в додаток будуть належним чином шифруватися, та матимуть змогу відтворення для проведення, будь яких маніпулятивних дій, котрі будуть цікаві користувачеві. В додаток ціновий аспект взаємодії з хмарним середовищем, можливо і буде трішки дорожчим, але це зовсім не вплине, в рахуванні мінімізації ризику втрати інформації, або заволодіння нею зловмисником**.**

# 1 ТЕОРЕТИКО-МЕТОДИЧНІ ОСНОВИ

## 1.1 Історія створення хмарних платформ

Вже понад 40 років аутсорсингове зберігання та обробка інформації є актуальними. Ідея що до доступності даних в будь-якій точці планети та будь-коли виникла в Джозефа Ліклайдера ще у 1960-х під час його роботи над ARPANET мережею, яку вважають початком Інтернету. Концепція була досить вдалою, але вона випередила час розвитку сприйняття інформаційно-комунікаційних технологій на понад 20 років. На той час звичайному користувачу було занадто складно співпрацювати з функціоналом, та цілковито зрозуміти доцільність цієї системи. Вже в 1983 американською онлайн інформаційною службою CompuServe було виведено концепцію хмарного сховища на відкритий простір з доступом для споживачів, та можливістю використання 128 кілобайтів дискового простору, що тоді давало змогу збереженню будь-яких файлів[1]. З того часу і по сьогодення системи віддаленого зберігання обробки інформації є мейнстрімом.

## 1.2 Факт повсякденності хмарних платформ завдяки безпеці

В наші дні хмарні послуги, платформи та і в цілкому хмарне середовище зайняли більшу частину Інтернет мереж. Це тому, що існують такі аспекти котрі впливають на безпеку всього масиву інформації, як об’єкта даних, а саме це захист, цілісність, конфіденційність даних. Дані мають свій життєвий цикл, або цикл існування. Він починається з запису, або створення нової інформації, і приходить кінцю в момент її видалення власником, зазвичай це момент настає суміжно з їх не актуальністю. Під час циклу існування даних вони повинні бути ретельно захищені. І мова йде не лише про безпечне зберігання даних, але й про безпечну передачу інформації з одного вузла до іншого. В цьому випадку небезпека являє собою можливість втрати конфіденційної складової даних та неможливості бути впевненим цілісному прибуттю до кінцевого вузла. Сьогодні хмарні платформи, центри обробки, зберігання даних перебувають в перманентному стані напруги, бо атаки зловмисників, хакерів є постійними їх докучаннями. Не в залежності стійкої

надійності механізму захисту від втручання осіб, котрі шукають метод заволодіння чужою інформацією, але механізми своєю чергою встановлюють та оновлюють. Захист даних в досягнені безпеки, конфіденційності є першочерговим для користувачів загальнодоступних чи гібридних системах зберігання, обробки даних.

### 1.3 Розвиток методів захисту інформації

Для захисту інформації, як даних існують різноманітні способи: пильно перевіряти особистість котра намагається отримати доступ до інформації (використовуючи автентифікацію, чи політики доступу до сервісу), видозмінювати інформацію для всіх інших крім власника з можливістю мати релевантний доступ після (тут може бути стеганографія – це тайнопис, що має контекст лише для тієї особи котра повинна отримати інформацію, також підходить скремблювання – це перетворювання інформації на незрозумілий, хаотичний, стохастичний шум для всіх інших, крім зазначеної особи, котра в змозі знайти істинну взаємодію для отримання реального потоку даних, існує ще такий метод, як шифрування – це перетворення об’єкта даних на шифр, завдяки різним алгоритмам цифрового, переміщення бітів, котрий може також бути розшифровувальним, що значить перетворити весь об’єкт на своє початкове, та цілісне значення, лише власник ключа доступу, або пас-фрази має змогу на оборотній процес, анонімізація – це також один з методів захисту інформації, що несе в собі зміну або видалення ідентифікації даних, що унеможливлює, варіант мати, хоча б якусь взаємодію з інформацією , крім власника володіючого цією інформацією, затушовування – це перетворення чутливих, оригінальних даних на щось релевантне сприйняттю, але не відповідаюче дійсності, з можливістю повернути все в цілісну форму для особи котра володіє даними). Існують й інші методи, але головний принципи взаємодії юзера та даних, повинен заборонити не санкціонованого перетину бар'єра на втручання в інформацію, не маючи прав(методів, ключів, карт) на володіння даної інформації. Під час одного, чи послідовності цих процесів, юзер має змогу крім захисту, також перевірити свої дані на момент їх відповідності, та визначити випадкову або навмисну зміну інформації. В свою чергу, ці методи можуть дорого

коштувати в розумінні ресурсо не ефективного витрачання пам’яті на хмарній платформі, серверном просторі, та можуть займати довгий період часу реалізації в залежності використання методу, та початкового об’єму масиву інформації.

На основі цієї інформації, у мене виникла ідеє по створенню нового відносно швидкого, та більш традиційного методу захисту інформації: фрагментації даних. Фрагментація це метод з використанням досить складного алгоритму поділу даних на частини або фрагменти, враховуючи характер, розмір даних і безпеку, як головний аспект. Для економічної складової обробки даних, щоб бути ефективним, фрагменти будуть мати різні методи захисту в залежності від їх рівня критичності або конфіденційності. Характер даних, буде варіюватися залежно від їх типу, це може бути текст, зображення, відео матеріал, також це буде мати категоричний вплив на тип проводження алгоритмічного аналізу, методу, процесу, для отримання значної ефективності. Використовуючи тип, характер, метод взаємодії ми будемо мати змогу ефективного розсіювання даних по різних фізичних машинах використовуючи деякі стохастичні фактори, котрі будуть унеможливлювати збірку повноцінного файлу, як об'єкту не маючи хешу інформації про його місце знаходження. Це дасть нам змогу анонімізувати дані від юзерів, провайдерів, та залишити лише відомі координати особі котра має до них доступ. Це інкрементує рівень безпеки зберігання інформації.

Припускаємо, що об’єм даних, котрий має достатньо великий розмір певною мірою не має одного рівня критичності, чи конфіденційності, це дає змогу аналізувати, а потім використовувати, більш різноманітний спектр алгоритмів по захисту даних. Дані котрі підходять до кінцевого життєвого циклу також можливо відокремити від всієї основної маси постійно використовуваних, це можливо описати як сегментацію. При використанні методу розсіювання, децентралізування, фрагментації даних, дуже доречно використовувати різноманітні в залежності від кваліфікації даних, додаткові методи захисту шифрування, на мій погляд, сюди неймовірно підходить асиметричний тип шифрування при реалізації котрого користувачеві буде надано хеш значення даних котрі розсіяні по різним серверам для можливості спочатку знаходження

фрагментів, також буде виданий приватний ключ та пас-фраза з котрими він зможе провести швидке розшифрування його фрагментно, а наступним етапом ми перейдемо до компонування фрагментів в цілу одиницю даних(впровадження дефрагментуючого процесу), маючи крім хешу значень, абстрактну реплікацію даних на фізичному пристрої у самого користувача(котра має знаходження по серверах хмарного середовища, тотальну суму всієї одиниці інформації, як даних, приватні ключі, хеш фрази, використовувані типи шифрування, деякі помітки, що до статистичних відображень), що дасть змогу безпечно відновити весь об'єкт даних котрий потрібен користувачеві, на певному етапі. Слід зазначити що вся взаємодія користувача з хмарним сховищем, повинна бути проведена в хмарному середовищі на захищеній віртуальній машині від'єднаної від самого середовища хмари, що дасть можливість обмежити провайдера для переглядів любої інформації на хмарі, не маючи місцеперебування всього файлу, ключів отримання доступу інформації, і додаткових аспектів взаємодії з даними, що знов же таки підвищую рівень безпеки для користувачів. Локальна, абстрактна, псевдо база даних, на основі котрої, ми зможемо конфігурувати всю систему маючи певний спектр інформації котра зберігається на фізичному носії у самого користувача, це дає змогу залишити безпеку на самого користувача без відповідальності провайдера.

## 1.4 Сучасний стан дослідження в області безпеки децентралізованих систем зберігання даних

Без різниці яке програмне забезпечення, чи вирішення використовується для збереження конфіденційності, першочергово ми повинні використовувати лише захищене апаратне забезпечення (так звану надійну територію декількох комп’ютерів) в певний момент коли наші дані перебувають в най вразливішому стані в період їхнього існування. Зокрема найуразливіші стани виникають, в місцях де інформація підлягає фрагментуванню, дефрагментуванню, сегментації, шифруванню, запису певних важливих метаданих, коли знаходиться на перед децентралізованому етапі.

Це обумовлюється чистотою самої інформації котра має період часу існування. Ще залишаються місця де виникає генерація самої інформації в потік, як даних. Також ґрунтовно створюється, має процес відображення, як візуалізації, для прямого сприйняття, обробки юзером, кінцевим споживачем, і цей момент також повинен мати ретельний захист, від несанкціонованого вторгнення, чи читання, взаємодії, і най базовіший факт використання надійної території.

Надійна територія може слугувати, як скриня чи сейф для депонування над конфіденційної інформації. І навіть втому випадку, якщо ця інформація вже надійно захищена різними алгоритмами. Безліч публікацій, давно і по сьогоднішні дні відображають це, наприклад [2] або [3]. Безперечно, з точки зору фрагментації дані не повинні входити, та виходити через один порт за одну ітерацію з безпечної зони, якби це існувало, наш потік даних був схильний до ризику перехопленню, або наша цілісна частина котрій ми довіряємо, стала б нашим отвором в безпеці передачі інформації. І зловмисник мав би змогу побачити весь прохідний трафік інформації.

Для уникнення потрібно розробити методи обробки збору, передачі інформації в ізольовану частину і на зовні до решти систем інфраструктури хмарного середовища. Хоча вся ця ізольована система має достатньо високу вартість, і більшість провайдерів зараз нехтують безпекою даних, як інформацією своїх споживачів. Відокремити ізольовану частину від хмарної інфраструктури поки не є, а можливо і не буде вірним варіантом розгортання системи, опираючись на зазначені факти вище, якщо ми маємо наміри досягти по справжньому високого рівня конфіденційності.

### 1.5 Фрагментація як захист

Фрагментація – не є чимось новим, як ідея. Ця концепція вже давно використовується в різних сферах як в економіці чи історії. Як говорять банкіри, «щоб мати спокій душі, краще не складати всі гроші в однин сейф».

В інформатичних науках, розуміння фрагментації, є також в різних варіаціях програм та використання. Це може бути використовувано в операційних системах,

для покращення, та оптимізації управління дисковим простором. А також в розподілених системах, для покращення рівня продуктивності. Є досить багато різноманітних технологій котрі мають можливість підтримки процесів фрагментації даних, наприклад: Raid, Apache Hbase, Google File System тощо…

Фрагментація даних, як захист інформації, була винайдена, ще в стародавні часи. Це сталося зразу, після виникнення винаходу писемності, одне і головне значення котре утворювалось завдяки їй, це безпека, а точніше, використання фрагментації, як збереження таємниць. Це один з перших примітивних, але достатньо діючих, для запобігання несанкціонованого доступу, алгоритмів шифрування. Шифр складався з певної кількості частин, і кожна частина являла собою, так званий ключ. Якщо ми припустимо ситуацію, в котрій певна важлива інформація буде написана на листі паперу, і нам потрібно передати цю інформацію іншій особі, таким чином, щоб ніхто крім неї не скористатися цією інформацією при нагоді. Для реалізації цієї задачі, нам потрібно згенерувати два масиву критеріїв характеру шифру, котрі будуть визначені перед трансфером між особами адресанту та адресату інформації. В першому критерії, адресант та адресат визначають кількість частинок на котрі буде подрібнений лист паперу (контрольну вагу), що стосовно другого критерію масиву, то його значення в маркуванні послідовності для повторного відтворення, це може бути послідовність певних знаків, або символів, котрими промарковані всі подріблені папірці. Це створює додатковий ключ (ключ відображення), для повторення раніше зазначеної для взаємодії послідовності. Знаючи цей алгоритм і реалізуючи його ми шифруємо раніше створене повідомлення, вказуючи першим раніше обумовленим критерієм 10 частинок блоків з інформацією, а другим знову ж таки, раніше обумовленим, вказуємо послідовність відміток на папірцях повідомлення, та ховаємо його в конверт. При загадкових обставинах конверт потрапляє в руки шахраєві котрий відкриває його під час відправки адресату. Бачучи лист паперу розділеного на 10 шматків, кожен з яких має надписи з обох боків, він може розгледіти йому знайому мову, та зібрати цей пазл, але якщо ми припустимо, що кількість розділених блоків збільшиться, хоча б до 100, розшифрувати таке повідомлення, вже нагоди не

уявляється, навіть коли всі ключи формування контрольної ваги в нього. Інформація залишається захищеною. Ми навіть маємо змогу покращити наший алгоритм додаючи замилюючий фактор, наприклад контрольна сума, вона може бути меншою за кількість блоків, це можливо зробити додавши блоків з хибними значеннями, або повторним використанням вже існуючих, але одночасно в декількох частинах.

Зраз виникає все більше причин, котрі потребують рішення саме з використанням фрагментації(розсіювання), як алгоритму зберігання і захисту інформації. Архітектури розподілених сховищ постійно розширюються, користувачами і машинами, та створюється неймовірна кількість даних щоденно, центральні процесори потребують більш ефективніших архітектур для аналітики, обробки заявок, що най головніше передачі даних з одного сервера до іншого. Зазвичай хмарне середовище, має підтримку одночасно декількох взаємопов’язаних центрів обробки інформації, як даних. Серед котрих, можна розподіляти, зберігати блоки інформації ефективно. Дані можуть бути розосереджені таким чином [4], так як центри оброки інформації мають велику сукупність розділених серверів та дистрибутивну пам’ять. Це дає змогу нам розсіяти велику кількість фрагментів, зберігаючи їх для взаємодії на різних машинах, чим менше фрагментів котрі мають можливість відтворення в одному середовищі, тим безпечніше стає процес зберігання обробки інформації. Також слід зазначити, не припустимість з’єднування даних різноманітним типами посилання, на кшталт систем блокчейн, щоб при нагоді отримання відповідного ключа доступу лише до одного з фрагментів, зловмисник не мав змоги знайти місце знаходження іншого фрагменту, тим самим не міг зібрати до купи файл. Таким чином знання одного фрагмента не пояснює його взаємодію з іншим, не створюється симантичної послідовності. Та і в цілком система не могла йому відображати, хоч щось релевантне для сприйняття, навіть при проникненні на саме середовище сервера зберігання оброки інформації. Відчуття зловмисника повинно бути порівнянним з пошуком багатьох голок в не відомих йому стогах сіна.

### 1.6 Що таке фрагментація інформації в даному проекті

В даному проекті фрагментація інформації являє собою методологію розділення даних. Дані будуть ділитись за певними алгоритмами в залежності від контрольної суми ваги, розміру файлу. Одиниця розділення, та зберігання файлу з котрих він весь потім збирається, має назву фрагмент. Кожен фрагмент буде мати свою оболонку, котра буде захищати його в залежності від цінності самого файлу. Сама оболонка - це кінечна форма захисту фрагменту. В неї є досить багато атрибутів, котрі є лише в самого власника файлу до котрого належить фрагмент для отримання коректного доступу та цілісної дефрагментації. І ця властивість захищеності, безумовно лежить в методології, як одній з ідеологій захисту інформації.

Сама система фрагментації даних, безумовно відповідає другому принципу Керкгоффа (див. оригінальна стаття (французькою мовою) [5] , також в приклад можливо привести вичерпну книгу з історії криптографії[6]. Зловмисник навіть маючи повний алгоритм роботи фрагментації, не в змозі отримати доступ до дефрагментованої інформації, так як сама послідовність конкатенацій для створення фрейму інформації лише у юзера, ключ до фрагментів також знаходиться у власника, тому інформація в безпеці.

Обумовлено $R$ множину колекцією масиву фрагментів, як повною одиницею даних, котру власник хоче піддати захисту.

Позначимо її через $n = |R|$, де $n$ є кількістю всіх фрагментів. (1.1)

З точки зору термінології вданому проекті, поняття сегментації та фрагментації даних, мають категорично різні значення. Сегмент є результатом сегментації. Його суть полягає не в простому розділенню даних, як одиниці інформації за певним алгоритмом, як це було в методології фрагментації. Сегментація повинна робити інформацію класовою. Так першочергово, весь об’єкт інформації буде розділений, але його поділ має критерії. В цілком значення сегмента в людському, чи машинному розумінні є більш зрозумілим чим фрагмента. Фрагмент являє собою відірвану частину чогось, як загальної інформації наприклад послідовності,

матриці бітів, це може бути маленьке не зрозуміле значення, якщо не придати йому контекстної уваги. Сегмент в свою чергу являє собою сином фрагмента, так як там також є шматки або блоки інформації, але вони підлягають певній визначній категорії (сортоване відокремлення). Сегмент особливий вид фрагмента.

Фрагментація даних, в розумінні захисту інформації, на перший погляд досить проста методологія. Але насправді достатньо тяжко спроектувати, та розробити рішення на рівні інфраструктури проекту, котре буде задовольняти вимогам ефективності системі хмарного сховища, з зазначеним моментом дефрагментації (компонуванням файлів), без неочікуваних витрат.

Фрагментація дає змогу відокремити дійсно критично важливу інформацію, для проведення додаткового бар'єру захисту, завдяки методам шифрування, за для збереження ефективності всієї системи хмари. Важливу роль в фрагментації відіграє розмір фрагмента, кожен не фрагментований файл повинен формувати кількість під файлів, за певним алгоритмом. Якщо в системі будуть зберігатися занадто малі за розміром фрагменти, то при збірці файлу, ефективність системи буде терпіти нестерпні накладки на критичному рівні. Накладки можуть привести до втрати одного або декількох фрагментів, що вже є вірогідно, не дуже добре для користувача. Якщо це будуть занадто великі фрагменти інформації, то вони стають більш вразливими до не санкціонованого доступу інформації. Слід зазначити що алгоритму розділенню на фрагменти, потрібно приділити ретельну увагу, зробити фрагментацію для кожного з популярних типів даних, використовуваних в сховищах. Також об'єктно орієнтований підхід буде влучним, для організації, взаємодії з даними.

### 1.7 Що таке абстрактна карта інформації

Абстрактна карта інформації являє собою сукупність певних даних, котрі дають можливість коректної роботи системи розподіленого зберігання, оброки інформації. В нашому проекті, вся карта знаходиться у самого користувача на фізичному носії. Фізичний носій може бути, як флешкартой, чи жорстким диском. В абстрактну карту інформації входять конфігураційні дані, котрі формують

обліковий фрейм користувача і дають змогу аутенфікуватися в саму його систему. Також в абстрактній карті знаходиться інформація, що до розташування певного фрагменту інформації на сервері хмарної платформи, для можливості дефрагментації їх в майбутньому. Файли котрі знаходяться на фізичному носії є з нульовою вагою суми інформації, це абстрактне розуміння файлу. В кожному з абстрактних файлів зберігається, справжнє ім'я файлу, шлях до певної кількості фрагментів, котрі будуть знайдені на серверах обробки, зберігання інформації, та їх послідовність збирання до купи, після файли будуть скопозиційовані на безпечній зоні, так званої віддаленої, безпечної, розподіленої, віртуальної машини, котра в свою чергу піднімається провайдером хмарної платформи для користувача. Також на фізичному носії зберігаються ключи, та пас-фрази доступу до цієї інформації для можливого розшифрування її вже на самій віртуальній машині. Крім цих факторів, також зберігаються дані юзера для облікової сторінки, та можливості відновлення його в системі. Для коректного відображення даних фізичний носій депонує в собі ще контрольну вага кожного не фрагментованого, повноцінного файлу для перевірки цілісності після збирання розсіяних частинок, та розшифрування їх і збирання до купи в один цілий файл. Крім всіх вже перерахованих фактів інформаційних об'єктів, є ще змога поліпшувати безпеку новими об'єктами, та вносити їх в систему завдяки об'єктності та масштабованості всього проекту.

### 1.8 Що таке дефрагментація інформації в даному проекті

Дефрагментація в цьому проекті являє собою метод компонування фрагментів, для повного відображення візуалізації даних, як інформації. Місце де проводиться дефрагментація інформації є з підвищеною вразливістю, тому що це єдине місце де в кінцевому результаті є можливість компрометації інформації. Також це місце є досить цікаве для зловмисників, через однократному потраплянні інформації з абстрактної карти користувача. Також слід зазначити, якщо це місце не буде мати надійного захисту, зловмисник отримавши доступ до нього зможе відтворити абстрактну карту інформації користувача. Дефрагментація це той

метод при відтворення котрого йому потрібен найретельніший захист, так як для його відтворення потрібні такі дані, як контрольні суми файлів, отримання ключів та пас-фраз, місце знаходження кожного фрагменту для його кмпонування, правильна послідовність зшивання фрагментів. Чим більше фрагментів, тим складніший процес дефрагментації. Також слід зазначити, що ефективність цього методу можливо підвищити різними іншими алгоритмами, завдяки котрим можлво дефрагментувати не весь об'єм інформації, а лише потрібний для самого користувача. Крім того дефрагментація повинна повертати 100 відсотків від початкового розсіяного файлу, для коректної взаємодії з ним, це ще можливо охарактеризувати, як повна цілісність файлу. Також слід уникати, або вирішувати різні фактори, котрі в свою чергу можуть спричинити втраті цілісності файлу. Не використовувати крім раціональних (цілих) чисел, ніяких інших. Також слід передбачити надійність каналів передачі інформації. Враховуючі ці аспекти, слід реалізовувати метод дефрагментації.

Дефрагментацію ще можливо розглядати, як закінчення циклу існування фрагментів. Якщо користувач виконав метод компонування даних, як дефрагментацію, після закінчення роботи, модифікації, оновлення, обробки іформації, при завершенні сеансу, всі дані будуть знову фрагментовані, та R множина фрагментів буде розсіяна по дистрибутивному сховищу, маючи нові шляхи зберігання фрагментів. Також зміняться на нові, всі інші атрибути абстрактної карти інформації. Це дає змогу звести, ще одну стіну безпеки між даними, як інформацією користувача хмарної платформи, та зловмисника.

## 1.9 Продуктивність, та її покращення в дистрибутивній системі обробки зберігання даних

Фрагментування та дефрагментування інформації, як даних досить ресурсо затратний механізм роботи. В розумінні ресурсів виступає першочергово час, та енергія. Також слід зазначити навантаження на оперативну пам'ять сервера, та і на постійну пам'ять тако. А саме тому що головинм критерієм фрагментації в нашуму випадку виступає захист, щоб задовольнити цей критерій потрібно депонувати

інформацію дистрибутивно, між декількаами серверами. Сам критерій продуктивності є критично важливий, для повноцінного функціонування, та використанням юзером системи. Ця технологія не може коштувати дешево, в порівнянні з не захищеним середовище зберіганням даних. Додаткову продуктивність можливо отримати завдяки використанню паралельних обрахунків. Розпаралелювання процесів, дасть змогу значно пришвидшити використання фрагментації. Також в нашому проекті ми використовуємо такі поняття, як шифрування та дешифрування даних, котрі слугують підвищеню безпеки. Ми можемо прискорити і ці ефективно затратні процеси завдяки розділеним розрахункам. Але потрібно звернути рительну увагу при підборі методів та алгоритмів шифрування, задля запобігання конфліктності, та перевірки підтримки розпаралелюваних процесів. Також нам потрібно визначитись з розмірами фрагментів для процесів їх взаємодії в майбутньому. Якщо ми будемо використовувати надто малі фрагменти, чи занадто великі, система дистрибутивних обрахунків тільки погіршить нашу ефективність. Виходячи з даної ситуації, нам допоможе застосовування варіацій закону Амдала [7].

Архівування є також ризиковим процесом, при можливості інкремнтації нашої продуктивності, ми можемо не нароком змінити розташування нашого масиву фрагментів користувачів. Реплікації в такому випадку будуть досить доречні, але слід зазначити, що реплікувати ми будемо лише попереднє місце розташування всіх файлів для більшої безпеки. Якщо ми визначили певні сектори пам’яті сервера де зберігаються рідко використовувані фрагментовані дані, нам потрібно першочергово зробити реплікації фрагментованих даних з їх ідентифікаційний номером цих секторівя, поті ми можемо спокійно робити архівування, поставити лічильник на запроси даних, котрі призводять до розархівування архіву, і після закінчення взаємодії з даними перенести їх в більш вживанні сектори. У випадку критичної ситуаццїї не відповідальності, можемо зробити бекап місць розташування інформації, та відновити систему взаємодії.

Ще один фактор котрий має важливий вплив на ефективність інфраструктури хмарного сховища, це пошук по системі чи сховищу певної інформації. Є певні

методи поліпшення даного процесу, це кешування, індексанція, використування хеш-таблиць, та словників.

