COMP3334 Project Report, Spring 2025
A Secure Multi-User File Management System

Wang Ruijie*, Zhu Jin Shun, Zeng Tianyi, Liu Yuyang
Department of Computing, Hong Kong Polytechnic University

April 13, 2025

Abstract

In this project, we designed and implemented a multi-user file management system in a secure manner in Python with the following functionalities: (1) user account management features with cryptographically secure hash functions for database storage and user authentication, ensuring confidentiality in registration, login, account management, and email-based one-time password (OTP) authentication; (2) file management features including file upload, download, and sharing, with secure storage and retrieval of files using the Advanced Encryption Standard (AES) algorithm; (3) auxiliary features such as access control and log auditing; (4) other software features that prevent common attack methods such as SQL injection and replay attacks. We will further discuss our hypothetical threat model, implementation details of the system, and test cases in this report to demonstrate our results in a comprehensive way.

Contents

*Team coordinator

1 Teamwork declaration

We declare that this project is a joint effort of the entire team with equal contribution of each member. We provide the following breakdown of our work:

Name	Student ID	Workload	Major Contribution
Wang Ruijie	22103808d	25%	email OTPs, algorithm design, code review, LaTeX
Zhu Jin Shun	22101071d	25%	account and file management, algorithm implementation
Zeng Tianyi	22098941d	25%	file management, log auditing, demonstration
Liu Yuyang	22100493d	25%	system testing, report writing

2 Introduction

We are going to present a comprehensive technical review of our project, a secure multi-user management system, from both the theoretical and practical aspects. We commence by Section 3 to demonstrate two hypothetical threat models, which are the bases for our design and implementation of the security features. Next, we move to present the system's architectural framework through three foundational pillars: the generalized software architecture, normalized relational schemas, and critical cryptographic constructs. This tripartite exposition establishes a holistic understanding of the system's macroscopic organization and theoretical underpinnings. Following this comprehensive overview, we employ a layered decomposition to examine each functional component, beginning with abstraction models and progressively descending to concrete implementation specifics. We are also going to discuss the technical difficulties we encountered during the development process and how we solved them, or potential defects requiring further enhancement. At last, we provide a set of test cases to validate the usability and other non-functional requirements on the system, and propose possible directions for future works.

Our artifact is mainly based on algorithms or libraries proposed by famous cryptographers and computer scientists, including but not limited to:

- the Advanced Encryption Standard (AES) algorithm, which is a symmetric encryption algorithm widely used for secure data transmission;
- the Cipher Block Chaining (CBC) mode of operation, which is a method for encrypting data in blocks to enhance security;
- the Rivest-Shamir-Adleman (RSA) algorithm, which is a public-key cryptosystem used for secure data transmission;
- the bcrypt password hashing algorithm, which is designed to be computationally intensive to resist brute-force attacks;
- the One-Time Password (OTP) authentication method, which generates a unique password for each login attempt to enhance security.

We sincerely appreciate their contributions to the field of cryptography and computer science, which have greatly influenced our work.

3 Threat models

Our system consists of two components: a client-side application and a server-side application. The client-side application is responsible for user interaction, while the server-side application handles data storage and processing. Meanwhile, our system is equipped with email-based OTP authentication, which is associated with a third-party email sending service and receiving service provider. Therefore, two corresponding threat models are proposed to analyze the security of our system from both the client-side and server-side perspectives.

3.1 Attackers on the server-side

We assume that attackers on the server-side have full access to view the contents in the database, the technical details of implementation of the system in the both sides, and the communication between the client-side and the server-side applications.

However, they can neither modify the implementation and the deployment of the system, nor execute any operation on the database. They are also assumed to be unable to influence the availability of the system, that is, the server machine will execute our program and return the computational results honestly. Meanwhile, the third-party that provides email services is assume to be a trusted party which will not leak any information to the attackers. Attackers are also unable to conduct any attacks in the social engineering aspect such as shoulder surfing. As for the encryption algorithm, we assume that the attackers cannot afford the computational cost to brute-force any computationally infeasible key space adapted in our system.

3.2 Attackers on the client-side

We assume that attackers on the client-side also acts passively. They are supposed to be able to steal legitimate accounts (know the account password) from other users possibly by means of social engineering but will not modify or destroy the files uploaded by the original user, or modify account information including the current password and email address. They own identical access privileges to all functions implemented as a normal user, but they are unaware of the technical details of the system in the both sides. The client-side device is no longer safe if any information is stored in a local file which is accessible for attackers on the client-side, while we assume that the random-access memory (RAM) of the client-side device is not accessible for attackers. Hence it satisfies our security requirement if any information is in plain text in the RAM but not in a local file.

