

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

(повне найменування назва факультету)

кібербезпеки

(повна назва кафедри)

Дипломна робота

РОЗРОБКА СИСТЕМИ КРИПТОГРАФІЧНОГО ЗАХИСТУ

ІНФОРМАЦІЇ У ХМАРНОМУ СХОВИЩІ

Виконав: студент групи ПМК-41с
спеціальності

125 «Кібербезпеки»

(шифр і назва спеціальності)



Maxwell

Левашов М. О.

(підпис)

(прізвище та ініціали)

Керівник

(підпис)

Трушевський В.М.

(прізвище та ініціали)

Рецензент

(підпис)

доц. Ковальська І.В.

(прізвище та ініціали)

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет Прикладної математики та інформатики

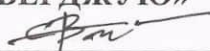
Кафедра Кібербезпеки

Спеціальність 125 «Кібербезпека»

(шифр і назва)

«ЗАТВЕРДЖУЮ»

Завідувач кафедри



"31" серпня 2022 року

ЗАВДАННЯ

НА ДИПЛОМНУ У РОБОТУ СТУДЕНТА

Левашов Максим Олегович

(прізвище, ім'я, по батькові)

1. Тема роботи Phishing. Розробка програм атаки та захисту.

керівник роботи доц.Трушевський В.М.

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

затверджені Вченою радою факультету від "13" вересня 2022 року № 15

2. Строк подання студентом роботи 13.06.2023р.

3. Вихідні дані до роботи

4. Зміст дипломної роботи (перелік питань, які потрібно розробити)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 31 серпня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів дипломної роботи	Термін виконання	Примітки
1	Уточнення постановки завдання	21.03.2023	
2	Аналіз літератури	28.03.2023	
3	Обґрунтування вибору рішень	31.03.2023	
4	Збір даних	07.04.2023	
5	Теоретичні відомості	18.04.2023	
6	Оформлення та запуск веб-додатку	20.04.2023	
7	Побудова системи хмарного сховища	01.05.2023	
9	Оформлення презентацій	06.06.2023	
10	Отримання рецензій	10.06.2023	
11	Подання роботи на кафедру	12.06.2023	
12	Захист в ЕК	15.06.2023	

Студент

(підпис)

Левашов М.О.
(ініціали, прізвище)

Керівник роботи

(підпис)

Трушевський В.М.
(ініціали, прізвище)

РЕФЕРАТ

Пояснювальна записка дипломного проекту складається зі вступу, шести розділів, що містять 12 рисунків, висновків та списку використаних джерел з 2 найменувань. Загальний обсяг роботи становить 49 сторінок.

Об'єкт дослідження: системи криптографічного захисту інформації, хмарні сховища, алгоритми шифрування, методи аутентифікації та надійні способи передачі даних, які можуть бути застосовані для забезпечення конфіденційності, цілісності та доступності даних у хмарному середовищі.

Метою роботи даної дипломної роботи є вивчення існуючих механізмів безпечного зберігання даних користувача у хмарних середовищах, дослідження їх протоколів безпеки та алгоритмів шифрування, розуміння переваг та обмежень кожної системи, розробка конкурентно здатної системи розподіленого зберігання даних, враховуючи криптографічні засоби захисту інформації.

Галузь застосування. Розробка систем криптографічного захисту важлива для інформаційної безпеки в хмарних технологіях, включаючи забезпечення безпеки зберігання та передачі даних, аутентифікацію, захист від несанкціонованого доступу та зловмисного використання. Це може бути використано в широкому спектрі областей, включаючи банківську справу, електронну комерцію, персональному використанні, та багато інших.

Ключові слова: Cloud Storage, Cloud Computing, Key Management in Cloud, Data Encryption , Information Security, Secure Data Transmission, Cryptography Algorithms.

ABSTRACT

The explanatory note of the diploma project consists of an introduction, six chapters containing 12 figures, conclusions, and a list of 12 references used. The total volume of the work is 49 pages.

Object of research: cryptographic information security systems, cloud storage, encryption algorithms, authentication methods and reliable data transmission methods that can be used to ensure the confidentiality, integrity and availability of data in the cloud environment.

The purpose of the work is to study existing mechanisms for securely storing user data in cloud environments, investigate their security protocols and encryption algorithms, understand the advantages and limitations of each system, and develop a competitively priced distributed data storage system, taking into account cryptographic means of information protection.

Field of application. The development of cryptographic protection systems is important for information security in cloud technologies, including ensuring the security of data storage and transmission, authentication, protection against unauthorized access and misuse. It can be used in a wide range of areas, including banking, e-commerce, personal use, and many others.

Keywords: Cloud Storage, Cloud Computing, Key Management in Cloud, Data Encryption, Information Security, Secure Data Transmission, Cryptography Algorithms.

ЗМІСТ

ВСТУП.....	8
РОЗДІЛ 1: ХМАРНІ ТЕХНОЛОГІЇ: БЕЗПЕКА, ВРАЗЛИВОСТІ.....	10
1.1 Огляд хмарних технологій.....	10
1.2 Проблеми безпеки в хмарних технологій.....	10
1.3 Заходи безпеки в хмарі.....	12
1.4 Висновок.....	13
РОЗДІЛ 2: МЕТОДИ ШИФРУВАННЯ	13
2.1 Симетричне шифрування.....	13
2.1.1 Advanced Encryption Standard (AES).....	14
2.2 Асиметричне шифрування.....	16
2.4 Порівняльний аналіз методів шифрування.....	17
2.5 Висновки.....	17
РОЗДІЛ 3: БЕЗПЕКА ДАНИХ У ХМАРНИХ СЕРВІСАХ.....	18
3.1 Захист передачі даних: Протоколи та практики.....	18
3.2 Архітектура хмарних сховищ.....	18
3.2.1 Централізоване сховище.....	18
3.2.2 Децентралізоване сховище.....	19
3.2.3 Клієнт-серверне сховище.....	19
3.2.4 Пірінгове (P2P) сховище.....	20
3.3 Резервне копіювання та відновлення.....	21
3.4 Заходи безпеки в хмарних сховищах.....	21
3.5 Роль шифрування в захисті даних.....	21
РОЗДІЛ 4: ТЕМАТИЧНІ ДОСЛІДЖЕННЯ ТА РЕАЛІЗАЦІЇ.....	22
4.1 Приклад 1: Шифрування та зберігання даних на Google Диску.....	22
4.2 Приклад 2: Шифрування та зберігання даних на Storj.io.....	22
4.3 Приклад 3: Шифрування та зберігання даних на Dropbox.....	23
4.4 Порівняння хмарних сховищ.....	25
4.5 Висновки.....	25
РОЗДІЛ 5: ПОСТАНОВКА ЗАДАЧІ ПРАКТИЧНОЇ ЧАСТИНИ.....	25
РОЗДІЛ 6: ПРОГРАМНА РЕАЛІЗАЦІЯ.....	26
6.1 Створення веб-застосунку.....	26
6.1.1 Вибір інструментів для розробки сайту.....	27

6.1.2 Архітектура та логіка відображення сайту.....	28
6.1.3. Маршрутизація у Flask.....	30
6.1.4 Відображення файлів у веб-додатку.....	31
6.2 Створення бази даних.....	32
6.3 Система авторизації користувачів.....	34
6.3.1 Реєстрація.....	35
6.3.2 Вхід у систему.....	36
6.3.3 Безпека.....	38
6.4 Шифрування та зберігання файлів.....	38
6.4.1 Приватний та публічний ключ.....	38
6.4.2 Вузли.....	39
6.4.3 Завантаження файлів на сервер.....	39
ВИСНОВКИ.....	44
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	46

ВСТУП

В епоху інформаційних технологій, коли щосекунди генерується і передається величезна кількість даних, важливість захищених систем зберігання і передачі даних не викликає сумнівів. Стрімка оцифровка процесів та інформації, експоненціальне зростання обсягів даних, що генеруються, а також необхідність доступу до них будь-де і будь-коли призвели до підвищення попиту на надійні та безпечні хмарні сервіси зберігання даних. Однак безпека і цілісність даних, що зберігаються на хмарних платформах, є нагальною проблемою, яка привертає увагу як дослідників, так і професіоналів галузі. Незважаючи на значний прогрес, складність і різноманітність загроз для хмарних сховищ даних продовжують розвиватися, створюючи значні виклики для галузі.

Актуальність цього питання підкреслюється численними гучними витоками даних за останнє десятиліття, коли такі корпорації, як Facebook, Yahoo та Equifax, стали жертвами кіберзагроз, що призвело до компрометації мільярдів даних користувачів. Ці інциденти не лише завдають компаніям фінансових збитків, але й призводять до значної шкоди репутації та потенційно руйнівних особистих наслідків для людей, чиї дані було скомпрометовано.

Більше того, дедалі ширше використання хмарних сервісів у всіх секторах, державному, бізнесі, охороні здоров'я і навіть особистому користуванні - означає, що під загрозою опиняється широкий спектр конфіденційної інформації. Така ситуація вимагає значного покращення безпеки хмарних сервісів зберігання даних, що робить цю сферу не тільки актуальною, але й критично важливою для вивчення та розвитку.

Метою цієї дипломної роботи є глибоке вивчення механізмів існуючих хмарних систем зберігання даних, дослідження їхніх протоколів безпеки та алгоритмів шифрування, розуміння переваг та обмежень кожної системи, а також пошук потенційних вдосконалень

Поштовхом для зосередження уваги на аспекті шифрування є його ключова роль у забезпеченні безпеки даних. Шифрування є потужним інструментом захисту конфіденційної інформації. Вивчаючи різні методи шифрування, їх застосування в сучасних хмарних системах зберігання даних та їх потенційні слабкі сторони, дослідження буде спрямоване на розробку хмарної системи зберігання даних, яка ефективно реалізує надійні методи шифрування.

Об'єкт дослідження - хмарні системи зберігання даних, таких як Google Drive, Dropbox та Storj.io, дослідження виявить їхні сильні та слабкі сторони. Буде зроблена спроба використати ці сильні сторони та усунути ці недоліки при створенні нової хмарної системи зберігання даних.

Таким чином, дипломна робота має на меті поєднати теоретичні дослідження з практичним застосуванням.

