Secure Modularized Online Cloud Storage System with Multi-layer Security

CHAR Cheuk Tung George[1], Kent Max Chandra[1], WANG Youkang Albert[1]
COMP3334 Project Team 63
[1]Hong Kong Polytechnic University
{22055747D,22107416D,22097007D}@connect.polyu.hk
*

Abstract

This report presents a comprehensive secure file storage system that implements multi-layered encryption and advanced authentication mechanisms. Our solution addresses critical security challenges through a hierarchical key management architecture consisting of password-derived master keys, randomly generated sub-master keys, and asymmetric cryptography. The system provides strong protection against 2 major vulnerabilities listed in our threat model: server compromises (as a passive attacker) and unauthorized access from client end. Besides enforcing different security measures, our system also enables various secure file operations and secure file sharing between authorized users. Through careful implementation of cryptographic primitives and secure protocols, our solution ensures that files remain confidentially encrypted end-to-end, with the server having zero knowledge of file contents or encryption keys.s

Keywords - Cloud Storage Security, Multilayer Encryption, Hashing, Authentication, One-Time Password

1 Introduction

Data security has become increasingly critical as organizations and individuals migrate to cloud storage solutions. Traditional approaches rely primarily on perimeter security (e.g., firewalls) and simple access controls, which prove insufficient when there're **Cloud Server Breaches** leading to unauthorized client data access on the server side or **Compromised Client Devices** enabling attackers to bypass perimeter safeguards entirely and access the victim client's online files using the client end. Our project build a **threat model** based on these 2 vulnerabilities by implementing a modularized online cloud storage system where files are encrypted before leaving the client device while being stored on the server with sophisticated file access control, decryption keys are never exposed to either client or server ends via multi-layer key hierarchy with good encapsulation, all operations are authenticated and logged, together with many other security features.

*Project page: `https://github.com/Georgehyuvuhjb/COMP3334_project/tree/main/COMP3334_GP_3`

2 Threat Model

2.1 Adversarial Scenarios & Security Objectives

Our developed online cloud storage system is specially designed for 2 types of adversary scenarios that cause vulnerabilities in today's cloud storage systems: **Server-side Passive Adversary** and **Unauthorized User Adversary**.

Server-side Passive Adversary The machine hosting the Server program is modeled as an honest-but-curious adversary. It faithfully executes the server-side protocol (e.g., storing files, managing user accounts) but actively monitors all communications and stored data, trying to decrypt files uploaded by clients. Our system could prevent the server from decrypting client-owned files, even with full access to ciphertexts and supporting client file operations.

Unauthorized User Adversary An unauthorized user gains physical or logical access to a legitimate user's device (e.g., stolen credentials, compromised workstation) and attempts to access the victim's remotely stored files via the Client program. Our system could prevent the unauthorized user from accessing legitimate user's stored files even when the attacker has physical control over victim's device.

Other Security Considerations We explicitly apply **access control** to the files owned by different users by setting a folder for each user when first registering on our cloud system. To prevent users from accessing others' files by navigating outside their own folder using "../", we explicitly avoid this problem by providing users with a numbered list of files, and users could only type in the index of the file to access it instead of inputting the entire file path. **SQL injection** is another common consideration for cloud storage, but since our system doesn't involve a database, this problem is automatically resolved. **Authentication** is also a general necessity for cloud system, and we authenticate the client during every file operation. Last but not least, we also implement **Non-repudiation** feature on the system by logging all the operations.

2.2 Trust Assumptions

During the implementation of our security system, we need to assume trust over some entities and operations, some prominent ones include:

- The communication channels are safe. (i.e., **No Man-In-The-Middle Attack!**)
- Server doesn't perform active attacks such as modifying the files.
- The server adheres to protocol specifications and does not collude with unauthorized users.
- The client program's integrity is maintained on the user's device unless compromised by the unauthorized user adversary.
- Underlying cryptographic primitives (e.g., encryption schemes, hash functions) are secure against computationally bounded adversaries.

Therefore, we don't consider attacks outside our threat models, which include but not limit to: *Man-In-The-Middle Attack*, *Active Server-Side Attacks*, *Client-Side Code Tampering*, *Replay Attacks*, *Hardware Compromise*, etc.

2.3 Efficiency and Usability Considerations

Based upon addressing vulnerabilities listed in the threat model, we also aim to maximize the efficiency and usability of our system. Here are just a few examples.

Accessible and Device-agnostic We ensure that our clients can access the system everywhere using different machines by **storing all the keys in the server but not locally**. Meanwhile, we implement sufficient encryption and other measures, ensuring that server can't access the key although all the keys are stored on it.

Small Security Overhead As our system doesn't consider attacks outside the threat model, the overhead for security is comparative light.

More details about our design and efforts for making our system more efficient and usable will be elaborated in the coming sections.

3 Modularized System Overview

Our secure online cloud storage system can be logically divided into 5 modules: **Registration Module**, **Login & Password Reset Module**, **File Operation Module**, **File Sharing Module**, **Log Audit Module**. Registration and Login & Password Reset are closely related, but since complicated and different security measures are applied in these 2 modules, we seperate them into 2 sections for discussion. In the Login & Password Reset Module, we'll also discuss our implementation of **OTP (Our Extended Functionality)**. File Operation Module deals with Upload/Download/Add/Edit/Delete operations on client's files. In both File Operation and File Sharing Module, we discuss our implemented security measures for **Access Control**. Finally, we also talk about Log Audit and the admin permissions for that.

4 Algorithm & Analysis for Registration Module

4.1 Hierarchical Key Management Architecture

Our system implements a sophisticated key hierarchy with three distinct layers to ensure file security while maintaining usability.

Master Key (MK) protects the Sub-Master Key and Private Key through encryption. It is derived using SHA256 [1] with the user's password, CLIENT_CONSTANT to produce a 32-byte key. The Master Key is never stored in any persistent medium, exists only in client memory during active sessions, and is derived on-demand from the user's password whenever needed.

Figure 14 shows the generation workflow of Master Key. This workflow ensures that the Master Key is derived from two components - one provided by the user (password) and one built into the client application (CLIENT_CONSTANT). This approach prevents server-side derivation of the Master Key even with knowledge of the user's password hashes.