3.3 Attacking methods

We suppose that all attacking methods that are capable of modifying the system or the database are not applicable in our system. The major attacking methods that we are concerned about include but not limited to SQL injection by inputting malicious SQL statements in the user interface, replay attacks by reusing the password sent to the server-side, or language redundancy analysis on the cipher text.

4 Implementation of cryptographic algorithms and the general system architecture

In our system, three major cryptographic algorithms we followed and implemented are the password hashing algorithm, the Advanced Encryption Standard (AES) algorithm along with Cipher Block Chaining (CBC) mode of operation for file encryption, and the Rivest-Shamir-Adleman (RSA) algorithm for description in file sharing. We introduce them ahead of all other functionalities to emphasize their importance as the foundational pillars of our system. In the following sections, we are going to demonstrate our concerns and the implementation details of these three algorithms in Python. We will continue discussing the method of embedding the cryptographic algorithms into the realization of the functions in Section 5.

4.1 Password hashing algorithm

We include the `bcrypt` library in our system, which is known to be capable of defending the brute-force attack, to implement a secure password hashing algorithm. To hash the password, we use the `gensalt()` function to generate a public random salt and then hash the password with its `hashpw()` function.

```
hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
```

To verify the password, we use the `checkpw()` function to directly compare the hashed input password with the hashed password.

```
bcrypt.checkpw(input_password.encode('utf 8'), hashed_password)
```

The salt used to hash the password is stored in the database along with the hashed password, and it is used to verify the password during login. The salt is generated randomly for each user, and it is unique to each password. This ensures that even if two users have the same password, their hashed passwords will be different due to the different salts to increase the difficulty of brute-force attacks by rainbow tables on the basis of high security assurance of `bcrypt` algorithms.

4.2 Advanced Encryption Standard (AES) algorithm and Cipher Block Chaining (CBC) mode of operation

We combine the AES algorithm with the CBC mode of operation to encrypt and decrypt files and RSA secret keys stored locally in our system. To encrypt any data, we first generate a random 16-byte initialization vector (IV) and a key derived from the user's password using using `bcrypt` hashing with a different `salt` compared to the salt used for storing a password Section 4.1. Our concern is that the hashed password storing in the database is visible to attackers, then once attackers obtain the number of iterations, they can derive a key for AES from the hashed password if the salt is the same. In the following code blocks, we present an example of deriving an AES key from a password which should be entered by the user, and how to encrypt a file with methods offered by `cryptography.hazmat.primitives`.

```
def derive_key(password, salt, iterations=10):
    # Derive a key from the password and salt using PBKDF2
    current_value = password.encode('utf 8')
    for _ in range(iterations):
        current_value = hashlib.pbkdf2_hmac('sha256', current_value, salt.encode('utf 8'), 1)
```

```
    return current_value

def encrypt_file(input_path, key, iv, output_path):
    # Encrypt the file using AES
    if len(key) not in {16, 24, 32}:
        raise ValueError("Key must be 16, 24, or 32 bytes long.")
    try:
        with open(input_path, 'rb') as f:
            data = f.read()

        # Pad the data
        padder = sym_padding.PKCS7(algorithms.AES.block_size).padder()
        padded_data = padder.update(data) + padder.finalize()

        # Encrypt the data
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(padded_data) + encryptor.finalize()

        with open(output_path, 'wb') as f:
            f.write(ciphertext)
        print(f"\nFile encrypted successfully and saved to {output_path}")

    except Exception as e:
        print(f"An error occurred during encryption: {e}")
```

Please note that the length of key are limited to 16, 24, or 32 bytes only. Then, it is both necessary and confidential to hash the password to get the desired length.

4.3 Rivest-Shamir-Adleman (RSA) algorithm

The system will generate an RSA secret key and an RSA public key for each user when they register; the public key will be stored in the database while the secret key will be encrypted by an AES key derived from the password and an IV and stored locally in a local `json` file. The following code block shows the generation of RSA elements during registration with the `cryptography.hazmat.primitives.asymmetric` library included:

```
def register()
  # Omit other registration code here
  # Generate RSA key pair
  private_key = rsa.generate_private_key(
    public_exponent=65537,
      key_size=1024
  )
  iv = os.urandom(16)
  # Encrypt the private key with AES based on the password
  encrypted_private_key = encrypt_key(password, iv, private_key)

  # Derive the public key and send it to the database
```

```
public_key = private_key.public_key()
# Omit the sending code here

# Store the encrypted private key and IV in a local JSON file
filename = "users.json"
user_data = {}
if os.path.isfile(filename):
  with open(filename, 'r') as f:
    user_data = json.load(f)
    user_data[username] = {
    "iv": base64.b64encode(iv).decode('utf 8'),
    "encrypted_private_key": base64.b64encode(encrypted_private_key).decode('utf 8')
  }

with open(filename, 'w') as f:
    json.dump(user_data, f, indent=2)
```

4.4 Combined workflow of cryptographic algorithms

Since we adapt three different cryptographic algorithms in our system, it is essential to illustrate the combination of them in the system. It can also be regarded as the major workflow without real program constructs from the algorithmic perspective as shown in Figure 1 and Figure 2.