### 1.10 Хмарні обчислення, та віртуальне середовище

Зазвичай провайдери хмарних платформ обрахунків пропонують користувачам 3 види послуг взаємодії з хмарною платформою. Перший вид є Software as a Service: цей вид дозволяє користувачам використовувати лише відокремлене програмне забезпечення хмарної інфраструктури провайдера, це інтернет сервіси. Прикладами можуть слугувати такі сервіси, як Google Workspace, Microsoft 365, Salesforce. Звичайна програма але знаходиться в онлайн Інтернет доступі. Другий вид є Platform as a Service: якщо перший вид SaaS давав лише можливість використання певного апп застосунку, без інсталяції його на комп'ютер, чи любий інший обчислювальний пристрій користувача, то PaaS дає змогу розгортання власної системи певних задач користувачів, котрі в майбутньому зможуть мати вже своїх користувачів на основі взаємодії з хмарою, та інстальованого в неї програмного забезпечення. PaaS це платформа як послуга, яка в собі вже містить певні алгоритми роботи для конструювання власних систем, також у користувача є можливість на використання вже інсуючих, розроблених хмарною платформою. Прикладами виду Platform as a Service, можуть слугувати такі сиситеми, як Google App Engine, Microsoft Azure, AWS Elastic Beanstalk. Третій вид взаємодії з хмарною платформую є Infrastructure as a Service: це най повноцінніший доступ до всього виділеного простору хмарної платформи провайдера, користувач може розгортати, розробляти, любе йому цікаві рішення, системи, платформи в рамках виділеного, орендованого простору провайдером. Прикладом Infrastructure as a Service, можуть слугувати такі системи, як Amazon Web Services , потім Microsoft Azure, та Google Cloud Platform. Це вид послуги корисний в зберіганні коштів, та не витрачання їх на власну інфраструктуру серверів.

Також є ще відокремлений варіант взаємодії з провайдером хмарної платформи. Це конструювання гібридних хмарних архітектур платформ вже як

надбудови. Гібридна інфраструктура, це поєднання з використанням мережевих технологій власної приватної інфраструктури з певними частинами публічних, хмарних платформ. Це дає змогу раціонально розміщати та використовувати кофіденційну інформацію, з більш ретельним захистом, також потужності середовищ, завдяки власним рішенням платформи. Гібридне підключення публічних до приватних хмар, відбувається завдяки мережевим технологіям з'єднання. Це може бути, як VPN (Virtual Private Network) віртуальна приватна мережа, також прямі з'єднання, та конструювання побудови таких мереж, як SDN (software-defined networking). В перевагах гібридних мереж над приватними, та публічними є такі фактори, як гнучкість, економічність, безпека, та масштабованість. Гнучкість дозволяє контролювати навантаження між відокремленими друг від друга, приватними та публічними мережами. Економічність дає змогу бути раціональнішим, що до сегменту зберігання даних, таких, як критично важливих, та просто важливих, це дає змогу зробити концентрацію більш важливих даних в приватній мережі, а дані котрі мають менше значення, що до безпеки, залишити в публічній. Безпека критичних даних, вже повністю залежить від користувача,  а не провайдера, тому розгортання рішень взаємодії можуть бути власні. Масштабованість є легкою реалізацією додаткових обчислювальних ресурсів, у випадку цієї потреби публічна хмарна платформа надасть певний об'єм кількості потужностей. Також при авріях, на одній з хмарних платформ, гібридна система надасть можливість перенаправлення потужностей, та швидкого відновлення повноцінної роботи. Під час впровадження нових інноваційних рішень, гібридна система буде досить доречною, тому як є можливість тестування, та відлагодження методів взаємодії спочатку  на публічній хмарній палатформі для більшої безпеки, та успішного перенесення на приватну.

### 1.11 Шифрування як метод захисту інформації

Шифрування є одним з най ефективніших, та наймасовіших методів захисту інформації, в інформаційному просторі. Головним його критерієм є захист інформації від не санкціонованого доступу. Основний метод реалізації полягає у

видозміні певних даних, котрі підлягають шифруванню, таким чином, щоб при моменті відтворення їх, тип даних був не можливий для будь якого сприйняття без певного ключа доступу до даних. Ключ доступу зазвичай виступає певною послідовністю цифр та символі, або одиниць та нулів, але інколи є ключі, як кастомізовані алгоритми дешифрування. З приводу алгоритмів котрі дешифрують дані на основі алгоритму шифрування, то дані котрі були зашифровані певним чином, достатньо вразливі, та шифр є не надійний.

Є декілька наймасовіших типів шифрування, а саме таких як: симетричне шифрування, асиметричне шифрування, гібридне шифрування. Симетричне шифрування, це шифрування в котрому використовується ключ, як для шифрування, так і для розшифрування. Прикладами алгоритмів симетричних шифрувань слугують такі види, як Advanced Encryption Standard та Data Encryption Standard. Головними перевагами симетричного шифрування являються такі аспекти, як швидкість, та ефективність. Головним недоліком симетричного вида шифрування є момент передачі ключа, а точніше безпека передачі між сторонами взаємодії. Асиметричне шифрування відмінно від симетричного використовує пару ключів публічний та приватний ключ . Публічний ключ використовується для шифрування даних, натомість приватний в асиметричному порядку використовується для дешифрування даних. Приклад алгоритмів асиметричного шифрування слугують такі види, як: RSA (Rivest-Shamir-Adleman) та ECC (Elliptic Curve Cryptography). Основним аспектом різниці, та переваги над симетричним шифруванням , полягає в безпечному розповсюджені публічного ключа шифрування, так як приватний котрий шифрує дані знаходиться лише в особи котра володіє даною зашифрованою інформацією. Недолік значний, але він менш важливий, чим компрометація зашифрованої інформації, це швидкість шифрування, вона може бути трішки повільнішою ніж у симетричного типу шифрування. Гібридне шифрування, містить в собі аспекти переваги, як і симетричного, так і асиметричного шифрування. Один з варіацій алгоритмів роботи гібридного шифрування полягає в симетричному шифруванні інформації і асиметричному шифруванню симетричного ключа, це дає змогу підвищити захист

інформації. Гібридний метод шифрування реалізовує один з алгоритмів системи PGP (Pretty Good Privacy).

Також існує такий вид шифрування, як гомоморфне. Гомоморфне шифрування являє собою інноваційним методом захисту інформації, який дає змогу взаємодії обчислення над вже зашифрованими даними інформації, без необхідних мір розшифрування. Це означає, що для взаємодії певним чином за даними, вони можуть залишатися в зашифрованому виді, при цьому дані знаходяться в певній мірі в безпечному стані. Що головне результати обрахунків інформації, також знаходяться в зашифрованому стані. А повноцінне розшифрування можливе тільки завдяки відповідного ключа доступу. У гомоморфного шифрування є власні під типи шифрування, такі як : додаткове гомоморфне шифрування, множинне гомоморфне шифрування, повне гомоморфне шифрування. Додаткове гомоморфне шифрування, дозволяє конкатинації вже зашифрованих даних. Прикладом слугує Пайльє (Paillier). Множинне гомоморфне шифрування дає змогу множенню шифрованих файлів Прикладом слугує RSA (з використанням специфічних налаштувань). Та повне гомоморфне шифрування даних. Повне гомоморфне шифрування даних дає можливість використання будь яких обчислювальних операцій над вже зашифрованими даними. Прикладом слугує алгоритм Гентрі (Gentry's scheme). Основними перевагами гомоморфного шифрування є: конфіденційність, безпека, та гнучкість. Що до конфіденційності, вона проявляється в обробці інформації, як даних без явного проявлення інформації, для сприйняття, та розуміння її зміста. Безпека підвищується за рахунок, без необхідного використання додаткових алгоритмів захисту під час обробки, так як дані вже являються зашифровані. Гнучкість представляє виконання різноманітних обрахункових операцій над зашифрованими даними. Перейдемо до недоліків гомоморфного шифрування, вони тут присутні також. В гомоморфному шифруванні є два головних недоліки: недолік продуктивності, та недолік складного впровадження. Недолік продуктивності, оснований на різних досить складних алгоритмах реалізації шифрування, має критерій досить низької обчислювальної інтенсивності, що і робить його

недоліком. З приводу складного впровадження, як недоліка, воно базується на високому технічному рівні взаємодії, також впровадження повинно проводитися достатньо кваліфікованими спеціалістами з високими технічними знаннями, і вже маючим обширним досвідом реалізації даного алгоритму.

# 2 ФОРМУВАННЯ ІНФРАСТРУКТУРНОГО РІШЕННЯ

## 2.1 Формування алгоритму фрагментації

Фрагментація даних, це процес розсіювання інформації на фрагменти. Формування алгоритму фрагментації, я почну з опису основних факторів кінцевих фрагментів, з котрими будуть проводитись подальші операції взаємодії, такі як: шифрування, переміщення по певним частинам дистрибутивного сховища, розшифрування, та дефрагментація. До основних факторі фрагментації відноситься: кількість фрагментів з одного файлу відносно контрольній суми ваги файлу, та масив послідовного конкатинування. Для формування аспекту кількості фрагментів з одного файлу, буде врахована сума ваги файлу, для повноцінного розділення, та можливості коректного дефрагментування з можливістю відображення. Що до масиву послідовного конкатинування, він буде формованний на основі генерації індексів з врахуванням знаходження самого елемента в ньому, як посилання на фрагмент даних.

У цьому проекті об’єкт, як одиниця даних для проведення фрагментації являє собою одиницю файлу в об’єктному стилі. Так існує можливість депонування всього віртуального середовища користувача в дата фрейм і вже потім реалізації методу фрагментації, але виникає низка недоліків взаємодії, обробки, дефрагментації, захисту даних, як інформації. Недолік обробки включає в себе значне зниження ефективності роботи всієї системи, за рахунок рівномірного значення всіх фрагментів даних, без можливості їх відокремлення та відображення користувачу певного спектра, котрий йому потрібен в не залежності всього масиву зберігаємого його інформації. Недолік дефрагментації включає в себе знову ж таки довший, ресурсо-часу затратний процес компонування розсіяних даних, як всього фрейма користувача. Також при можливій неочікуваній критичній ситуації порушення цілісності даних, під час процесу дефрагментації, є варіація втрачання куди більшого об’єму інформації. Недолік взаємодії також тягне за собою аспект ефективності, під час процесу відображення даних потрібно завантажити весь фрейм,  що займає суттєво більше часу. Що стосовно захисту, фрагментація буде

проведена зразу на великому за масою, контрольної суми ваги фрейму, вірогідно фрагменти будуть мати більший розмір, так як середній або малий розмір фрагмента може призвести до одночасної великої затрати ресурсів хмарної платформи зберігання обробки інформації.

Розділення файлів на категорії, по його контрольній сумі вагів, відокремлює три основні типи розміру файлу, це: малі, середні, та великі файли. Мали файли будуть мати розмір від 0 байтів до 100 мегабайтів зберігаємої в ньому інформації. Середні файли будуть мати розмір від 100 мегабайтів до 1 гігабайту зберігаємої в ньому інформації. Великі файли будуть мати свій початок з 1 гігабайту закінчуючи 100 гігабайтами.

Стосовно малих файлів, було розроблено формулу, в котрій визначається сума фрагменту, який є одиницею *R*.

$$size(R_i) = (D \times r)\text{, де } r\ [0.1; 0.5]. \tag{2.1}$$

В даній формулі використовуються такі змінні, як :

*R* – це множина колекції масиву фрагментів, $n = |R, \dots T|$, де n є кількістю всіх фрагментів, а *T* у випадку необхідності є додатковою множиною;

*D* – це сума не фрагментованого файлу;

*r* – це псевдо випадкове число, на проміжку від 0.1 до 0.5 включно;

*size( )* – це розмір кожного фрагмента з *R*.

Після визначення суми фрагменту, ми можемо дізнатися кінцеву кількість фрагментів на котрі буде фрагментований файл.

$$k = \frac{D}{size(R)} \tag{2.2}$$

В даній формулі *k* являє собою тотальну кількість всіх фрагментів з котрих складається *R*.

Стосовно середніх та великих файлів, було розроблено формулу, в котрій визначається відмінно від маленьких файлів вже кількість фрагментів, які являються об’єктами $R$.

$$k = \left(\frac{D}{r \times f}\right) \tag{2.3}$$

В даній формулі використовуються такі змінні, як :

$D$ – це сума не фрагментованого файлу;

$r$ – це псевдо випадкове число, на проміжку від 0.1 до 0.5 включно;

$k$ – це тотальна кількість всіх фрагментів з котрих складається R;

$f$ – це бажаний розмір фрагменту.

$$f = (D \times y), \text{де } y \text{ це константа котра дорівнює } 0.25. \tag{2.4}$$

Таким чином ми формуємо кількість фрагментів для середніх та великих даних. Різниця формування між малими та середніми, великими даними є досить значною. Це рішення було спроектовано, та реалізовано, з причин підвищення подальшої ефективності роботи всієї програмної частини, таких аспектів як: компонування, дефрагментації, шифрування. Для більш коректної роботи інфраструктури хмарної платформи, без неочікуваних витрат . Якщо в системі будуть зберігатися занадто малі за розміром фрагменти, то при збірці файлу, ефективність системи буде терпіти нестерпні накладки на критичному рівні. Накладки можуть привести до втрати одного або декількох фрагментів, що вже є вирогідно, не дуже добре для користувача. Якщо це будуть занадто великі фрагменти інформації, то вони стають більш вразливими до не санкціонованого доступу інформації.

Також під час фрагментації слід враховувати такій термін, як залишкові байти, вони виникають за неможливістю передбачення всієї суми файла, як даних користувача попередньо. Це є занадто критично для ігнорування, приклад: користувач має файл розміром 105 мегабайт, котрий підлягає розділенню на 10 частин за нашим алгоритмом. Нами очікується, що буде 10 фрагментів кожний з

яких по 10 мегабайтів і одна частина 5 мегабайтів. Ні такого не буде, якщо не впровадити обробку залишкових бітів. В інакшому випадку ми втрачаємо 5 останніх мегабайт цього файлу, та потім цей файл не підлягає коректній дефрагментації, та відображенню взаємодії з користувачем. Перелік значних, не приємних факторів, котрим сприяє не оброблена залишкова частина: втрата даних, не коректна реконструкція файлу, зниження цілісності даних.

Було розроблено алгоритм уникнення залишкових байтів файлу під час фрагментації:

Перший крок: визначаємо тип файлу за критерієм розміру;

Другій крок: визначаємо кількість фрагментів в залежності від розміру зазначеного файлу з використанням певної формули;

Третій крок: визначаємо суму ваги одного фрагмента, а саме ділим суму ваги файлу, як його розмір на кількість фрагментів;

Четвевартий крок: створюємо буфер обробки інформації(читання та запису), головним критерієм э розмір однієї частини фрагмента;

П’ятий крок: проходження циклом по файлу зчитуючи заданий об’єм даних у буфер та переписуючи фрагменти по новим файлам.

Шостий крок: перевірка контрольної суми всіх розділених фрагментів з початковим файлом, та якщо є потреба створення додаткового фрагмента інформації(перевірка на залишкові байти).

Слід зазначити, що існує два типа фрагментації, такі як: статична та динамічна фрагментація.

Статична фрагментація являє собою розділення певного файла користувача, на раніше зазначену кількість фрагментів. Кількість фрагментів завжди залишається статичною в не залежності від типу, чи розміру фалу. Статична фрагментація має дві ключові переваги, такі як: простота реалізації, та продуктивність. Простота реалізації відбувається за рахунок фіксованого розміру фрагментів, та меншій кількості фактів котрі потрібно реалізувати для коректної праці медоту. Також під час статичної фрагментації розсіювання даних, як інформації, для подальшої взаємодії, не потрібно реалізовувати більшість методів

котрі дають змогу коректної роботи за рахунок перевірки контрольної суми фрагментів, так як вся платформа використовує це пропорціонально в залежності від файлу. В подальшій взаємодії з даними фрагментами, потрібно реалізувати метод дефрагментації(компонування) фрагментів, з урахуванням статичної фрагментації, реалізація методу буде куди простіша. Продуктивність в статичній фрагментації є фактором швидкого та ефективного, мало ресурсо-часо затратного процесу. Це відбувається завдяки вже передбачуваному, та оптимізованому процесу обробки даних. Головний фактор це фіксований розмір фрагментів даних. Всі подальші процеси взаємодії відлагодження системи вже налагоджені(простота, та ефективність), також можливо встановлювати певний ліміт фрагментів з котрим хмарна платформа завдячуючи своїм потужностям зможе обробити данні най ефективнішим шляхом. В статичній фрагментації, слід зазначити присутність недоліків, і вони також значні, як і переваги. З недоліків головними можна назвати такі, як: неоптимальне використання пам’яті, зменшення продуктивності при великих файлах. До неоптимального використання пам’яті, наводжу приклад: інснує файл вагою 1.5 мегабайти, для нього буде виділено два фрагменти, кожен з яких вагою по 1 мегабайту. В цьому випадку хмарне сховище втрачає половину мегабайта, а точніше половина одного мегабайта з двох, зовсім не буде використано. Другій недолік це зменшення продуктивності при великих файлах, а саме під час передачі, шифрування, дефрагментації, обробки фрагмента, котрий буде достатньо великий для повноцінного функціювання в розподіленій системі депонування інформації. Також цікавість зловмисників до великих фрагментів куди більша, чим до малим. Тому рівновага безпеки використання фрагментації, як захисту, не може бути використана на повну. Статична фрагментація наразі застосовується в таких місцях, як: файлові системи під час формування диску, розділяє його на кластери та фрагменти, потім використовується в базах даних при розділенні таблиць, та індексів, на фіксовані розмірами сторінки.

Динамічна фрагментація являє собою більш різносторонні рішення в порівнянні з статичною фрагментацією. Динамічна фрагментація дозволяє зробити процес обробки зберігання даних більш ефективнішим, завдяки можливості

пристосування до навколишнього стану системи. Головним критерієм котрий впливає, на ефективність системи хмарної платформи, це можливість керуванням кількості фрагментів пі час фрагментації. Даний процес розподілення інформації, може залежити від багатьох умов. Це може бути, як і корисно, так трішки не безпечно, але не тоді коли система інфраструктури побудована на основі вірних рішень конструктора. Такі умови, як розмір файлу, тип файлу, або необхідність впровадження специфічних вимог до продуктивності. Базовими характеристикам динамічної фрагментації слугують факти, як: змінний розмір фрагментів, та адаптивність. Змінний розмір фрагментів, має змогу змінюватись в залежності від типу чи розміру самого файлу, вимог продуктивності, можливого заисту завдячуючи не значному стохастичному впливу, чи інших критеріїв котрі будуть застосовані провайдерами хмарних середовищ, також в свою чергу ці атрибути забезпечують більш гнучкій та ефективний процес взаємодії. Адаптивність це можливість адаптації(вже зазначеної гнучкості) до умов навантаження на хмару, зміна обсягу файлу, і додаткові критерії провайдера. Безперечно є переваги, та недоліки. Почнемо з переваг, головними аспектами котрі можливо охарактеризувати, як переваги є: оптимізація та ефективне використання пам’яті, покращення продуктивності, зменшення результата фрагментації, як кількості фрагментів. Оптимізація та ефективне використання пам’яті є фактором, котрий дозволяє позбутися занадтого резервування пам’яті, що досить відрізняє динамічну фрагментацію від статичної, дає можливість ефективно використовувати доступний користувачеві простір зберігання даних. Покращення продуктивності, маючи всі фактори котрі були перераховані, як базові, покращення продуктивності виникає за рахунок адаптивного розміру фрагментів за певних умов. Провайдер різними методами, вже існуючими, експериментальними, котрі основані на доречному підборі фрагментів до певного типу даних і ситуації, може значно підвищити рівень продуктивності. Що до зменшення результата фрагментації, як кількості фрагментів, являє собою перевагу, із-за знову ж таки фактора, котрий вже був врахований, це меншої кількості фрагментів, а якщо бути точнішим, то краще фактор “меншості”, замінити на фактор “оптимальної кількості фрагментів”, на

основі методів провайдера, тоді це буде точно вірно. Настала черга недоліків, головними аспектами котрі можливо охарактеризувати, як недоліки динамічної фрагментації є: складність в відтворенні та реалізації, вищі накладні витрати. Складність в відтворенні та реалізації, при реалізації методів фрагментації, та її перевизначенні при певних обставинах, та додаткових значних критеріях, реалізація потребує більшого часу, та більш точних алгоритмічних обрахунків з врахуванням статистичних необхідностей користувачів. Вище накладні витрати, варіюються в залежності від певних атрибутів котрі використовуються в ході фрагментації. Не завжди ці атрибути можуть дешево коштувати з точки зору ресурсів, але ефективність дистрибутивного зберігання даних значно поліпшується. Застосування систем, методів динамічної фрагментації інформації, як даних. Сектори застосування динамічної фрагментації схожі з статичною фрагментацією, але на більш професійному рівні. Це може бути впроваджене в системах зберігання даних, таких сучасних файлових системах, як ZFS, чи Btrfs. Також слід зазначити використання в дистрибутивних системах. Та системах управління базами даних(СУБД), для ефективного отримання доступу, швидкої обробки, великого масиву інформації, оптимізації індексування. Динмаічна фрагментація має низку популярних алгоритмів, оптимізації використання сховища, таких як: перший підходящий (First-Fit), найменший підходящий (Best-Fit), найгірший підходящий (Worst-Fit).

Перший підходящий або (First-Fit) – це доволі простий та досить ефективний метод управління простором сховища інформації, головний аспект цього алгоритму розміщення нових даних, як інформації, в нашому випадку фрагментів в першому доступному певної, відокремленої частини хмарного сховища, котра задовольняє критерії їх зберігання. Цей метод використання пам’яті дозволяє швидко знайти підходяще місце для зберігання нових файлів, тим самим зменшити витрачання часу на пошук. Під значенням пошуку першого ліпшого місця для певного фрагменту, мається на увазі перебір по списку всіх доступних частин хмарного сховища. У разі якщо місце задовольняє всі вимоги фрагмента інформації, алгоритм резервує пам’ять та починає переписувати туди фрагмент, все

інше сховище залишається не змінним, натомість алгоритм не починає перевірку ще теоретично можливих місць. Цей алгоритм простий в реалізації, адже не містить складних аспектів. Також слід зазначити швидкість, та ефективність виконання даного алгоритму, так як прохід в пошуках місця для фрагмента зупиняється одразу ж після знаходження, без додаткових перевірок. Переваги алгоритму першого підходящого: швидкість, простота реалізації, фактори дійсності цих переваг ми вже описали. Перейдемо до недоліків алгоритму, а саме зовнішня не збалансована фрагментація, та нерівномірний розподіл. Зовнішня фрагментація це характеризується відсутності збалансування сховища по критерієм ваги даних, може призвести до погіршення ефективності взаємодії, на основі менш ефективного використання пам’яті. Одне з не найкращих аспектів алгоритму являє недоторканість до певних місць на диску, так як перший підходящий блок може розрівати декілька байтів, ці байти накопичуються з часом, і декілька відсотків всього сховища залишаються не використанні.

Найменший підходящий (Best-Fit) це метод управління пам’яті, простором сховища хмарної платформи, дисковим простором алгоритму динамічної фрагментації, головним виконуючим дію аргумента є система пошуку найменшого блоку з точки зору раціонального зберігання в ньому фрагмента, інформації. Відмінну від першого підходящого (First-Fit) методу, цей метод дає змогу відокремити не використовуємий простір на на диску, та підвищити продуктивність використання пам’яті. Зменшую фрагментацію в розумінні відокремлених частинок пам’яті від реального фрагмента. Алгоритмічним фактором котрий слугує виконанню даного методу, є реалізація пошуку, котрий проходячи через список всіх релевантних блоків пам’яті та визначає найменший блок для фрагмента співвідносного значення суми ваги фрагменту. Після знаходження його дані фрагмента записуються в нього. Перевагами виступає ефективне використання пам’яті, та зменшення не використаного простору в хмарному сховищі. З приводу ефективного використання пам’яті, ця перевага дозволяє залишати в сховищі більші блоки для майбутніх фрагментів завдяки використання найменшого можливого блоку, та зберігає кількості не

фрагментованих блоків для наступних фрагментів. Зменшення не використаного простору в хмарному сховищі, ще відбувається завдяки щільному завантаженню фрагментів. Головними недоліками методу найменшого підходящого місця зберігання фрагментів алгоритму динамічної фрагментації є складність реалізації, та час виконання даного методу. Складність реалізації полягає в провадженню додаткових структур даних, таких як збалансовані дерева або купи, для більш чіткішого пошуку місця фрагменту, а це ускладнює управління всією системою обробки даних. Час виконання даного методу має два фактори котрі впливають на продуктивність, пошук кожного фрагмента, та зростання фрагментів з часом. Пошук кожного фрагмента, алгоритму потрібно кожен раз при фрагментації обходити всі вільни комірки для розташування фрагмента, якщо в системі буде велика кількість запитів, це може спричинити затримки. Зростання фрагментів з часом, йде мова про можливе збільшення кількості розпилених даних, котре в свою чергу збільшить об'єм комірок пам'яті, котрі потрібно перевірити, це може призвести до зниження продуктивності системи. В загалом метод найменшого підходящого фрагмента ефективний, але якщо є обмеження по часу запиту, краще його не використовувати.

Метод найгіршого підходящого (Worst-Fit) – це один з можливих методів алгоритму динамічної фрагментації котрого основною функцією є управління пам'яттю або дисковим простором. В послідовності дій цього методу вкладено пошук найбільш доступних блоків пам'яті для фрагментів. Головна ідея являє собою можливість залишення великих блоків на майбутні фрагменти, для потенційного зменшення ймовірно не коректної фрагментації диску. Метод найгіршого підходящого, має свої основні принципи роботи методу, а саме пошук найбільшого блоку, та стратегія мінімізації фрагментації. Пошук найбільшого блоку це основна концепція для майбутніх взаємодій цього методу. Для реалізації пошуку найбільшого блоку потрібно, пройти списком блоків котрі є доступні, і вибрати найбільший релевантний блок, котрий має достатньо місця для зберіганная фрагменту. Після успішного знаходження цього блоку, дані розміщуються в ньому.