Sub-Master Key (SMK) encrypts individual file keys for storage. It is generated using 32 bytes of cryptographically secure random data and stored encrypted with the Master Key using AES-256-GCM. The Sub-Master Key is unique per user and remains constant even when the password changes, which enables password changes without requiring file key re-encryption.

File Keys (FK) encrypt individual file contents to provide isolation between files. Each file key consists of 32 bytes of cryptographically secure random data, unique per file. Different files have different keys even when they contain the same content, which limits compromise scope to individual files. File keys are stored encrypted with the Sub-Master Key using AES-256-GCM. This deterministic derivation ensures that each file has a unique encryption key derived from the user's Sub-Master Key, providing isolation between files. File Keys are generated using the **SHA256 hash of Sub Master Key (SMK) + file name**

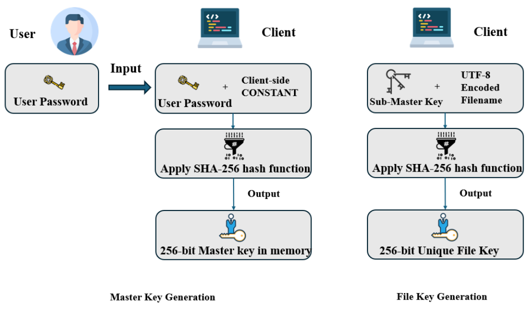


Figure 1. Master Key and File Key Generation Workflow

RSA Key Pair facilitates digital signatures and asymmetric operations for sharing. The system generates an RSA-2048 key pair with secure exponent generation. The private key is encrypted with the Master Key while the public key is stored in plaintext. These keys provide authentication via digital signatures, facilitate secure key sharing between users, and provide non-repudiation for user actions.

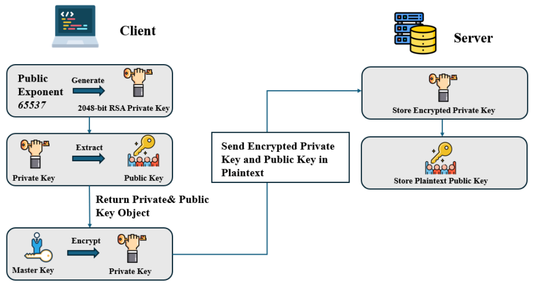

Client
Server
Public Exponent 65537
Generate
2048-bit RSA Private Key
Store Encrypted Private Key
Private Key
Extract
Public Key
Send Encrypted Private Key and Public Key in Plaintext
Store Plaintext Public Key
Return Private& Public Key Object
Master Key
Encrypt
Private Key

Figure 2. RSA Key Pair Generation Workflow

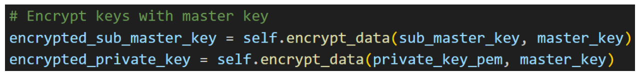
4.2 Registration Workflow & Manage Key

The workflow for user registration is shown below.

1. Client generates random 32-byte Sub-Master Key.
2. Client generates RSA-2048 key pair.
3. Master Key from password using SHA256.
4. Client encrypts Sub-Master Key using Master Key with username as authentication data.
5. Client encrypts Private Key using Master Key.
6. Client transmits to server: *Username*, *Password hash*, *Server salt*, *Encrypted Sub-Master Key*, *Encrypted Private Key*, *Public Key (plaintext)*.
7. Server stores user information and encrypted keys.

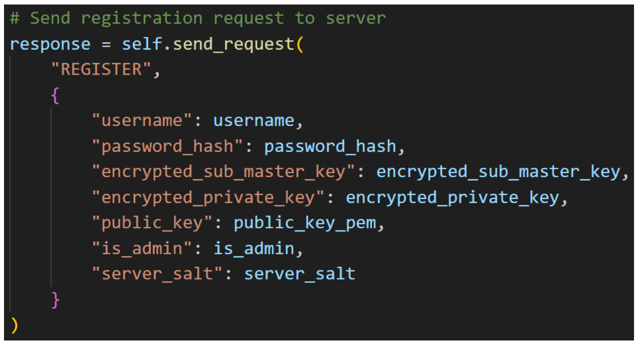
4.3 Technical Implementations

Here are some important steps for code implementation.



```
# Encrypt keys with master key
encrypted_sub_master_key = self.encrypt_data(sub_master_key, master_key)
encrypted_private_key = self.encrypt_data(private_key_pem, master_key)
```

Figure 3. Encrypt keys with master key



```
# Send registration request to server
response = self.send_request(
    "REGISTER",
    {
        "username": username,
        "password_hash": password_hash,
        "encrypted_sub_master_key": encrypted_sub_master_key,
        "encrypted_private_key": encrypted_private_key,
        "public_key": public_key_pem,
        "is_admin": is_admin,
        "server_salt": server_salt
    }
)
```

Figure 4. Send registration request to server

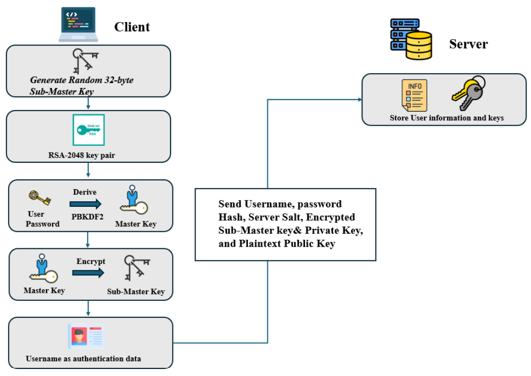

Client
Server
Generate Random 32-byte Sub-Master Key
Store User information and keys
RSA-2048 key pair
User Password
Derive
PBKDF2
Master Key
Send Username, password Hash, Server Salt, Encrypted Sub-Master key& Private Key, and Plaintext Public Key
Master Key
Encrypt
Sub-Master Key
Username as authentication data

Figure 5. RSA Key Pair Generation Workflow

5 Algorithm & Analysis for Login and Password Reset Module

5.1 Extended Functionality: OTP Submodule

A **One-Time Password (OTP)** is a dynamically generated, short-lived code used to authenticate a user for a single login session or transaction. Unlike traditional static passwords, OTPs are valid only once and expire after a brief period — typically seconds to minutes. This temporary nature significantly reduces the risk of unauthorized access, even if the code is intercepted, as it cannot be reused or exploited after expiration.