Розділ 1: Хмарні технології: Безпека, вразливості

Хмарні технології з їхнім спільним пулом конфігурованих обчислювальних ресурсів революціонізували сучасні IT-інфраструктури, дозволивши компаніям і приватним особам отримувати доступ до даних і зберігати їх віддалено. Однак разом з цим розвитком з'явилася проблема забезпечення безпеки та конфіденційності даних, зважаючи на постійне зростання кількості кіберзагроз та вразливостей.

1.1 Огляд хмарних технологій

Хмарні технології - це використання віддалених серверів, розміщених в Інтернеті, для зберігання, управління та обробки даних, на відміну від використання локальних серверів або персональних комп'ютерів. Цю технологію можна розділити на різні моделі надання послуг, включаючи інфраструктуру як послугу (IaaS), платформу як послугу (PaaS) і програмне забезпечення як послугу (SaaS).

Хмарні технології виводять управління безпекою даних на новий рівень складності, головним чином через розподілений характер зберігання та обробки даних. Хоча вона пропонує масштабованість, гнучкість та економічно ефективні переваги, вона також створює ризики, такі як несанкціонований доступ до даних, витік або втрата даних. Ці загрози безпеці вимагають впровадження надійного шифрування та криптографічних методів.

1.2 Проблеми безпеки в хмарних технологій

Хоча хмарні технології пропонують безпрецедентну зручність, масштабованість та економію коштів, вони також несуть з собою унікальний набір викликів. Серед них безпека залишається однією з найактуальніших проблем, яка

часто слугує бар'єром на шляху до впровадження хмарних технологій для багатьох підприємств.

У хмарних технологій дані зазвичай зберігаються у спільному середовищі, що збільшує ризик несанкціонованого доступу, порушень і втрати даних. Крім того, сама природа хмарних обчислень, що базується на Інтернеті, наражає їх на широкий спектр кіберзагроз, включаючи хакерство, фішинг та розподілені атаки на відмову в обслуговуванні (DDoS).

Зокрема, у сфері безпеки хмарних технологій часто виникають такі проблеми, що викликають занепокоєння:

1. **Витік даних:** У хмарному середовищі величезні обсяги даних зберігаються в одному місці. Хоча це забезпечує ефективність, це також є привабливою мішенню для кіберзлочинців. Якщо станеться витік, конфіденційні дані клієнтів, інтелектуальна власність або комерційна таємниця можуть бути викриті.

2. **Втрата даних:** Дані, що зберігаються в хмарі, можуть бути втрачені з різних причин, таких як катастрофічний збій, випадкове видалення постачальником хмарних послуг або фізична катастрофа, наприклад, пожежа чи землетрус.

3. **Недостатнє управління ідентифікацією, обліковими даними, доступом та ключами:** Несанкціонований доступ до даних і послуг часто є наслідком відсутності надійного управління ідентифікацією та доступом. Хмарні сервіси повинні впроваджувати надійну аутентифікацію користувачів, механізми контролю доступу та безпечні процеси управління ключами.

4. **Незахищені API:** Доступ до хмарних сервісів часто здійснюється через API (інтерфейси прикладного програмування), які, якщо вони розроблені та налаштовані ненадійно, можуть бути потенційно вразливими.

5. **Захоплення облікового запису:** Захоплення облікового запису - це атака, коли зловмисник отримує доступ до хмарного облікового запису користувача, часто за допомогою фішингу, вразливостей програмного забезпечення або використання викрадених облікових даних для входу в систему. Потрапивши всередину, зловмисник може маніпулювати даними, підслуховувати дії та перенаправляти транзакції.

6. **Атаки на відмову в обслуговуванні (DoS):** DoS-атака має на меті зробити комп'ютер або мережевий ресурс недоступним для його користувачів, тимчасово або на невизначений час порушуючи роботу сервісів хоста, підключеного до Інтернету.

1.3 Заходи безпеки в хмарі

Для того щоб забезпечити надійну безпеку у хмарі, потрібно реалізовувати наступні пункти:

- Шифрування даних.
- Захищені протоколи передачі даних.
- Управління доступом і авторизація.
- Резервне копіювання і відновлення даних.
- Фізична безпека центрів обробки даних.
- Антивірусний захист.
- Захист від DDoS-атак.
- Логування та аудит безпеки.

Це детальніше обговорено у наступних розділах роботи.

1.4 Висновок

Хоча хмарні технології продовжують розвиватися, вони несуть з собою ряд вразливостей. Однак ці вразливості можна зменшити за допомогою надійної стратегії безпеки, яка включає шифрування, криптографію, безпечні методи передачі даних і суворі політики управління доступом. Актуальність тематики даної роботи базується на нових викликах сучасного світу з домінуванням дистанційного навчання та віддаленого працевлаштування, де все більшої важливості набуває безпека зберігання даних у хмарі.

Розділ 2: Методи шифрування

2.1 Симетричне шифрування

Симетричне шифрування, яке часто називають шифруванням з секретним ключем, - це метод, коли один і той самий ключ використовується для шифрування і розшифрування даних. Цим ключем необхідно безпечно обмінятися між сторонами, перш ніж відбудеться передача даних. Процес є швидким і обчислювально ефективним, що робить його придатним для кодування великих обсягів даних. Такі алгоритми, як Data Encryption Standard (DES), Advanced Encryption Standard (AES) і Blowfish, є прикладами симетричних методів шифрування.

У хмарних сховищах симетричне шифрування зазвичай використовується для захисту даних у стані спокою. Наприклад, після завантаження файлу на хмарний сервіс дані можуть бути зашифровані за допомогою симетричного ключа, що гарантує, що навіть якщо фізичний носій інформації буде скомпрометований, дані залишаться нечитабельними без ключа. Зокрема, Amazon S3, популярний хмарний сервіс зберігання даних, використовує AES-256 для шифрування на стороні сервера, щоб захистити дані користувачів.

2.1.1 Advanced Encryption Standard (AES)

У сфері симетричних методів шифрування AES (Advanced Encryption Standard) вважається одним з найбезпечніших і найефективніших алгоритмів. AES підтримує різні режими роботи, кожен з яких пропонує унікальні можливості та компроміси. Два найпоширеніші режими - це AES-256 CBC (Cipher Block Chaining) і AES-256 GCM (Galois/Counter Mode). Ці режими відрізняються за своїми властивостями і придатністю для конкретних випадків використання. Давайте порівняємо їх більш детально:

AES-256 CBC (Cipher Block Chaining):

- **Робота:** AES-256 CBC працює з блоками фіксованого розміру (зазвичай 128 біт) і вимагає вектора ініціалізації (IV).
- **Процес шифрування:** Кожен блок перед зашифруванням піддається операції XOR з попереднім блоком зашифрованого тексту, що гарантує, що один і той самий блок відкритого тексту не призведе до отримання одного і того ж блоку зашифрованого тексту.
- **Заповнення:** Вимагає додавання останнього блоку відкритого тексту, якщо він не має повного розміру.
- **Безпека:** Забезпечує конфіденційність, але не пропонує перевірку цілісності або автентичності. При неправильній реалізації вразливий до атак padding oracle.
- **Варіанти використання:** AES-256 CBC зазвичай використовується в сценаріях, де конфіденційність є першочерговим завданням, наприклад, для шифрування дисків або застарілих систем, які вимагають сумісності зі старими алгоритмами шифрування.

AES-256 GCM (Galois/Counter Mode):

- **Робота:** AES-256 GCM поєднує в собі блоковий шифр AES з режимом роботи лічильника і механізмом автентифікації, відомим як GMAC (код автентифікації повідомлень Галуа).
- **Процес шифрування:** GCM працює з потоком даних, а не з фіксованими блоками. Він використовує унікальне значення лічильника для кожного блоку, яке потім шифрується і поєднується з відкритим текстом для отримання зашифрованого тексту.
- **Автентифікація та цілісність:** GCM забезпечує не тільки конфіденційність, але й цілісність та автентичність. Механізм автентифікації GMAC гарантує, що будь-яка модифікація або підробка зашифрованого тексту буде виявлена.
- **Нонсе:** Вимагає унікального нонсе (номер, який використовується один раз) для кожної операції шифрування, щоб забезпечити унікальність значень лічильника.
- **Продуктивність:** AES-256 GCM зазвичай працює швидше, ніж AES-256 CBC, завдяки можливостям розпаралелювання.
- **Варіанти використання:** AES-256 GCM зазвичай використовується в мережевих комунікаціях, де конфіденційність і цілісність мають вирішальне значення. Він підходить для захищених протоколів зв'язку, таких як TLS (Transport Layer Security).

Таким чином, хоча AES-256 CBC і AES-256 GCM засновані на одному і тому ж блочному шифрі AES-256, вони відрізняються за принципом роботи, властивостями захисту і сферами використання. AES-256 CBC забезпечує конфіденційність, але не має вбудованої перевірки цілісності, що робить його придатним для сценаріїв, де потрібна сумісність зі старими системами. З іншого боку, AES-256 GCM забезпечує і конфіденційність, і цілісність, що робить його добре придатним для захищених мережевих комунікацій, де цілісність даних має вирішальне значення.

2.2 Асиметричне шифрування

Асиметричне шифрування, також відоме як шифрування з відкритим ключем, використовує два різні, але математично пов'язані ключі: один відкритий (для шифрування) і один закритий (для розшифрування). Цей метод вирішує проблему обміну ключами при симетричному шифруванні. Якщо хтось хоче надіслати зашифровані дані, він може використати відкритий ключ одержувача, який знаходиться у відкритому доступі. Розшифрувати ці дані можна лише за допомогою відповідного закритого ключа, який одержувач тримає в таємниці. До відомих алгоритмів асиметричного шифрування належать RSA (Rivest-Shamir-Adleman), DSA (алгоритм цифрового підпису) та ECC (криптографія еліптичних кривих).

У хмарних сховищах асиметричне шифрування часто використовується під час передачі даних, які часто називають даними в дорозі. Це гарантує, що дані, які передаються від клієнта до хмарного сервера, залишаються захищеними від атак прослуховування. Яскравим прикладом є протокол HTTPS, який використовується під час передачі даних до хмари та з неї, і який використовує асиметричну систему шифрування, зазвичай RSA або ECC, для створення безпечного каналу.