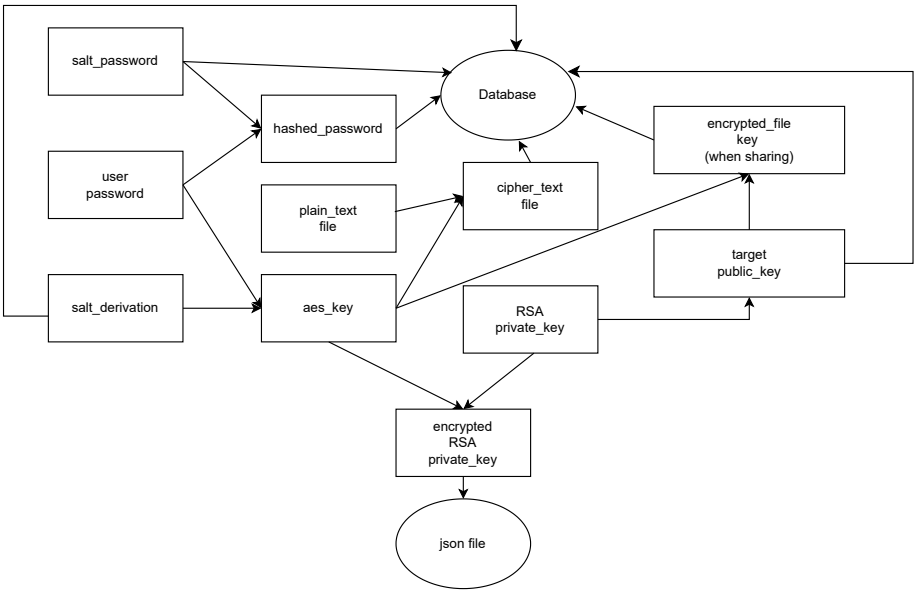

Figure 1: The encryption workflow of the cryptographic algorithms

Now we proceed to the general software architecture and database schemas to show how we implement some basic operations inside the functionality. For the sake of simplicity, we will not repeat them in Section 5.