We specially implemented **TOTP (Time-Based One-Time Password)** [2], defined in RFC 6238, a type of OTP algorithm that generates temporary codes using a **shared secret key** and **the current timestamp**. Unlike HOTP (event-triggered), TOTP codes refresh periodically—typically every **30 seconds** on a synchronized clock between the authentication server and the user's device (We use **Google Authenticator**). This time-bound approach ensures codes expire quickly, minimizing the risk of interception and replay attacks.

5.2 Login - Initial Authentication

1. Client requests server salt for username
2. Server responds with stored salt value
3. Client derives Master Key locally using password and a constant defined in the client program
4. Client calculates password hash and sends to server
5. Server verifies password hash against stored value
6. If valid, server prompts for Google Authenticator code
7. User opens their Google Authenticator app and reads current TOTP code
8. User enters the 6-digit TOTP code into client application
9. Client sends TOTP code to server for verification
10. Server validates TOTP against user's stored secret using current timestamp
11. If TOTP is valid, server responds with encrypted Sub-Master Key and encrypted Private Key
12. Client decrypts Sub-Master Key and Private Key using Master Key
13. Authentication complete, client maintains secure session state

The two-factor authentication process adds an additional layer of security beyond password verification, requiring a time-limited OTP before granting access to encrypted keys.

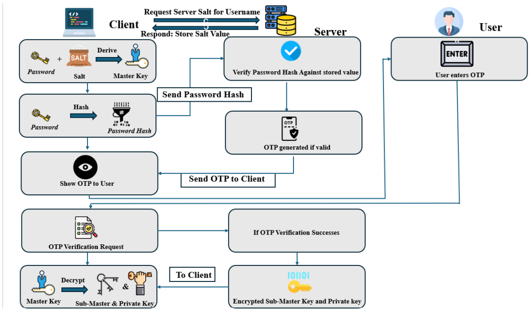


Figure 6. Login Authentication Workflow

Analysis The implementation of **TOTP** via Google Authenticator could guard against several attacks. **Phishing/Replay Attacks** which intercepted passwords or session tokens are useless without the TOTP (which expires in 30 seconds); **Credential Stuffing** attack can't work as well since attackers cannot bypass TOTP verification even with stolen passwords. Also, TOTP's 6-digit code (1M combinations) requires **16K attempts/second** for 30-second window. This makes brute-force exploitation practically infeasible.

Also, **Master Key (MK)** is derived using PBKDF2 with password, CLIENT_CONSTANT, and server salt. Even when **Server is compromised**, attackers cannot derive MK without CLIENT_CONSTANT, rendering stolen password hashes/salts insufficient for decryption. **Offline Brute-Force** can't work as well since PBKDF2's computational cost (e.g., 100K iterations) slows brute-force attempts to 1K guesses/second (vs 1B/sec for unsalted SHA256).

5.3 Login - Fetching Key from Server improves efficiency of following operations

Every time when a user logs in, client will fetch **encrypted Sub-Master Key (SMK)** and **encrypted Private Key** (used for signing digital signature and encryption during sharing) from server. This is in preparation for the following operations.

Analysis This operation will improve the efficiency of various following operations because users don't need to fetch SMK and private key again and again. These keys will be decrypted by the master key, used, and removed upon termination of the current session. Such design of key ensures the best of 3 worlds: (i) **High Security** (since no key is stored locally and the keys stored on server is always encrypted, attacker can't access the key even they have control of the client or server side); (ii) **High Efficiency** (Since the key is stored in RAM after loaded and we don't need to fetch keys from server more than once during the session); (iii) **High Usability & Scalability** (since users don't need to store any keys on the client end, this system is device agnostic and user could access the system from any machine).

5.4 Signature - Authentication of User in Various Operations

We utilize signature to authenticate the user (make sure it is really the user) during various operations, such as change password, download file, share file, and delete file operations. It is an important core submodule supporting all these operations.

Signature Signing The Signature Signing basically follows these steps:

1. Input: Data string to be signed
2. Client verifies Private Key is available in memory
3. Client creates signature using RSA-PSS [3] with: *SHA-256 hash function*, *MGF1 mask generation function*, *Maximum salt length for increased security*
4. Client encodes signature with base64
5. Output: Base64-encoded digital signature

Signature Verification The verification workflow is similar:

1. Server receives: *Username*, *data string*, and *claimed signature*
2. Server retrieves user's public key from storage
3. Server decodes base64 signature to binary format
4. Server attempts signature verification using: *User's public key RSA-PSS algorithm*, *SHA-256 hash function*, *Matching padding configuration*
5. If verification succeeds, return true; otherwise, log the failure and return false

This verification process ensures that only commands with valid signatures from authenticated users are processed, providing non-repudiation for all system operations.

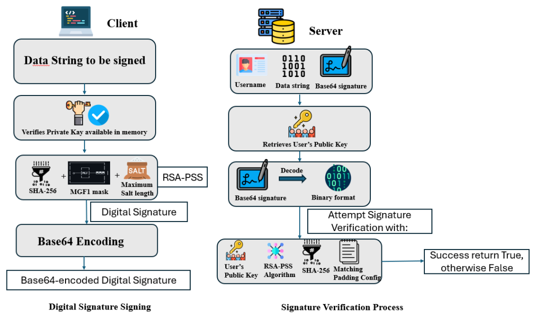


Figure 7. Digital Signature Signing & Verification Workflow

Analysis **Signature** binds requests to user identity and specific timestamps, effectively preventing tampering/replay Attacks. **SHA-256 hashing** and **RSA-2048** operations balance security with computational feasibility (around 50ms/signature on modern hardware).