2.3 Гібридне шифрування

Гібридне шифрування - це метод, який поєднує переваги як симетричного, так і асиметричного шифрування. Він використовує швидкість і ефективність симетричного шифрування та безпечний обмін ключами асиметричного шифрування. Процес зазвичай передбачає використання асиметричного шифрування для безпечного обміну ключем симетричного шифрування, який потім використовується для шифрування фактичного повідомлення або даних.

Гібридне шифрування відіграє вирішальну роль у захисті даних як під час передачі, так і під час зберігання в хмарному сховищі. Процес починається, коли

клієнт ініціює підключення до хмарного сервера. Використовуючи асиметричний протокол шифрування, такий як RSA, клієнт і сервер безпечно обмінюються симетричним ключем. Потім цей симетричний ключ використовується для шифрування даних, що передаються, використовуючи швидкість і ефективність симетричного шифрування. Симетричний ключ також може бути використаний для шифрування даних у стані спокою, коли вони потрапляють на хмарний сервер, що додає додатковий рівень безпеки.

2.4 Порівняльний аналіз методів шифрування

Хоча кожен метод шифрування має свої переваги та можливості використання в хмарних сховищах, вибір часто залежить від конкретних вимог до даних та архітектури системи.

Симетричне шифрування є швидким і ефективним, ідеально підходить для шифрування великих обсягів даних. Однак обмін ключами створює ризик для безпеки, особливо в хмарному середовищі, де дані і користувачі географічно розподілені.

Асиметричне шифрування вирішує проблему обміну ключами, але є важчим з точки зору обчислень, що робить його повільнішим і менш придатним для великих обсягів даних. Однак його здатність забезпечувати безпечні, автентифіковані канали зв'язку робить його безцінним для захисту даних під час передачі.

Гібридне шифрування поєднує переваги обох методів, забезпечуючи безпечний обмін ключами та ефективне шифрування даних. Тому воно широко використовується в хмарних системах зберігання даних.

2.5 Висновки

AES-GCM у парі з RSA часто використовують великі та надійні компанії для шифрування даних, це свідчить про те що вони достатньо надійні, тому в якості методів шифрування, програму буде реалізовано саме з цією парою алгоритмів.

Розділ 3: Безпека даних у хмарних сервісах

3.1 Захист передачі даних: Протоколи та практики

Забезпечення безпеки даних під час передачі є наріжним каменем будь-якої надійної хмарної служби. Для забезпечення безпечного каналу передачі даних часто використовуються такі протоколи, як протокол захищених сокетів (SSL), протокол безпеки транспортного рівня (TLS) і HTTPS (HTTP Secure). Ці протоколи використовують шифрування, як правило мають, гібридний підхід, для захисту даних під час передачі, захищаючи їх від перехоплення або зміни.

Окрім використання захищених протоколів, для посилення безпеки під час передачі даних застосовуються додаткові методи, такі як білі списки IP-адрес (дозволяючи з'єднання лише з довірених IP-адрес), віртуальні приватні мережі (VPN) та наскрізне шифрування.

3.2 Архітектура хмарних сховищ

Сервіси хмарного зберігання даних можуть працювати з використанням декількох типів архітектурних моделей, кожна з яких пропонує унікальні переваги та потенційні вразливості:

3.2.1 Централізоване сховище

Централізоване хмарне сховище - це традиційний підхід, коли всі дані зберігаються та керуються на центральному сервері або кластері серверів. Ця модель зазвичай використовується в організаційних структурах, де даними централізовано керує спеціальний ІТ-відділ. Прикладом централізованої системи зберігання є внутрішній мережевий диск компанії, де всі файли організації зберігаються в одному місці.

Хоча така модель спрощує управління даними і забезпечує узгодженість контролю доступу та форматів даних, вона може створювати вузькі місця,

особливо при великих обсягах даних або великій кількості користувачів, які отримують доступ до них одночасно. Крім того, центральний сервер стає єдиною точкою вразливості, де збій або порушення безпеки може скомпрометувати всі дані, що зберігаються.

3.2.2 Децентралізоване сховище

Децентралізоване хмарне сховище - модель, яка все частіше використовується в сучасних хмарних архітектурах - розподіляє дані між численними одноранговими вузлами, як правило, в мережі на основі блокчейну. Кожен фрагмент даних розбивається на кілька частин, які розподіляються по мережі. Ця модель може підвищити безпеку даних, оскільки атака на один вузол не скомпрометує весь набір даних.

Прикладом децентралізованої системи зберігання є IPFS (InterPlanetary File System) - протокол, розроблений для створення постійного і децентралізованого способу зберігання та обміну файлами.

Однією з головних переваг децентралізованого сховища є підвищена доступність даних і відмовостійкість. Якщо один вузол виходить з ладу або піддається компрометації, дані все одно можна отримати з інших вузлів мережі. Однак це досягається за рахунок збільшення складності управління даними та потенційної затримки в отриманні даних через розподілену природу сховища.

3.2.3 Клієнт-серверне сховище

У моделі клієнт-серверного сховища сервер (або сервери) зберігає дані та керує ними, а клієнти (користувачі або програми) взаємодіють з цими даними за потреби. Сервер зберігає контроль над даними, керуючи доступом, застосовуючи протоколи безпеки та забезпечуючи узгодженість даних.

Класичним прикладом клієнт-серверної моделі зберігання даних є веб-сервіс електронної пошти, наприклад, Gmail. Сервери Google зберігають

електронні листи (дані), а користувачі (клієнти) взаємодіють зі своїми електронними листами через веб-інтерфейс.

Хоча ця модель спрощує управління даними, централізуючи їх на сервері, вона може створювати потенційні єдині точки відмови. Якщо сервер скомпрометований або виходить з ладу, всі клієнти втрачають доступ до своїх даних.

3.2.4 Пірінгове (P2P) сховище

У пірінгових (P2P) системах зберігання всі учасники мережі виступають і як клієнти, і як сервери. Це означає, що кожен учасник мережі може надавати простір для зберігання, ділитися ресурсами, а також обслуговувати або споживати вміст. Така архітектура дозволяє підвищити надмірність і надійність, оскільки дані можна зберігати в різних місцях і отримувати їх, навіть якщо деякі учасники мережі виходять з мережі.

BitTorrent, популярний протокол обміну файлами P2P, є гарним прикладом такої моделі зберігання. Файли діляться на частини і розподіляються між одноранговими користувачами. Коли одноранговий користувач хоче завантажити файл, він отримує фрагменти від кількох інших однорангових користувачів одночасно, розподіляючи навантаження і збільшуючи швидкість завантаження.

Хоча P2P-сховища можуть підвищити відмовостійкість даних і ефективність їх розподілу, вони також ускладнюють управління даними і створюють потенційні проблеми з безпекою. Оскільки всі учасники можуть надавати і споживати контент, управління контролем доступу і забезпечення цілісності даних на всіх вузлах може бути складним завданням.

3.3 Резервне копіювання та відновлення

Стратегії резервного копіювання та відновлення є невід'ємною частиною безпеки даних у хмарних сервісах. Регулярне резервне копіювання гарантує, що у

випадку втрати даних, чи то через випадкове видалення, збій обладнання або порушення безпеки, можна буде відновити останню копію даних.

Стратегії відновлення часто передбачають зберігання декількох резервних копій у географічно розподілених місцях для захисту від регіональних катастроф або перебоїв у роботі. Деякі хмарні сервіси також пропонують функцію версійності, яка зберігає попередні версії файлів і дозволяє користувачам повернутися до попереднього стану, забезпечуючи додатковий рівень захисту від пошкодження або випадкової модифікації даних.

3.4 Заходи безпеки в хмарних сховищах

На додаток до шифрування, безпеку в хмарних сховищах можна підвищити за допомогою численних стратегій. Такі технології, як системи виявлення вторгнень (IDS) і системи запобігання вторгненням (IPS), можуть виявити і запобігти потенційним загрозам. Крім того, регулярний аудит журналів активності може допомогти виявити незвичну або підозрілу поведінку.

3.5 Роль шифрування в захисті даних

Як обговорювалося в Розділі 3, шифрування має ключове значення для безпеки даних. Хоча в попередньому розділі розглядалися методи шифрування, важливо розуміти, що шифрування має бути правильно реалізоване на всіх етапах обробки даних - під час передачі (дані в дорозі), під час зберігання (дані в стані спокою) і навіть під час обробки (дані в процесі використання).

Самі ключі шифрування потребують надійного управління, що часто передбачає безпечне зберігання ключів, періодичну ротацію ключів і надійний контроль доступу. Хмарні сервіси часто пропонують послуги з управління ключами шифрування, щоб допомогти користувачам безпечно керувати своїми ключами шифрування.

Крім того, для конфіденційних даних такі методи, як токенізація, коли конфіденційні елементи даних замінюються не конфіденційними еквівалентами, або анонімізація, яка передбачає видалення ідентифікаційної інформації з даних, можуть додати ще один рівень захисту.

Розділ 4: Тематичні дослідження та реалізації

4.1 Приклад 1: Шифрування та зберігання даних на Google Діску

Google Диск - це багатофункціональна хмарна платформа для зберігання даних, що пропонує користувачам безліч можливостей для зберігання, обміну та спільної роботи над файлами. В основі його привабливості лежить безкоштовна інтеграція з іншими сервісами Google, такими як Документи, Таблиці та Слайди, що робить його ідеальним вибором для продуктивності та співпраці.

Якщо говорити про безпеку, то в Google Діску використовуються передові методи шифрування. Файли під час передачі захищені 256-бітовим шифруванням SSL/TLS, а в стані спокою дані захищені 128-бітовим шифруванням AES. Однак суттєвою суперечкою є те, що Google зберігає контроль над ключами шифрування. Хоча така схема не становить загрози для безпеки, вона означає, що Google технічно має можливість розшифрувати ваші дані. Тому користувачі, які прагнуть максимальної конфіденційності, можуть використовувати сторонні інструменти шифрування перед завантаженням конфіденційних даних.

4.2 Приклад 2: Шифрування та зберігання даних на Storj.io

Storj.io використовує інший підхід до зберігання даних. Використовуючи технологію блокчейн, він пропонує децентралізоване хмарне сховище, відходячи від традиційної моделі центрального сервера. Ваші дані зберігаються не в одному місці, а розкидані по глобальній мережі вузлів, що підвищує їх доступність та стійкість до втрати.