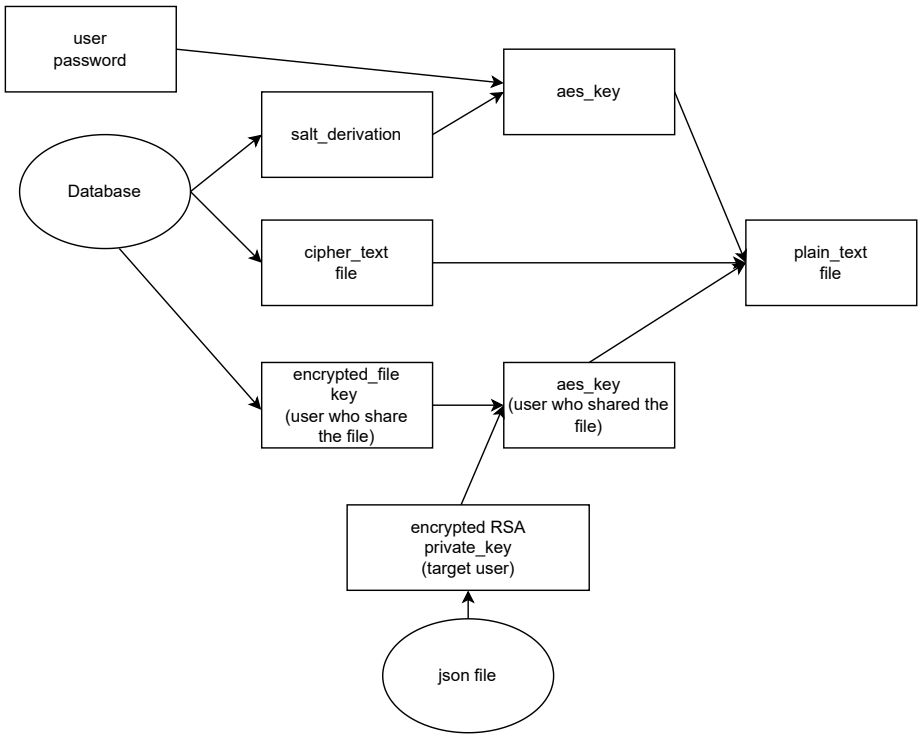

Figure 2: The decryption workflow of the cryptographic algorithms

4.5 Client and server communication

We adapt the popular client-server software architecture with the star topology. The server is the only centralized node that is responsible for storing all the data, sending SQL codes to the database, and processing all the requests from the clients. The server-side application will return different responses (`200, 400, 404`) and `500` status codes to the client-side application based on the request method and the corresponding SQL query. Then the client-side application will process the response and display the results to the user and react accordingly. The client-side application take the role of the user interface, encryption, and decryption, as our threat models declare that the communication channel is compromised. We use the `http` library to realize the inter-process communication between the client-side and the server-side applications.

Once the client-side has received user input which is pre-processed by relevant functions to ensure confidentiality, it will send a query to the server-side application. For example, when the client is trying to send the hashed password to login an account, it executes:

```
connection = http.client.HTTPConnection('localhost', 8000)
login_path = f"/login?username={urllib.parse.quote(username)}
  &password={urllib.parse.quote(hashed_password)}"
connection.request('GET', login_path) # Request method

response = connection.getresponse()
connection.close()
```

```
if response.status == 200:
  log_audit("LOGIN", username, f"User {username} logged in")
  return True, username  # Login successful
else:
  log_audit("LOGIN", username, f"User {username} login failed", status="FAILED")
  print("Login failed. Invalid username or password.")
  return False, None # Login failed
```

Meanwhile, the server-side application will handle the request by:

```
class MyHandler(http.server.SimpleHTTPRequestHandler):

  def do_GET(self):
    if self.path.startswith('/login'):
        self.handle_login()
    # Omit other requests here

  def handle_login(self):
    query = urllib.parse.urlparse(self.path).query
    params = urllib.parse.parse_qs(query)
    username = params.get('username', [None])[0]
    password = params.get('password', [None])[0]

    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()
    cursor.execute('SELECT password FROM users WHERE username = ?', (username,))
    result = cursor.fetchone()
    conn.close()
    # Compare...
```

The server-side application will parse the request and extract the username and password from the query string with the library `urllib`, based on the request method (GET or POST), and we do not give a strict classification of the two request methods. We can see from the example code above that the server function is also responsible to establish the connection with the SQLite database and execute SQL queries, rather than the client-side application establish a connection directly.

One potential defect in our design is that we do not implement a protocol for the client-side application to verify the identity of the server-side application by certificates, or to examine whether the server-side application has been launched when the client-side application is sending requests. Hence, our system might suffer from the man-in-the-middle attacks who can impersonate the server-side application and send malicious responses to the client-side application.

4.6 Database schemas on the server-side

If the server-side application does not find an existing database, it will create a new SQLite database with the name `File_System.db` in the source code directory. One can check the database by using the SQLite command line tool or any SQLite database browser. We provide the schema of the database as follows:

Please note that the attributes in red are the primary keys of the corresponding tables, the (tuples of) attributes in cyan are unique keys, and the attributes in green are foreign keys that reference

Table Name	Attributes
users	id, email, email_status, username, password, salt_password, salt_derivation, public_key, role
files	id, filename, user_id, file_data
file_encryption_iv	id, filename, iv, username, created_at, updated_at
shared_files	owner, file_id, target_username, encrypted_real_key
audit_logs	log_id, timestamp, username, details, status
otps	username, email, top_code, created_at

an another table. All the IDs are auto-incremented integers, the `created_at` and `updated_at` attributes are timestamps, and the `file_data` attribute is a `BLOB` (Binary Large Object) type that stores the encrypted file data. All other attributes are defined in `TEXT` or `VARCHAR` types.

5 Design and implementation of system functionalities

In this section, we are going to discuss the theoretical design, the implementation, and the workflow of each system functionality in detail. We will also discuss technical difficulties we encountered during the development process and how we solved them.

5.1 Register

Theoretical design Firstly, the user interface will prompt the user to enter their username and password. Then, the system will check whether the username already exists or not in the database, as we set the username as a unique key to identify different accounts. If it does, the system will prompt the user to try again to complete the registration. If it does not, the client-side application will hash the password with the `bcrypt` library and transmit the hashed password and the corresponding random salt to the server-side application to store in the database. The system will create an RSA key pair for the user, encrypt the secret key with the AES key derived from the password, and store the encrypted secret key in a local `json` file. We have already demonstrated the algorithmic details of encrypting user passwords and the generation and protection of RSA keys in Section 4.1 and 4.3 respectively, which are the basis of the registration process.