5.5 Password Change System

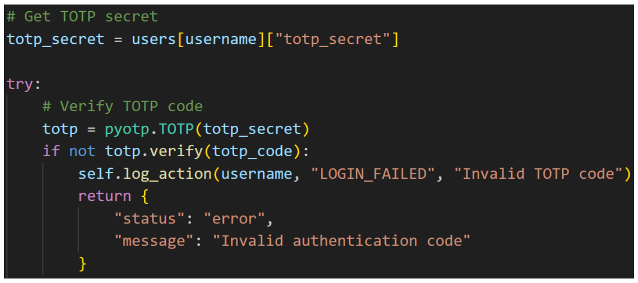
Password Change is another important feature in a secure online system. Our system supports password change while ensuring efficiency by using both a Master Key and Sub-Master Key, below is the the workflow:

1. Client verifies current password with server
2. Client inputs and confirms new password
3. Client evaluates new password strength
4. Client derives new Master Key from new password
5. Client re-encrypts Sub-Master Key with new Master Key
6. Client re-encrypts Private Key with new Master Key
7. Client generates new random server salt
8. Client calculates new password hash with new salt
9. Client generates timestamp and signature: username + timestamp + "CHANGE_PASSWORD"
10. Client sends to server:
    - New password hash
    - New server salt
    - Re-encrypted Sub-Master Key (SMK)
    - Re-encrypted Private Key
    - Authentication (username, timestamp, signature)
11. Server verifies signature
12. Server updates stored password hash and salt
13. Server updates stored encrypted keys
14. Server logs password change action
15. Server returns success response

Analysis We use Master Key (MK) to encrypt Sub Master Key (SMK), and use SMK to encrypt the other keys. The **advantage** of using a SMK here is that when the user changes to a new password, there's no need to re-encrypt the other keys, and we just need to encrypt the SMK using the newly generate MK from the newly user-defined password. This greatly improves the efficiency of the system.

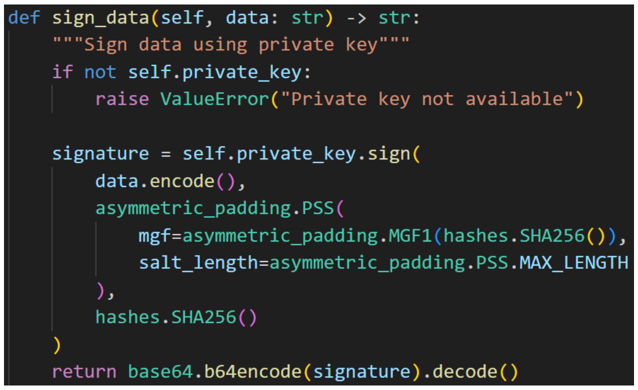
5.6 Technical Implementations

Here are some important steps for code implementation.



```
# Get TOTP secret
totp_secret = users[username]["totp_secret"]

try:
    # Verify TOTP code
    totp = pyotp.TOTP(totp_secret)
    if not totp.verify(totp_code):
        self.log_action(username, "LOGIN_FAILED", "Invalid TOTP code")
        return {
            "status": "error",
            "message": "Invalid authentication code"
        }
```

Figure 8. server check whether TOTP is correct



```
def sign_data(self, data: str) -> str:
    """Sign data using private key"""
    if not self.private_key:
        raise ValueError("Private key not available")

    signature = self.private_key.sign(
        data.encode(),
        asymmetric_padding.PSS(
            mgf=asymmetric_padding.MGF1(hashes.SHA256()),
            salt_length=asymmetric_padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return base64.b64encode(signature).decode()
```

Figure 9. generate digital signature

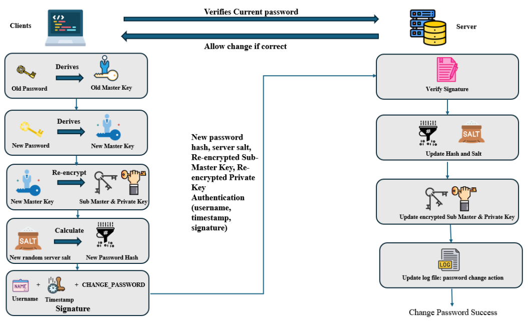


Figure 10. Password Change Workflow

6 Algorithm & Analysis for File Operation Module

6.1 File Encryption Design During Upload

File Encryption is the core security submodule supporting secure file upload. Therefore, we discuss file encryption as a separate submodule with others.

1. Client generates File Key from Sub-Master Key and filename.
2. Client creates random 16-byte initialization vector (IV).
3. Client applies PKCS7 padding to file data.
4. Client encrypts padded file data using AES-CBC with

File Key and IV.
5. Client prepends IV to encrypted data and encodes with base64.
6. Client encrypts File Key with Sub-Master Key.
7. Client sends encrypted file data and encrypted key to server as separate objects.
8. Server stores encrypted file and encrypted key in separate locations.

This approach ensures that file data remains encrypted throughout the entire storage lifecycle. The separation of file content encryption from key encryption creates a secure hierarchy where the server never possesses the ability to decrypt files.

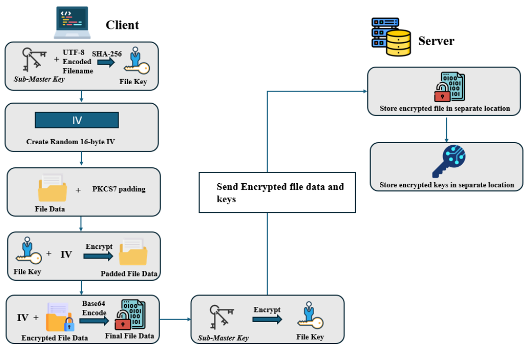


Figure 11. File Encryption Mechanism

Analysis All the File Keys (FK) are derived via SHA256(SMK + filename) and are thus unique. This prevents **mass decryption** if one file is compromised (cryptographic containment). We implemented **AES-CBC with PKCS7 Padding**. The uniqueness of **IV** prevents pattern analysis (it meets NIST SP 800-38A standards), and it blocks replay attacks as well (IVs ensure ciphertext non-determinism). Also, Encrypted File Key and file content are stored separately, making attack harder as that would require compromise of both storage systems for decryption.

6.2 File Upload Workflow

File Upload utilizes the file encryption submodule and adds additional secuirty measures using the workflow below:

1. Client reads file from local storage
2. Client generates File Key from Sub-Master Key and filename
3. Client encrypts file content with File Key
4. Client encrypts File Key with Sub-Master Key
5. Client generates current timestamp
6. Client creates content hash of encrypted file
7. Client signs authentication data: username + timestamp + ”UPLOAD” + content_hash
8. Client sends to server: *Encrypted file content*, *Encrypted file key*, *Filename*, *Content hash*, *Authentication (username, timestamp, signature)*
9. Server verifies signature authenticity
10. Server stores encrypted file in user’s directory
11. Server stores encrypted file key separately
12. Server updates file database with metadata
13. Server returns success response with file ID

This workflow ensures end-to-end encryption where the file is never exposed in plaintext during transmission or storage.