Далі йде принцип мінімізації фрагментації простору хмарного сховищного середовища , в його основі полягає залишити для майбутніх використань, блоки пам’яті в котрі потім будуть записуватись фрагменти. Якщо цього не зробити в середовищі сховища, буде можливість створення великої кількості дрібних блоків, що можуть так і залишитись не використаними. Переваги методу найгіршого підходящого, із головних переваг, то це зменшення фрагментації дискового простору та ефективність для великих запитів. Ми вже описали перевагу зменшення фрагментації дискового простору, але головним її аспектом є те, що у випадку коли ми залишаємо великі блоки пам’яті для майбутніх фрагментів, але у нас все одно йде поті маленьких фрагментів за своєю вагою, у нас є можливість розбити великий фрагмент на розмір потрібного нам зараз використаного фрагменту. Далі ми розглянемо перевагу, як ефективність для великих запитів, знаючи що всі мали блоки були використані а великі можливо розділити, варіація розміру дає змогу отримувати фрагменти будь якого величини. Недоліки методу найгіршого підходящого, їх три, а саме: складність реалізації, час виконання, неоптимальність для малих запитів. Складність реалізації відбувається за рахунок інфрастрктурних вимог, так як потрібного пройти циклом по всім можливим місцям розміщення інформації. Недолік часу виконання виникає, тому що обхід всіх вільних блоків в пошуках вільного місця в перегляді комірок пам’яті для фрагменту може бути ресурсо та часо затратними процесами. Не оптимальність під час малих запитів, тоді коли всі малі блоки для фрагментів будуть зайнятті, розбивання великого блоку може зайняти певний час. В сумі метод найгіршого підходящого, може використовуватись в різних інформаційних системах, але все ж головним його аспектом являється усунення можливої фрагментації сховища, як не використання доступних, але малих комірок пам’яті.

## 2.2 Формування абстрактної карти інформації

Абстрактна карта інформації являє собою сукупність певних даних, котрі дають можливість коректної роботи системи розподіленого зберігання, оброки інформації. В нашому проекті основним носієм абстрактної карти виступає жорсткий диск, також це може бути і віртуальне сховище на вашому комп'ютері, навіть звичайна флешка пам'яті.

При формуванні абстрактної карти, було виведено певну кількість змінних котрі є критично важливими для отримання доступу до віддаленого безпечного дистрибутивного хмарного сховища інформації. Це такі змінні як: ідентифікатор користувача; приватні та публічні ключи; пас-фрази; назви, та контрольна сума ваги файлів, котрі знаходять в сховищі; розташування всіх фалів у віртуальному середовищі; розташування фрагментів в самому сховищі хмарної платформи; послідовність конкатенації фрагментів, для коректної дефрагментації. Завдячуючи всім вище перерахованим змінним, чи масивам змінних, ми можемо проводити взаємодію обрахунків, обробку інформації.

Ідентифікатор користувача, також (User ID) – це унікальний номер або ім'я, котре дозволяє ідентифікувати користувача в системі, та відрізнити його від іншого в програмі, чи сервісі. Його формат може в собі містити, як числові значення, ряди з цифер та літер, також може бути електронна адреса або інші формати. Поняття ідентифікація (ідентичність) – це фактор можливості користувача, зробити доказ, що він являється, саме той, зазначеною особою, котрій належить певний спектор можливостей, чи інформації. Процес виявлення ідентичності, відбувається за рахунок співставлення, чи порівняння певного значення, котре має потенційний користувач, з тим значенням котре є зареєстроване на інформаційній платформі. Якщо значення збігають відбувається проходження першої частини бар'єру відповідності. Так як, зазвичай, і в цьому проекті, ім'я користувача не є чимось секретним, хоча можливість доступу іншим користувачам в раках системи до нього закрита, існує таке розуміння, як код доступу до певного середовища мережі, де проходить аутентифікація. Другою змінною доступу являє собою, так званий пароль, параметри його можуть бути досить обширні та різні, їх визначенням

займається платформа місця аутентифікації. В нашому проекті користувач сам вводить ідентифікатор, а пароль генерується системою. Ідентифікатор користувача має такі основні функції, як: аутентифікація; керування доступом; індивідуалізація; трекінг та аналітика. Аутентифікація - це процес котрий дозволяє при використовувані ідентифікатора виконати перевірку відповідності особи користувача для успішного входу до системи. Керування доступом - це процес котрий дозволяє при використані ідентифікатора надати певний обсяг доступності в залежності від прав користувача, до певних ресурсів або інформації, також є можливість ранжування доступу, надання або обмеження що до можливих дій взаємодії з ресурсом, чи іншими користувачами. Індивідуалізація - це процес котрий дозволяє при використані ідентифікатора надати користувачеві вже зроблені ним певні налаштування його робочого простору, але не завжди це є безпечною функцією, так я зазвичай сервер досить часто цікавить зловмисників, і в разі доступу до нього є можливість втрати певних налаштувань, так як в нашому проекті система є дистрибутивна і в зловмисника не має змоги відтворити інформацію користувачів для взаємодії, він буде намагатися отримати певну інформацію про користувача, для подальшого використання в компрометуючих цілях. Тому було прийняте рішення, що до переносу локальних характеристик робочого простору користувача на його особисте персональне сховище. Трекінг та аналітика - це процес котрий дозволяє при використані ідентифікатора знайти цікавий матеріал споживання саме певному користувачеві, але в розробленій платформі хмарного середовища це не впроваджено. В нашій ситуації, це може бути корисно для аналітики спів постанови розміру даних та частоти їх використання, таким чином ти дані котрі не використовуються в продовж тривалого терміну, можуть бути архівованими, та хмарне середовище може бути оптимізовано за для ефективнішого використання пам’яті.

Приватні та публічні ключі (Private key and Public key) – це є під частиною в папці зв’язці ключів на персональному сховищі користувача в нашому проекті. Також основне їх призначення відбувається в момент асиметричного шифрування, в нього є ще інша назва, як криптографія з відкритим ключем. Асиметричний метод

шифрування, як технологія з головним призначенням захисту інформації від несанкціонованого втручання. Також приватні та публічні ключі використовуються, ще в інших сферах, наприклад як цифровий підпис документу, чи за для безпечного обміну інформацією. Цифровий підпис документу, має можливість остаточного засвідчення всіх його читачів, що цім документом володіє відповідно підпису людина чий належить цей підпис. Що до безпечного обміну інформації завдяки цифрового підпису, наразі є достатньо багато ніш де це використовується, таких як: електрона пошта, документообіг, транзакції, системи управління версіями, розповсюдження програмного забезпечення. Електрона пошта використовує метод цифрового підпису, як гарантію того що під час відправлення повідомлення, саме повідомлення не було перехоплене кимось, та видозмінене. Є програмні рішення такого типу підпису, такі як PGP (Pretty Good Privacy) та S/MIME (Secure/Multipurpose Internet Mail Extensions), також вони дозволяють реалізовувати ще багато різних криптографічних методів захисту інформації. Документообіг в цій сфері використовується цифровий підпис за для затвердженню документу, юридичною вагою, та ідентифікувати автора документа. Такі типи документів, як угода, заява, договір може бути використаний цифровий підпис. Головна мета підтвердження автентичності. Транзакції також мають можливість на використання цифрового підпису, сфера фінансів та комерція потребує безпеки. Наприклад при використанні наших електронних гамаців під час покупки товару в онлайн магазині, ми створюємо транзакцію котру підписуємо цифровим підписом, цим самим гарантуємо, що є власниками даного гаманця. Системи управління версіями в цьому випадку ми використовуючи цифровий підпис даємо можливість підтвердженню написаного нами коду. Розповсюдження програмного забезпечення, цей випадок гарантує, що при використанні певних програм, користувачі можуть бути впевнені, що вони користуються саме ліцензійним продуктом певної компанії. Також користувач впевнений в захисті від шкідливо вмістного програмного забезпечення. Цифрові підписи дають достатньо високій рівень безпеки, тому що при їх створені використовується поєднання двох видів ключів, публічного, та приватного. Приватний ключ з точки зору базового

використання – це секретний, відомий лиш власнику ключ, який використовуються задля розшифрування зашифрованого файлу, чи створенні цифрового підпису. Публічний ключ з точки зору базового використання – це відкритий для всіх охочих ключ, головна мета котрого є можливість зашифрувати певну інформацію, або зробити перевірку відповідальності цифрового підпису. Публічний ключ не завжди може бути чимось секретним, та може бути розповсюджений на певну аудиторію, для перевірки власника. Потрібно зазначити, що це все стосовно асиметричної моделі шифрування. Головними перевагами використанням публічних та приватних ключів є безпека, гнучкість та масштабованість. Безпека, поки приватний ключ у власника інформації, інформація не може бути скомпрометована. Гнучкість, використання приватного та публічного ключів може бути корисним для багатьох додатків. Масштабованість може бути ефективно впроваджена завдяки можливості використання публічного ключа.

Пас-фраза це один з типів аутентифікаційних даних, головними критеріями котрих є забезпечення можливості отримання доступу до певних ресурсів системи. Пас фраза схоже значення, та вигляд з паролем, але критерії в порівнянні з паролем в неї трішки інші, вона має довший вигляд, вона краще запам’ятовується користувачами, це може бути уривок з вірша, або послідовність не сумісних слів котрі поєднуються за рахунок конкатенації в одну довгу суцільну фразу. І має куди вищу ступінь безпеки в порівняні із звичайним паролем захисту. У пас-фраз є дві основні переваги це безпека, та легкість використання. Безпека з’являється із-за стійкості до атак брутфорсом (атак послідовного перебору). Легкість використання залежить від можливості правильної, логічної, послідовності комбінації слів для запам’ятовування. Пас фрази мають декілька найчастіших випадків для застосування, це шифрування даних, надання доступу до системи, при використанні цифрових гаманців, та в технологіях котрі займаються забезпеченням криптовалюти. Пас-фрази під час шифрування даних, зазвичай використовуються під час нової генерації приватних, та публічних ключів, як додатковий захист поміж приватного ключу. Часто використання пас-фраз відбувається при в ході в різни облікові системи юзерів, завдяки легкості використання в розумінні

запам’ятовуванні. Криптовалюта часто використовує при захисті приватних ключів пас-фрази, як додатковий захист. Пас-фрази зазвичай генеруються програмним забезпеченням на основі різних запропонованих користувачем словників. Головним фактором використання програмного забезпечення є генерація випадкових чисел стохастичним чином, після чого певне слово зі словника буде додане до попереднього кількість слів можна обрати самому, чим довше, тим краще. Що стосовно створювання пас фаз без використання програмного забезпечення за рахунок власного інтелекту, це також можливо, є декілька варіантів, перший і більш безпечний, спробувати створити комбінацію на основі мнемотехнічних, логічних послідовностей слів з використанням верхніх, та нижніх літералів в фразі в неочікуваних місцях для послаблення можливості компрометації. Другій спосіб використання цитат, чи розповсюджених фраз котрі можуть бути взломані за рахунок атаки словниками. В цьому проекті пас-фрази відіграють роль захисту під час шифрування вже фрагментованих даних. Це відбувається за рахунок використання словника Pretty Good Privacy з певним різновидом слів та комбінування їх, після чого ця пас фраза пов’язана, ще з декількома атрибутами, а файл в ролі об’єкта котрий поділився на певну кількість фрагментів, був розділений зашифрований, та записаний завдяки алгоритму запису JSON формат в абстрактну карту інформації, для подальшої взаємодії з фрагментами.

Кожен файл в абстрактній кратці має власну назву. Наприклад якщо це повноцінний файл то назва його залежить лише від користувача. Якщо йде мова про фрагменти, то під час процесу фрагментації для кожного фрагменту створюється його власна назва, власна назва фрагмента в дистрибутивній системі зберігання даних слугує його ідентифікаторм, вона формується на основі англійського алфавіту та можливості використання цифр додаючи їх у випадковій послідовності разом з літералами. Все ім’я може складатися починаючи з 10 і до 20 літер, головним фактором слугує унікальність і не можливість повторюваності. Назви ключів фрагментів відповідають назвам фрагментам та зберігаються в одній зв’язці з атрибутах абстрактний файлів.

Контрольна сума ваги файлу, також знаходиться в атрибутах кожного абстрактного файлу, на абстрактній карті інформації. Вона використовується в багатьох методах в ході всього циклу даних. Основною її метою слугує перевірку коректної обробки інформації,  а точніше під час фрагментації, в порівнянні суми всіх фрагментів з файлом, під час шифрування, під час дефрагментації.

Розташування всіх фалів у віртуальному середовищі, атрибут котрий несе в собі робочій простір користувача. А точніше під час приєднання користувачем до безпечної, відокремленої віртуальної машини котра в свою чергу знаходиться на хмарній платформі. У користувача є можливість створити власну ієрархію розташування файлів в робочому середовищі, всі зміни розташування зберігаються в файл JSON де ключем є назва файлу, а його значенням є шлях до самого файлу. Таким чином є можливість відтворення робочого столу, та всього простору користувача для комфортної його роботи на хмарній платформі.

Розташування фрагментів в самому сховищі хмарної платформи, відбувається аналогічним чином, як зі збереженням робочого простору. В той час коли фрагменти після їх обробки будуть займати своє положення їхня пряма адреса буде зберігатися в JSON файл, а також це буде атрибутом кожного фрагмента. В цьому випадку ключем виступає ім’я або ідентифікатор фрагменту, а його значення це місце розташування на сервері дистрибутивній, хмарній платформі.

Послідовність конкатенації фрагментів, для коректної дефрагментації. Крім тих факторів, що потрібно коректно фрагментувати, шифрувати, дешифрувати, то також важливим фактором є коректна дефрагментація. Головним критерієм коректної дефрагментації в нашому алгоритмі є правильна відносно послідовності фрагментації, послідовність дефрагментації. І ця послідовність зберігається також у JSON, але тепер це вже масив значень. Кожне його значення являє назву фрагмента, а його індекс, те місце в котрому він знаходиться в масиві, послідовність починаючи з нуля, закінчуючи кількістю фрагментів, воно і є відповідно котрому їх потрібно з’єднувати почергово. Це дасть нам змогу повноцінно відтворити файл, без можливості втрати інформації.

Всі вище перелічені змінні, котрі зберігаються в локальному сховищі користувача, є критично важливі для повноцінної взаємодії з хмарною платформою. Всі ці змінні являють абстрактним розумінням атрибутів котрих вони набувають підчас роботи користувача в системі. Кожне значення котре було описане, воно зберігається в JSON файл, а сам файл в особисте сховище користувача. Таким чином при необхідності отриманні доступу до певних даних для взаємодії, відображенням інформації, користувач сам є одним і не повторним ключем до своєї інформації, навіть серверний провайдер не має зовсім ніякого доступу до інформації користувача, як даних. Також слід зазначити, що після кожної дефрагментації, під час фрагментації кожен фрагмент має нове місцезнаходження на сервері, це відтворює ще один бар'єр захисту інформації.

### 2.3 Застосування криптографії, як шифрування, та дешифрування фрагментів

В даному проекті використовується такий вид шифрування, як асиметричне. Асиметричне шифрування, як вже зазначалося в розділі 2.2 ще має назву криптографія з відкритим ключем. Його метод використовує поєднання двох ключів публічного та приватного, для шифрування та дешифрування інформації. Завдяки такому виду шифрування у користувача є можливість цілковито безпечно передавати свої дані.

Пара ключів, кожен користувач в ході фрагментації, чи інших умовах отримання доступу розповсюдження, генерує приватні та публічні ключи. Публічні ключи використовуються під час шифрування, та можуть бути розповсюджений серед учасників середовища. Приватний ключ зберігається в локальному, фізичному носії самого власника інформації, і є секретним. Також в нашу випадку під час асиметричного шифрування використовується пас-фраза, її головна мета захистити приватний ключ навіть тоді, коли він потрапив до рук зловмисників.

Так як ми реалізовуємо асиметричний метод шифрування, з використанням PGP (Pretty Good Privacy), а він в свою чергу є поєднанням елементів симетричного,

та асиметричного шифрування. Якщо використовувати метод PGP для передачі певної важливої інформації. Це буде відбуватися в три етапи, генерація сесійного ключа, шифрування повідомлення, шифрування сесійного ключа. Під час генерації сесійного ключа, відбувається тимчасова генерація сесійного ключа, для симетричного методу шифрування, він в свою чергу і використовується при прямому шифруванні фрагмента передачі, як файлу. Генерація сесійного ключа є стохастичним процесом, і унікальним для кожного фрагменту. Далі йде етап шифрування повідомлення, текст чи фрагмент починає шифруватися завдяки симетричному сесійному ключу. Алгоритм шифрування котрий використовується в той час – це AES (Advanced Encryption Standard). Далі йде третій етап шифрування, це шифрування сесійного ключу. До його входить такий критерій як публічний ключ отримувача котрий буде використаний за для шифрування сесійного ключу. Щоб отримати доступ до даних назад, потрібно мати приватний ключ одержувача, лише він здатний на розшифрування сесійного ключу. Також це можливо охарактеризувати, як повідомлення котре потрібно відправити шифрується на основі публічного ключа отримувача. Лише при використанні відповідного приватного ключу є можливість отримати доступ до фрагмента, що впроваджує безпеку інформації. У випадку використання цифрового підпису, приватний підпис засвідчує особу відправника, а публічний перевіряє документ на автентичність даних. Переваги використання асиметричного шифрування є безпека та не можливість підміни власника. Безпека виникає за рахунок будь якого каналу передачі інформації, та не потребує додаткових заходів підвищення безпеки. Неможливість підміни власника є значною перевагою, розумінні не можливого процесу компрометування. Є і недоліки пов’язані з обчислювальними витратами ресурсів якщо порівнювати з симетричним методом шифрування. Менеджмент ключів, полягає у ретельному управлінні ключами, що не завжди так легко.

Advanced Encryption Standard – це є досить популярним та часто вживаним стандартним алгоритмом симетричного шифрування. Він був створений для покращення попереднього стандарту Data Encryption Standard, ще в 2001 році. AES має власні особливості, такі як розмір ключів шифрування, розмір шифрованого

блоку, різні режими роботи. Розмір ключів даного симетричного методу шифрування може бути зазвичай 128, або 256. Чи більша буде довжина тим ретельніший буде захист. Розмір шифрованого блоку, не може перевищити 128 біт, без різниці по відношенню до ключа. В AES є декілька режимів роботи з певними особливостям безпеки, такі як CBC (Cipher Block Chaining), CFB (Cipher Feedback), ECB (Electronic Codebook), та OFB (Output Feedback). CBC (Cipher Block Chaining) завдячуючи цьому режиму, всі блоки тексту попередньо перед шифрування підлягають функції XOR з використанням зашифрованого блоку. Починається з першого блоку він XOR-ується з проведенням вектором ініціалізації 4. Це дає змогу однаковим вхідним текстам давати кожному різні блоки шифр-тексту, це може відбуватися навіть коли текст знаходиться в одному повідомлені. CFB (Cipher Feedback) завдячуючи цьому режиму шифр працює самосинхронізуючись відносно потоку. AES використовує функцію шифрування попереднього шифртексту, а потім використовується функція XOR відносоно цього тексту. Це дозволяє видозмінювати розмір котрий буде зашифрований і не є кратним розміру блоку. ECB (Electronic Codebook), відносно найпростіший режим роботи, в його розумінні полягає шифрування по блочно всього тексту, не залежно друг від друга. Не підходить для шифрування значних обсягів даних, в тому випадку, якщо дані, як інформація мають досить багато повторюваних блоків. OFB (Output Feedback) – це режим забезпечує потокове шифруванням з генерацією блоків ключового потоку, котрі після підлягають обробці функції XOR з використанням вхідного тексту, для конвертування в шифртекст. Так як вихід шифру може повернутися до входу, будь-яка помилка, не може вилинути на основу зашифрованого тексту. При застосуванні AES підвищується захист конфіденційності даних, безпечний обмін даними, також є перевагою те що даний тип шифрування стандартизований та застосовується в державних секторах. Це досить безпечний алгоритм для використання як захисту інформацію методом шифрування. Досить стійкий до більшості криптографічних атак. Враховуючи всі аспекти реалізації, його швидкість обробки все одно залишається ефективною. Навіть пристрої котрих ресурси є обмеженими цілковито мають змогу на його використання.

RSA (Rivest–Shamir–Adleman) це також один з най популярніших алгоритмів асиметричного шифрування, досить широко використовуваний. Його алгоритм дій схожий з більшістю асиметричних алгоритмів, також має як і публічний, так і приватний ключ шифрування та розшифрування даних. Його часто використовують в цифрових підписах документу, та і в цілком захисту інформації. Алгоритм генерації публічного та приватного ключа. Першим кроком слугує обирання двох великих, простих чисел, ми їх позначимо, як *pp* та $qq$, слід зазначити що вибирання потрібно проводити випадковим чином. Наступний крок є обчислення модуля *nn. NN* являє собою добутком двох цих простих чисел *pp* та *qq,* потім вони будуть використовуватись, як модуль для обох ключів, також їхня бітова довжина відповідає довжині майбутніх ключів. Наступним кроком є обрахування функції Ейлера:

$$\phi(n) = (p - 1) \times (q - 1) \tag{2.5}$$

В даній функції нам потрібно знайти змінну *n.* Наступний крок полягає в виборі відкритої експоненти. Число з змінною *e,* потрібно вибирати випадковим чином, але головне щоб воно було протягом проміжку $2 < e < \phi(n)$. Саме значення $e$ також є відоме, як відкрита експонента має бути взаємно простим відносно значення $\phi(n)$. Якщо ми будемо використовувати занадто малі значення відкритої експоненти, це може значно послабити силу захисту алгоритму RSA. Наступний крок дозволяє нам знайти приватну експоненту, вона повинна бути обернена за модулем числа. Вся суть відкритої експоненти $e$ полягала в основу створення публічного ключа. Приватна експонента $d$, також відома як секретна, полягала в основу створення приватно ключа. Знаходження приватної експоненти $d$ полягає використанні розширеного рішення Евкліда. А саме приватна експонента d з використанням розширеного рішення Евкліда заходиться, як обернена за модулем $\phi(n)$ до публічної експоненти $e$.

$$d \times e = 1 \times (mod(\phi(n))) \tag{2.6}$$

Наступний крок є кроком розділу ключа. Після того коли всі необхідні обчислення було проведено, та певна кількість чисел знайдена, ж можливість формування публічного та приватного ключа. Приватний ключ має такі складові, як *d* та *n*, а публічний з пари *e* та *n.* Число котре ми знайшли як приватну експоненту повинно бути засекреченим. Основна функція $d$ це можливість дешифрації повідомлення. Також слід зберігати числа $p$ та $q$ та функція $\phi(n)$, щоб зловмисник не міг на основі вхідних даних згенерувати такий самий приватний, та публічний ключ. Момент шифрування відбувається завдяки впровадженню такої рівності як:

$$c = m^e \times (mod\ n) \tag{2.7}$$

Змінні цієї формули є такі як:

$m$ – текст;

$c$ – зашифрований текст.

Також існує й інверсійна рівність запропонованій, для можливості дешифрування, а саме:

$$c = c^d \times (mod\ n), \tag{2.8}$$

В нашому проекті використовується алгоритм шифрування RSA фрагментів під час фрагментації. Існує окремий клас котрий створює пару ключів (публічний, та приватний) також використовується додатковий захист у вигляді пас-фрази для приватного ключа. Всі фрагаменти перед своїм розсіюванням шифрується довжина ключа є 2048 біт, на разі це повністю задовольняє потреби захисту від можливих атак, вражень вражень. Ключи їх доступу являються атрибутами абстрактного об’єкта файлу з ієрархічною структурою на абстрактній карті інформації.

### 2.4 Формування алгоритму дефрагментації

Дефрагментація – це метод компоновки всіх фрагментів абстрактногофайлу, для можливого відтворення. Також для відтворення файлу потрібно мати всі атрибути, а саме: ключі, пас-фрази, місце розташування фрагментів в

дистрибутивному сховищі. Це дасть змогу провести операцію побітової конкатенації успішно. Сама дефрагментація відбувається на віртуальній машині віддаленого сервера. Процес дефрагментації досить тривалий с точки зору ефективності. Також під час дефрагментації головним аспектом котрої є наявність всіх мета конфігураційних файлів, потрібно звернути найбільш пильну увагу на присутність чужих очей в системі, так як в цей момент на віртуальній машині знаходиться вся потрібна інформація що до отримання розташування файлів, як розсіяних фрагментів, їх ключів, контрольна сума ваги, пас фрази, правильна послідовність збору фала. Також слідує зазначити, що всі файли котрі підлягають дефрагментації повинні бути в сто відсотковій цілісності, так як коректне їхнє відображення залежить саме від цього. Процес дефрагментації починається з пошуку всіх фрагментів в хмарному середовищі, потім наступним етапом є завантаження їх на віртуальну машину в шифрованому вигляді. Після завантаження їх починається процес дешифрування. Для цього потрібно такі аспекти як ключ приватний ключ до фрагмента, пас фраза до приватного ключа, алгоритм почергово проходить по всім фрагментам котрі були зібрані в віртуальному середовищі. Далі, коли ми вже маємо всі дешифровані фрагменти, наступним крокоп виступає перевірка цілісності кожного з фрагмента. Вона відбувається за рахунок повної суми файлів котра була залишена на минулому сеансі користувача. Якщо ми бачимо співставивши контрольну суму та суму біт всіх фрагментів котрі у нас зараз є, і вона не є однаковою потрібно визначити, що за фрагмент був порушений цілісності. Для цієї реалізації ми проводим вже співпоставлення ваги кожного фрагменту з вагою абстрактного фрагменту із абстрактної карти інформації. Знайшовши пошкоджений фрагмент потрібно зробити запит на ще один такий самий після чого провести дешифрування, та перевірити на цілісність. Після того як ми впевнені, що всі фрагменти цілі, можемо починати етап дефрагментації, а саме маючи правильну послідовність з’єднання фрагментів, як цілісності файлу, ми повинні відтворити конкатинацію їх з іншими фрагментами послідовно завдяки їхнім ідентифікаторам. Наступний крок дозволяє

нам перевірить релевантність, та можливість відображення всіх файлів, якщо це можливо, ми видаляємо їх копії фрагментів з основного сховища.