Techinical details We are concerned with the data type transferring processes for RSA key pairs for storage and transmission happening during registration:

```
def serialize_private_key(key: rsa.RSAPrivateKey)  > bytes:
    return key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

def serialize_public_key(key: rsa.RSAPublicKey)  > bytes:
    return key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
```

```
def deserialize_private_key(data: bytes)  > rsa.RSAPrivateKey:
    return serialization.load_pem_private_key(
        data,
        password=None,  # Not encrypted
        backend=default_backend()
    )

def deserialize_public_key(data: bytes)  > rsa.RSAPublicKey:
    return serialization.load_pem_public_key(
        data,
        backend=default_backend()
    )
```

By the functions above, we can serialize the RSA key pairs to `bytes` format and then encoded by `base64` to be transmitted to and stored in the database. We also need to decode the `bytes` format and deserialize them to the original RSA key pairs when we need to use them, such as in the file sharing process.

5.2 Login by password

The user interface will prompt the user to enter their username and password. Then then client-side application will query the database to obtain the hashed password and the salt of the user, and hash the input password with the `bcrypt` library, which has been introduced in Section 4.1. If the database response in not `200 OK` status code, we can assert that the username does not exist in the database, and the system will prompt the user to try again.

5.3 Login by email

Theoretical design If a user has not verified the provided email address, the system will not allow the user to login by email. Once the user has verified the email address and he or she has selected to login by email, the system will first obtain the email address and the email status to verify if the function is application. Moreover, it might result in DDoS attacks if the user repeatedly tries to login by email, hence the system will also check whether there has been an OTP that has not expired in the database. If there is, the system will prompt the user that an OTP has already been sent to the email address and the user should check the email inbox instead of updating the OTP by a new one. After that, the system will generate a random combination of six characters of digits and letters, and send it to the provided user email. The generated OTP is required to be hashed with the `bcrypt` library and stored in the `otps` table in the database along with the email address and the timestamp of the generation. When the user receives the OTP and inputs it, the system will first check whether the current OTP has expired or not by comparing the current timestamp with the timestamp of the generation, to prevent attackers from obtaining the OTP in plain text from the database to log in before the real user. If the OTP is still valid, the system will hash the user input OTP and compare it with the hashed OTP stored in the database. If they match, the system will allow the user to login by email. In the case that an OTP has expired or the hashed input OTP is matched to the hashed OTP stored in the database (the user logs in successfully), the system will request the database to delete the OTP record to prevent replay attacks by reusing the OTP.

Technical details We include the `smtplib` library and `email.mime.text` to develop the email sending function from the program to the third-party email service provider. To generate the OTP, we uses the `choice` function from the `secrets` library to randomly permutate the characters in the string `string.ascii_letters + string.digits` to generate a random combination of six characters. We provide the code here:

```
def generate_otp(length=6):
    chars = string.digits + string.ascii_uppercase
    # Remove '0' and 'O' to avoid confusion
    chars = chars.replace('0', '').replace('O', '')
    return ''.join(secrets.choice(chars) for _ in range(length))
```

We then show how the program send the OTP to a third-party email service provider by `smtp`:

```
def send_otp_to_email(username, email):
    otp = generate_otp()
    sender = "example@gmail.com"
    password = "example password"

    msg = MIMEText(f"From Team 19: Your verification code is {otp}\nValid for 5 minutes.")
    msg['Subject'] = 'COMP3334 Project Login Verification Code'
    msg['From'] = sender
    msg['To'] = email

    try:
        with smtplib.SMTP_SSL('smtp.gmail.com', 465) as server:
            server.login(sender, password)
            server.sendmail(sender, [email], msg.as_string())
    except Exception as e:
        print(f"Error sending email: {e}")
    # Synchronize the OTP with the database...
```

5.4 List uploaded files

To list all files uploaded by the user, it suffices to query the database to obtain the file names and their corresponding IDs where the username is the same as the current user. The system will then display the file names and IDs in a list format. If there is no file uploaded by the user, no content will be displayed.

5.5 Upload a file

Theoretical design The client-side application will prompt the user to enter the file path of the file to be uploaded on the local device, hence it is necessary to check whether the file exists or not. If it does, the system will ask the user to enter the password to derive the AES key and encrypt the data read from the file. Before that, the client-side application must retrieve the previous salt value for derivation uploaded to the database instead of the salt to encrypt the password itself. The IV used to encrypt will be uploaded to the `file_encryption_iv` table in plain text in the database along with the file name, username, and timestamps, while the encrypted file data is stored in the `files` table. For each user, the system should set that the file name is unique, and if the user tries

to upload a file with the same name, the system will block the operation and prompt the user to try again after deleting the existing file.