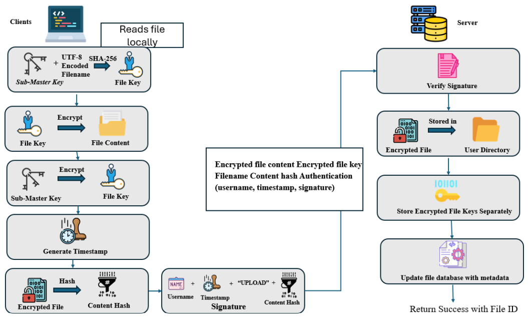


Figure 12. File Upload Process

6.3 File Download Workflow

Below is the file download workflow.

1. Client retrieves list of available files (owned and shared)
2. User selects file to download
3. Client generates timestamp and creates signature: username + timestamp + ”DOWNLOAD” + filename
4. Client sends download request with authentication
5. Server verifies signature and access permissions
6. Server returns encrypted file content and encrypted key
7. For **owned files**: Client decrypts File Key using Sub-Master Key
8. For **shared files**: Client decrypts File Key using Private Key (asymmetric decryption)
9. Client decrypts file content using the obtained File Key
10. Client saves decrypted file to local storage at specified location

The download process handles owned and shared files differently, using the appropriate key decryption mechanism based on file ownership.

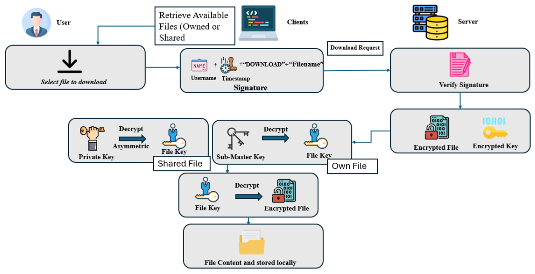


Figure 13. File Download Process

Analysis SMK decryption for owned files while asymmetric keys for shared files. This enforces least-privilege access mitigate potential vulnerabilities of unauthorized sharing (requires valid recipient keys)

6.4 File Adding / Editing Workflow

File Adding and Editing could actually be formulated as a file upload task. **File Adding** is exactly the same as file upload; **File Editing** is mainly implemented as user modifying file on his/her client end, and re-uploading the modified file to replace the original one. We have implemented this file replacement mechanism based on new uploads, enabling file editing functionality.

6.5 File Deletion Workflow

The file deletion enables clients to delete their own files. It follows the following workflow.

1. Client displays list of user's owned files
2. User selects file to delete
3. Client requests confirmation from user
4. Client generates timestamp and creates signature: username + timestamp + "DELETE" + filename
5. Client sends deletion request with authentication
6. Server verifies user is the file owner
7. Server identifies all users with whom the file is shared
8. For each shared user: (i) Server removes their access reference to the file; (ii) Server deletes their encrypted key for this file
9. Server deletes the original encrypted file
10. Server deletes the file's encrypted key
11. Server updates sharing registry to remove file entry
12. Server logs deletion action for audit purposes
13. Server returns success response

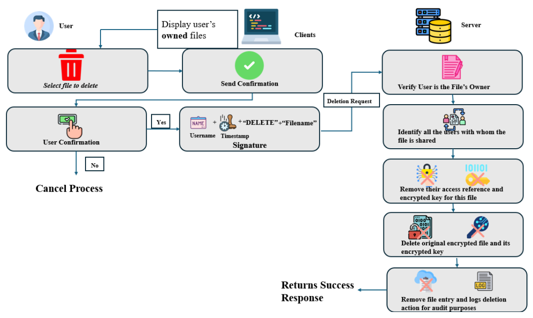

User
Display user's owned files
Clients
Server
Select file to delete
Send Confirmation
Verify User is the File's Owner
Deletion Request
User Confirmation
Yes
NAME
Username
Timestamp
+"DELETE"+"Filename"
Signature
Identify all the users with whom the file is shared
No
Cancel Process
Remove their access reference and encrypted key for this file
Delete original encrypted file and its encrypted key
Returns Success Response
LOG
Remove file entry and logs deletion action for audit purposes

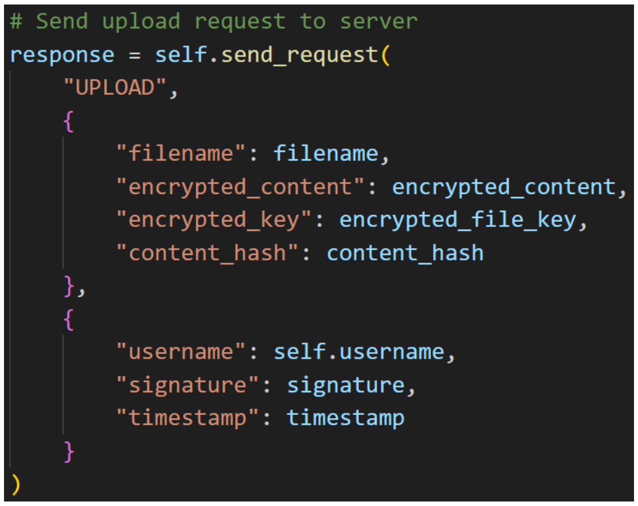
Figure 14. File Deletion Process

Our implemented deletion process ensures that when a file is deleted, all shared access is properly revoked and all related cryptographic material is removed from the system.

Analysis The system deletes all copies of FK (owner + shared users) upon request, which is in compliance with Cascading Cryptoshredding. Also, the deletion Requires RSA-PSS on username + timestamp + "DELETE" + filename for authentication. This prevents unauthorized deletions.