### 2.5 Формування алгоритму перевірки цілісності файлу

Перевірка на цілісність фалу є досить важливим аспектом, повноцінного функціонування системи, компонування фрагментів шляхом дефрагментації, та можливості відображення файлу як одиниці складеної з фрагментів.

В цьому проекті за для перевірки цілісності фрагмента реалізована функція, котра дозволяє дізнатися наявність спів відношення бітів, як кількості. Вона реалізована завдяки наявності хешу, після шифрування. Також завдяки цієї функції, є можливість зробити перевірку, не тільки на пошкодження фрагменту під час зберігання, чи транзакції його, але й на підміну фрагменту, як інформації. Цей метод являє собою най розповсюдженішим, та най ефективнішим. Для початку нам потрібно перенести наші фрагменти у віртуальну, безпечну середу. Потім нам слід зробити дешифрування, використовуючи приватний ключ, та пас-фразу. Після успішного дешифрування нам потрібно зробити обрахунки початкового хешу фрагмента. В ході цього процеса буде використовуватись хеш-функція SHA-256,її головною задачею, буде полягати коректна обчислення хешу даних. Також це можливо назвати відбитком фрагмента.

Хеш-функція така як SHA-256 (Secure Hash Algorithm 256-bit) являє собою криптографічну функцію. Її головна мета  це генерація 256-бітного, аналогічно що 32-байтного хеш відбитка інформації котра йде на вхід. Послідовність роботи хеш-функції SHA-256, чи її алгоритм це отримання вхідних даних, конкатинування довжини, використання ініціалізаційний вектору, і обрахункова обробка даних. Отримання вхідних даних це процес отримання тексту без обмеженої границі довжини, та конвертування його в фіксовану 256-бітну довжину. В ході цього процесу алгоритм SHA-256 першою дією робить розмір файлу кратним 512 бітам. Цей процес називається падінгом, він починає додавати біти з одиниці, а потім йде нуль, таким чином він додає біти поки розмір всього файлу чи фрагменту не стане,

кратним 512 бітам, також це є число котре менше наступного на 64 біти. Потім цих 64 відсутніх біти додаються та воно являє собою вихідне повідомлення. Далі йде процес ініціалізації восьми буферів хешу. SHA-256 використовує 32-бітні регістри, за для буферів хешу. Кожен регістр має власну ініціалізовану константу. Наступним кроком наступає обчислювальний процес, обробка даних виконується по блока, кожен блок займає в пам'яті 512 бітів. Починаючи з першого і кожний наступний блок підлягається розширенню до 64 слів, кожне слово займає 32 біта. Весь цей процес дозволяє підготовити дані до обробки. Також вцьому алгоритмі присутній основний цикл SHA-256, його мета виконувати ітерації, точніше обходити кожний блок по 64 рази. В цих ітераціях є три головних аспекта впливу на даних, а саме: застосування різних функцій складності, використання функції вибору, та більшості, і використання методу додавання констант раунду. Застосування різних функцій складності являє собою різноманітні операції над даними, такі як зсув бітів, чи інверсія, це дасть змогу розподілити всі вхідні дані по хешу. Застосування функції вибору чи функції більшості, це дві функції котрих головна мета є змішення даних, при змішуванні йде вибір поточних буферних значень на основі інших буферів. Додавання константи раунду, мається на увазі константа ітерації , вона додається з кожним проходом по циклу, це дає змогу відокремити кожен раунд, та запобігти реверс інжинірингу. Наступним етапом після. Після проходження основного циклу операцій, настає процес оновлення хешу. Кожен блок котрий був пройдений оновлює вісім своїх буферів хешу. Цей процес буде відбуватися протягом всіх циклів до самого закінчення операції. Наступний, та фінальний крок, це виведення результату, результат являє собою поєднання всіх восьми 32-бітних регістрів. Весь цей процес впроваджений лише для категоричної зміни хешу в можливості хоч найменшої підміни тексту. Алгоритм SHA-256 є безпечним, ефективним, та універсальним. Безпечний тому що стійкий до пошуку та реалізацій колізій, в нашому випадку колізія являє собою однакове значення хешу, але різний вхідний матеріал. Цей алгоритм ефективний, тому що генерація хеш-кода є досить швидкою. Та цей алгоритм можна назвати

універсальним, тому що він досить безпечний, та цей критерій потребує впровадження в багатьох сервісах, тим паче в хмарних платформах.

# 3 РЕАЛІЗАЦІЯ ПЕРШОРЯДНИХ ФУНКЦІЙ СИСТЕМИ

## 3.1 Реалізація перевірки наявності абстрактної карти

Під час кожного підключення користувача до системи, користувач повинен пройти перевірку наявності абстрактної карти. Під час цієї перевірки, головним критерієм виступає наявність ідентифікатора користувача та приватного ключа доступу з пас фразою. Також в програмній реалізації передбачено клас “flashDrive” котрий має поле типу boolean з найменуванням “authorization”. Тип boolean являє собою імітований перемикач. Якщо користувач вже був раніше зареєстрований в системі, то ця змінна набуде значення true, що є правдою. Також якщо такий користувач не був раніше зареєстрований, змінне значення залишиться false, що є противагою true, неправдою. На (Рисунку 3.1) ми зможемо переглянути вітальне вікно користувача.

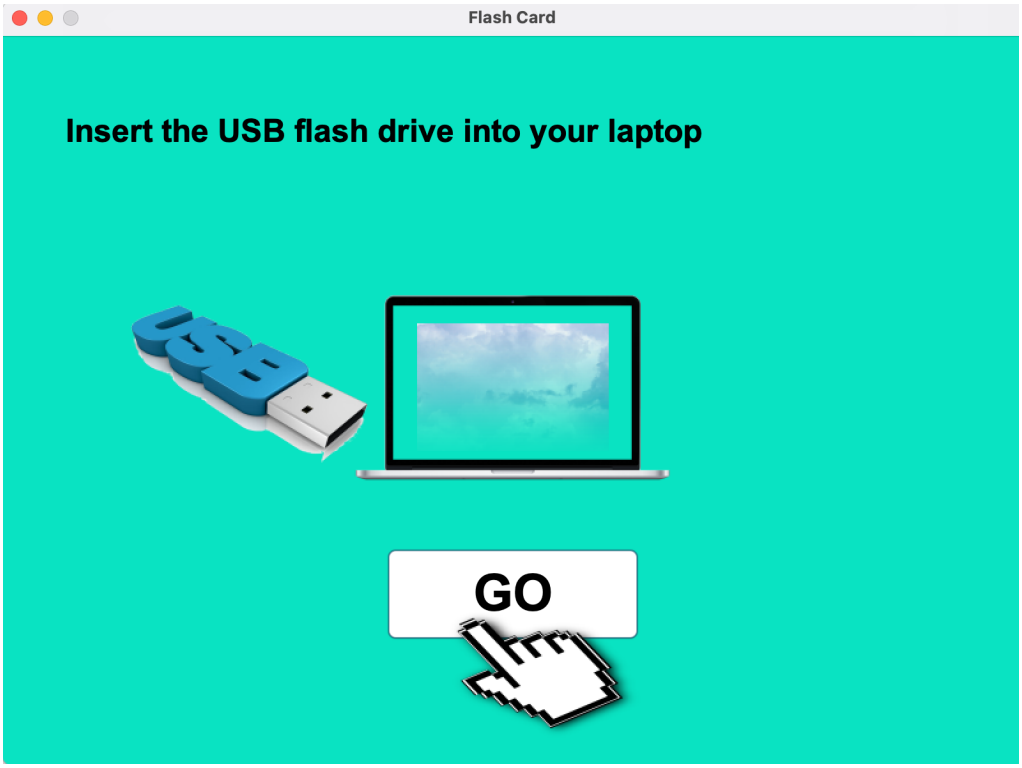



Рисунок 3.1 Вітальне вікно користувача

При натисканні на кнопку “GO”, буде проходити головна перевірка на аутентифікацію цього користувача в системі. У випадку відсутності користувача в

системі, базуючись на наявності його ідентифікатора, можливому приватному ключу, та перемикачу “ authorization ”, буде запропонована можливість реєстрації. На (Рисунку 3.2) зображена можливість реєстрації.

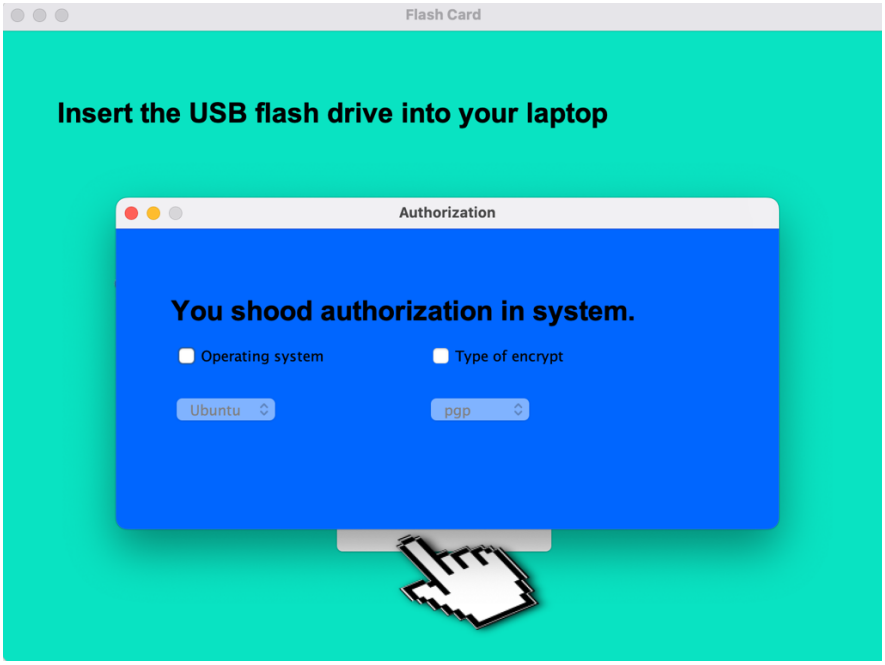



Рисунок 3.2 Можливість реєстрації

Також на (Рисунку 3.2), користувач бачить два прапорця (Checkbox), це елементи графічного інтерфейсу користувача GUI - graphical user interface. Завдяки цім прапорцям, користувач може обрати операційну систему майбутньої віртуальної машини, та тип шифрування котрий буде використовуватись в ході обробки інформації. Значення котрі вибрав користувач, записуються в JSON файл як його ідентифікаційна інформація. Приватний та публічний ключ, генерується в момент вибору будь якого з цих об’єктів. На (Рисунку 3.3) зображена ієрархія папок користувача, при отриманні ключів та встановленні аккаунта.

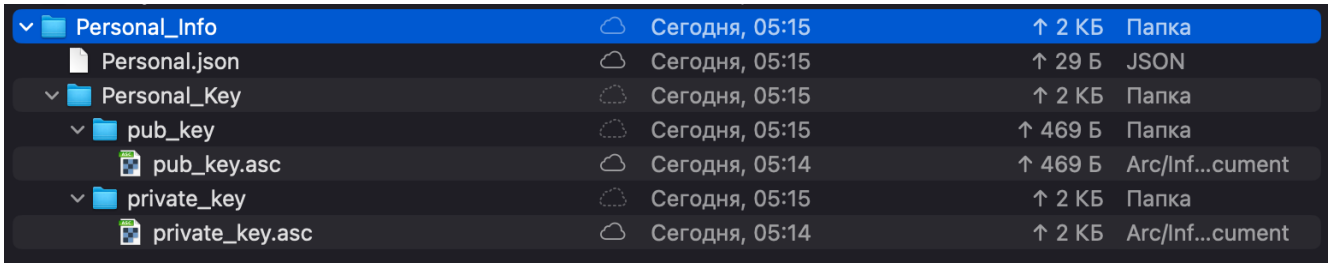



Рисунок 3.3 Зображення ієрархії папок та файлів користувача, при отриманні ключів, та встановленні акаунта

Для створення приватного, та публічного ключа, для успішної авторизації користувача, був розроблений в класі “KeyGenerator” метод “generateKeyPairForEntry”. В основі методу “generateKeyPairForEntry” полягає реалізація алгоритму генерації приватного та публічного ключа, потім відправлення його за певним шляхом створеним при ініціалізації класу “FlashDrive”, “Flash Card/Personal_Info/Personal_Key/pub_key/ pub_key.asc” в файл з розширенням ”.asc”, таким же чином був відправлений і приватний ключ, завдяки “FlashDrive”, “Flash Card/Personal_Info/Personal_Key/private_key/private_key.asc”. Це асиметрична пара ключів PGP. В основу лягла вже розроблена бібліотека “Bouncy Castle”. Вхідними параметрами методу “generateKeyPairForEntry” є outputPathForPublic, outputPathForPrivate, passphrase. OutputPathForPublic – ця змінна містила в собі посилання на файл, в котрий ході виконання методу був записаний публічний ключ. OutputPathForPrivate – ця змінна містила в собі посилання на файл, в котрий ході виконання методу був записаний приватний ключ. Passphrase для отримання додаткового захисту приватного ключа, на основі ідентифікатора користувача, використовується пас-фраза. Сам алгоритм дії методу “generateKeyPairForEntry” є таким: перевірка вхідних даних, ініціалізація генератора ключів, генерація пари ключів, створення об’єкта pgp ключа, створення та налаштування pgp секретного ключа. Перевірка вхідних даних являє собою перевірку пас-фрази на можливу відсутність інформації, чи значення null. У випадку, коли дані є не релевантними використовується виняток “IllegalArgumentException”. Ініціалізація генератора ключів, ми використовуємо в нашому методі алгоритм шифрування RSA з криптографічною бібліотекою “Bouncy Castle”, тому робимо екземпляр класа, де реалізація слугує ініціалізації об’єкта “KeyPairGenerator”, друга частина цього об’єкта повертається з статичного класа самого “KeyPairGenerator”, де приймає два параметри, алгоритм RSA, та криптографічний провайдер "BC" (Bouncy Castle). Першочерговий розмір ключа вказується на 2048 біт, достатньо надійно. Наступним етапом виступає генерація асиметричної пари ключів. Далі ми створюємо об’єкт pgp key, при цьому використовуємо об’єкт обгортання “PGPKeyPair”, та встановлюємо в ньому

поточну дату. Створення та налаштування pgp секретного ключа відбувається за рахунок “PGPSecretKey” та асоційованого об’єкту “PGPKeyPair”. Для сертифікації ключа використовується SHA-1 дайджест. Приватний ключ шифрується використовуючи алгоритм CAST5, та пас-фразу. Приватний та публічний ключ після всіх частин генерації зберігається кожен у свій файл в форматі ASCII Armor.

### 3.2 Реалізація алгоритму фрагментації даних

Для реалізації алгоритму фрагментації було розроблено клас “DataOfFile” цей клас представленням об’єкту даних, як файл цілісному виді, також його можна конвертувати як структур даних для зберігання посилань всіх фрагментів на котрі буде ділитися файл. Клас “DataOfFile” має такі поля як: об’єкт файлу для представлення цілісного файлу, назву файлу, шлях (місце розташування) файлу, розширення файлу, шлях до місця зберігання фрагментів після їх розсіювання, загальний розмір файлу в бітах для контрольної суми ваги, вказується також мінімальний розмір фрагментів з котрих буде складатися файл, завершена кількість фрагментів, шлях котрий показує на місце розташування фрагментів при зборці, словник котрий зберігає шляхи до всіх фрагментів. В конструкторі класу визначається ініціалізація об’єктів, об’єкта файлу, та місце зберігання, та знаходження інформації, як даних, Також визначає всі необхідні атрибути для подальшої взаємодії. Крім полів є й методи, а саме: гетери, та сетери для встановлення або читання атрибутів, setMapOfPathsOfFile додає або оновлює шлях до частин файлу в словнику, метод “ getFileExtension ” котрий дозволяє визначити розширення фалу, calculateSplitCount метод котрий використовується для розбиття на основі загального розміру файлу та мінімального розміру частини. Цей клас дозволяє шаблонно описувати файли, фрагментів файлів, для подальшої взаємодії з ними.

Також для даного класа є реалізація так званого хендлера (handler), він дозволяє оброблювати данні. Саме конструювання програмного забезпечення реалізовано на мові програмування Java, з використанням шаблону розробки MVC (Model-View-Controller), в даному випадку handler виступає в ролі контролера. Сам

клас має назву “HandleOfData”, там реалізовано два основних методи “splitFile”,та “mergeFiles”, вони є вдвох екземплярах кожен, перевизначені. В першому випадку вони працюють на основі даних, як об’єкта, у другому випадку змінні метода можна вказати в ході роботи. В цьому розділі ми розглянемо лише метод “ splitFile”. Вхідними параметрами метода “ splitFile” є об’єкт “ DataOfFile ”, в цьому об’єкті вже є вся необхідна інформація для роботи з файлом, а точніше фрагментації файлу. Сам алгоритм можливо поділити на шість кроків: визначення розміру частини; ініціалізація файлових потоків; читання та запис у частин; запис самих частин; контроль залишкових даних; реєстрація шляхів до частин; Визначення розміру частини, має декілька критеріїв загальний розмір фалу, мінімальна кількість частин, псевдо випадкове число, сама формула обрахування оптимальної кількості частин зазначена в розділі 2.1. Ініціалізація файлових потоків, це зроблено для можливості читання через FileInputStream, а точніше з використанням обгортки BufferedInputStream, для оптимізації читання. Для кожної частини створюється вихідний файл, куди буде проведений запис. Ці файли мають шаблон для абстрактної карти інформації "partX.txt", де X — номер частини. Також використовується буфера пам’яті, які відповідають розмірам частин. Для більш ефективної передачі даних між потоками. Запис частини, являє собою зчитування кожної частини з основного файлу та запису у відповідно йому частину завдяки BufferedOutputStream, це забезпечує буферізацію при записі фрагменту. Після завершення запису однієї частини, файл закривається, і процес повторюється для наступних частин. Контроль залишкових бітів виявляється, якщо кількість байт яких було прочитано, менше розміру буфера. Реєстрація до шляхів фрагментів являє собою процедурою запису шляху кожної нової створеної частини в словник “mapOfPathsOfFile”. Також в цьому методі реалізовано виключення, за для можливого зрозуміння, якщо виникнуть проблеми під час читання або запису файлу. На (Рисунку 3.4) ми можемо побачити результат процесу фрагментації.


Рисунок 3.4 Результат процесу фрагментації

На (Рисунку 3.4) ми може побачити розбиття файлу вагою 478.3 мегабайтів на 17 фрагментів, кожен з яких має вагу 28.1 мегабайт. Якщо порівнювати інформацію з (Рисунок 3.5) та з (Рисунок 3.6) ми можемо впевнитися, що залишкових бітів не залишилося, вага папки де зберігаються фрагменти та вага базового файлу є однакові.

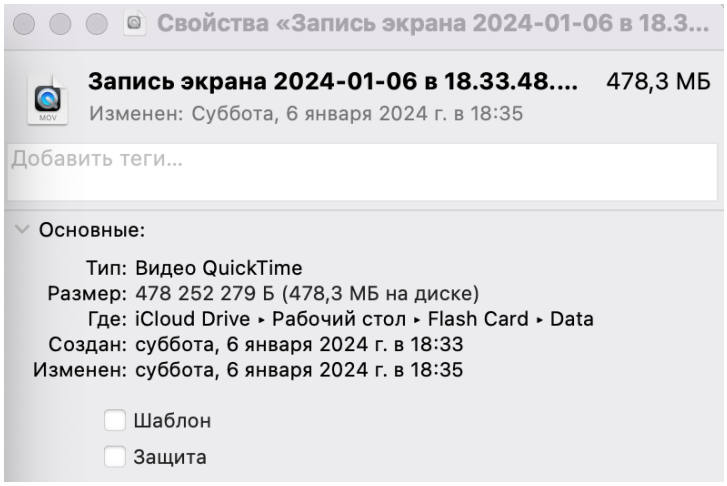



Рисунок 3.5 Параметри базового файлу

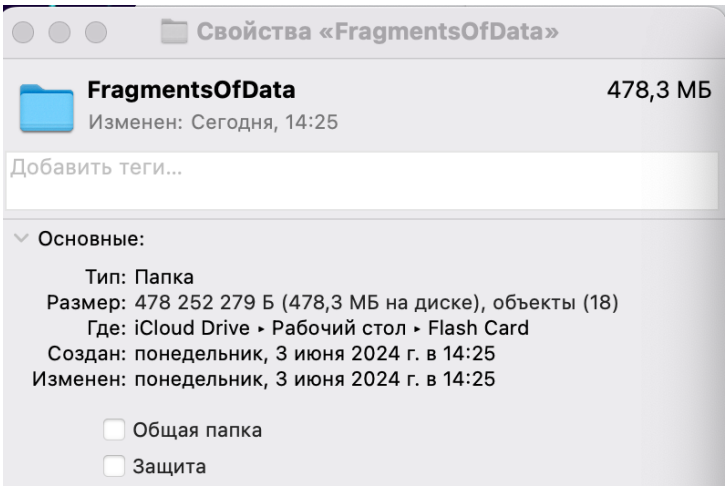



Рисунок 3.6 Параметри суми всіх фрагментів

### 3.3 Реалізація алгоритму дефрагментації даних

Для реалізації алгоритму дефрагментації, було розроблено клас “DataOfFile”. Про цей клас було описано в розділі 3.2. Ми вже розглянули метод фрагментації, зараз будемо опрацьовувати алгоритм дефрагментації (поєднування даних). В класі “DataOfFile” є такий метод, як “mergeFiles”, його головною метою є компонування всіх фрагментів визначеного файлу в один, для можливого відтворення. Параметри цього методу приймають в себе об’єкт, та шлях розміщення файлу , що містить всі необхідні метадані, для обробки інформації. В нього є власна послідовність, як алгоритм дії, а саме: підготовка  вихідного файлу; ініціалізація потоків для запису; зчитування та запис частин у вихідний файл; контроль залишкових байтів. Підготовка вихідного файлу являє визначення повного шляху до вихідного файлу, який включає назву та його розширення, як тип. Ініціалізація потоків для запису має два аспекти, перший аспект є відкриттям FileOutputStream, для вихідного файлу. Другий аспект використовується BufferedOutputStream, для буферизованого запису в файл.Наступним кроком виступає зчитування та запис частин у вхідний файл, для його реалізації використовується послідовність розділення котра була записана в словник, разом з місце находження файлів. По черзі відкриваються файли кожної частини. Для кожного фрагменту файлу ініціюється FileInputStream, обгорнутий у BufferedInputStream. Маючи розмір кожної частини, конкретно під неї виділяється буфер певного розміру, дані зчитаються з кожного фрагмента, та

шляхом запису та конкатинації записуються в новий файл. Послідовність цих дій буде, до того моменту, поки дані всі фрагменти не будуть поєднанні в один файл. Також проводить контроль залишкових бітів, якщо файл не дорівнює зазначеному розмірі його частин. Після того як всі ітерації були успішно закінчені, всі потоки запису в даний файл закриваються, та процеси завершуються. Також в даном методі присутнє виключення IOException, для можливості перехоплення помилок при читанні, та записі. Як можемо бачити на (Рисунку 3.7) дефрагментація пройшла успішно.

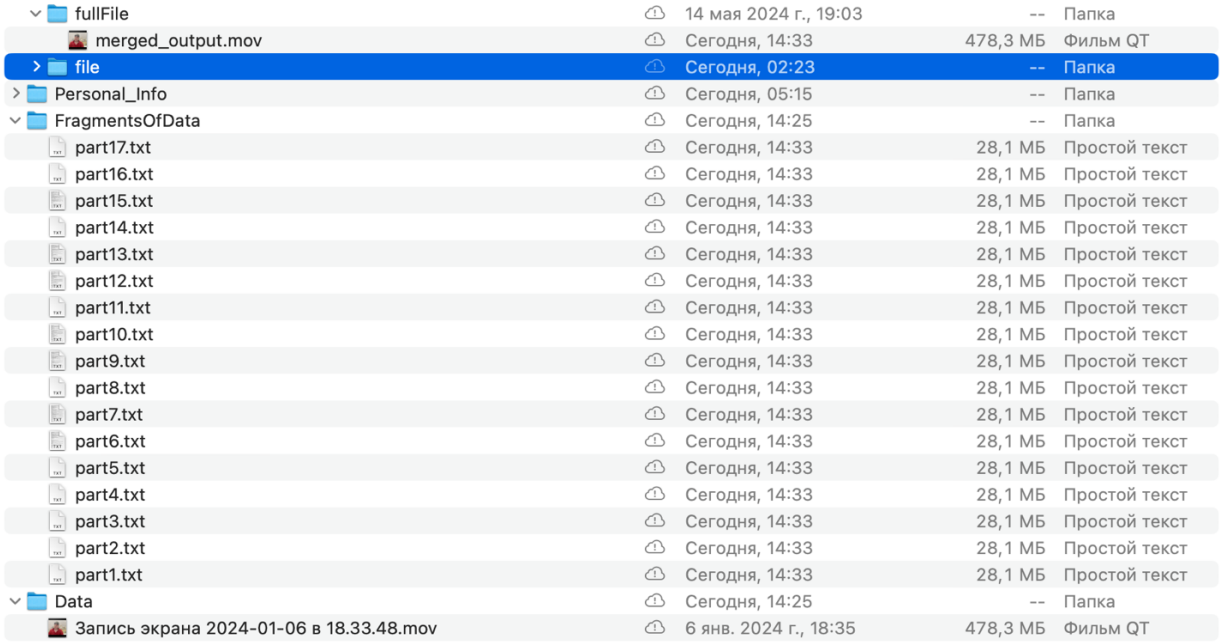



Рисунок 3.7 Дефрагментація пройшла успішно

Вся інформація котра була складена з фрагментів успішно з компонувалась в один файл. На (Рисунку 3.8) ми можемо впевнитись в цілісності файлу, маючу вже з базового файлу точну кількість байт,  а саме 478 252 279.