Technical details It is possible that attackers may use the file path to conduct SQL injection attacks. We apply parameter binding to prevent SQL injection attacks by using the ? placeholder in the SQL query. We illustrate the processing of the user input to the parameter binding in the code block below:

```
# Client–side application sending the request
print("Sending the encrypted file data...")
headers = {
  'Content Type': 'application/octet stream',
  'Filename': os.path.basename(file_path),
  'User ID': username  # Username identifies the user
}
connection.request('POST', "/upload_file", encrypted_data, headers)

# Server–side application after receiving the request
conn.execute('BEGIN')
# Insert the file, associating it with the user
cursor.execute('INSERT INTO files (filename, file_data, user_id) VALUES (?, ?, ?)',
  (filename, file_data, user_id))
conn.commit()
```

Other that, it is essential to the validity of the file name, which may be a tool for conducting directory traversal attacks. we implement a general method to check whether the name of the provided file comply with the standard format of file names in the system:

```
def detect_legal_filename(filename):
    # Step 1: Decode URL
    decoded = urllib.parse.unquote(filename)
    # Step 2: Unification
    decoded = decoded.replace("\\", "/")
    # Step 3: Extract pure name

    # Defense 1: Directory traversal attack
    if re.search(r'(\.\.|%2e%2e)', filename, re.IGNORECASE):
        print("Directory traversal attack detected!")
        return False

    # Defense 2: Black–list characters
    illegal_chars = r'[<>:"/\\|?*\x00 \ x1F]'
    if re.search(illegal_chars, filename):
        print("Illegal characters found!")
        return False

    # Defense 3: Retain device names (Windows)
    base_name = os.path.splitext(filename)[0].upper()
    reserved = {"CON", "PRN", "AUX", "NUL", "COM1", "LPT1"}
```

```
    if base_name in reserved:
        print("System retained characters detected!")
        return False

    # Defense 4: Limit length
    if len(filename) > 255:
        print("File name is too long!")
        return False

    # If all checks pass, the filename is valid
    return True
```

5.6 Download a file

The system will prompt the user to enter the file name to be downloaded, and examine the validity of user input in the same way discussed in Section 5.5. Since the AES key to decrypt the uploaded encrypted file data is never stored in the database, the client-side application will request the user to enter the password again to derive the AES key along with the salt value obtained from the database. Then, the client-side application will download the encrypted file into a local folder with a given output path named by the username, and decrypt the downloaded file with the derived AES key and the IV stored in the database.

5.7 Delete a file

To delete a file, it suffices to check whether the target file name exists in the database and is uploaded by the user If it does, the system will delete the file data from the `files` table and the IV from the `file_encryption_iv` table in the database. In the case that the file has been shared with other users, the system will also delete the corresponding record in the `shared_files` table. The methods of defending SQL injection is the same as the ones in Section 5.5.

5.8 Share a file

Theoretical design A user (the owner of a file) can indicate the username of the target user to share the file with, and the system will check whether the target user exists in the database or not. If it does, the system will check whether the file has been shared with the target user or not. If it has, the system will prompt the user to try again. If not, a tuple of the file ID, the target username, the owner username, and the encrypted derived AES key will be inserted into the `shared_files` table in the database. Since we adapt the strategy of password-based key derivation for the AES key, it is not appropriate to assume the target user knows the password of the owner user to derive the desired AES key by himself or herself. Hence, the client-side application will retrieve the target user's public key from the database and encrypt the derived AES key with the public key. It ensures that no one other than the target user can decrypt the AES key with his or her own secret key. We also prohibit the owner user to share a file with himself or herself, and the system will prompt the user to try again if it is the case.

Technical details We present how the derived AES key of the file owner is encrypted with the target user's public key in the code block below:

```
# Retrieve the target user's public key
connection = http.client.HTTPConnection('localhost', 8000)
connection.request('GET', f"/get_public_key?username={urllib.parse.quote(target_user)}")
response = connection.getresponse()
if response.status != 200:
    print("Failed to retrieve the public key. Please check the username.")
    return

target_user_public_key_bytes = response.read()
# Remember to decode the public key bytes
# Deserialize the public key
target_user_public_key = deserialize_public_key(target_user_public_key_bytes)
connection.close()

# Retrieve the owner user's salt for derivation
connection = http.client.HTTPConnection('localhost', 8000)
connection.request('GET', f"/get_salt_derivation?username={urllib.parse.quote(username)}")
response = connection.getresponse()
if response.status != 200:
    print("Failed to retrieve the salt derivation. Please check your username.")
    return
salt_derivation = response.read().decode('utf 8 ')
connection.close()

# Derive the AES key from the password
real_key = derive_key(password, salt_derivation)

# Encrypt the AES key with the target user's public key
encrypted_real_key = target_user_public_key.encrypt(real_key, asym_padding.OAEP(
    mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
    algorithm=hashes.SHA256(),
    label=None
))

#Upload the tuple to the database...
```