6.6 Technical Implementations

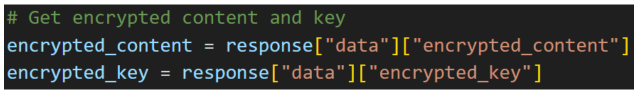
Here are some important steps for code implementation.



```
# Send upload request to server
response = self.send_request(
    "UPLOAD",
    {
        "filename": filename,
        "encrypted_content": encrypted_content,
        "encrypted_key": encrypted_file_key,
        "content_hash": content_hash
    },
    {
        "username": self.username,
        "signature": signature,
        "timestamp": timestamp
    }
)
```

Figure 15. Send upload request to server

For here, we can see that it contains some important data such as encrypted content, encrypted key and digital signature.



```
# Get encrypted content and key
encrypted_content = response["data"]["encrypted_content"]
encrypted_key = response["data"]["encrypted_key"]
```

Figure 16. Get encrypted content and key from server

7 Algorithm & Analysis for File Sharing Module

7.1 File Sharing Workflow

Our system also supports sharing files between different users. Below is the file sharing workflow:

1. Client displays list of user's owned files
2. User selects file to share and specifies target username
3. Client requests target user's public key from server
4. Server returns target user's public key
5. Client generates the File Key for the selected file
6. Client encrypts File Key with target user's public key
7. Client generates timestamp and signature: username + timestamp + "SHARE" + filename + target_username
8. Client sends to server:
   - Filename to share
   - Target username
   - Key encrypted for target
   - Authentication (username, timestamp, signature)
9. Server verifies:
   - Signature authenticity

    - File ownership
    - Target user exists
10. Server stores encrypted key for recipient
11. Server copies the original encrypted file and add to the receiver's folder
12. Server updates sharing registry
13. Server logs sharing action
14. Server returns success response

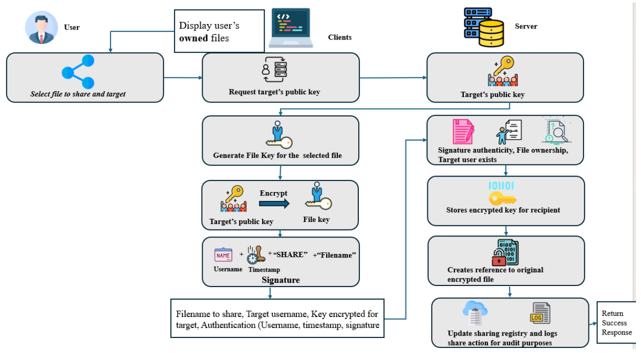

User
Display user's owned files
Clients
Server
Select file to share and target
Request target's public key
Target's public key
Generate File Key for the selected file
Signature authenticity, File ownership, Target user exists
Encrypt
Target's public key
File key
Stores encrypted key for recipient
Username
Timestamp
+"SHARE" +"Filename"
Signature
Creates reference to original encrypted file
Filename to share, Target username, Key encrypted for target, Authentication (Username, timestamp, signature
Update sharing registry and logs share action for audit purposes
Return Success Response

Figure 17. File Sharing Process from Sender

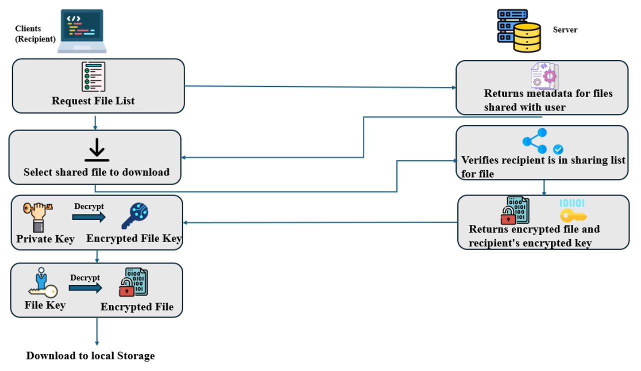

Clients (Recipient)
Server
Request File List
Returns metadata for files shared with user
Select shared file to download
Verifies recipient is in sharing list for file
Private Key
Decrypt
Encrypted File Key
Returns encrypted file and recipient's encrypted key
File Key
Decrypt
Encrypted File
Download to local Storage

Figure 18. File Sharing Process from Receiver

7.2 Sharing Mechanism Design Analysis

Analysis There're several advantages of our implemented file share system:

Maximum Privacy Files stay encrypted always—even when shared. This meets strict standards like GDPR and HIPAA automatically.

Bulletproof Sharing The system uses the recipient's public key to lock files (like a personal digital padlock), and only the intended recipient can unlock.

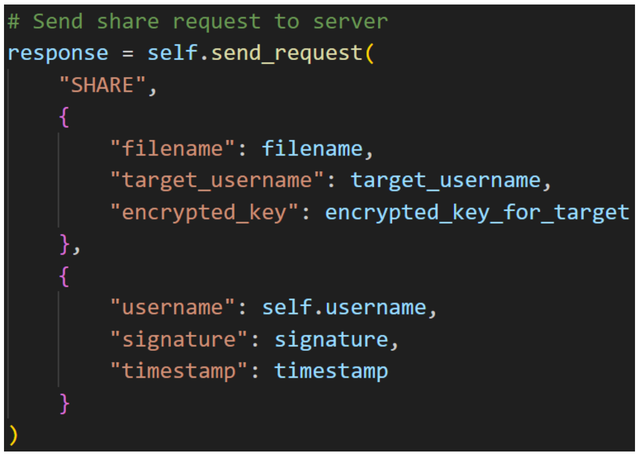
No Forgery Possible Every share request is signed with your private key, which makes it impossible to fake a share request (like a tamper-proof wax seal). This Stops hackers from sharing files without your permission.

Efficiency Sharing doesn't require re-encrypting the file, this makes the system significantly faster than systems that re-encrypt for each user. Also if one user's access is **revoked**, others are unaffected. Changing passwords also won't break existing shares.

Other Built-in Features Automatic logs of who shared what (for audits). Server checks ownership before allowing shares.