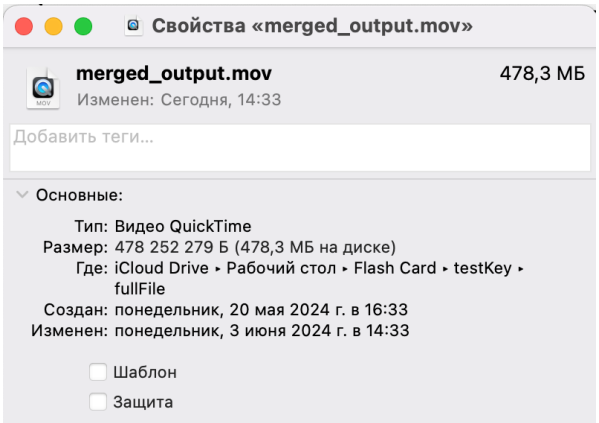



Рисунок 3.8 Параметри дефрагментованого файлу

### 3.4 Реалізація алгоритму шифрування даних

Для можливості шифрування даних було створено клас “EncryptionData”в котром було реалізована певна кількість методів, в даному розділі ми розглянемо лише два з них, це метод “ encryptFile”, та метод ” processEncryptFiles”. Це два різні методи, але вони виконують одну функцію шифрування даних.

Метод encryptFile використовує асиметричний метод шифрування RSA з використанням бібліотеки Bouncy Castle, також метод PGP котрий був описаний в розділі 2.3. Метод PGP дозволяє використанню публічного, приватного ключу, та додатковий захист для приватного ключу у виді пас-фрази. Пас фраза формується з словника, котрий знаходиться на флешці користувача. На (Рисунку 3.9), ми можемо побачити як виглядає словник з котрого формується пас-фраза. Вміст словника складає 3931 слово. Котрі будуть генерувати пас-фразу різної довжини, але це буде відображено далі в цьому розділі.

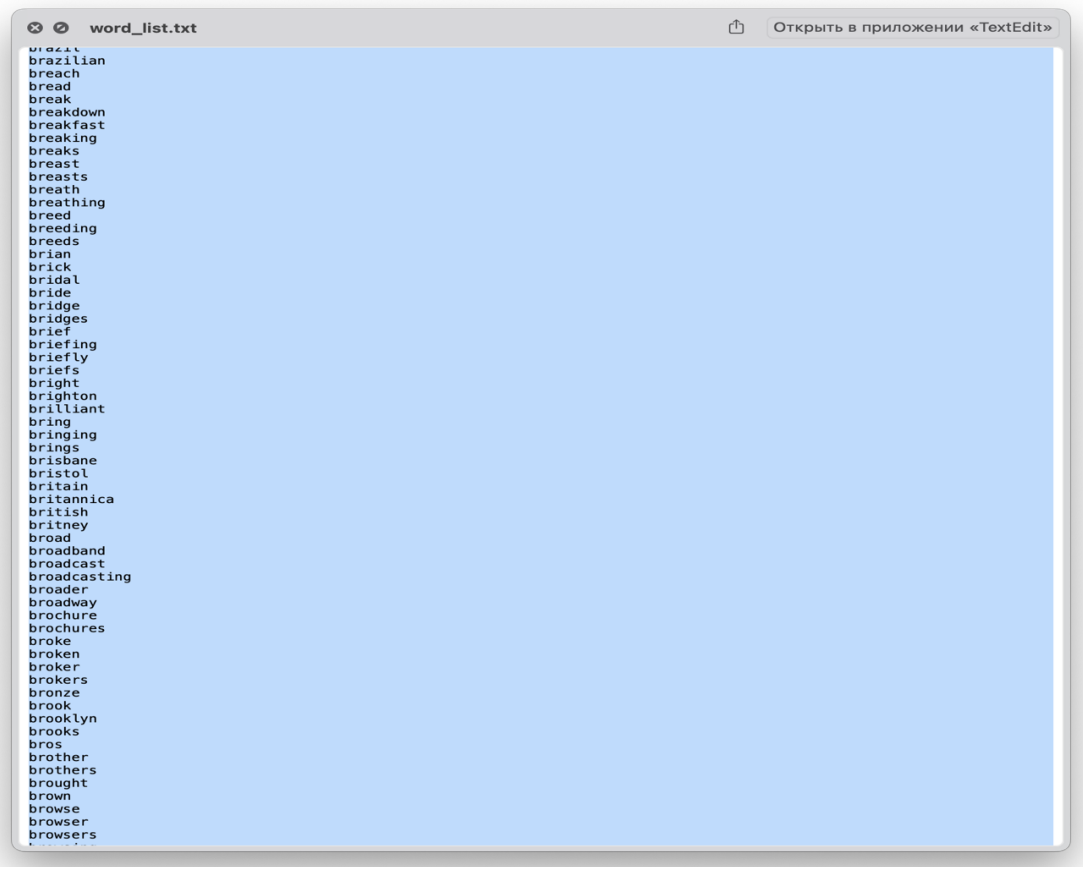



Рисунок 3.9 Відображення словника для формування пас-фраз

Шифрування в даному випадку забезпечує безпечне передавання, зберігання фрагментів, як інформації. Детальніше про шифрування в цьому проекті написано в розділі 2.3. Метод encryptFile має такі вхідні параметри (або змінні методу), як фрагмент котрий потрібно зашифрувати, майбутній публічний ключ до файлу, про алгоритм створення котрого згадувалось в розділі 3.1, та місце знаходження вже зашифрованих фрагментів. Цей алгоритм шифрування має 5 кроків, а саме: ініціалізація ключа; підготовка даних до шифрування; ініціалізація генератора зашифрованих даних; шифрування та запис даних; логування процесу шифрування. Ініціалізація ключа відбувається таким чином, першочергово відкривається файл публічного ключа, використовуючи декодер “PGPUtil.getDecoderStream”, ми в змозі прочитати вміст публічного ключу, його головним аспектом є перетворення потоком даних в потрібний йому формат. Інформація з потоку, котра була прочитана декодером, потім обробляється для створення PGPPublicKeyRing (колекції ключів), з котрого в свою чергу дістається PGPPublicKey. Далі йде процес підготовки фрагментів до шифрування, в нього входить використання такого класу як ByteArrayOutputStream, котрий належить пакету java.io. Мета використання ByteArrayOutputStream полягає в зручному запису в пам’ять, де дані зберігаються в буфері, котрий при необхідності може змінювати свій розмір, а в момент вилучення дані являють з себе масив байтів. Далі вхідні дані читаються та конвертуються в літеральни дані PGP в форматі PGPLiteralData.BINARY, далі йде запис в даних знову в ByteArrayOutputStream (bOut), для можливості подальшого експорту масиву байтів для шифрування. Потім настає етап ініціалізації генератора зашифрованих даних, при цьому створюється PGPEncryptedDataGenerator, в конструкторі передається значення CAST5, він ще має назву як CAST-128, це симетричний алгоритм шифрування розробленим Карлом Снайером (Carlisle Adams) та Стаффордом Таваресом [8]. Його головна мета розбити дані на блоки, та шифрувати. Блоки можуть становити від 64 біт, аключ може мати довжину від 40 до 128 біт, шаг зміни дорівнює 8 бітам. Поті додається метод шифрування, котрий використовує публічний ключ. Процес шифрування та запису даних, цей крок відкриває зашифрований потік, та

зв’язується з вхідним файлом. Всі дані йдуть через цей потік. Після завершення запису потік закривається. Також слід зазначити, що весь процес шифрування підлягає логуванню. На (Рисунку 3.10) ми можемо бачити послідовану логовану частину, шифрування даних.

```
2024-06-03 20:13:52Public key for part8 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part8_publicKey.asc
2024-06-03 20:13:52Private key for part8 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part8_privateKey.asc
peoples delivers opinions recovery
2024-06-03 20:13:52Public key for part9 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part9_publicKey.asc
2024-06-03 20:13:52Private key for part9 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part9_privateKey.asc
cake evaluating talks alien
2024-06-03 20:13:52Public key for part14 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part14_publicKey.asc
2024-06-03 20:13:52Private key for part14 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part14_privateKey.asc
proven seem railroad aged
2024-06-03 20:13:52Public key for part2 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part2_publicKey.asc
2024-06-03 20:13:52Private key for part2 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part2_privateKey.asc
tier corpus coordination teach
2024-06-03 20:13:53Public key for part3 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part3_publicKey.asc
2024-06-03 20:13:53Private key for part3 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part3_privateKey.asc
holly annoying worldcat two
2024-06-03 20:13:53Public key for part15 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part15_publicKey.asc
2024-06-03 20:13:53Private key for part15 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part15_privateKey.asc
```

Рисунок 3.10 Послідовна логована частина, шифрування даних

На цьому зображені ми можемо бачити успішну шифрацію фрагментів, та запис приватного та публічного ключа в певні назначені для цього директорії. Також тут видно генерування пас-фраз для кожного приватного ключа, прикладом може слугувати “ peoples delivers opinions recovery”, та вивід поточного часу. Може здивувати не послідовне шифрування пакетів, але це цілковито нормально, так як ми використовуємо потокову обробку інформації. На (Рисунку 3.11) зображено розміщення приватних, та публічних ключів, на локальному сховищі користувача.

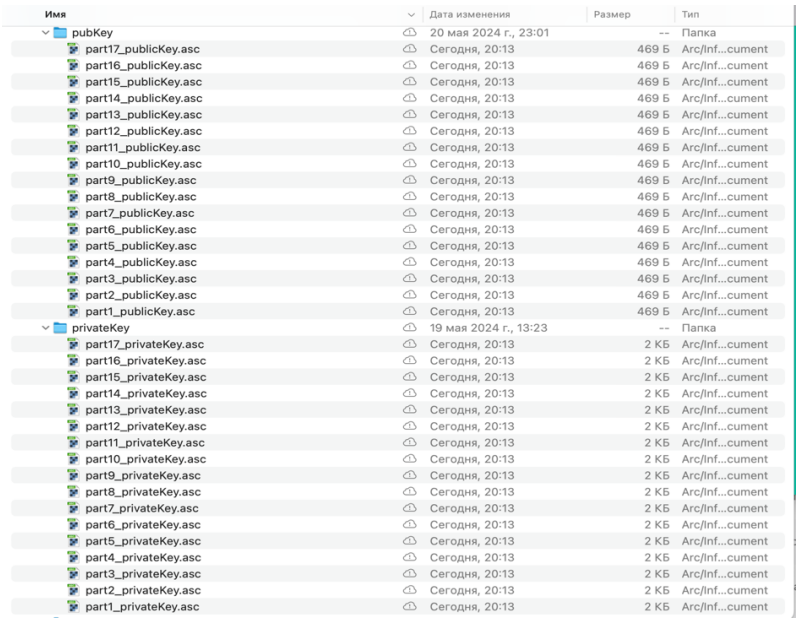



Рисунок 3.11 Розміщення приватних, та публічних ключів

Приватні ключи займають куди більше пам’яті, ніж публічні. Потрібно зазначити, що шифрування самих фрагментів, пройшло успішним образом, і це зображено на (Рисунку 3.12).

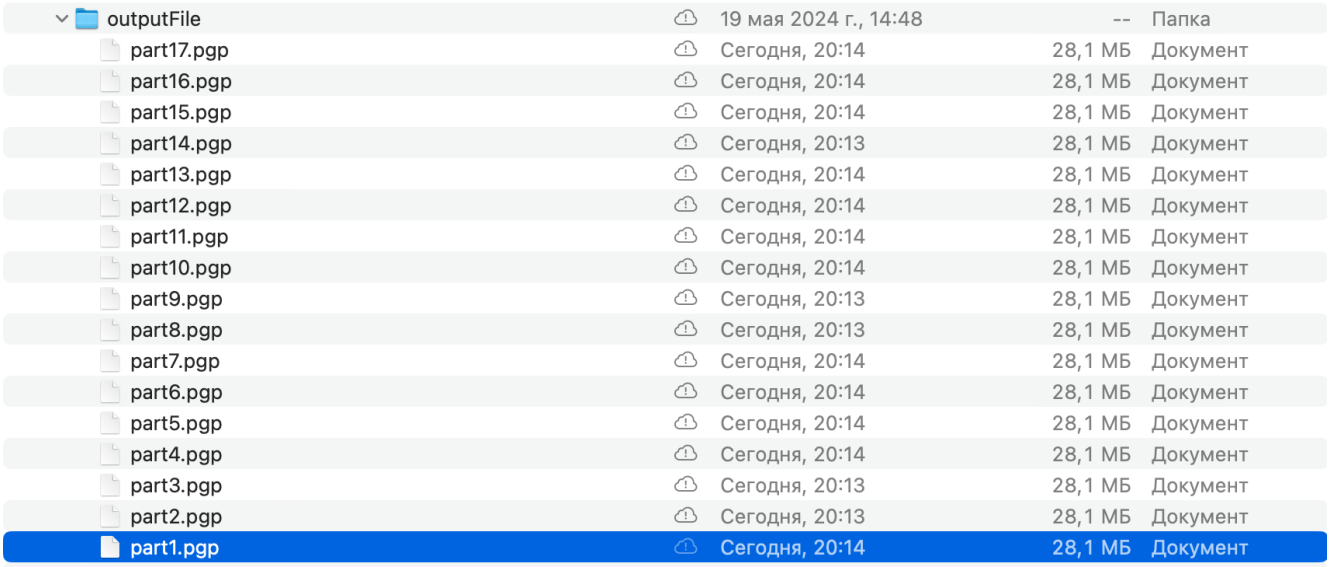



Рисунок 3.12 Шифровані фрагменти

Розмір зашифрованих фрагментів змінився стосовно базових (не зашифрованих), так як ми використовуємо асиметричний алгоритм з блоковим шифруванням. Під блок виділяється фіксоване місце і зазвичай останній блок є більшим за кількість залишкових бітів, тому розмір зашифрованого файлу є більшим. Це ще має назву падінг. Детально описано про блокове шифрування в розділах 1.1, та 2.3. На (Рисунку 3.13) зображена різниця між зашифрованою інформацією фрагментів, та базовою, котра була вхідною.

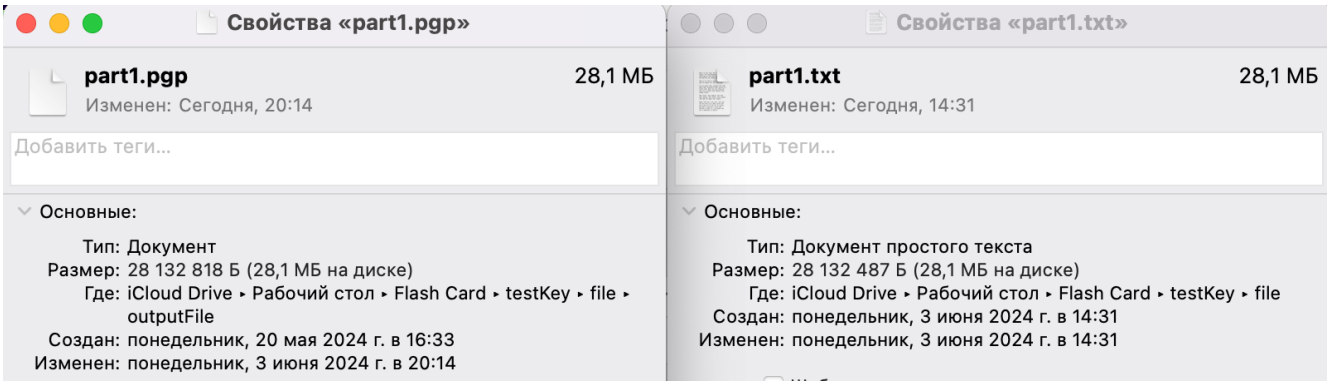



Рисунок 3.13 Зображена різниця між зашифрованою інформацією фрагментів, та базовою, котра була вхідною

Файл котрий має розширення “.txt” він є базовим, а з розширенням “.pgp” це є зашифрованим файлом.

Ми говорили про метод “encryptFile”, насправді це компонент методу “processEncryptFiles”. Реалізація методу “encryptFile”, є суто в шифруванні запропонованого фрагменту. В пошуках потрібних фрагментів для шифрування використовується метод “processEncryptFiles”. Другій метод під назвою “processEncryptFiles” інкапсулює в собі метод “encryptFile”. Головна мета методу “processEncryptFiles” полягає в коректному, автоматизованому пошуку фрагментів, та їх послідовності при шифруванні. Алгоритм дії методу “processEncryptFiles”, можливо поділити на п’ять етапів. До першого етапу відноситься коректне визначення місцезнаходження фрагменту. Метод приймає три параметри, масив шляхів до фрагментів, шлях до діректорії з публічними ключами , та масив місцезнаходжень, куди відповідні фрагменти будуть переміщуватись. Другий етап це експорт фрагментів до місця шифрування. Відносно зазначеному масиву знаходження, відповідно індексу починає пошук фрагментів, для почергового шифрування. Третій етап, для кожного знайденого фрагмента метод здійснює отримання базової назви файлу. Видалення стандартного розширення ”.txt”, потім воно буде замінене на “.pgp”. Для кожного фрагмента існує раніше згенерований публічний ключ, цей метод знаходить його, та використовує. Потім визначається на основі вхідної інформації, кінцеве розташування зашифрованого фрагменту, для його переміщення, та встановлюється раніше зазначене розширення “.pgp”. Четвертий передбачає перед початком основного процесу шифрування зробити перевірку, на наявність, та коректність формування публічного ключа, якщо щось не так, виводиться в консоль логування пряме повідомлення о проблені з приватним ключем. П’ятий етап заключний, виклик інкапсульованого методу “encryptFile”, котрий в свою чергу вже робить шифрування конкретного фрагмента.

Пас-фраза (passphrase) являє собою додатковим захистом для приватного ключа. Вона генерується разом з публічним ключем. Вся детальна інформація про пас-фразу описується в розділі 2.1. В цьому проекті для генерації пас-фрази був створений окремий клас з найменуванням “ PassphraseGenerator”. Даний клас

виконує генерацію пас-фраз на основі словника, котрий вказаний на (Рисунку 3.9), далі. Клас має такі поля, як шлях до словника, кількість слів генерації, екземпляр SecureRandom, котрий використовується для генерації псевдовипадкових значень, задля вибору певних слів зі словника. Також слід зазначити, що кількість самих слів також генерується випадковим чином. Метод “generatePassphrase”, виконує головну функцію генерації, що випадкової кількості слів, що і випадковість їх з словника. В словнику знаходиться 3931 слово. Алгоритм методу “generatePassphrase” можливо описати в шість кроків. Перший крок полягає в зчитуванні всіх можливих рядків в словнику , кожен рядок являє собою лише одне слово. З цих слів потім складається пас-фраза. Другим кроком є ініціалізація об’єкта StringBuilder, він виконує роль поєднання слів в пас-фразу. Третій крок це формування випадкової змінної шляхом перевизначення поля класу та використання бібліотеки Math.random, для створення псевдо випадкового числа від двійки до десяти, вона має назву “numWords”. Четвертий крок полягає вже у виборі слів, цикл проводить ітерації рівно, як зазначено в numWords разів, під час циклу використовується метод бібліотеки SecureRandom, а саме random.nextInt(words.size()), де words.size() це кількість слів у словнику, кожна ітерація це нове випадкове слово котре додане до листу, як структури даних. Після чого наступає п’ятий крок, і починається запис завдяки стрінг білдеру, використовується перевірка останнього елемента, якщо він останній, то в цьому випадку після нього пробіл не ставиться, якщо ні, то ставимо. Шостим кроком є перетворення об’єкта стрінг білдера в рядок. На (Рисунку 3.14) буде зображено, текстові документи, де зберігаються згенеровані пас-фрази.

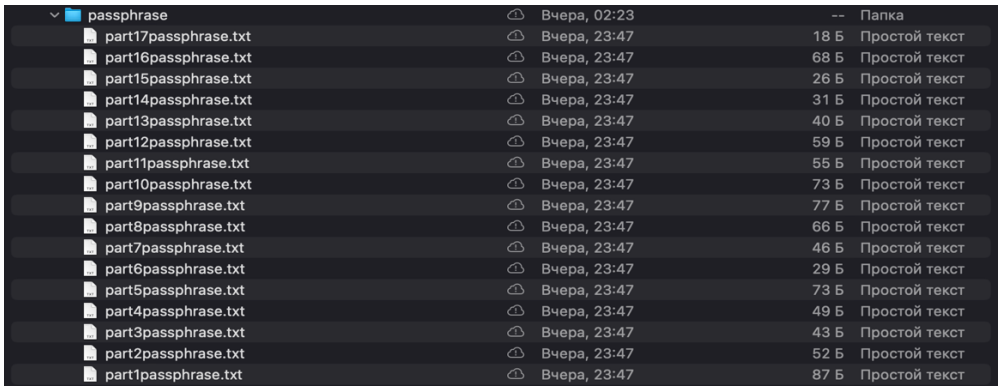



Рисунок 3.14 Текстові документи, де зберігаються пас-фрази

Дивлячись на розмір фалу кожної пас фрази, ми можемо зрозуміти, що кількість слів в кожній є різна відповідно їх розміру.

### 3.5 Реалізація алгоритму дешифрування даних

Для реалізації можливості дешифрування фрагментів, та подальшої дефрагментації їх, було розроблено клас, котрий відповідав за дешифрування фрагментів, як даних. Він має назву ”DecryptionData”. В ньому присутні два методи, вони реалізовані по типу методів “encryptFile” та “processEncryptFiles” з класу “EncryptionData”. Вони мають таку назву, як “decryptFile” та “decryptFilesInDirectory” . В свою чергу метод “decryptFilesInDirectory” інкапсулює в собі сам метод котрий проводить дешифрування “decryptFile”.

Почнемо з методу “decryptFile”, його головна мета є розшифрування вже зашифрованих фрагментів за допомогою асиметричного методу шифрування PGP (Pretty Good Privacy). Він має такі змінні методу, як: “encryptedFile” – це зашифрований файл котрий підлягає розшифровці; ”privateKeyFile” – це значення приватного ключа для можливого розшифрування; ”passphrase” – це пас-фраза, котра була використана для створення приватного ключа, та без неї не отримати доступ до приватного ключа, хоч він і використовувався при шифруванні даних; ”utputFile” – це місця куди будуть потім зберігатися фрагменти, воно може бути як і одним, так це може бути і масивом котрий робить дистрибутивне розподілення.
Цей метод реалізований 6-етапним алгоритмом. В нього входять такі пункти, як: зчитування всієї інформації із змінних методу; створення PGP об’єкту для подальшої роботи з шифрованими файлами; завантаження секретних ключів; отримання доступу до приватного ключа; повноцінна розшифровка даних; перенесення вже розшифрованих даних в їх локацію розміщення. Першим етапом є зчитування всієї інформації із змінних методу, для його реалізації відкривається encryptedFile, та перетворення його потоку з використанням методу PGPUtil.getDecoderStream, бібліотеки Bouncy Castle, котрий використовується для отримання потоку, та його читання даних з типом PGP. Потім потрібно зробити відкриття, для майбутнього читання privateKeyFile. Другим етапом є створення

PGP об’єкту, в нього входить створення PGPObjectFactory з потоком вхідного файлу, для можливості сприйняття PGP об’єктів. Поті нам потрібно отримати всі зашифровані фрагменти, записати їх в лист, та вибрати один для поточної розшифровки. Третім етапом є завантаження колекції секретних ключів, щоб реалізувати цей етап, потрібно створити колекцію PGPSecretKeyRingCollection з потоку приватного ключа. Далі отримання секретного ключу, відповідно до використаного для шифрування, використовуючи getKeyID() з encP. Четвертим етапом є отримання доступу до приватного ключа завдяки пас-фразі, в програмно розумінні використовується extractPrivateKey з використанням об’єкта JcePBESecretKeyDecryptorBuilder, котрий налаштований з пас-фразою, для отримання приватного ключа. П’ятим етапом є повноцінна розшифровка даних, в нього входить відкриття потоку даних із encP, та створення PGPObjectFactory для обробки розшифрованих даних. Та шостий останній етап, це перенесення вже розшифрованих даних в їх локацію розміщення. В нього входить перевірка на коректне розшифрування даних, потім читання даних з PGPLiteralData, перенесення їх по локаціях. Закриття всіх потоків. На (Рисунку 3.15) зображено параметри базового фрагмента, та відповідно на (Рисунку 3.16) зображено параметри того ж фрагмента, але цей фрагмент вже був зашифрований, та розшифрований.