Модель шифрування Storj.io вирізняється тим, що її метод шифрування на стороні клієнта пропонує підвищену безпеку та конфіденційність. Перед тим, як покинути пристрій, ваші дані шифруються, і тільки ви маєте доступ до ключів, що робить ваші дані недоступними для Storj.io і будь-яких потенційних зловмисників. Така практика підвищує безпеку даних, але вона також усуває певні функції, такі як попередній перегляд або редагування файлів у хмарі.

Метод шифрування, який використовує Storj, - це AES-256-GCM, тип симетричного шифрування. Це означає, що для шифрування та розшифрування даних використовується один і той самий ключ. Метод AES-256-GCM вважається дуже безпечним і широко використовується для захисту конфіденційної інформації.

Однак важливою особливістю процесу шифрування Storj є те, що тільки користувачі мають доступ до своїх ключів шифрування. Це гарантує, що ніхто, включаючи Storj, не може отримати доступ до даних користувача без ключів шифрування, забезпечуючи додатковий рівень конфіденційності та безпеки даних користувачів.

На додаток до шифрування, Storj.io також практикує шардінг. Кожен файл розбивається на кілька фрагментів і розподіляється по різних вузлах, що ще більше підвищує стійкість і безпеку даних.

4.3 Приклад 3: Шифрування та зберігання даних на Dropbox

Dropbox - відомий гравець на ринку хмарних сховищ, який вирізняється зручним інтерфейсом і простотою використання. Він пропонує функції, подібні до Google Drive, такі як резервне копіювання, синхронізація та спільний доступ до файлів.

Dropbox забезпечує безпеку даних користувачів за допомогою 256-бітного шифрування AES для даних у стані спокою та SSL/TLS для даних під час передачі. Подібно до Google Drive, Dropbox зберігає контроль над ключами шифрування, а це означає, що вони можуть технічно розшифрувати ваші дані в

разі потреби. Для додаткової конфіденційності користувачі можуть використовувати сторонні інструменти шифрування перед завантаженням своїх файлів.

Унікальні переваги Dropbox включають можливість відновлення видалених файлів, детальну історію версій і сумісність з численними сторонніми додатками. Однак він не пропонує власного набору інструментів для підвищення продуктивності, як це робить Google Drive.

4.4 Порівняння хмарних сховищ

Порівняння Google Drive, Storj.io та Dropbox підкреслює відмінності в їхньому підході до хмарних сховищ. Google Диск і Dropbox - це централізовані сервіси, які зберігають дані на своїх серверах, причому ключовою відмінністю є набір інструментів для підвищення продуктивності Google Діску. Dropbox, з іншого боку, значною мірою покладається на сторонні інтеграції для аналогічної функціональності.

На противагу цьому, Storj.io працює за децентралізованою моделлю, розподіляючи дані користувачів по великій мережі вузлів. Такий підхід призводить до більшої стійкості даних і кращої конфіденційності завдяки шифруванню на стороні клієнта, але за рахунок певних зручностей, які пропонують централізовані сервіси.

Вибір між цими сервісами залежить від індивідуальних потреб і пріоритетів користувачів. Якщо пріоритетом є інтегровані інструменти для підвищення продуктивності та згуртована екосистема, то Google Диск буде найбільш відповідним. Якщо важливими є конфіденційність даних та надійна децентралізована модель зберігання, краще обрати Storj.io. Тим часом, Dropbox пропонує баланс між зручними функціями та сумісністю з численними сторонніми додатками.

4.5 Висновки

Проаналізувавши переваги та недоліки вищеперелічених хмарних сховищ, було вирішено розробити децентралізоване хмарне середовище. Але щоб зробити програму більш компактною, буде уникнено спробу розробити окремий клієнт для серверу, тому, приватні ключі будуть зберігатися на самому сервері, але на відміну від Google Drive, сам сервер не знатиме як їх розшифровувати, що в свою чергу, забезпечує більшу надійність на конфіденційність.

Розділ 5: Постановка задачі практичної частини

Пошук більш безпечної, приватної та надійної системи зберігання даних спонукає до розробки прототипу децентралізованої хмарної платформи для зберігання даних. Платформа, має на меті імітувати функціональність реальних хмарних систем, але з урахуванням обмежень наявних ресурсів. Система, яку ми прагнемо створити, слугуватиме мікросвітом повноцінної децентралізованої мережі хмарних сховищ з локалізованими "вузлами", представленими окремими папками.

Мета - побудувати систему, в якій кожен файл буде фрагментований на кілька частин, кожна з яких зберігатиметься на окремому вузлі, імітуючи таким чином децентралізовану схему зберігання даних. Для подальшої імітації стійкості реальних систем, ці частини також матимуть дублікати, розподілені по всій системі. Така надмірність дозволяє системі підтримувати доступність даних навіть тоді, коли деякі вузли імітують офлайн-статус, відображаючи реальні сценарії виходу вузлів з ладу.

Інтерфейс користувача буде представляти собою веб-додаток, реалізований за допомогою бібліотеки Flask на мові Python. Цей додаток запускатиме локальний сервер, надаючи інтерфейс, знайомий користувачам звичайних хмарних сервісів зберігання даних, таких як Google Drive.

Додаток матиме форму авторизації користувача, що дозволить безпечний та персоналізований доступ до файлів кожного користувача. Використовуючи

Bootstrap, ми розробимо форму реєстрації та входу з надійною перевіркою для захисту від SQL-ін'єкцій та інших поширених вразливостей безпеки.

Після входу користувачі будуть мати змогу завантажувати файли в систему. Ці файли будуть фрагментовані, зашифровані та розподілені між доступними вузлами. Користувач також матиме можливість вивантажувати свої файли. Під вивантаження файлу, система збиратиме його з частин, розшифруватиме і повертатиме користувачеві.

По суті, запропонована система буде імітувати повномасштабний, децентралізований сервіс хмарного зберігання даних, але в більш керованому, локальному налаштуванні. Вона має на меті забезпечити розуміння внутрішньої роботи таких систем, їхніх переваг з точки зору безпеки, таким як фрагментація файлів і шифрування, та стійкості до збоїв у роботі вузлів.

Розділ 6: Програмна реалізація.

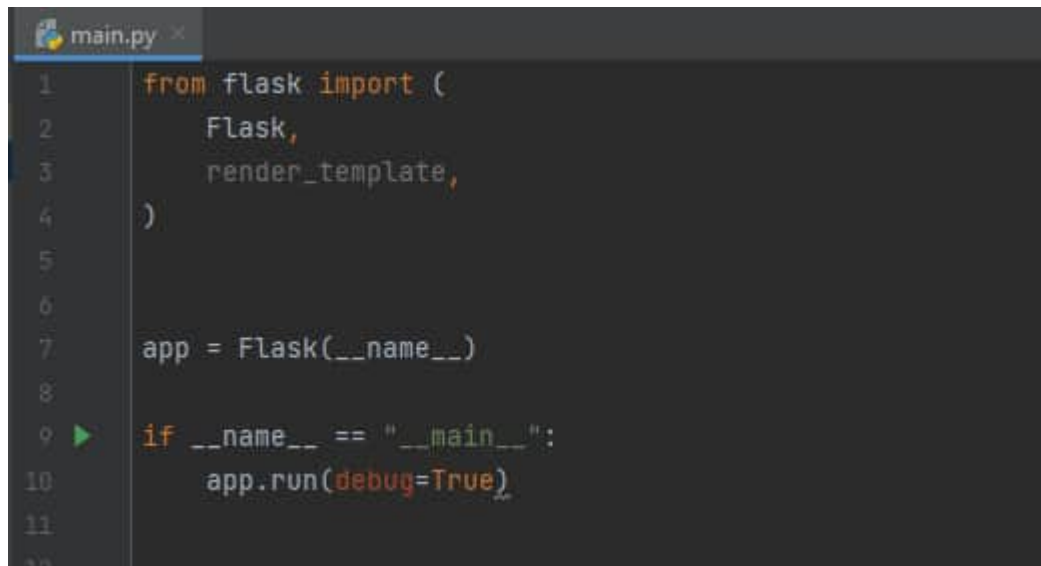
При виборі технології розробки основною увагою приділялась мові Python, а саме її широкому виборі корисних бібліотек, які значно спрощують процес розробки. Хоча Python може не мати особливої репутації щодо оптимізації та швидкості виконання процесів, ця мова приваблює розробників своїм простим та зрозумілим синтаксисом. Вона не вимагає значних витрат часу та має незліченні бібліотеки для різноманітних сценаріїв використання.

6.1 Створення веб-застосунку

Для того щоб користувач міг використовувати функціонал програми, в програми має бути інтерфейс. Він повинен бути інтуїтивно зрозумілий для користувачів.

6.1.1 Вибір інструментів для розробки сайту

Переглядаючи всі можливі інструменти для створення, менеджменту, запуску веб-додатків, обрав бібліотеку Flask. Flask — мікрофреймворк для веб-додатків, створений з використанням Python. Його основу складає інструментарій Werkzeug та рушій шаблонів Jinja2. З його допомогою, маємо змогу підняти локальний веб-сайт, написавши пару рядків коду. *див. рис. 6.1.1.1*



```

1  from flask import (
2      Flask,
3      render_template,
4  )
5
6
7  app = Flask(__name__)
8
9  if __name__ == "__main__":
10     app.run(debug=True)
11
12

```

Рисунок 6.1.1.1

Написання веб-сторінок потребує знання з:

а) **HTML** - мова для розмітки документів для веб-сторінок для подальшого їх зчитування та розставлення браузером,

б) **CSS** - спеціальна мова стилю сторінок, що використовується для опису їхнього зовнішнього вигляду.

в) **JavaScript** - мова програмування, яка дає можливість реалізувати складну поведінку веб-сторінки.

Для зручності та економії часу, використовував сервіс Bootstrap. Bootstrap - це безкоштовний набір інструментів з відкритим кодом, призначений для створення веб-сайтів та веб-застосунків, який містить шаблони CSS та HTML для типографіки, форм, кнопок, навігації та інших компонентів інтерфейсу, а також додаткові розширення JavaScript.