5.9 Read shared files

Theoretical design As explained in 5.8, the client-side application of the target user can retrieve the shared file information from the `shared_files` table in the database. Since the AES key belongs to the owner user and the target user have no knowledge about the owner user's password, the target user can only obtain the decrypt the AES key (of the owner user) by his own RSA secret key. However, the target user's RSA secret key is encrypted by the AES key derived from the target user's password, and the target user needs to enter the password to derive the AES key to decrypt the RSA secret key stored in the local `json` file. After that, the client-side application can retrieve the encrypted file data and the IV from the database in accordance with the owner username and the file ID. The downloading process is basically identical to downloading a file uploaded by the

target user, and in addition, we realize the function of reading the plain text of the shared file on the command-line interface if it in `txt` format.

Technical details We present the method of decrypting the RSA secret key of the target user with given target user's username and password in the code block below:

```
def decrypt_my_secret_key_from_json(username, password):
    with open('users.json', 'r') as f:
        user_data = json.load(f)
        user_entry = user_data[username]
        iv = base64.b64decode(user_entry["iv"])
        encrypted_private_key = base64.b64decode(user_entry["encrypted_private_key"])
        private_key = decrypt_key(password, iv, encrypted_private_key)
        return private_key
```

Meanwhile, due to some mechanisms of Windows, the data retrieved from the database may be in `bytes` format, and we need to decode it to `utf-8` format before displaying it on the command-line interface:

```
file_name = selected_file['file_name'].decode('utf 8')
  if isinstance(selected_file['file_name'],bytes) else selected_file['file_name']
file_owner = selected_file['owner'].decode('utf 8')
  if isinstance(selected_file['owner'], bytes) else selected_file['owner']
file_encrypted_real_key = base64.b64decode(selected_file['encrypted_real_key'])
```

5.10 Reset the password

Theoretical design Resetting the password is similar to register a new account without changing the username. If the hashed current password is identical to the output of the hash function on the new password, the system will reject the operation and prompt the user to try again. The system will also check whether the new password is identical to the old one in plain text, and if it is, the system will also reject the operation.

Technical details A problem we encountered during the implementation of the password reset function is that after it is executed, the files uploaded to the server are still encrypted by the AES key derived from the old password, and the system will not be able to decrypt them with the new password. Meanwhile, the RSA secret key is also encrypted by the AES key derived from the old password. We need to decrypt all these previously encrypted elements and re-encrypt or derive them with the new password.

5.11 Update the email

Theoretical design Once users have already registered and logged in, they can update their email address by selecting the corresponding option in the user-selection menu. The system should ensure that the new email address is not identical to the existing one for the same account by sending a request to query the current email address of the user from the database. Other that that, the email address input should not violate the standard format by passing a regular expression check. Please we allow multiple users sharing the same email address. The outcome of the update operation is to store the email address in the `users` table. However, it does not mean that the

user can start to login by email from then on; he or she has to select to verify the email, and the `email_status` of the user is still `False`.

Technical details We adapt a regular expression to check the format of the email address with the `re` library as shown below:

```
regex = r'\b[A Za z0 9._%+ ]+@[A Za z0 9. ]+\.[A Z|a z]{2,}\b'
  if not re.fullmatch(regex, new_email):
    print("Invalid email. Please check the email address.")
```

5.12 Verify the email

After the user has updated the email address, he or she can select to verify the email address by sending a one-time password (OTP) to the new email address. If the hashed user input OTP is matched to the hashed OTP stored in the database, the system will update the `email_status` of the user to `True` and allow the user to login by email. Other design and implementation details are similar to the ones in Section 5.3.

5.13 Export audit logs

Theoretical design Whenever an important operation is activated (an operation is selected by a user from the user-selection menu), the `audit_log` function should receive a list of parameters to describe the execution of the operation. They are `action_type`, `username`, `details`, and `status`. Afterwards, the function should send the parameters to the server-side application, where a corresponding handler will insert them into the `audit_logs` table in the database. Moreover, only an admin account is allowed to export the audit logs recorded in the database, and hence if the user is not an admin, the option of exporting logs will not be displayed in the user-selection menu. The exported log files will be stored in the admin's client-side device.