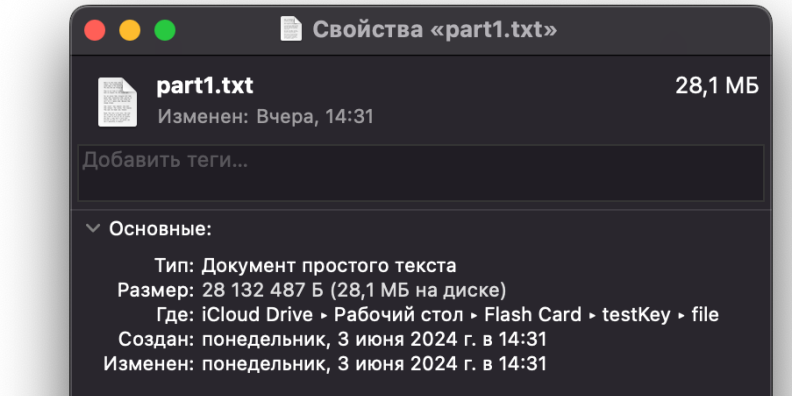



Рисунок 3.15 Зображення параметрів базового фрагменту

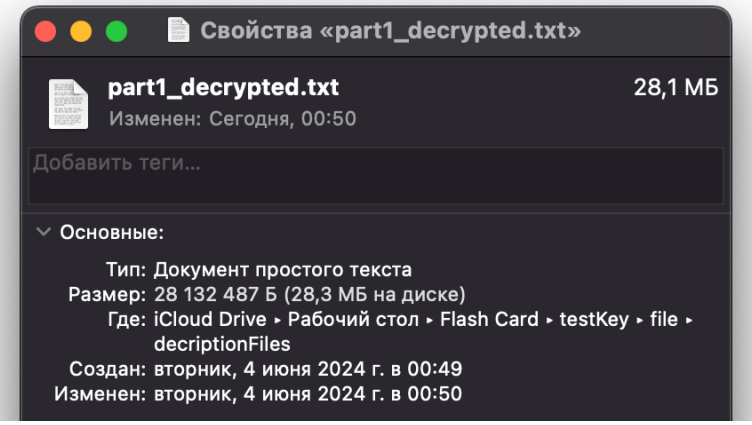



Рисунок 3.16 Зображення параметрів вже розшифрованого фрагменту

З використання зазначеної інформації, ми можемо зробити висновок, що файли є ідентичними, ми не втрачаємо дані. На (Рисунку 3.17), ми можемо побачити перелік всіх розшифрованих фрагментів.

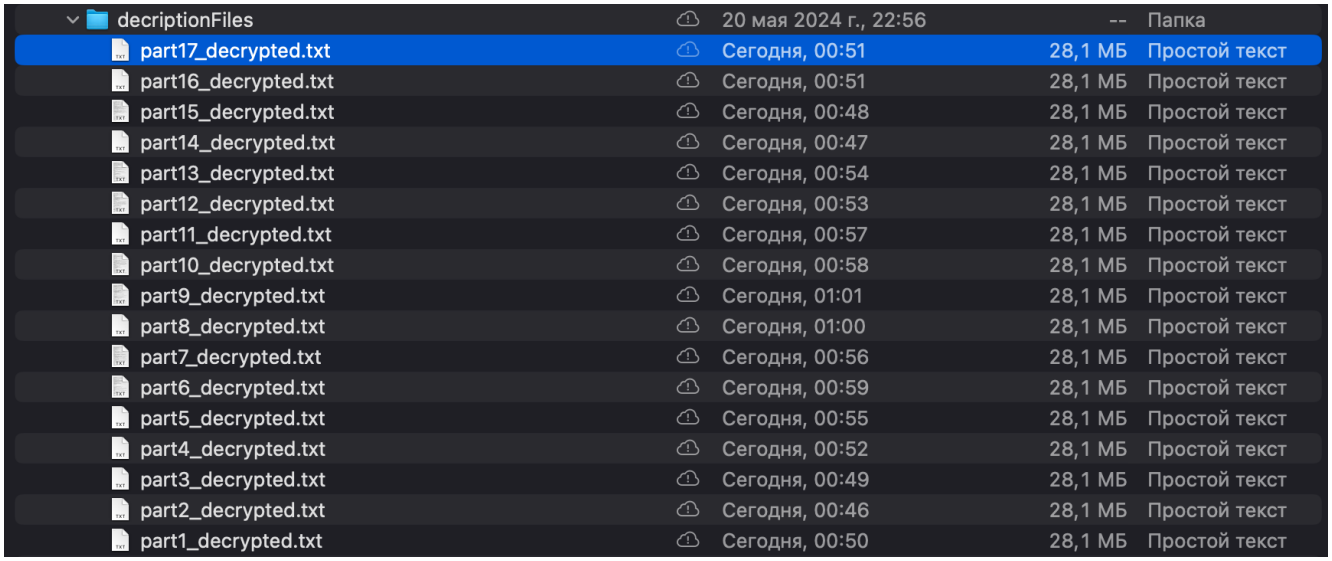



Рисунок 3.17 Зображення всіх дешифрованих фрагментів файл

# ВИСНОВКИ

Було проведено розробку методів реалізації безпечної, децентралізованої, системи, зберігання обробки інформації, використовуючи досить різні методи. Головними методами захисту слугували фрагментація, шифрування, відтворення абстрактної, локальної карти інформації користувача, застосування різних методів криптографії. Кожен з цих методів детально описаний в ході роботи.

Ці методи достатньо складні в реалізації з точки зору формування інфраструктурної системи хмарного середовища. Але якщо їх впровадити в роботу у великому, хмарному середовищі, системи безпеки зберігання даних користувачів значно зросте. Так як не можливо в стохастичних обставинах зібрати все до купи. Головним вразливим місцем даної системи виступає власно сам користувач, та володар фізичного локального сховища, для зберігання всіх ключів, та пас-фраз, а також і абстрактної карти інформації. Навіть провайдер хмарного сховища, не в силах відтворити послідовність всіх збережених фрагментів одного файлу в додаток з використанням асиметричного шифрування. Це дає змогу покласти всі ризики на користувача, крім фактору, котрий може по впливати на цілісність інформації, як фізичного знищення серверу. Слід використовувати архівування, та реплікації, що на фізичному носії користувача, що на сервері провайдера. Вході розробки, дослідження було приведено значну увагу цілісності фрагментів в момент дефрагментації, та  дешифрування, розглядання як вже готовий файл для можливості відтворення, та взаємодії.

Також існує безліч алгоритмів, котрі б змогли підвищити безпеку ще на декілька щаблів, таких як додаткові фрагменти під час фрагментації, або використання різних типів шифрування одночасно на основі сегментації даних. Крім безпеки, також можливо підвищити ефективність за рахунок той самої сегментації, але вже на об’єктному рівні сприйняття інформації.

# ПЕРЕЛІК ПОСИЛАНЬ


1) Історія створення хмарних сховищ [Електронний. ресурс] - https://uk.wikipedia.org/wiki/Хмарні_сховища#Історія
2) [Fray…86] J.-M. Fray, Y. Deswarte, and D. Powell: “Intrusion-Tolerance Using Fine-Grain Fragmentation-Scattering” IEEE Symp. on Security and Privacy pp194-203, 1986.
3) [Aggarwal...05] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu “Two Can Keep A Secret: A Distributed Architecture for Secure Database Services” in Proc. of 'CIDR' 05, Asilomar, California, pp. 186- 199, January 2005.
4) [Bessani…11] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa “DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds.” 6 th ACM SIGOPS/EuroSys’11 on Comp.
5) [Kerckhoff 1883] A. Kerckhoff “La Cryptographie militaire” Journal des Sciences Militaires, v. 9, pp 5-38, Jan. 1883, and pp 161-191, Feb. 1883.
6) [Khashan…14] O. A. Khashan, A. M. Zin, & E. A. Sundararajan “Performance study of selective encryption in comparison to full encryption for still visual images.” Journal of Zhejiang University SCIENCE C, 15(6), pp435–444, 2014.
7) Закон Амдала, виконання програми залежно від числа процесорів [Електронний ресурс] https://uk.wikipedia.org/wiki/Закон_Амдала
8) CAST-128 [Електронний ресурс] - https://uk.wikipedia.org/wiki/CAST-128