6.1.2 Архітектура та логіка відображення сайту

У бібліотеці Flask, є спеціальна функція, яка відповідає за відображення HTML документів, виглядає воно наступним чином, *див. рис. (6.1.2.1)*

```
@app.route('/profile', methods=['GET', 'POST'])
def profile():
    if not g.user:
        return redirect(url_for('login'))
    else:
        get_user_files()
        clear_temp_folder()
        if request.method == 'POST':
            file_index = request.form.get('file')

            if files[int(file_index)].isSelected == 1:
                files[int(file_index)].isSelected = 0 #unselect
            else:
                files[int(file_index)].isSelected = 1 #select

        return render_template('index.html', files=files)
```

Рисунок 6.1.2.1

HTML документ зберігається у спеціальній папці templates, ця папка є обов'язковою, тільки з неї можемо підтягувати та відображати HTML документи. Цю папку можна змінити на іншу у конфігурації застосунку.

При використанні окремих CSS, JavaScript або інших файлів в HTML-документах, їх зберігають у спеціалізованій директорії - static, розташованій у корені програми. Для полегшення навігації, для кожного HTML-документа було створено відповідні піддиректорії в рамках директорії static. Важливо відзначити, що при вказівці шляху до додаткових файлів у HTML-документі, слід дотримуватися формату, представленого на відповідному *рисунок 6.1.2.2*

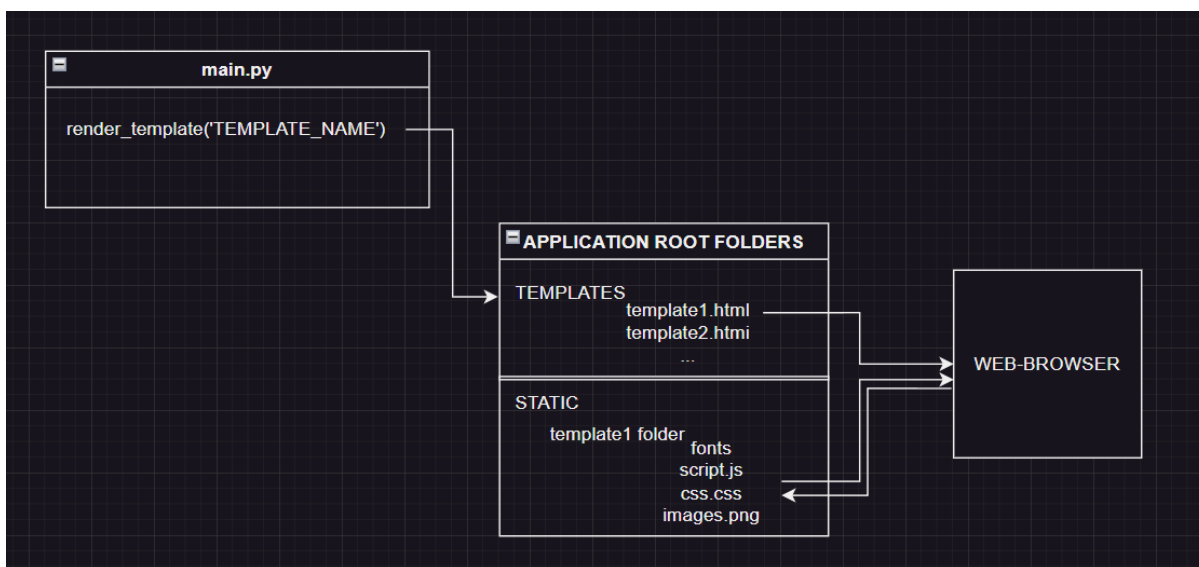
```

!----->
<script src="{{ url_for('static', filename='login/vendor/jquery/jquery-3.2.1.min.js') }}"></script>
!----->
<script src="{{ url_for('static', filename='login/vendor/animation/js/animation.min.js') }}"></script>
!----->
<script src="{{ url_for('static', filename='login/vendor/bootstrap/js/popper.js') }}"></script>
<script src="{{ url_for('static', filename='login/vendor/bootstrap/js/bootstrap.min.js') }}"></script>
!----->
<script src="{{ url_for('static', filename='login/vendor/select2/select2.min.js') }}"></script>
!----->
<script src="{{ url_for('static', filename='login/vendor/daterangepicker/moment.min.js') }}"></script>
<script src="{{ url_for('static', filename='login/vendor/daterangepicker/daterangepicker.js') }}"></script>
!----->
<script src="{{ url_for('static', filename='login/vendor/countdown/countdown.js') }}"></script>
!----->
<script src="{{ url_for('static', filename='login/js/main.js') }}"></script>

```

Рисунок 6.1.2.2

Функція `url_for()` в бібліотеці Flask використовується для створення URL-адреси для заданого маршруту (route) або функції. Вона приймає ім'я маршруту або функції та опціональні аргументи, і повертає відповідний URL-адрес. Це дозволяє легко створювати динамічні посилання на сторінки або ресурси веб-додатка, уникнувши жорстко заданого URL-адреси вручну. Дві фігурні дужки - `{{ ... }}` - дають змогу виконувати вшитий у HTML код, в даному випадку, у змінну `src` повертається тип даних `string`, з конкретною локацією додаткового файлу.



Діаграма архітектури сайту - Рисунок 6.1.2.3

Спочатку виконується запит з `main.py` на відображення HTML документа у

веб-браузер. Далі, браузера йде запит на додаткові файли на сервер, і якщо вони існують, вони теж починають відображатись або виконувати певну логіку. Фактично, якщо додаткових файлів не буде знайдено на сервері, то у веб-браузері буде відображатись тільки “сирий” HTML код.

6.1.3. Маршрутизація у Flask

Маршрутизація є однією з ключових функцій у фреймворку Flask, оскільки вона визначає, які функції повинні бути виконані для кожного запиту.

Декоратор `@app.route()` - У Flask маршрутизація відбувається за допомогою декоратора `@app.route()`. Декоратор `@app.route()` дозволяє вказати шлях до ресурсу та HTTP-методи, які можуть обробляти цей ресурс. Наприклад, декоратор `@app.route('/login', methods=['GET', 'POST'])` вказує, що функція, яка слідує за ним, буде викликатися при запиті на шлях `/login` і може обробляти як GET, так і POST запити.

Параметри шляху - Flask дозволяє використовувати параметри у шляху, що дозволяє створювати динамічні маршрути. Наприклад, ми можемо використовувати `<username>` як параметр у шляху, наприклад, `/user/<username>`. Цей параметр стає доступним у функції-обробнику як аргумент. Наприклад, функція `def user_profile(username)` буде викликатися для шляху `/user/john`, і значення "john" буде передано в якості аргументу `username`.

HTTP-методи - Flask підтримує різні HTTP-методи, такі як GET, POST, PUT, DELETE та інші. Методи можуть обробляти ресурси за допомогою параметра `methods` у декораторі `@app.route()`. Наприклад, декоратор `@app.route('/login', methods=['GET', 'POST'])` означає, що ресурс `/login` може обробляти як GET, так і POST запити.

Приклад таких функцій відображено на *Рисунку (6.1.3.1)*

```

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Обробка POST-запиту
        username = request.form['username']
        password = request.form['password']
        # Логіка перевірки логіну та паролю

        return f'Вхід користувача {username}'

    # Обробка GET-запиту (відображення форми входу)
    return '''
        <form method="post" action="/login">
            <label for="username">Ім'я користувача:</label>
            <input type="text" id="username" name="username"><br>
            <label for="password">Пароль:</label>
            <input type="password" id="password" name="password"><br>
            <input type="submit" value="Увійти">
        </form>
    '''

@app.route('/user/<username>')
def user_profile(username):
    # Логіка відображення профілю користувача
    return f'Профіль користувача: {username}'

```

Рисунок 6.1.3.1

6.1.4 Відображення файлів у веб-додатку

Основна сторінка сайту, буде відображатись файли користувачів. Бібліотека Flask має змогу вставляти Python код в HTML файли. Таким чином, використовуючи клас Grid (сітка) в HTML, з коду перекидається лист файлів в HTML документ, в якому відбувається цикл for (Рисунок 6.1.4.1), і таким чином , кожен файл, сортується та відображається на сторінці.

```

<div class="content">
  {% for file in files %}
  <div class="file {% if file.isSelected == 1 %}blue{% endif %}">
    <b>
      
      <p class="myText name">{{ file.name }}.{{ file.extension }}</p>
    </b>
    <form class="myForm" method="POST">
      <input type="hidden" class="fileNameInput" name="file" data-index="{{ files.index(file) }}">
    </form>
  </div>
  {% endfor %}
</div>

```

Рисунок 6.1.4.1

6.2 Створення бази даних

Для повноцінного функціонування сайту, було вирішено зберігати дані у базі даних.

Для повноцінної безпеки, база даних має бути зашифрована паролем, та мати систему користувачів, авторизації та інше. Наприклад, база даних postgresql включає в себе вище перелічені функції, та є надійно зашифрована. Для її встановлення потрібно запускати окремий сервер. Оскільки цей проект є лише імітацією хмарного сховища, підійде звичайний sqlite3, який можна використовувати з мовою Python.

SQLite3 - це вбудована реляційна база даних, яка забезпечує легку та ефективну систему управління даними в додатках. Вона працює на основі файлів і не потребує окремого сервера для своєї роботи. SQLite3 підтримує стандартні можливості SQL, такі як створення таблиць, вставка, вибірка, оновлення та видалення даних. Вона є популярним вибором для вбудованих систем, мобільних додатків та невеликих проектів, де вимоги до обсягу та обробки даних не великі.

Для того щоб ідентифікувати користувачів, було створено таблицю USERS, з стовпцями id, name, email, password, salt.

Додатково до цього, було створено таблицю KEYS з парою RSA ключів, таким чином, що id з таблиці USERS, відповідав id з таблиці KEYS.

Також було створено таблицю FILES, яка зберігає дані про файли користувачів, а саме назву, розширення файлу, та зашифровані розташування останніх N частинок файлу.

Таблиця NODES зберігає дані про всі вузли (папки сховища) проекту, ідентифікує їх, показує чи вузол онлайн, та показує кількість доступного місця у мегабайтах для кожного вузла.