7.3 Technical Implementations

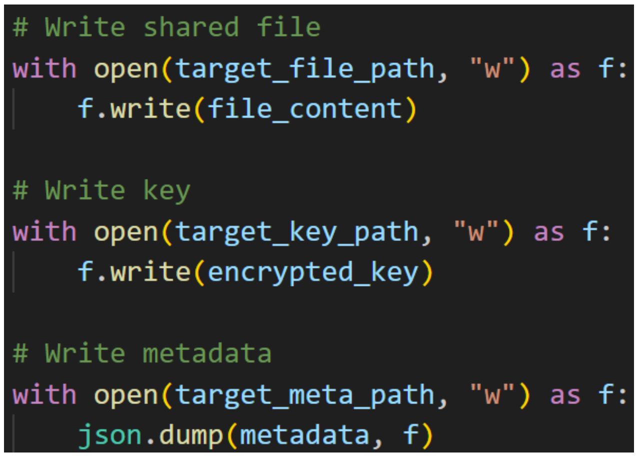
Here are some important steps for code implementation.



```
# Send share request to server
response = self.send_request(
    "SHARE",
    {
        "filename": filename,
        "target_username": target_username,
        "encrypted_key": encrypted_key_for_target
    },
    {
        "username": self.username,
        "signature": signature,
        "timestamp": timestamp
    }
)
```

Figure 19. Send share request to server

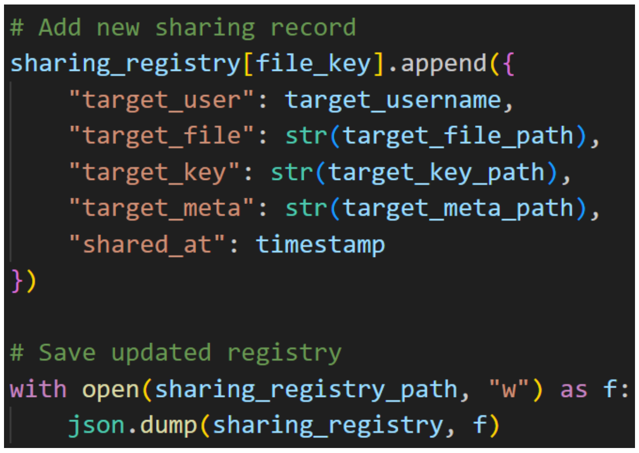
It contains the encrypted key for this share file encrypted by the receiver public key.



```
# Write shared file
with open(target_file_path, "w") as f:
    f.write(file_content)

# Write key
with open(target_key_path, "w") as f:
    f.write(encrypted_key)

# Write metadata
with open(target_meta_path, "w") as f:
    json.dump(metadata, f)
```

Figure 20. Server handle share request

Server will copy the sender file, key and metadata to the receiver server location.



```python
# Add new sharing record
sharing_registry[file_key].append({
    "target_user": target_username,
    "target_file": str(target_file_path),
    "target_key": str(target_key_path),
    "target_meta": str(target_meta_path),
    "shared_at": timestamp
})

# Save updated registry
with open(sharing_registry_path, "w") as f:
    json.dump(sharing_registry, f)
```

Figure 21. Sharing record

The server will use the sharing records stored in the sharing registry to track files. When a user who shared a file wants to delete it, the server will consult this registry to identify all locations where the file has been shared, and then the server will be responsible for deleting all instances of this shared file.

8 Log Audit Module

A Log file is stored on the server, documenting every user's actions who is performing actions on the server. The Log file ensures non-repudiation of users' actions. Only **admin users** can access the log files (read access, not write access to prevent malicious admin user attack). To guarantee the Log file's legitimacy and correctness in repudiating the users' activities on the file system, several properties of the log file are enforced.

1. Only the Admin accounts of the file system are allowed to view the log files, but not the regular users.
2. Even Admins don't have the permission or methods to modify the Log file.
3. When the Admins request the Log file, the content is printed on the console's screen, meaning a copy of the content in the Log file is transmitted to the Admin requesting it, so the Log file's content remains unchanged, and so does its repudiating integrity.
4. The only actions that are allowed to write to the Log file are through system-supported file actions, but the write actions are performed in the background without notifying the user on the console.

9 General Implementation Summary & Other Details

9.1 Libraries Used

Cryptography.io The system architecture leverages four essential libraries that work together to provide secure, cross-platform functionality. Cryptography.io serves as the foundation for security operations, delivering AES-256-GCM encryption/decryption, RSA key generation and signatures, PBKDF2 key derivation, and secure random number generation through a high-level API that implements cryptographic best practices.

JSON For data handling, the JSON library manages serialization needs across multiple domains including user information storage, file metadata tracking, sharing relationship management, and client-server communication, providing a standardized format compatible with all major programming languages.

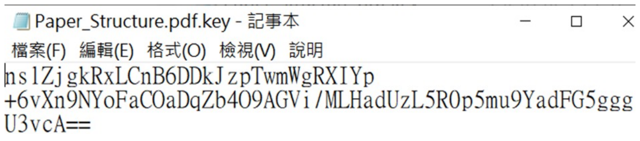
Base64 Base64 encoding ensures secure binary data transport by converting encrypted files, cryptographic keys, signatures, and other binary elements into text format, preventing transmission issues related to binary protocols or character encoding.

Pathlib The Pathlib library complements these security and data components by providing object-oriented filesystem path manipulation that creates user directories, manages file paths securely, and verifies file existence across different operating systems.

9.2 Technical Challenges

9.2.1 Managing Large File Transfers Large files caused memory issues and timeouts during upload/download operations. To solve this problem, we implemented chunked file transfer processing. **Chunked File Transfer Workflow:**

1. Determine optimal chunk size (e.g., 4MB)
2. Calculate total number of chunks for file
3. For each chunk:
   - Extract chunk data from file
   - Upload/download chunk individually
   - Store with sequence number in filename
4. Verify all chunks were transferred successfully
5. Combine chunks into final file
6. Remove individual chunk files
7. Report completion status


Paper_Structure.pdf.key - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

nslZjgkRxLCnB6DDkJzpTwmWgRXIYp
+6vXn9NYoFaCOaDqZb4O9AGVi/MLHadUzL5R0p5mu9YadFG5ggg
U3vcA==

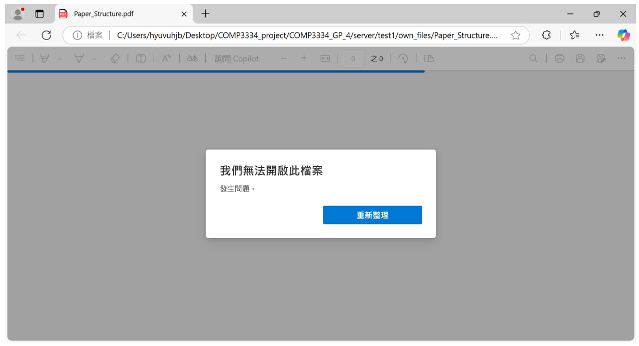
Figure 22. Chunked File Transfer

We also added progress tracking and resumable transfers by tracking completed chunks in a manifest file, allowing resumption of interrupted transfers, and verifying chunk integrity individually. This solution enables the system to handle files of arbitrary size without memory limitations or timeouts.