## Додаток А

Лістинг програми:

```
import javax.swing.*;
import java.awt.*;
import java.io.File;


public class Main {
    public static void main(String[] args) throws Exception {
        // Створення вікна
        JFrame frame = new JFrame("Flash Card");

        // Налаштування розмірів вікна
        frame.setSize(800, 600);
        frame.setResizable(false);

        // Встановлення положення вікна по центру екрана
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize(); // Отримання розмірів екрана
        int x = (screenSize.width - frame.getWidth()) / 2; // Розрахунок координати X
        int y = (screenSize.height - frame.getHeight()) / 2; // Розрахунок координати Y
        frame.setLocation(x, y); // Встановлення положення вікна



        float[] hsb = Color.RGBtoHSB(9, 227, 194, null); // Конвертуємо RGB у HSB
        Color backgroundColor = Color.getHSBColor(hsb[0], hsb[1], hsb[2]); // Отримуємо об'єкт Color з HSB
        frame.getContentPane().setBackground(backgroundColor);

        // Налаштування розміщення компонентів
        frame.setLayout(null); // Використовуємо null layout для точного розміщення компонентів

        // Напис "Insert the flash card into your laptop"
        JLabel label = new JLabel("Insert the USB flash drive into your laptop");
        label.setFont(new Font("Arial", Font.BOLD, 25));
        label.setBounds(50, 50, 500, 50); // Встановлюємо розміщення та розмір напису
        frame.add(label); // Додаємо напис до вікна
```

```
        // Завантаження зображення Laptop
        ImageIcon Laptop = new ImageIcon("PhotoPNG/Laptop.png");
        Image imageOFLaptop = Laptop.getImage().getScaledInstance(250, 150,
Image.SCALE_SMOOTH); // Змінюємо розмір зображення на 150x150
        ImageIcon resizedLaptop = new ImageIcon(imageOFLaptop);
        JLabel laptopLabel = new JLabel(resizedLaptop);
        laptopLabel.setBounds(275, 200, resizedLaptop.getIconWidth(),
resizedLaptop.getIconHeight());
        frame.add(laptopLabel);

        // Завантаження зображення USB_Flash_Card
        ImageIcon USB_Flash_Card = new
ImageIcon("PhotoPNG/USB_Flash_Card.png");
        Image image_OF_USB_Flash_Card =
USB_Flash_Card.getImage().getScaledInstance(250, 150, Image.SCALE_SMOOTH); //
Змінюємо розмір зображення на 150x150
        ImageIcon resized_USB_Flash_Card = new
ImageIcon(image_OF_USB_Flash_Card);
        JLabel USB_Flash_Card_Label = new JLabel(resized_USB_Flash_Card);
        USB_Flash_Card_Label.setBounds(75, 200,
resized_USB_Flash_Card.getIconWidth(), resized_USB_Flash_Card.getIconHeight());
        frame.add(USB_Flash_Card_Label);

        // Завантаження зображення Pointer
        ImageIcon Pointer = new ImageIcon("PhotoPNG/Pointer.png");
        Image imageOfPointer = Pointer.getImage().getScaledInstance(150, 100,
Image.SCALE_SMOOTH); // Змінюємо розмір зображення на 150x150
        ImageIcon resizedPointer = new ImageIcon(imageOfPointer);
        JLabel pointerLabel = new JLabel(resizedPointer );
        pointerLabel.setBounds(350, 450, resizedPointer .getIconWidth(), resizedPointer
.getIconHeight());
        frame.add(pointerLabel);

        // Завантаження зображення ScreenShot
        ImageIcon ScreenShot = new ImageIcon("PhotoPNG/ScreenShot.png");
        Image screenShotOfPointer = ScreenShot.getImage().getScaledInstance(450, 450,
Image.SCALE_SMOOTH); // Змінюємо розмір зображення на 150x150
        ImageIcon resizedscreenShot= new ImageIcon(screenShotOfPointer);
        JLabel screenShotLabel = new JLabel(resizedscreenShot );
        screenShotLabel.setBounds(325, 225, resizedPointer .getIconWidth(),
resizedPointer .getIconHeight());
        frame.add(screenShotLabel);
```

```
        // Кнопка "GO"
        JButton button = new JButton("GO");
        button.setBounds(300, 400, 200, 75); // Встановлюємо розміщення та розмір кнопки
        button.setFont(new Font("Arial", Font.BOLD, 40));
        frame.add(button); // Додаємо кнопку до вікна

        button.addActionListener(e -> {
            // Визначення шляху, за яким зазвичай підключаються флеш-карти

            try {
                FlashDrive personal = new FlashDrive();
            } catch (Exception ex) {
                throw new RuntimeException(ex);
            }


            String flashDrivePath = "/Users/nikitateremko/Desktop/Flash Card/Personal_Info";

            // Створення файлового об'єкту для перевірки існування флеш-карти
            File flashDrive = new File(flashDrivePath);

            // Перевірка наявності флеш-карти
            if (flashDrive.exists() && flashDrive.isDirectory()) {
//                JOptionPane.showMessageDialog(null, "Flash drive is inserted into your laptop.");

if(HendlerOfFlashCard.checkExistOfFileOnFlashDrive(flashDrivePath,"Personal.json") ) {

HendlerOfFlashCard.checkAndCreateFileConfigOnFlashDrive(flashDrivePath, "Personal.json");
                    ConstructorOfFrame firstAuthorization = new ConstructorOfFrame(600, 300, "Authorization");
                    firstAuthorization.setBackground(0, 102, 255);
                    firstAuthorization.newLabel("You shood authorization in system.", 25, 50, 50,  500, 50);

                    JComboBox<String> comboOsBox = new JComboBox<>(new String[]{"Ubuntu"});
                    firstAuthorization.dropdownCheckbox("Operating system", 50, 100, 150, 30, comboOsBox);
```

```
                JComboBox<String> comboTypesOfEncryptBox = new
JComboBox<>(new String[]{"pgp"});
                firstAuthorization.dropdownCheckbox("Type of encrypt", 280, 100, 150,
30, comboTypesOfEncryptBox);

            }
            else{

            }
        } else {
            JOptionPane.showMessageDialog(null, "No flash drive found in your
laptop.");
        }
    });


    // Налаштування видимості вікна
    frame.setVisible(true);

    // Налаштування дії при закритті вікна
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    String inputFile = "/Users/nikitateremko/Desktop/Flash Card/Data/Запись экрана
2024-01-06 в 18.33.48.mov";
    String occureFolder = "/Users/nikitateremko/Desktop/Flash
Card/FragmentsOfData";
    DataOfFile dataFile = new DataOfFile(inputFile,occureFolder);
    HandleOfData.splitFile(dataFile);
    dataFile.setOutputFile("/Users/nikitateremko/Desktop/Flash
Card/testKey/fullFile/merged_output");
    HandleOfData.mergeFiles(dataFile);

    System.out.println("Name of file : " + dataFile.getNameOfFile()+"\n"+ "Total
count memory : "+ dataFile.getTotalCountMemoryOfFile()+"\n" + "Count fragments :
"+dataFile.getCountSplitingOfFile());




    String folderForPublicKey = "/Users/nikitateremko/Desktop/Flash
```

```
Card/testKey/file/pubKey";
        String sourceFolder = occureFolder;
        String folderForPrivateKey = "/Users/nikitateremko/Desktop/Flash
Card/testKey/file/privateKey";

KeyGenerator.generatePublicKeyForFiles(sourceFolder,folderForPublicKey,folderFor
PrivateKey, new PassphraseGenerator());

        EncryptionData.processEncryptFiles(sourceFolder, folderForPublicKey,
"/Users/nikitateremko/Desktop/Flash Card/testKey/file/outputFile" );
        DecryptionData.decryptFilesInDirectory("/Users/nikitateremko/Desktop/Flash
Card/testKey/file/outputFile", "/Users/nikitateremko/Desktop/Flash
Card/testKey/file/privateKey" ,"/Users/nikitateremko/Desktop/Flash
Card/testKey/file/passphrase","/Users/nikitateremko/Desktop/Flash
Card/testKey/file/decriptionFiles");

        HandleOfData.mergeFiles("/Users/nikitateremko/Desktop/Flash
Card/testKey/file/decriptionFiles", "/Users/nikitateremko/Desktop/Flash
Card/testKey/file/decriptionFiles/fullFile", "/Users/nikitateremko/Desktop/Flash
Card/testKey/file/decriptionFiles/fullFile");

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.security.SecureRandom;
import java.util.List;

public class PassphraseGenerator {

    private static final String WORD_LIST_PATH = "/Users/nikitateremko/Desktop/Flash
Card/testKey/word_list/word_list.txt"; // Шлях до файлу зі словником
    private int num_words = 4; // Кількість слів у фразі-паролі
    private static final SecureRandom random = new SecureRandom();

    public String generatePassphrase() throws IOException {
        List<String> words = Files.readAllLines(Paths.get(WORD_LIST_PATH)); //
Зчитування всіх слів з файлу
        StringBuilder passphrase = new StringBuilder();
        num_words = (int)(Math.random() * 10) + 1;
        for (int i = 0; i < num_words; i++) {
            String word = words.get(random.nextInt(words.size())); // Вибір випадкового
слова
            passphrase.append(word);
            if (i < num_words - 1) {
```

```
            passphrase.append(" "); // Додавання пробілу між словами
        }
    }

    return passphrase.toString();
  }



}


import org.bouncycastle.bcpg.ArmoredOutputStream;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.openpgp.*;
import org.bouncycastle.openpgp.operator.PGPDigestCalculator;
import org.bouncycastle.openpgp.operator.jcajce.JcaPGPContentSignerBuilder;
import
org.bouncycastle.openpgp.operator.jcajce.JcaPGPDigestCalculatorProviderBuilder;
import org.bouncycastle.openpgp.operator.jcajce.JcaPGPKeyPair;
import org.bouncycastle.openpgp.operator.jcajce.JcePBESecretKeyEncryptorBuilder;

import java.io.*;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Security;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Date;

public class KeyGenerator {

  static {
    Security.addProvider(new BouncyCastleProvider());
  }

  public static void generatePublicKeyForFiles(String sourceFolder, String
outputPathForPublic, String outputPathForPrivate, PassphraseGenerator
generatorPassphrase) throws Exception {
    String passphrase;
    File folder = new File(sourceFolder);
    File[] files = folder.listFiles(new FilenameFilter() {
      public boolean accept(File dir, String name) {
        return name.startsWith("part") && name.endsWith(".txt");
```

```
        }
    });

    String passphraseDirPath = "/Users/nikitateremko/Desktop/Flash
Card/testKey/file/passphrase";
    File passphraseDir = new File(passphraseDirPath);
    if (!passphraseDir.exists()) {
        passphraseDir.mkdirs(); // Створює директорію, якщо вона не існує
    }




    if (files != null) {
        for (File file : files) {
            String baseName = file.getName().substring(0,
file.getName().lastIndexOf('.'));
            String outputPathPub = outputPathForPublic + "/" + baseName +
"_publicKey.asc";
            String outputPathPriv = outputPathForPrivate + "/" + baseName +
"_privateKey.asc";
            passphrase = generatorPassphrase.generatePassphrase();
            String passphraseNameInDir = baseName + "passphrase.txt";
            File passphraseFile = new File(passphraseDir, passphraseNameInDir); //
Файл для збереження passphrase
            try (BufferedWriter writer = new BufferedWriter(new
FileWriter(passphraseFile))) {
                writer.write(passphrase); // Запис passphrase в файл
            } catch (IOException e) {
                System.err.println("Error writing passphrase to file: " + e.getMessage());
            }
            generateKeyPair(outputPathPub, outputPathPriv, passphrase);
            LocalDateTime now = LocalDateTime.now();
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
            String formattedDateTime = now.format(formatter);
            System.out.println( formattedDateTime + "Public key for " + baseName + "
generated and saved to: " + outputPathPub);
            System.out.println( formattedDateTime + "Private key for " + baseName + "
generated and saved to: " + outputPathPriv);
            System.out.println(passphrase);
        }
    } else {
        System.out.println("No files found in the specified directory.");
```

```
        }
    }

    public static void generatePublicKey(String outputPathForPublic, String
outputPathForPrivate ) throws Exception {
        KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA", "BC");
        keyGenerator.initialize(2048);
        KeyPair keyPair = keyGenerator.generateKeyPair();

        PGPDigestCalculator sha256Calc = new
JcaPGPDigestCalculatorProviderBuilder().build().get(PGPUtil.SHA256);

        PGPKeyPair pgpKeyPair = new JcaPGPKeyPair(PGPPublicKey.RSA_GENERAL,
keyPair, new Date());

        try (OutputStream pubOut = new ArmoredOutputStream(new
FileOutputStream(outputPathForPublic))) {
            pgpKeyPair.getPublicKey().encode(pubOut);
        }

    }

    public static void generateKeyPairForEntry(String outputPathForPublic, String
outputPathForPrivate, String passphrase) throws Exception {
        if (passphrase == null || passphrase.isEmpty()) {
            throw new IllegalArgumentException("Passphrase cannot be null or empty.");
        }

        KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA", "BC");
        keyGenerator.initialize(2048);
        KeyPair keyPair = keyGenerator.generateKeyPair();

        PGPDigestCalculator sha1Calc = new
JcaPGPDigestCalculatorProviderBuilder().build().get(PGPUtil.SHA1);
        PGPKeyPair pgpKeyPair = new JcaPGPKeyPair(PGPPublicKey.RSA_GENERAL,
keyPair, new Date());

        String userId = "user@example.com"; // Замініть на реальний ідентифікатор
користувача

        PGPSecretKey secretKey = new PGPSecretKey(
                PGPSignature.DEFAULT_CERTIFICATION,
                pgpKeyPair,
                userId,
```

```
            sha1Calc,
            null,
            null,
            new
JcaPGPContentSignerBuilder(pgpKeyPair.getPublicKey().getAlgorithm(),
PGPUtil.SHA1),
            new JcePBESecretKeyEncryptorBuilder(PGPEncryptedData.CAST5,
sha1Calc).setProvider("BC").build(passphrase.toCharArray())
        );

        // Збереження публічного ключа
        try (OutputStream pubOut = new ArmoredOutputStream(new
FileOutputStream(outputPathForPublic))) {
            pgpKeyPair.getPublicKey().encode(pubOut);
        }

        // Збереження приватного ключа
        try (OutputStream privOut = new ArmoredOutputStream(new
FileOutputStream(outputPathForPrivate))) {
            secretKey.encode(privOut);
        }
    }

    public static void generateKeyPair(String outputPathForPublic, String
outputPathForPrivate, String passphrase) throws Exception {
        KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA", "BC");
        keyGenerator.initialize(2048);
        KeyPair keyPair = keyGenerator.generateKeyPair();

        // Використання SHA1 замість SHA256 для обчислення контрольних сум
ключів
        PGPDigestCalculator sha1Calc = new
JcaPGPDigestCalculatorProviderBuilder().build().get(PGPUtil.SHA1);

        PGPKeyPair pgpKeyPair = new JcaPGPKeyPair(PGPPublicKey.RSA_GENERAL,
keyPair, new Date());

        // Generate secret key
        PGPSecretKey secretKey = new PGPSecretKey(
            PGPSignature.DEFAULT_CERTIFICATION,
            pgpKeyPair,
            "user@example.com",
            sha1Calc,
            null, // no additional user attributes
```

```
            null, // no additional signatures
            new
JcaPGPContentSignerBuilder(pgpKeyPair.getPublicKey().getAlgorithm(),
PGPUtil.SHA1),
            new JcePBESecretKeyEncryptorBuilder(PGPEncryptedData.CAST5,
sha1Calc).setProvider("BC").build(passphrase.toCharArray())
        );

        // Save public key in armored format
        try (OutputStream pubOut = new ArmoredOutputStream(new
FileOutputStream(outputPathForPublic))) {
            pgpKeyPair.getPublicKey().encode(pubOut);
        }

        // Save private key in armored format
        try (OutputStream privateOut = new ArmoredOutputStream(new
FileOutputStream(outputPathForPrivate))) {
            secretKey.encode(privateOut);
        }
    }
}

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class HendlerOfFlashCard {

//
    public static void checkAndCreateFileConfigOnFlashDrive(String folderPath, String
fileName) {
        // Формуємо шлях до файлу JSON
//          FlashDrive personal = new FlashDrive();
        Path filePath = Path.of(folderPath, fileName);

        // Перевіряємо, чи файл існує
        if (!Files.exists(filePath)) {
            try {
                // Створюємо об'єкт ObjectMapper для роботи з Jackson
```

```
            ObjectMapper objectMapper = new ObjectMapper();

            // Створюємо порожній об'єкт JSON
            ObjectNode rootNode = objectMapper.createObjectNode();

            // Записуємо порожній об'єкт JSON у файл
            objectMapper.writeValue(new File(filePath.toString()), rootNode);

            System.out.println("Created new JSON file: " + filePath);
        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {
        System.out.println("JSON file already exists: " + filePath);
    }
}

public static void checkAndCreateFolderConfigOnFlashDrive(String folderPath,
String folderName) {
    // Формуємо шлях до папки
    Path folderPathComplete = Path.of(folderPath, folderName);

    // Перевіряємо, чи папка існує
    if (!Files.exists(folderPathComplete)) {
        try {
            // Створюємо папку
            Files.createDirectories(folderPathComplete);

            System.out.println("Created new "+ folderName +": " + folderPath);
        } catch (IOException e) {
            System.err.println("Failed to create the folder: " + folderPathComplete);
            e.printStackTrace();
        }
    } else {
        System.out.println(folderName +" already exists: " + folderPath);
    }
}

public static boolean checkExistOfFileOnFlashDrive(String folderPath, String
fileName){
    Path filePath = Path.of(folderPath, fileName);
    if (!Files.exists(filePath)) {
            System.out.println("Created new "+ fileName +": " + filePath);
            return true;
```

```
        } else {
            System.out.println(fileName +" already exists: " + filePath);
        }
        return false;
    }
    public static void createEmptyFile(String filePath) throws IOException {
        Path path = Paths.get(filePath);

        // Створення пустого файлу, якщо він ще не існує.
        if (Files.notExists(path)) {
            Files.createFile(path); // Створює файл
            System.out.println("Пустий файл було створено: " + filePath);
        } else {
            System.out.println("Файл уже існує: " + filePath);
        }
    }

    // Метод для запису вибраного значення у JSON файл
    public static void writeToJsonFile(String selectedOption, String
keyFotSelectedOption) throws IOException {
        // Створення об'єкту ObjectMapper

          ObjectMapper objectMapper = new ObjectMapper();
        // Створення об'єкта JSON
//          ObjectNode rootNode = objectMapper.createObjectNode();
            ObjectNode rootNode = (ObjectNode) objectMapper.readTree(new
File("/Users/nikitateremko/Desktop/Flash Card/Personal_Info/Personal.json"));

        // Запис вибраного значення


        rootNode.put(selectedOption, keyFotSelectedOption);

        // Запис JSON об'єкта у файл
        try {
            objectMapper.writeValue(new File("/Users/nikitateremko/Desktop/Flash
Card/Personal_Info/Personal.json"), rootNode);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.*;
import java.util.Arrays;
import java.util.Comparator;

public class HandleOfData {

    public static void splitFile(DataOfFile datafile) throws IOException {
//        File file = new File(datafile.getFilePath());
        long fileSize = datafile.getTotalCountMemoryOfFile();
        long partSize = fileSize / datafile.getCountSplitingOfFile(); // Розмір кожної частини

        try (FileInputStream inputStream = new FileInputStream(datafile.getFile());
             BufferedInputStream bufferedInputStream = new BufferedInputStream(inputStream)) {

            byte[] buffer = new byte[(int) partSize];
            int bytesRead;

            for (int i = 1, j = 0; i <= datafile.getCountSplitingOfFile(); i++, j++) {
                String partName = datafile.getOccurFolder() + File.separator + "part" + i + ".txt";
                datafile.setMapOfPathsOfFile(partName, datafile.getOccurFolder());
                try (FileOutputStream outputStream = new FileOutputStream(partName);
                     BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(outputStream)) {

                    bytesRead = bufferedInputStream.read(buffer);
                    if (bytesRead == -1) {
                        break;
                    }
                    bufferedOutputStream.write(buffer, 0, bytesRead);
                }
            }
        }
    }

    public static void mergeFiles(DataOfFile datafile) throws IOException {
        String outputFile = datafile.getOutputFile() + "." + datafile.getFileExtension(datafile.getNameOfFile());
        long fileSize = datafile.getTotalCountMemoryOfFile();
        long partSize = fileSize / datafile.getCountSplitingOfFile(); // Розмір кожної частини
        try (FileOutputStream outputStream = new FileOutputStream(outputFile);
```

```
         BufferedOutputStream bufferedOutputStream = new
BufferedOutputStream(outputStream)) {

        for (int i = 1; i <= datafile.getCountSplitingOfFile(); i++) {
            String partName = datafile.getOccurFolder() + File.separator + "part" + i +
".txt";
            try (FileInputStream inputStream = new FileInputStream(partName);
                 BufferedInputStream bufferedInputStream = new
BufferedInputStream(inputStream)) {

                byte[] buffer = new byte[(int) partSize];
                int bytesRead;
                while ((bytesRead = bufferedInputStream.read(buffer)) != -1) {
                    bufferedOutputStream.write(buffer, 0, bytesRead);
                }
            }
        }
    }

   public static void mergeFiles(String pathOfFiles, String pathToMerge, String
outputFile) throws IOException {
        File directory = new File(pathOfFiles);
        File[] files = directory.listFiles((dir, name) ->
name.matches("part\\d+_decrypted"));

        // Сортування файлів за номером частини в назві
        if (files != null) {
            Arrays.sort(files, new Comparator<File>() {
                public int compare(File f1, File f2) {
                    int num1 = Integer.parseInt(f1.getName().replaceAll("\\D+", ""));
                    int num2 = Integer.parseInt(f2.getName().replaceAll("\\D+", ""));
                    return num1 - num2;
                }
            });

            // Створення шляху вихідного файлу
            File mergedFile = new File(pathToMerge, outputFile);

            try (OutputStream out = new BufferedOutputStream(new
FileOutputStream(mergedFile))) {
                byte[] buffer = new byte[1024];
                int bytesRead;
```

```
            for (File file : files) {
                try (InputStream in = new BufferedInputStream(new
FileInputStream(file))) {
                    while ((bytesRead = in.read(buffer)) != -1) {
                        out.write(buffer, 0, bytesRead);
                    }
                }
            }
        }
        System.out.println("Merged file created at: " + mergedFile.getPath());
    } else {
        System.out.println("No files to merge.");
    }
  }
}


import java.util.HashMap;
import java.util.Map;

public class FlashDrive {
   private long id;
   private String flashDrivePath;
   private Map<String, String> keys; // Карта для зберігання ключів (ім'я файлу ->
частина ключа)
   private boolean authorization;


   public FlashDrive() throws Exception {
      this.id = 1;
      this.keys = new HashMap<>();

HendlerOfFlashCard.checkAndCreateFolderConfigOnFlashDrive("/Users/nikitateremk
o/Desktop/Flash Card","Personal_Info");

HendlerOfFlashCard.checkAndCreateFolderConfigOnFlashDrive("/Users/nikitateremk
o/Desktop/Flash Card/Personal_Info","Personal_Key");
      String folderForPublicKey = "/Users/nikitateremko/Desktop/Flash
Card/Personal_Info/Personal_Key/pub_key/pub_key.asc";

HendlerOfFlashCard.checkAndCreateFolderConfigOnFlashDrive("/Users/nikitateremk
o/Desktop/Flash Card/Personal_Info/Personal_Key","pub_key");
      HendlerOfFlashCard.createEmptyFile("/Users/nikitateremko/Desktop/Flash
Card/Personal_Info/Personal_Key/pub_key/pub_key.asc");
```

```
        String folderForPrivateKey = "/Users/nikitateremko/Desktop/Flash
Card/Personal_Info/Personal_Key/private_key/private_key.asc";

HendlerOfFlashCard.checkAndCreateFolderConfigOnFlashDrive("/Users/nikitateremk
o/Desktop/Flash Card/Personal_Info/Personal_Key","private_key");
        HendlerOfFlashCard.createEmptyFile("/Users/nikitateremko/Desktop/Flash
Card/Personal_Info/Personal_Key/private_key/private_key.asc");
        KeyGenerator.generateKeyPairForEntry(folderForPublicKey,
folderForPrivateKey, "passphrase");
    }

    public void setId(long id){
        this.id = id;
    }

    public long getId(){
        return id;
    }


}


import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.openpgp.*;
import org.bouncycastle.openpgp.operator.bc.BcKeyFingerprintCalculator;
import org.bouncycastle.openpgp.operator.jcajce.JcePGPDataEncryptorBuilder;
import
org.bouncycastle.openpgp.operator.jcajce.JcePublicKeyKeyEncryptionMethodGenerato
r;

import java.io.*;
import java.security.NoSuchProviderException;
import java.security.SecureRandom;
import java.security.Security;

public class EncryptionData {
    static {
        Security.addProvider(new BouncyCastleProvider());
    }

    public static void encryptFile(File inputFile, File publicKeyFile, File outputFile)
throws IOException, PGPException, NoSuchProviderException {
        try (InputStream keyIn = PGPUtil.getDecoderStream(new
```

```
FileInputStream(publicKeyFile));
          OutputStream out = new FileOutputStream(outputFile)) {
         PGPPublicKeyRing keyRing = new PGPPublicKeyRing(keyIn, new
BcKeyFingerprintCalculator());
         PGPPublicKey publicKey = keyRing.getPublicKey();

         ByteArrayOutputStream bOut = new ByteArrayOutputStream();
         PGPUtil.writeFileToLiteralData(bOut, PGPLiteralData.BINARY, inputFile);
         byte[] data = bOut.toByteArray();

         PGPEncryptedDataGenerator encGen = new PGPEncryptedDataGenerator(
               new JcePGPDataEncryptorBuilder(PGPEncryptedData.CAST5)
                     .setWithIntegrityPacket(true)
                     .setSecureRandom(new SecureRandom())
                     .setProvider("BC"));
         encGen.addMethod(new
JcePublicKeyKeyEncryptionMethodGenerator(publicKey).setProvider("BC"));

         try (OutputStream cOut = encGen.open(out, data.length)) {
            cOut.write(data);
         }
      }
      System.out.println("Encryption of " + inputFile.getName() + " using " +
publicKeyFile.getName() + " completed successfully.");
   }

   public static void processEncryptFiles(String filesDir, String keysDir, String
encryptedDir) throws Exception {
      File filesDirectory = new File(filesDir);
      File[] files = filesDirectory.listFiles(new FilenameFilter() {
         @Override
         public boolean accept(File dir, String name) {
            return name.matches("part\\d+\\.txt"); // Regex для вибору файлів з назвами
"part1.txt", "part2.txt", ...
         }
      });

      if (files != null) {
         for (File file : files) {

            String baseName = file.getName().replace(".txt", ""); // Видалення
розширення ".txt"
            File publicKeyFile = new File(keysDir, baseName + "_publicKey.asc"); //
Знаходимо відповідний публічний ключ
```

```
            File outputFile = new File(encryptedDir, baseName + ".pgp"); // Визначення місця збереження зашифрованого файлу

            if (publicKeyFile.exists() && file.isFile()) {
                encryptFile(file, publicKeyFile, outputFile);
                System.out.println("Encrypted " + baseName + " successfully.");
            } else {
                System.out.println("Skipping " + baseName + ", public key not found or not a file.");
            }
        }
    }
}


import org.bouncycastle.openpgp.*;
import org.bouncycastle.openpgp.operator.bc.BcKeyFingerprintCalculator;
import org.bouncycastle.openpgp.operator.jcajce.JcePBESecretKeyDecryptorBuilder;
import org.bouncycastle.openpgp.operator.jcajce.JcePublicKeyDataDecryptorFactoryBuilder;

import java.io.*;
import java.nio.file.Files;
import java.security.NoSuchProviderException;

public class DecryptionData {
    public static void decryptFile(File encryptedFile, File privateKeyFile, String passphrase, File outputFile) throws IOException, PGPException, NoSuchProviderException {
        InputStream in = PGPUtil.getDecoderStream(new FileInputStream(encryptedFile));
        InputStream keyIn = new FileInputStream(privateKeyFile);

        PGPObjectFactory pgpF = new PGPObjectFactory(in, new BcKeyFingerprintCalculator());
        PGPEncryptedDataList encList = (PGPEncryptedDataList) pgpF.nextObject();
        PGPPublicKeyEncryptedData encP = (PGPPublicKeyEncryptedData) encList.getEncryptedDataObjects().next();

        PGPSecretKeyRingCollection pgpSec = new PGPSecretKeyRingCollection(
                PGPUtil.getDecoderStream(keyIn), new BcKeyFingerprintCalculator());

        PGPSecretKey pgpSecKey = pgpSec.getSecretKey(encP.getKeyID());
        PGPPrivateKey pgpPrivKey = pgpSecKey.extractPrivateKey(
                new
```

```
JcePBESecretKeyDecryptorBuilder().setProvider("BC").build(passphrase.toCharArray(
)));

        InputStream clear = encP.getDataStream(new
JcePublicKeyDataDecryptorFactoryBuilder().setProvider("BC").build(pgpPrivKey));

        PGPObjectFactory plainFact = new PGPObjectFactory(clear, new
BcKeyFingerprintCalculator());
        Object message = plainFact.nextObject();
        if (message instanceof PGPLiteralData) {
            PGPLiteralData ld = (PGPLiteralData) message;
            InputStream unc = ld.getInputStream();
            OutputStream fOut = new FileOutputStream(outputFile);

            int ch;
            while ((ch = unc.read()) != -1) {
                fOut.write(ch);
            }

            fOut.close();
        }
        clear.close();
        in.close();
        keyIn.close();
        System.out.println("Decryption of " + encryptedFile.getName() + " completed
successfully.");
    }


    public static void decryptFilesInDirectory(String encryptedFilesDir, String keysDir,
String passphraseDir, String outputDir) {
        File encDir = new File(encryptedFilesDir);
        File[] encryptedFiles = encDir.listFiles((dir, name) -> name.endsWith(".pgp"));

        if (encryptedFiles == null || encryptedFiles.length == 0) {
            System.out.println("No encrypted files found.");
            return;
        }

        for (File encryptedFile : encryptedFiles) {
            String baseName = encryptedFile.getName().replace(".pgp", "");
            File privateKeyFile = new File(keysDir, baseName + "_privateKey.asc");
            File passphraseFile = new File(passphraseDir, baseName + "passphrase.txt");
            File outputFile = new File(outputDir, baseName + "_decrypted.txt");
```

```
            if (!outputFile.getParentFile().exists()) {
                outputFile.getParentFile().mkdirs();
            }

            try {
                String passphrase = new String(Files.readAllBytes(passphraseFile.toPath()));
                decryptFile(encryptedFile, privateKeyFile, passphrase, outputFile);
                System.out.println("Decryption completed for: " + encryptedFile.getName());
            } catch (Exception e) {
                System.err.println("Failed to decrypt " + encryptedFile.getName() + ": " +
e.getMessage());
            }
        }
    }
}

import java.io.File;
import java.util.HashMap;
import java.util.Map;

public class DataOfFile {
    private File file;
    private String nameOfFile;
    private String fileExtension;
    private String filePath = "";

    private String occurFolder = "";
    private String outputFile = "";
    private long totalCountMemoryOfFile;
    private int minSizeForSpliting = 3;
    private int countSplitingOfFile;
    private Map <String, String> mapOfPathsOfFile ;




    public DataOfFile(String filePath, String occurFolder) {

        file = new File(filePath);
        this.filePath = filePath;
        this.nameOfFile = file.getName();
        this.fileExtension = getFileExtension(nameOfFile);
        this.occurFolder = occurFolder;
```

```
        this.totalCountMemoryOfFile = file.length();
        this.countSplitingOfFile = calculateSplitCount(totalCountMemoryOfFile,
minSizeForSpliting);
        mapOfPathsOfFile = new HashMap<>();
    }

    public void setNameOfFile(String nameOfFile) {
        this.nameOfFile = nameOfFile;
    }

    public void setTotalCountMemoryOfFile(int totalCountMemoryOfFile) {
        this.totalCountMemoryOfFile = totalCountMemoryOfFile;
    }

    public void setCountSplitingOfFile(int countSplitingOfFile) {
        this.countSplitingOfFile = countSplitingOfFile;
    }

    public static String setFileExtension(String fileName, String newExtension) {
        int dotIndex = fileName.lastIndexOf('.');
        if (dotIndex >= 0) {
            // Видаляємо існуюче розширення і додаємо нове
            return fileName.substring(0, dotIndex) + "." + newExtension;
        } else {
            // Якщо розширення не було, просто додаємо нове
            return fileName + "." + newExtension;
        }
    }

    public void setMapOfPathsOfFile(String key, String value) {

        if (!mapOfPathsOfFile.containsKey(key)) {
            mapOfPathsOfFile.put(key, value);
        } else {
            System.out.println("User with ID 4 already exists.");
        }
    }

    public void setOutputFile(String outputFile) {
        this.outputFile = outputFile;
    }

    public String getNameOfFile() {
        return this.nameOfFile;
```

```
    }

    public long getTotalCountMemoryOfFile() {
        return this.totalCountMemoryOfFile;
    }

    public int getCountSplitingOfFile() {
        return this.countSplitingOfFile;
    }
    public String getFilePath() {
        return this.filePath;
    }

    public String getOccurFolder() {
        return this.occurFolder;
    }

    public File getFile() {
        return this.file;
    }

    public static String getFileExtension(String fileName) {
        int dotIndex = fileName.lastIndexOf('.');
        if (dotIndex >= 0) {
            return fileName.substring(dotIndex + 1);
        }
        return ""; // Немає розширення
    }

    public Map<String, String> getMapOfPathsOfFile() {
        return mapOfPathsOfFile;
    }

    public String getOutputFile() {
        return outputFile;
    }

    public int calculateSplitCount(long totalCountMemoryOfFile, int
minSizeForSpliting) {
        if (minSizeForSpliting <= 0) {
            throw new IllegalArgumentException("Максимальний розмір частини має
бути більшим ніж 0");
        }
        while(totalCountMemoryOfFile % minSizeForSpliting != 0) {
```

```
            minSizeForSpliting++; // Додаємо ще одну частину, якщо є залишок
        }
        return minSizeForSpliting;
    }
}


import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;

public class ConstructorOfFrame {
    private int width;
    private int height;
    private String name;
    JFrame frame;

    public ConstructorOfFrame(int width, int height, String name){
        this.name = name;
        this.width = width;
        this.height = height;
        frame = new JFrame(this.name);
        // Налаштування розмірів вікна
        frame.setSize(this.width, this.height);
        frame.setResizable(false);
        // Встановлення положення вікна по центру екрана
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize(); //
Отримання розмірів екрана
        int x = (screenSize.width - frame.getWidth()) / 2; // Розрахунок координати X
        int y = (screenSize.height - frame.getHeight()) / 2; // Розрахунок координати Y
        frame.setLocation(x, y); // Встановлення положення вікна
        frame.setLayout(null);
        // Налаштування видимості вікна
        frame.setVisible(true);

        // Налаштування дії при закритті вікна
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void setHeight(int height){
        this.height = height;
    }
```

```
    public int getHeight(){
        return this.height;
    }
    public void setWidth(int width){
        this.width = width;
    }
    public int getWidth(){
        return this.width;
    }
    public void setBackground(int r, int g, int b) {
        float[] hsb = Color.RGBtoHSB(r, g, b, null); // Конвертуємо RGB у HSB
        Color backgroundColor = Color.getHSBColor(hsb[0], hsb[1], hsb[2]); //
Отримуємо об'єкт Color з HSB
        frame.getContentPane().setBackground(backgroundColor);
    }

    public void newLabel(String text, int size, int x, int y, int width, int height){
        // Напис "Insert the flash card into your laptop"
        JLabel label = new JLabel(text);
        label.setFont(new Font("Arial", Font.BOLD, size));
        label.setBounds(x, y, width, height); // Встановлюємо розміщення та розмір
напису
        frame.add(label); // Додаємо напис до вікна
    }

    public void dropdownCheckbox (String nameOfDropBox, int x, int y, int width, int
height, JComboBox<String> comboBox){
//       90, 100, 200, 30
        JCheckBox dropdownCheckbox = new JCheckBox(nameOfDropBox);
        dropdownCheckbox.setBounds(x, y, width, height);
//      JComboBox<String> comboBox = new JComboBox<>(new String[]{"Option 1",
"Option 2", "Option 3"});
        comboBox.setBounds(x, 150, 100, 30);
        comboBox.setEnabled(false);

        // Додавання компонентів до вікна
        frame.add(dropdownCheckbox);
        frame.add(comboBox);

        dropdownCheckbox.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent actionEvent) {
                comboBox.setEnabled(dropdownCheckbox.isSelected());
```

```
            // Запис вибраного значення у JSON файл
            System.out.println(nameOfDropBox + " " + dropdownCheckbox);
            try {
                HendlerOfFlashCard.writeToJsonFile(nameOfDropBox,
dropdownCheckbox.isSelected() ? comboBox.getSelectedItem().toString() : null);
            } catch (IOException e) {
                e.printStackTrace(); // або інша обробка винятку
            }           }
        });

        // Показати вікно
        frame.setVisible(true);
    }
}
```

Додаток Б

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Презентація

до кваліфікаційної роботи на здобуття освітнього ступеня бакалавра

на тему: «Розробка віддаленої безпечної децентралізованої системи зберігання та обробки даних »

Виконав: студент Теремко М. О.

Керівник: Гніденко М. П.

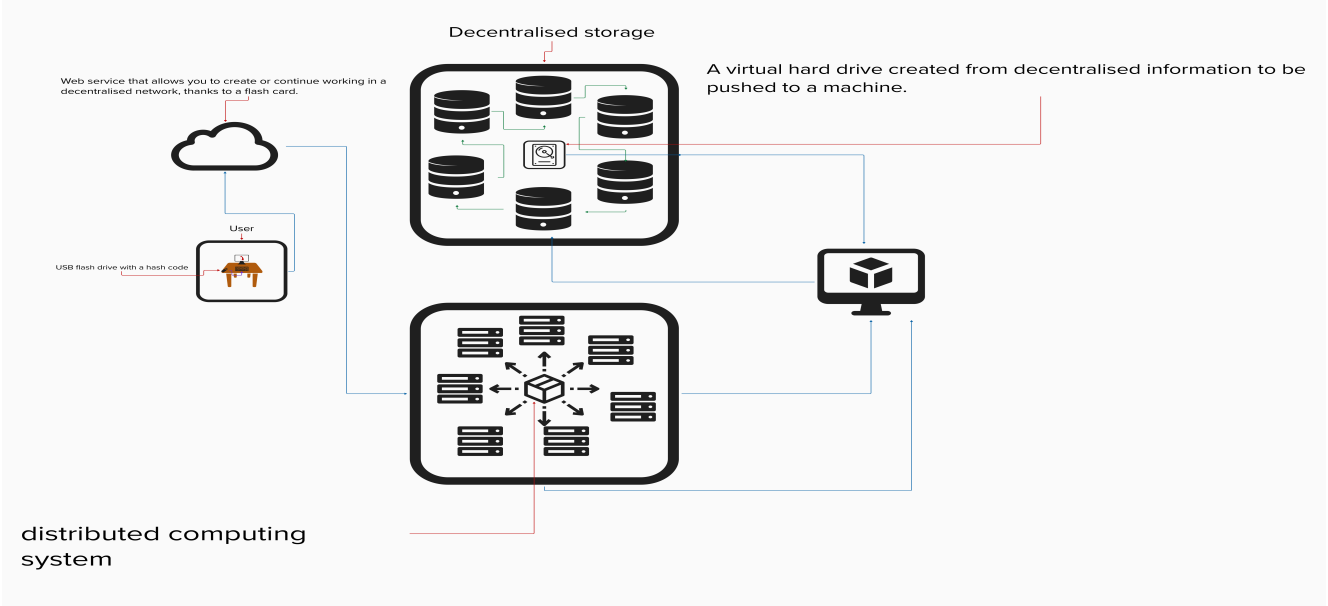
## Мета роботи, об`єкт та предмет дослідження

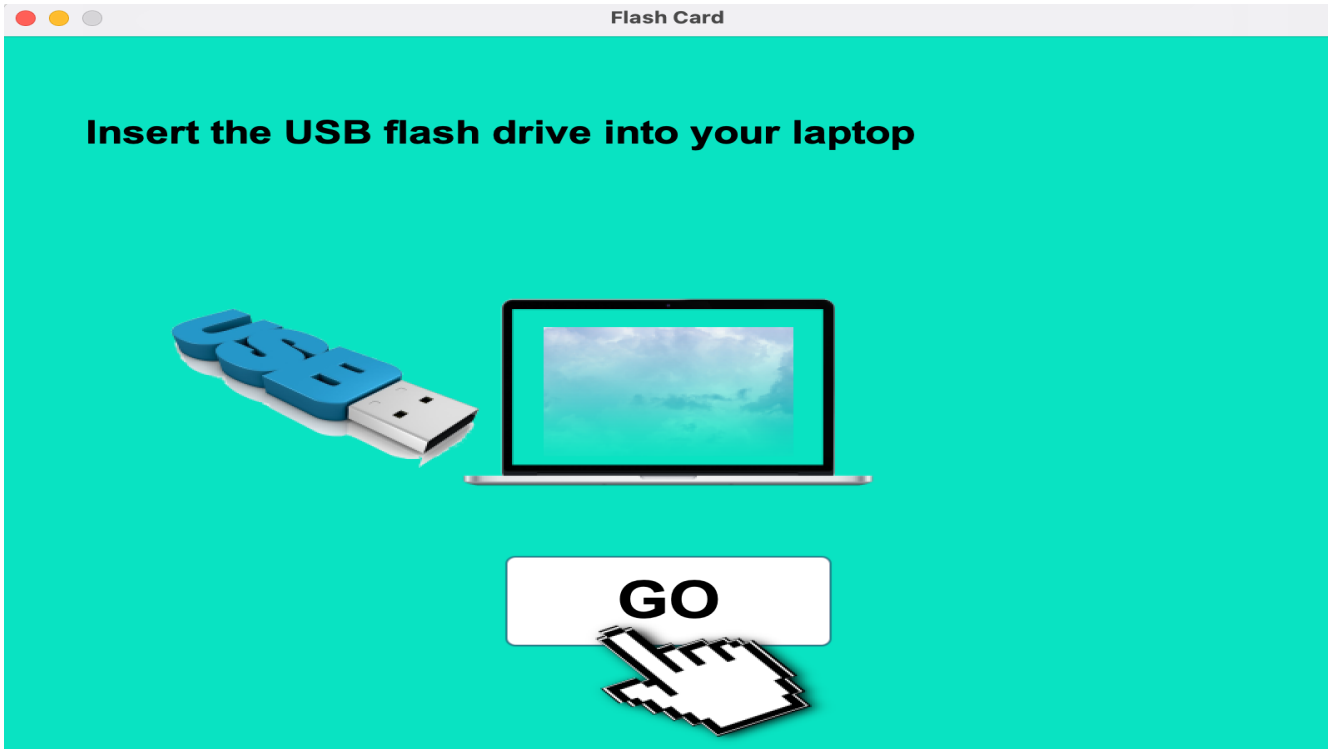
**Мета роботи – реалізація алгоритму з використанням методів дистрибутивності, фрагментації, шифрування, а також абстрактних особистих карт користувача, для безпечного зберігання обробки інформації, як даних. Головним аспектом виступає впровадження в майбутньому в хмарну інфраструктуру, для запобігання отримання не санкціонованого доступу, будь кого, крім власників інформації.**