Остання таблиця SETTINGS визначає загальні настройки проекту, такі як: на скільки частин ділити файл, скільки реплікацій робити, та максимальну можливу вмістимість вузла.

Було створено діаграму для кращого розуміння взаємозв'язку між таблицями *див. рис. 6.2.1*

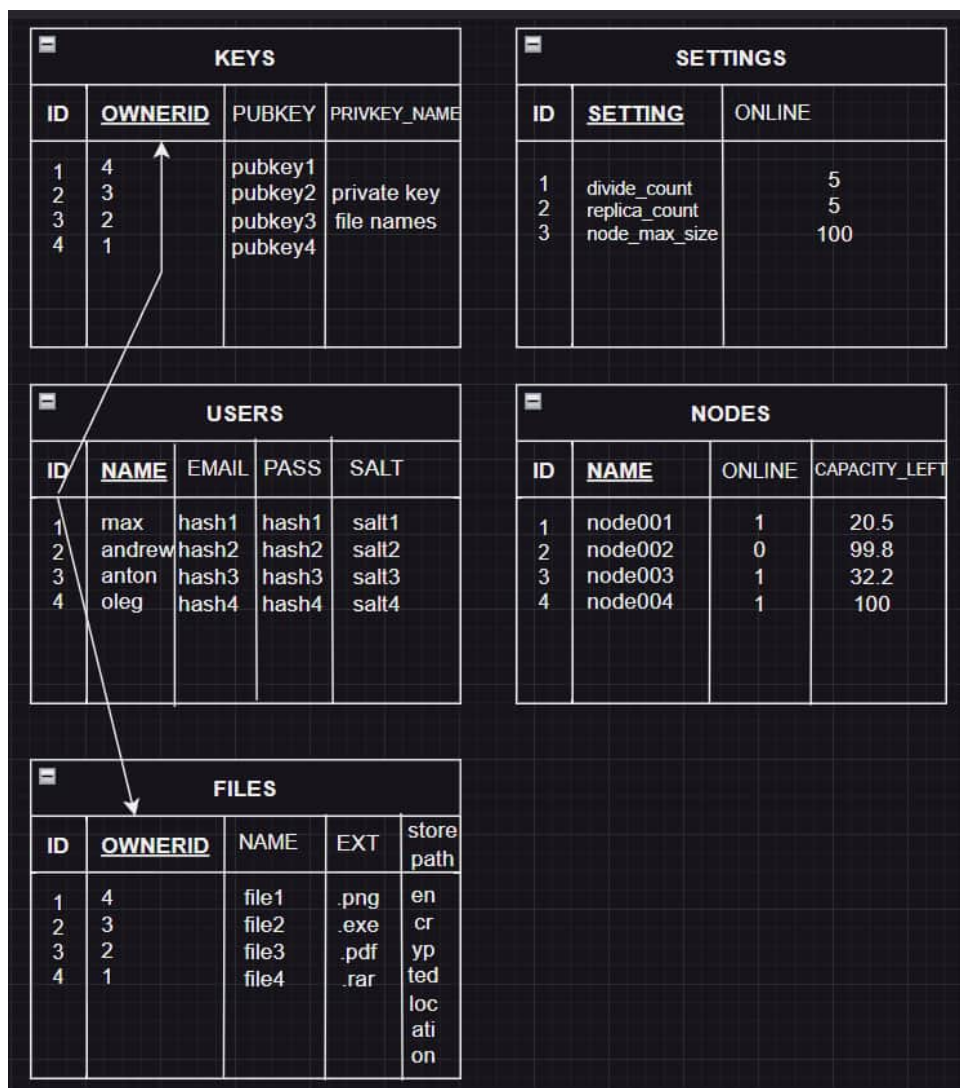


Рисунок 6.2.1

6.3 Система авторизації користувачів

У користувачів має бути можливість авторизуватись, для того щоб вони могли завантажувати, зберігати, та вивантажувати файли назад, з своєї персональної сторінки з веб-додатку.

Використовуючи готові шаблони HTML,CSS для логіну та реєстрації, з сервісу bootstrap, було вирішено покращити їх. А саме, як додатковий ступінь захисту, було добавлено у поля вводу - валідацію, для того щоб стати менш вразливим до SQL-ін'єкцій.

Для забезпечення безпеки даних при їх передачі на сервер, зокрема логіна, пароля та інших чутливих даних, було вирішено використовувати протокол HTTPS (HTTP over SSL/TLS). Він зашифровує дані, забезпечуючи їхню конфіденційність, цілісність та аутентичність. Це допомагає уникнути перехоплення або змін даних третіми особами.

SHA-256 (Secure Hash Algorithm 256-bit) - це одна з варіацій алгоритму хешування SHA-2, розробленого Агентством національної безпеки США. Вибір SHA-256 зумовлений декількома важливими характеристиками цього алгоритму:

а) **Висока безпека:** SHA-256 вважається одним з найбезпечніших алгоритмів хешування, доступних на сьогоднішній день. Він генерує унікальний хеш довжиною 256 біт для кожного набору вхідних даних. Це робить неможливим виявлення початкового вмісту на основі хешу безпосередньо, що надає високий рівень захисту від спроб взлому.

б) **Відсутність колізій:** Колізії виникають, коли два різних набори даних генерують однаковий хеш. В SHA-256 вони є надзвичайно малоймовірними, що забезпечує унікальність кожного хешу.

в) **Сумісність:** SHA-256 сумісний з більшістю сучасних систем безпеки, що дозволяє вам легко інтегрувати його в будь-яку інфраструктуру.

Ці чинники роблять SHA-256 вибором номер один для багатьох організацій та розробників, які прагнуть забезпечити надійне і ефективне хешування своїх даних.

6.3.1 Реєстрація

Заповнивши дані у реєстраційній формі, користувач тисне кнопку submit, і відправляє дані на сервер методом POST.

- 1) Спершу сервер зберігає “чисті” дані у змінні.
- 2) За допомогою бібліотеки hashlib, використовуємо хеш функцію sha256, на змінну з електронною поштою. Це вільується для того щоб не зберігати у базу даних електронну пошту у чистому вигляді, оскільки вона слугує як логін для авторизації, тому її не варто, судячи з мір безпеки, записувати у звичайному вигляді у базу даних.
- 3) Звіряємо з базою даних, чи є цей хеш вже у базі даних. Це зроблено для того, щоб користувачі не могли на одну і ту ж саму пошту реєструвати декілька акаунтів. Якщо знайдеться схожий хеш, то впливе помилка, що пошта уже зареєстрована у базі даних.
- 4) Далі, генерується випадкова сіль, яку ми додаємо до паролю та хешуємо за тією ж самою функцією. Це виконується для того, щоб у кожного паролю був свій унікальний хеш, тобто якщо у декількох користувачів буде однаковий пароль, хеш у них буде різним.
- 5) Після чого, записуємо все, у тому числі сіль, у таблицю USERS
Ось так воно виглядає в таблиці *див. рис. 6.3.1.1*

	ID	NAME	EMAIL	PASSWORD	SALT
	Філ...	Фільтр	Фільтр	Фільтр	Фільтр
1	1	Maxwell	fb19d95f847a5ece9a281cbcb72eb67c263a7496b...	6f9bb2a6390dc47afb31a1428d901c8cdae083bd1...	2#7bcd00>IC*r]EZ
2	2	Andrew	b6883a3d5b47c4d727649f78c58295536e4a2a40...	4efb575d6b3f6c5cb5f81aa2a481a1ede0cf154a0a...	*t?cA'S)zg)ha^\
3	3	Kiril	bb95f9dc8abdabe22afd20c2b9989104d7524636f...	e68b952670c031a7a36cbb445b0639f446c833956...	d.08!L5idgc)t)S#
4	4	Tatto	929ba4b64618c76d402ba9a580d0c21092474117...	a9fa4c42d3587a7c5e2baab5b3ef26ded06216598...	`\v>APRxHr+x/,gN
5	5	Stepan	b68ae8654de671f1881fca9ca98803d60b76dfef24...	2d2f5cb72d9d728a929d6fe58b63eb7d631a9d012...	#i>1W1RCKa!9`S%^

Рисунок 6.3.1.1

Код, який відповідає за реєстраційну форму див. рис. 6.3.1.2

```

if request.method == 'POST':
    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()

    email = hashlib.sha256((request.form["your-email"] + "DarudeSandStorm").encode()).hexdigest()

    salt = generate_salt()

    cursor.execute("SELECT EXISTS(SELECT 1 FROM USERS WHERE EMAIL = ? LIMIT 1)", (email,))
    emailAlreadyExists = cursor.fetchone()[0]

    if(emailAlreadyExists == False):
        name = request.form["full-name"]
        passPhrase = hashlib.md5((request.form["password"]).encode()).hexdigest()
        changePass(passPhrase)
        password = hashlib.sha256((request.form["password"] + "DarudeSandStorm" + salt).encode()).hexdigest()
        cursor.execute("INSERT INTO USERS (NAME, EMAIL, PASSWORD, SALT) VALUES (?, ?, ?, ?)",
            (name, email, password, salt))
        conn.commit()

```

Рисунок 6.3.1.2

6.3.2 Вхід у систему

Користувач вводить дані у логін-форму, та нажимає тисне login.

- 1) Спершу, у змінні зберігаються дані для входу.
- 2) Електронна пошта, хешується за алгоритмом SHA256.

- 3) Після чого, у базу даних відбувається запит, чи зберігає вона такий хеш. Якщо ні - то видає помилку, що неправильний логін, чи пароль.
- 4) Якщо ж знаходиться збіг, то з бази даних дістається хешований з сіллю пароль та сама сіль.
- 5) До паролю, котрий ввів користувач, додається сіль та хешується таким же чином, як і при реєстрації. Результати порівнюються, і якщо вони будуть однакові, то вхід буде успішним.