10 Test Cases

10.1 Test Case 1: Server directly open a stored files

Since the server is modeled as a passive adversary, all files must be encrypted before storage to ensure their confidentiality even if the server is compromised.


Paper_Structure.pdf
檔案
C:/Users/hyuvuhjb/Desktop/COMP3334_project/COMP3334_GP_4/server/test1/own_files/Paper_Structure....
詢問 Copilot

我們無法開啟此檔案

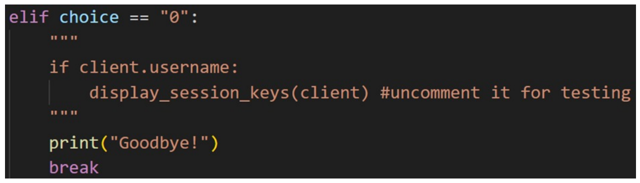
發生問題。

Figure 23. Try to open file stored in the server

We uploaded a PDF file to the server, and then tried to open it directly. We can see that because the file content is encrypted, it cannot be opened.

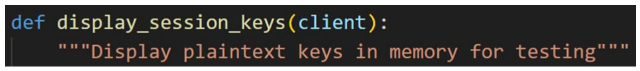
10.2 Test Case 2: Whether keys are encrypted in the server

In order to test the security of keys (whether they are encrypted on the server), we wrote a function to display the plaintext of keys during program execution. We can see that the plaintext of keys during program execution is different from the content of key files saved on the server, which proves that keys are stored encrypted on the server.



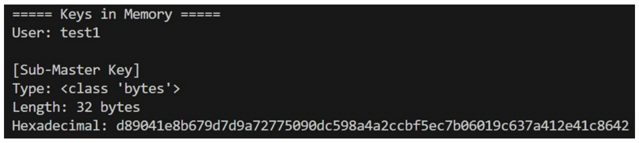
```
elif choice == "0":
    """
    if client.username:
        display_session_keys(client) #uncomment it for testing
    """
    print("Goodbye!")
    break
```

Figure 24. testing function



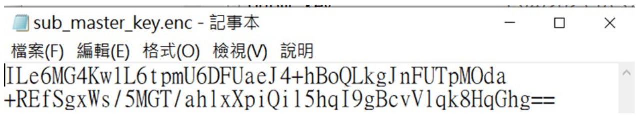
```
def display_session_keys(client):
    """Display plaintext keys in memory for testing"""
```

Figure 25. display key function



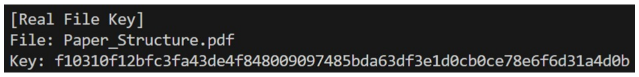
```
===== Keys in Memory =====
User: test1

[Sub-Master Key]
Type: <class 'bytes'>
Length: 32 bytes
Hexadecimal: d89041e8b679d7d9a72775090dc598a4a2ccbf5ec7b06019c637a412e41c8642
```

Figure 26. display sub master key plaintext


sub_master_key.enc - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

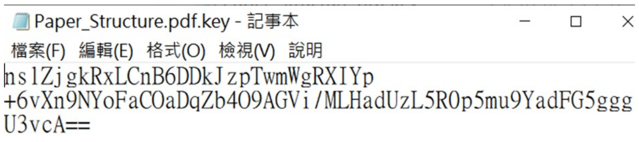
```
ILe6MG4Kw1L6tpmU6DFUaeJ4+hBoQLkgJnFUTpMOda
+REfSgxWs/5MGT/ah1xXpiQi15hqI9gBcvV1qk8HqGhg==
```

Figure 27. stored sub master key



```
[Real File Key]
File: Paper_Structure.pdf
Key: f10310f12bfc3fa43de4f848009097485bda63df3e1d0cb0ce78e6f6d31a4d0b
```

Figure 28. display file key plaintext


Paper_Structure.pdf.key - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

```
ns1ZjgkRxLCnB6DDkJzpTwmWgRXIYp
+6vXn9NYoFaCOaDqZb4O9AGVi/MLHadUzL5R0p5mu9YadFG5ggg
U3vcA==
```

Figure 29. stored file key

11 Conclusion and Future Work

In this project, we successfully designed and implemented a secure, modularized online cloud storage system that addresses critical vulnerabilities in modern cloud storage solutions. By employing a multi-layered encryption architecture, advanced authentication mechanisms, and robust access control measures, our system ensures end-to-end data confidentiality and integrity. Our testing and analysis confirm the system's resilience against the target vulnerabilities listed in our threat model, making the system suitable for environments requiring high levels of security and auditability.

For future improvement, the following points shall be considered:

Advanced Authentication Options We intend to add support for hardware security keys as an additional authentication method and implement biometric authentication integration for enhanced security. We'll also create contextual authentication based on user behavior patterns. The challenge will be maintaining consistent security across different authentication methods while enhancing user experience.

Performance Optimization Further optimizing of encryption and decryption processes are needed to handle the file operation of even larger files efficiently.

User Experience We may also build a user interface and provide more intuitive workflows for non-technical users.

12 Contribution Table

Student ID	Name	Contribution
22107416D	Kent Max Chandra	33.33%
22097007D	WANG Youkang	33.33%
22055747D	Char Cheuk Tung George	33.33%

13 Reference

1. D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "TOTP: Time-Based One-Time Password Algorithm," RFC 6238, Internet Engineering Task Force (IETF), May 2011.
2. K. Moriarty et al., "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, Internet Engineering Task Force (IETF), Jan. 2017.
3. K. Moriarty et al., "PKCS #1: RSA Cryptography Specifications Version 2.2," RFC 8017, Internet Engineering Task Force (IETF), Nov. 2016.