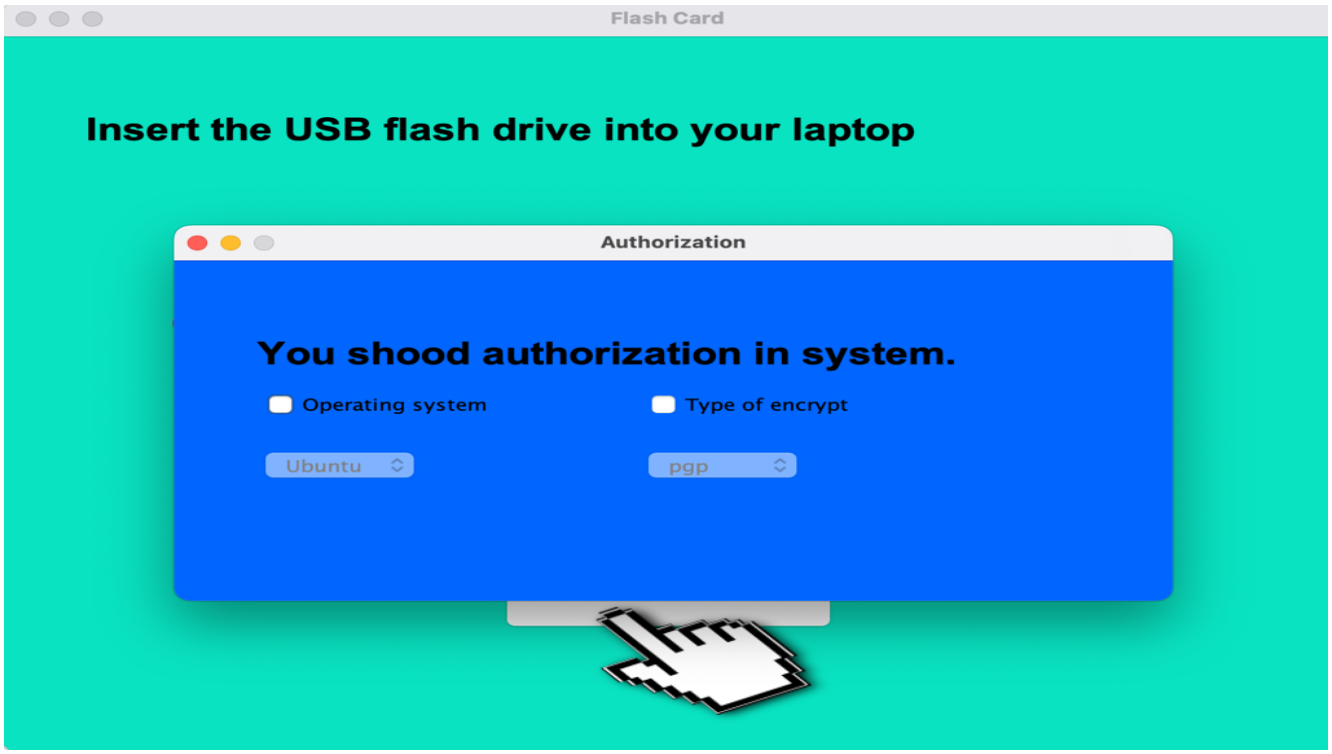
**Об`єкт дослідження – безпечне зберігання, маніпульювання, обробка даних користувачів інфраструктури хмарних платформ.**

**Предмет дослідження – методи запобігання несанкціонованого отримання доступу до даних користувача, які зберігаються в хмарі.**

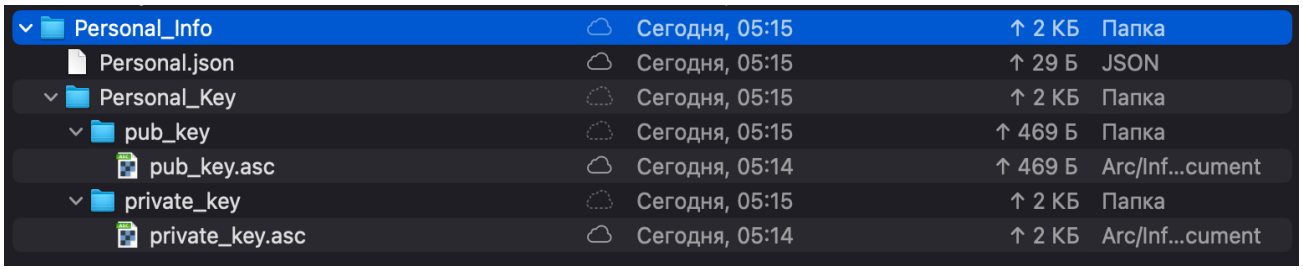
# Перехід до дистрибутивного хмарного середовища



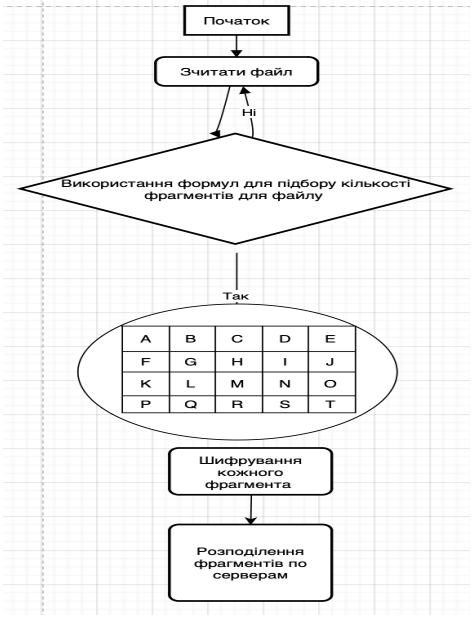






# Пересональна інформація користувача



# Алгоритм Фрагментації




# Нйважливіші властивості фрагментів

## All file fragments and their most important properties

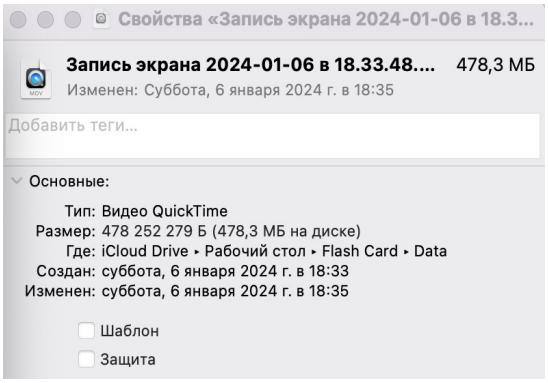
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight | Weight |
| Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key | Public Key |
| Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key | Private Key |
| Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase | Pass phrase |
| Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location | Location |
| Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate | Sequence of concatenate |
| Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption | Type of encryption |

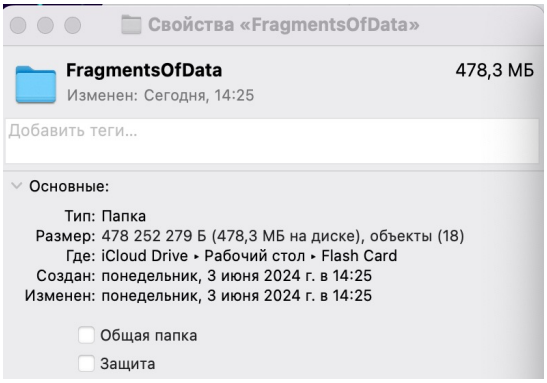
WEIGHT OF THE FILE

# Процес фрагментування

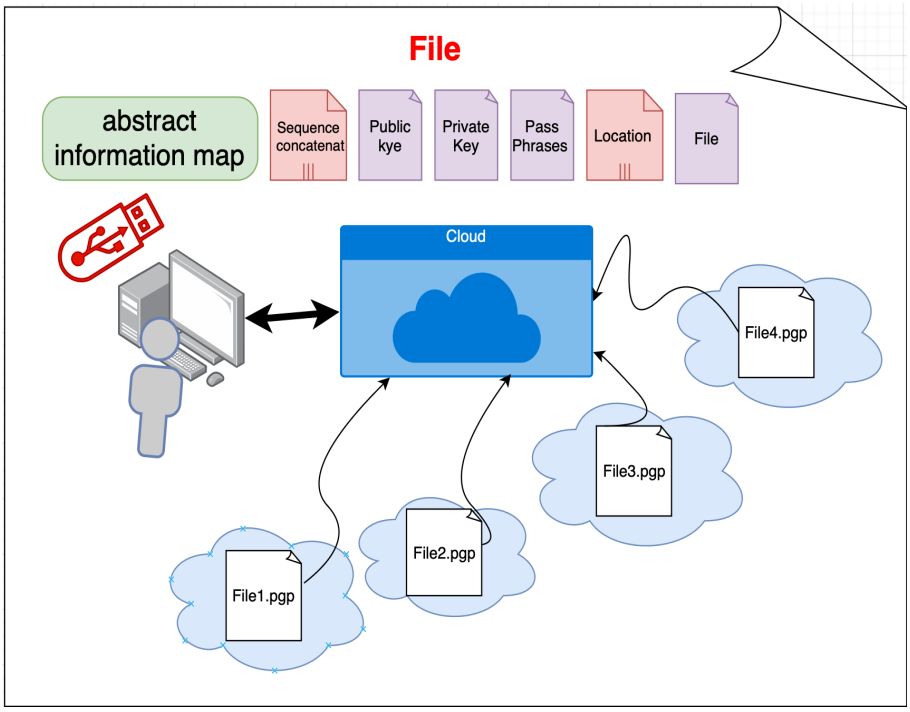
| | | | |
|---|---|---|---|
| FragmentsOfData | Сегодня, 14:25 | ↑ 478,3 МБ | Папка |
| part17.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part16.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part15.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part14.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part13.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part12.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part11.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part10.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part9.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part8.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part7.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part6.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part5.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part4.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part3.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part2.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| part1.txt | Сегодня, 14:33 | ↑ 28,1 МБ | Простой текст |
| Data | Сегодня, 14:25 | -- | Папка |
| Запись экрана 2024-01-06 в 18.33.48.mov | 6 янв. 2024 г., 18:35 | 478,3 МБ | Фильм QT |

# Врахування залишкових бітів, перевірка на цілісність після фрагментування



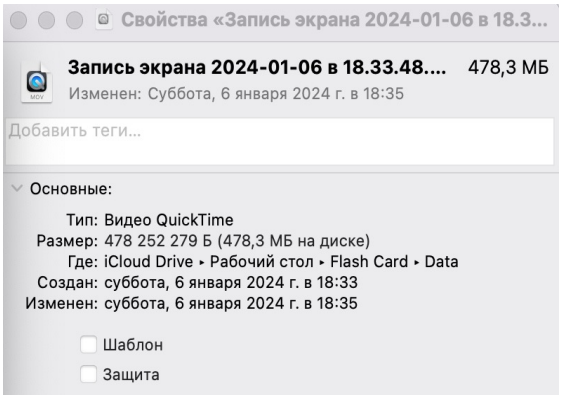
# Алгоритм Дефрагментації

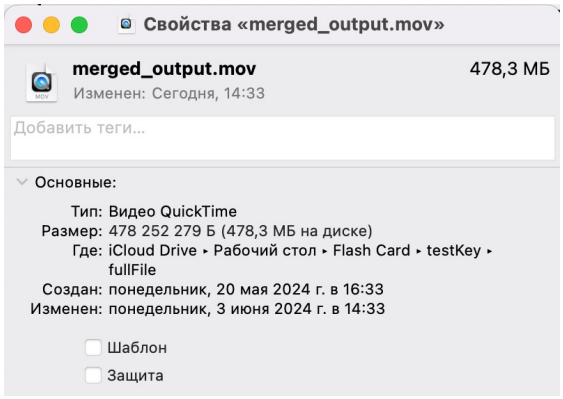



# Результати Дефрагментації

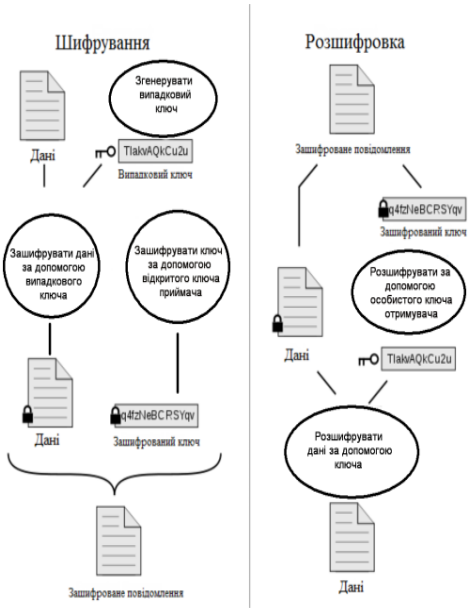
| | | | |
|---|---|---|---|
| fullFile | 14 мая 2024 г., 19:03 | -- | Папка |
| merged_output.mov | Сегодня, 14:33 | 478,3 МБ | Фильм QT |
| file | Сегодня, 02:23 | -- | Папка |
| Personal_Info | Сегодня, 05:15 | -- | Папка |
| FragmentsOfData | Сегодня, 14:25 | -- | Папка |
| part17.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part16.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part15.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part14.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part13.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part12.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part11.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part10.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part9.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part8.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part7.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part6.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part5.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part4.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part3.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part2.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| part1.txt | Сегодня, 14:33 | 28,1 МБ | Простой текст |
| Data | Сегодня, 14:25 | -- | Папка |
| Запись экрана 2024-01-06 в 18.33.48.mov | 6 янв. 2024 г., 18:35 | 478,3 МБ | Фильм QT |

# Порівняння кінцевих файлів




# Шифрування Даних



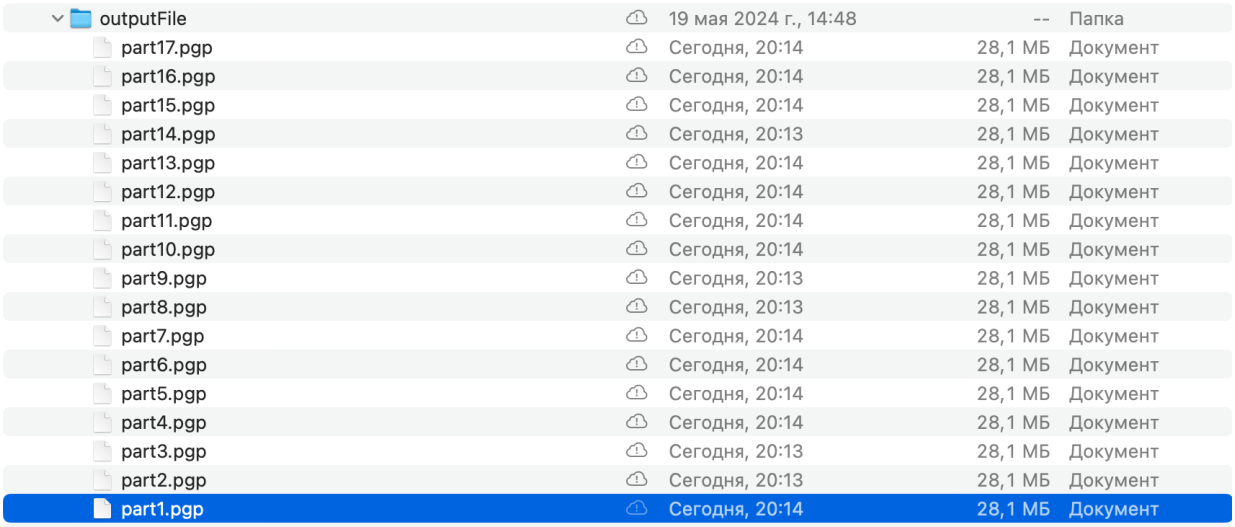
## Послідовна логована частина, шифрування даних

```
2024-06-03 20:13:52Public key for part8 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part8_publicKey.asc
2024-06-03 20:13:52Private key for part8 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part8_privateKey.asc
peoples delivers opinions recovery
2024-06-03 20:13:52Public key for part9 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part9_publicKey.asc
2024-06-03 20:13:52Private key for part9 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part9_privateKey.asc
cake evaluating talks alien
2024-06-03 20:13:52Public key for part14 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part14_publicKey.asc
2024-06-03 20:13:52Private key for part14 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part14_privateKey.asc
proven seem railroad aged
2024-06-03 20:13:52Public key for part2 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part2_publicKey.asc
2024-06-03 20:13:52Private key for part2 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part2_privateKey.asc
tier corpus coordination teach
2024-06-03 20:13:53Public key for part3 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part3_publicKey.asc
2024-06-03 20:13:53Private key for part3 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part3_privateKey.asc
holly annoying worldcat two
2024-06-03 20:13:53Public key for part15 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/pubKey/part15_publicKey.asc
2024-06-03 20:13:53Private key for part15 generated and saved to: /Users/nikitateremko/Desktop/Flash Card/testKey/file/privateKey/part15_privateKey.asc
```

## Розміщення приватних, та публічних ключів

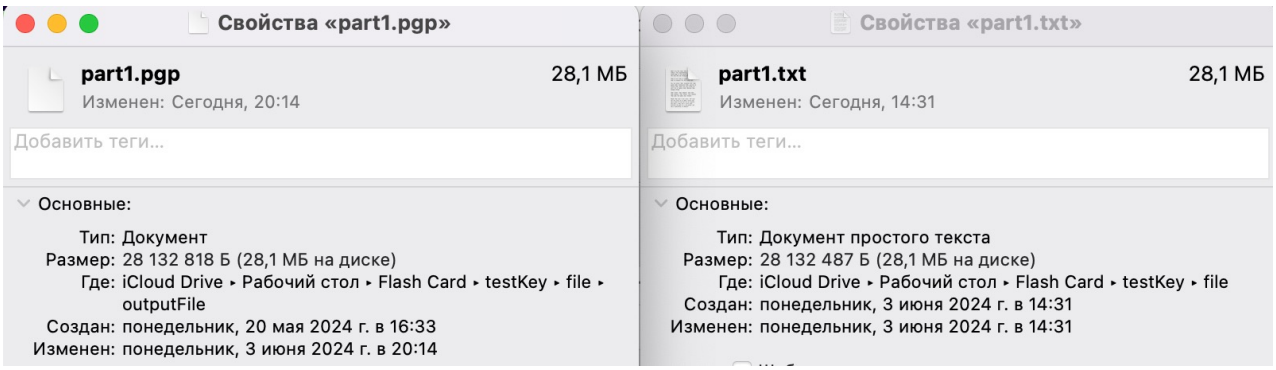
| Имя | Дата изменения | Размер | Тип |
|---|---|---|---|
| pubKey | 20 мая 2024 г., 23:01 | -- | Папка |
| part17_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part16_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part15_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part14_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part13_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part12_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part11_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part10_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part9_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part8_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part7_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part6_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part5_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part4_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part3_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part2_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| part1_publicKey.asc | Сегодня, 20:13 | 469 Б | Arc/Inf...cument |
| privateKey | 19 мая 2024 г., 13:23 | -- | Папка |
| part17_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part16_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part15_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part14_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part13_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part12_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part11_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part10_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part9_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part8_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part7_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part6_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part5_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part4_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part3_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part2_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |
| part1_privateKey.asc | Сегодня, 20:13 | 2 КБ | Arc/Inf...cument |

# Шифровані фрагменти



| | | | |
|---|---|---|---|
| outputFile | 19 мая 2024 г., 14:48 | -- | Папка |
| part17.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part16.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part15.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part14.pgp | Сегодня, 20:13 | 28,1 МБ | Документ |
| part13.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part12.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part11.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part10.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part9.pgp | Сегодня, 20:13 | 28,1 МБ | Документ |
| part8.pgp | Сегодня, 20:13 | 28,1 МБ | Документ |
| part7.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part6.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part5.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part4.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |
| part3.pgp | Сегодня, 20:13 | 28,1 МБ | Документ |
| part2.pgp | Сегодня, 20:13 | 28,1 МБ | Документ |
| part1.pgp | Сегодня, 20:14 | 28,1 МБ | Документ |

# Різниця між зашифрованою інформацією фрагментів, та базовою, котра була вхідною



Свойства «part1.pgp»

part1.pgp 28,1 МБ
Изменен: Сегодня, 20:14

Добавить теги...

Основные:

Тип: Документ
Размер: 28 132 818 Б (28,1 МБ на диске)
Где: iCloud Drive ▸ Рабочий стол ▸ Flash Card ▸ testKey ▸ file ▸ outputFile
Создан: понедельник, 20 мая 2024 г. в 16:33
Изменен: понедельник, 3 июня 2024 г. в 20:14

Свойства «part1.txt»

part1.txt 28,1 МБ
Изменен: Сегодня, 14:31

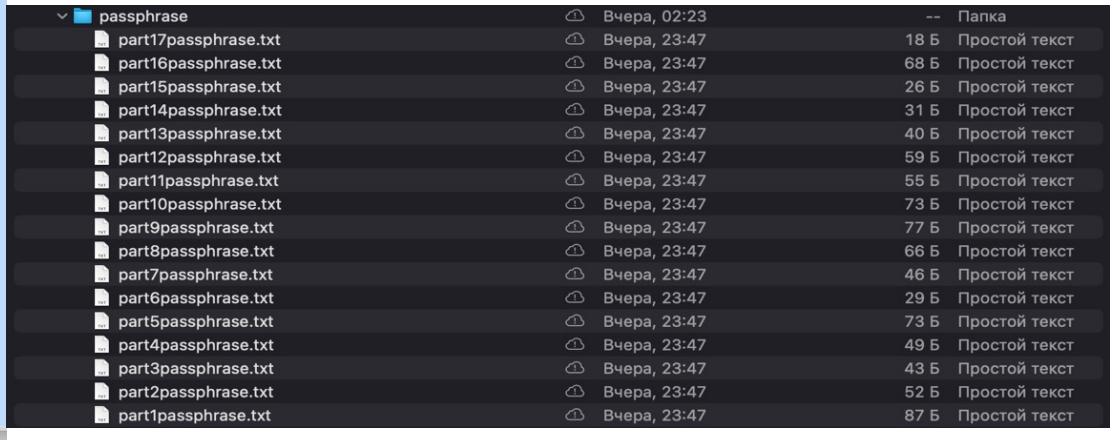
Добавить теги...

Основные:

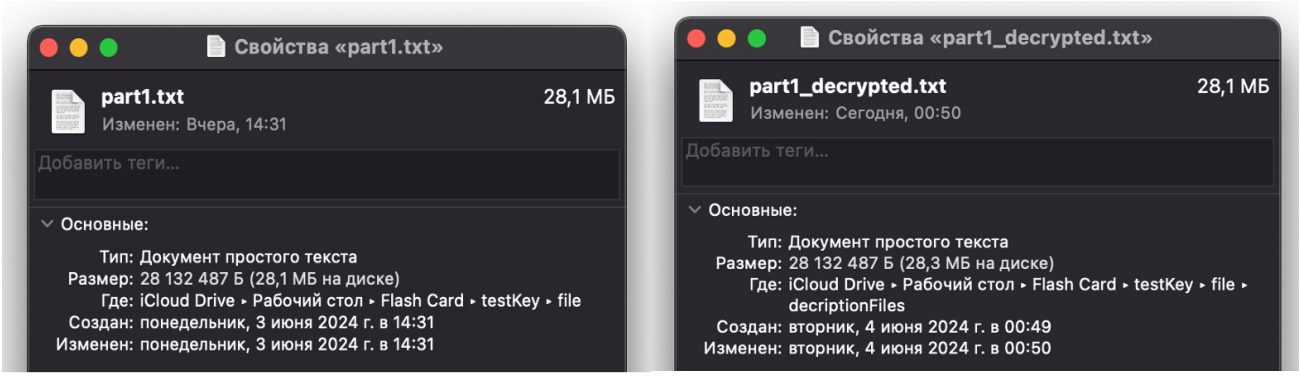
Тип: Документ простого текста
Размер: 28 132 487 Б (28,1 МБ на диске)
Где: iCloud Drive ▸ Рабочий стол ▸ Flash Card ▸ testKey ▸ file
Создан: понедельник, 3 июня 2024 г. в 14:31
Изменен: понедельник, 3 июня 2024 г. в 14:31

## Словник для формування пас-фраз


## Текстові документи, де зберігаються пас-фрази




## Порівняння базового фрагменту, та вже вже розшифрованого фрагменту



# Всі дешифровані фрагменти файлу

| | | | |
|---|---|---|---|
| decriptionFiles | 20 мая 2024 г., 22:56 | -- | Папка |
| part17_decrypted.txt | Сегодня, 00:51 | 28,1 МБ | Простой текст |
| part16_decrypted.txt | Сегодня, 00:51 | 28,1 МБ | Простой текст |
| part15_decrypted.txt | Сегодня, 00:48 | 28,1 МБ | Простой текст |
| part14_decrypted.txt | Сегодня, 00:47 | 28,1 МБ | Простой текст |
| part13_decrypted.txt | Сегодня, 00:54 | 28,1 МБ | Простой текст |
| part12_decrypted.txt | Сегодня, 00:53 | 28,1 МБ | Простой текст |
| part11_decrypted.txt | Сегодня, 00:57 | 28,1 МБ | Простой текст |
| part10_decrypted.txt | Сегодня, 00:58 | 28,1 МБ | Простой текст |
| part9_decrypted.txt | Сегодня, 01:01 | 28,1 МБ | Простой текст |
| part8_decrypted.txt | Сегодня, 01:00 | 28,1 МБ | Простой текст |
| part7_decrypted.txt | Сегодня, 00:56 | 28,1 МБ | Простой текст |
| part6_decrypted.txt | Сегодня, 00:59 | 28,1 МБ | Простой текст |
| part5_decrypted.txt | Сегодня, 00:55 | 28,1 МБ | Простой текст |
| part4_decrypted.txt | Сегодня, 00:52 | 28,1 МБ | Простой текст |
| part3_decrypted.txt | Сегодня, 00:49 | 28,1 МБ | Простой текст |
| part2_decrypted.txt | Сегодня, 00:46 | 28,1 МБ | Простой текст |
| part1_decrypted.txt | Сегодня, 00:50 | 28,1 МБ | Простой текст |