Код, який відповідає за логін-форму *див. рис. 6.3.2.1*

```

if email is not None and password is not None:

    conn = sqlite3.connect('database.db')
    cursor = conn.cursor()

    email = hashlib.sha256((email + "DarudeSandStorm").encode()).hexdigest()

    cursor.execute("SELECT EXISTS(SELECT 1 FROM USERS WHERE EMAIL = ? LIMIT 1)", (email,))
    emailAlreadyExists = cursor.fetchone()[0]

    if (emailAlreadyExists == True):
        passPhrase = hashlib.md5((password).encode()).hexdigest()
        changepass(passPhrase)

        salt = cursor.execute("SELECT SALT FROM users WHERE email = ?", (email,)).fetchone()[0]

        hashed_password = hashlib.sha256((password + "DarudeSandStorm" + salt).encode()).hexdigest()

        cursor.execute("SELECT COUNT(*) FROM users WHERE email = ? AND password = ?", (email, hashed_password))
        match = cursor.fetchone()[0]

        if match:
            user_id = cursor.execute("SELECT ID FROM USERS WHERE EMAIL = ?", (email,)).fetchone()[0]

            session["user_id"] = user_id

            g.password = password
            return redirect(url_for('profile'))

    else:

```

Рисунок 6.3.2.1

6.3.3 Безпека

Використання вище перелічені методів, а саме: захищений протокол передачі даних, хешування сучасним та надійним алгоритмом, використання солі, та валідації, робить авторизацію у веб-додатку максимально захищеною від зловмисників.

Також, варто зазначити, що веб-додаток використовує сесії, при авторизації, користувачеві автоматично надається `session_id`, який випадково генерується, та в подальшому використовується для ідентифікації користувача на різних сторінках веб-сайту. Не авторизовані користувачі не матимуть змоги переходити по доступним маршрутам, їх автоматично буде перенаправляти до вкладки з авторизацією, до поки, вони не ввійдуть у систему.

6.4 Шифрування та зберігання файлів

6.4.1 Приватний та публічний ключ

В програмі використовується здебільшого гібридний метод шифрування даних. В якості асиметричного шифрування, веб-додаток використовує пару RSA ключів. З публічним ключем проблем немає, оскільки його можна зберігати і у чистому вигляді. А ось на рахунок приватного ключа, тут вже різні сервіси зберігають їх по різному. В даному випадку, приватний ключ зберігається на самому сервері, але не у відкритому вигляді.

При реєстрації користувача, для нього автоматично генерується пара RSA ключів, публічний ключ у відкритому вигляді записується у базу даних, а саме у таблицю KEYS. Натомість приватний ключ, спершу зашифровується AES-256 шифром, використовуючи кодову фразу в якості ключа. До чистого паролю користувача додається текст, далі це хешується за алгоритмом MD5, і результат і є кодовою фразою. Після шифрування приватного ключа, йому дається випадково згенерована назва, і далі шифрований ключ відправляється у спеціально відведену директорію на сервері, в якій зберігаються приватні ключі користувачів. Для того

щоб і надалі використовувати цей ключ, його назва записується у базі даних.

Для розшифрування приватного ключа потрібен пароль користувача у чистому вигляді, який, в свою чергу, не зберігається у такому виді на сервері чи у базі даних, тому сервер не буде мати змогу дізнатися приватний ключ, на відміну від сервісу Google Drive.

Для роботи такого методу розшифрування ключа, сервер записує кодову фразу в оперативну пам'ять. Це змушує веб-додаток, час від часу змушувати користувачів заново проходити авторизацію для повторного збереження ключа в оперативну пам'ять. Зберігання кодової фрази в оперативній пам'яті, не є найефективнішим та найнадійнішим способом зберігати кодові фрази, але натомість, дає можливість працювати практично без окремої клієнтської частини, оскільки все що необхідно, вже є, або передається на сервер.

6.4.2 Вузли

Для того щоб імітувати реальну систему децентралізованого хмарного сховища, було розроблено систему директорій в якості вузлів. Вся інформація про вузли зберігається у відповідній таблиці NODES у базі даних сервера. Кожен вузол має параметр чи підключений від до мережі (онлайн/офлайн), та скільки даних у нього можна ще вписати. Якщо вузол буде “офлайн”, то при роботі шифрування/розшифрування, його буде виключено з доступних вузлів. Таким чином, у децентралізованій системі, це імітує реальні ситуації, коли у районі 10% вузлів, можуть бути “офлайн” , але тим не менш система досі буде працювати.

Загальна вмістимість вузлів визначається у таблиці SETTINGS у базі даних, після кожного вивантаження, видалення файлу, система рахує кількість доступного місця, та записує дані у таблицю.

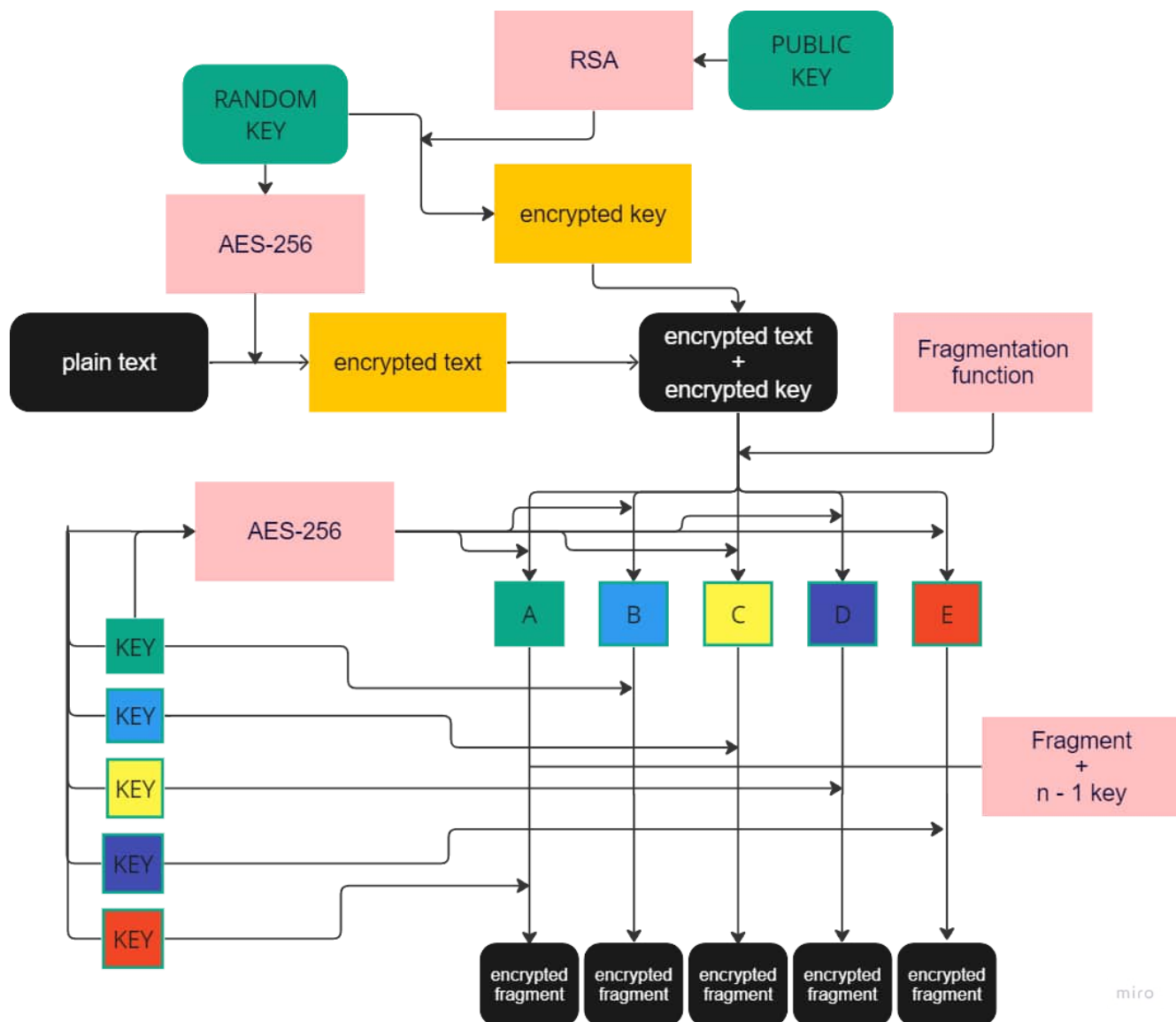
6.4.3 Завантаження файлів на сервер

При завантаженні файлів на сервер, виконується наступні дії:

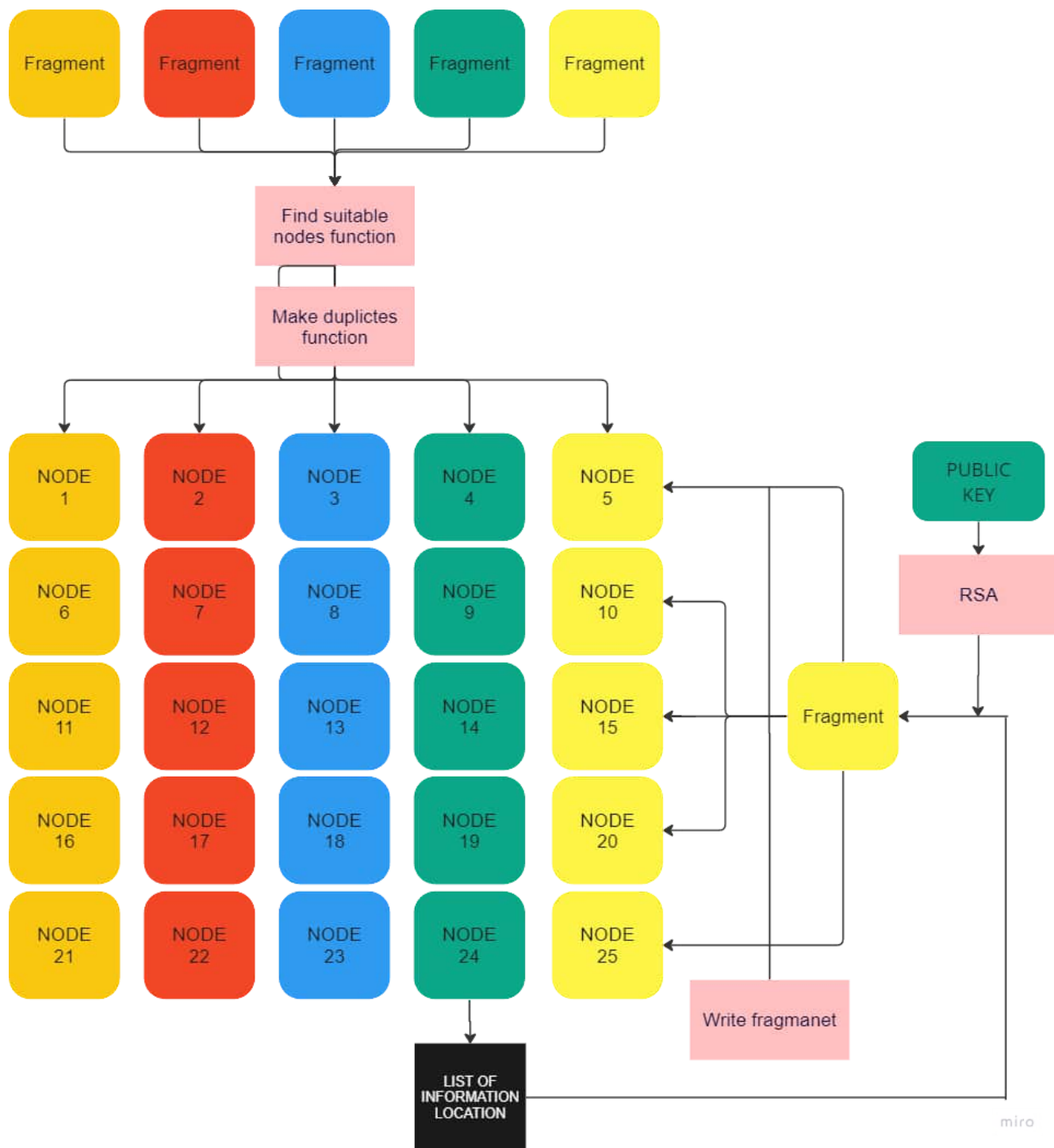
- 1) Програма приймає файл по захищеному HTTPS протоколу, та зберігає їх як лист байтів.
- 2) Відбувається ідентифікація файлу, в тимчасові змінні записуються параметри імені файлу, його розширення, та ідентифікаційний код його власника. Ці параметри, потрібні для коректного відображення файлів на самому сайті. Розширення файлу також потрібно для того, щоб на сайті, над назвою файлів користувачів, відображалась правильно піктограма, яка показує суть розширення.
- 3) Випадковим чином генерується AES-256 ключ, за допомогою цього ключа та цього алгоритму, шифрується весь вміст файлу.
- 4) У базу даних йде запит на відкритий RSA ключ користувача. Цим ключем шифрується попередній AES-256 ключ, яким вже було зашифровано файл. Та результат цього. вшивається у вже зашифрований файл.
- 5) З таблиці SETTINGS, витягується два значення, на скільки частин фрагментувати файл, та скільки його реплікацій робити.
- 6) Файл розподіляється на N кількість частин, таким чином що кожен фрагмент файлу, має різний розмір. Це зроблено для того, щоб зловмисник не зміг скласти файл скориставшись тим, що у всіх частин однаковий розмір.
- 7) Для кожної частини файлу, генерується випадковий AES-256 ключ, яким і шифрується кожен фрагмент файлу.

- 8) Згенерований ключ n фрагменту зберігається у $n+1$ фрагменті. А ключ останнього фрагменту, записується у найперший фрагмент.
- 9) За допомогою спеціальної функції, лист з усіх фрагментів відправляється шукати вузли, які будуть задовольняти розмір кожного фрагменту, таким чином, що кожен фрагмент повинен зберігатись у різному вузлі.
- 10) Крок 9 виконується ще пару разів, оскільки потрібно також створити N -кількість дублікатів (N - визначається у 5 кроці), та знайти для кожного дублікату вузол, який теж буде задовольняти розмір фрагменту, та буде унікальним для кожного фрагмента. В результаті створюється лист вузлів, де для кожного фрагменту буде N -кількість різних локацій.
- 11) Починаючи з останнього фрагменту, у нього записується всі можливі вузли $n-1$ фрагмента. Тобто практично кожен фрагмент має інформацію про свого попереднього фрагмента, але не знає свого наступного. Схожим чином працює система блокчейну. Це дає додатковий захист для файлу. Оскільки тільки знаючи локації останнього фрагменту, буде можливість зібрати файл цілим. Як додатковою мірою безпеки, всі локації шифруються відкритим RSA ключем користувача.
- 12) Всі частини файлу відповідно до їх локацій, зберігаються у відповідних вузлах. Інформація про файл, та зашифровані локації останнього фрагменту файлу зберігаються у таблицю FILES у базі даних.

Ця система шифрування, забезпечує надійний захист даних на сервері, оскільки включає в себе фрагментування, багаторазове шифрування різними ключами та методами, надійну систему зберігання ключів і взаємозв'язку та локацій фрагментів.



Візуалізація методів шифрування файлу - Рисунок 6.4.3.1



Візуалізація зберігання частин файлу у вузлах - Рисунок 6.4.3.2

ВИСНОВКИ

При завершенні дипломної роботи на тему шифрування та зберігання даних у хмарних сервісах, було отримано значне розуміння тонкощів технологій хмарного зберігання даних та заходів безпеки, що лежать в їх основі. Цей процес був одночасно навчальним та трансформаційним, від базового розуміння хмарних сервісів до створення безпечної, децентралізованої системи хмарного зберігання даних.

Дослідження включало глибоке занурення в роботу основних хмарних сервісів зберігання даних, таких як Google Drive, Dropbox та децентралізований Storj.io. Були вивчені підходи цих сервісів до шифрування даних, методології зберігання та користувацькі функції. Ці сервіси використовують поєднання симетричних та асиметричних методів шифрування і дотримуються різних протоколів для передачі та зберігання даних. Централізація чи децентралізація підходів до зберігання даних сильно впливає на вразливість сервісів до атак і втрати даних.

Google Drive і Dropbox дотримуються традиційної моделі централізованого зберігання даних з надійними протоколами шифрування. Однак факт, що вони зберігають ключі шифрування, створює теоретичний ризик доступу до даних з боку постачальників послуг. Натомість Storj.io працює за децентралізованою моделлю, яка пропонує додатковий рівень безпеки. Шифрування на стороні клієнта, яке використовує Storj.io, гарантує, що тільки користувачі мають доступ до своїх даних, підвищуючи конфіденційність.

В результаті всебічного дослідження було вирішено створити власну хмарну систему зберігання даних, гібридну, з використанням децентралізованої моделі для підвищення стійкості даних і поєднанням шифрування RSA і AES-256 для надійної безпеки. У цій системі кожна частина файлу шифрується індивідуально, за допомогою унікальних ключів, які потім шифруються за допомогою RSA для

додаткового захисту. Незважаючи на те, що приватні ключі зберігаються на сервері, вони також зашифровані, що є додатковим рівнем безпеки.

Створена система поєднує в собі безпеку децентралізованого шифрування на стороні клієнта Storj.io та надійність центральних серверів Google Drive і Dropbox. Вона пропонує користувачам додатковий рівень безпеки завдяки гібридному шифруванню, забезпечуючи при цьому доступність і надлишковість даних завдяки децентралізованій моделі.

Необхідно визнати, що сфера криптології стрімко розвивається. З появою квантових обчислень на горизонті криптографічні алгоритми, які вважаються безпечними сьогодні, можуть стати вразливими завтра. Тому слід стежити за останніми розробками квантово-стійких криптографічних алгоритмів, оскільки вони можуть зіграти вирішальну роль у майбутньому безпечного хмарного зберігання даних.

Забігаючи наперед, можна сказати, що децентралізація відіграватиме значну роль в еволюції хмарних сховищ. Оскільки занепокоєння щодо конфіденційності та безпеки даних стає все більш помітним, децентралізовані моделі зберігання даних, такі як та, що використовується Storj.io і в програмній реалізації, ймовірно, стануть більш поширеними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Морозюк А.А., Ніколюк П.К. Шифрування даних. Комп'ютерні технології обробки даних, 2022, С. 80-82.
2. Северина С.В. Інформаційна безпека та методи захисту інформації. Вісник Запорізького національного університету. Економічні науки, 2016, №1, С. 155-161.
3. “What is encryption?”. [Електронний ресурс]. – Режим доступу: <https://cloud.google.com/learn/what-is-encryption> (Дата звернення: 06.06.2023)
4. “How public and private key encryption works”. [Електронний ресурс]. – Режим доступу: <https://www.preveil.com/blog/public-and-private-key/> (Дата звернення: 06.06.2023)
5. “What is Cloud Encryption?”. [Електронний ресурс]. – Режим доступу: <https://www.crowdstrike.com/cybersecurity-101/cloud-security/cloud-encryption/> (Дата звернення: 06.06.2023)
6. Wikipedia.. [Електронний ресурс]. – Режим доступу: <https://www.wikipedia.org/> (Дата звернення: 06.06.2023)
7. “Cloud Encryption: Using Data Encryption in the Cloud”. [Електронний ресурс]. – Режим доступу: <https://www.business.com/articles/cloud-data-encryption/> (Дата звернення: 06.06.2023)
8. How Dropbox keeps your files secure. [Електронний ресурс]. – Режим доступу: <https://help.dropbox.com/security/how-security-works> (Дата звернення: 06.06.2023)
9. Storj.io. [Електронний ресурс]. – Режим доступу: <https://www.storj.io/solution-brief/encryption> (Дата звернення: 06.06.2023)
10. “Where Are the Keys? Managing Encryption in the Cloud”. [Електронний ресурс]. – Режим доступу: <https://www.pkware.com/blog/where-are-the-keys-managing-encryption-in-the-cloud> (Дата звернення: 06.06.2023)

11. “Is Google Drive secure and what steps can you take to improve it?”. [Електронний ресурс]. – Режим доступу: <https://www.comparitech.com/blog/information-security/google-drive-secure/> (Дата звернення: 06.06.2023)
12. “What is Decentralized Cloud Storage and How does it Work?”. [Електронний ресурс]. – Режим доступу: <https://www.arcana.network/blog/how-does-decentralised-storage-work> (Дата звернення: 06.06.2023)
13. “Уроки по Flask с нуля”. [Електронний відеохостинг]. – Режим доступу: <https://www.youtube.com/playlist?list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn> (Дата звернення: 06.06.2023)