Technical details We include the `logging` library in Python to implement the log function, which has already defined all parameters required and the output format.

5.14 Log out

To log out the user, it suffices to break the loop of the user-selection menu to the initial menu before the user logs in. No connection need to be closed since there will no ongoing connection with the server-side application when a user is already in the user-selection menu to select to log out. A log will be added by calling the `log_audit` function with the `LOGOUT` action type.

5.15 Exit the program

Exiting the program of the client-side application is also a simple function. When a user selects to exit the program, the client-side applications will first log out the user and directly return the main function to terminate the running program. No connection need to be closed as well.

6 Test cases

6.1 Key protection

We classify our key protection into three categories: passwords, AES keys, and RSA secret keys. We have introduced how algorithms are used to generate or protect them in Section 4.1, 4.2 4.3. If an attacker gains physical access to the client-side computer or the database and attempts to steal it in order to check for passwords, secret keys, and AES keys, they would face significant challenges. All passwords, secret keys, and AES keys are stored in the database as encrypted cipher text, ensuring that no plaintext versions are available. The only way to access the plain text is to log in and download the appropriate files. You can see the case from the database contents:

```
sqlite> SELECT * FROM users;
1|kingsoonchu@gmail.com|True|1|\$2b\$12\$mQeBjlHwpfQjBuRk2kdFQOm
tU9F3LSKehilUW40xYYp8KIuUSPMei|\$2b\$12\$mQeBjlHwpfQjBuRk2kdFQO|
\$2b\$12\$a5ID3DOZ2oep0anI4bcJn.|     BEGIN PUBLIC KEY
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDdrLZFysmqiKjz5r0yX0uwMon1
bBVa3JzhpJYafOsw88YYIFVjpE7qSsV5aziXmnSq+0kyOHVlVVEsWNSCT4cd1Paa
4ZQHVG072d5PtGO5396lVDn3hKoHYIuO8uFUlqhYDn8IdrKcRzeIWNu2DY87pko7
LEgYbINvtADuG6enSQIDAQAB
     END PUBLIC KEY
```

Compared to the public key in plain text, it is easy to see that the and the password is also hashed (the sixth column). When sharing a file to another user, the AES key decrypting the encrypted file data is also encrypted in the `shared_file` table (the fourth column), which can be decrypted by the target user's RSA secret only.

```
sqlite> SELECT * FROM shared_files;
1|1|0|2jo9kHULwPEBwGEb0qccDI0twch9gH/z46q/2JAlg6wm76VLunHvFbXaGIiR
MnxhepCAbby8SQrniidxcaoK9bx+sNNJbKqc+rUs1xWPGbgxR24hIXxKjaK05trjy3
2YMJQT4OrDB/hd0jVvENDgAKHzhwKwkm5S/GWKDYb1Mec=
```

If an attacker attempts to retrieve the RSA private key stored locally, he can not decrypt it unless he obtained the user's password in plain text. We can see from the local `json` file to check the encrypted private key:

```
"kingsoon": {
    "iv": "aL5kd22vv+DgyWajvwH1yw==",
    "encrypted_private_key": "9J4ikx92oR4FiM1O4yfnjjy9XS6wO6JFAD2
    N9qgIo1QfvGgZwvBiddyJRx2RSVybYaNPGlew1OLHKXmdo8yQ4WPHKFOgJrA+
    mcHdiV4ZzZD8xWQenwdwwbVyparE12Lc1jiU2Mg7zlIl7bOMJDJbrWPmjmml
    9R2tc6QWeKi6YM+FJO80ccf9NAGtygbJR6Yq8ccFKoOIu8QOKQeUuDJsz1v3
    X054cXQu/8yXXA4Hl2UJWvnnM0hUNImcG1DyBwAI6yXprbSxk2kI+/HvlRiY
    ab+bgyXS3KZ3iI/Om5SLdUl9ONGFlbB261Ejn+/wXn1JwwY0wsc6RtDmb/Vp
    GfGJxgIcm0QtQLf7wze4lVWlfALd2FPbVLADksY4DbfUqGoeg2gMcXsf4LRb
    # Too long, omit the rest
  }
```

In conclusion, the security of the entire crypto-system is completely based on the confidentiality of the user's password, which is memorized by the user and not stored in anywhere else in the system. The attacker just can not derive and obtain it if the user does not leak his or her password.

6.2 SQL injection

When SQL injection occurs during the input stage, an attacker can exploit vulnerabilities on the client side during login by entering:

```
Username: admin' --
Password: anything
```

However, the system defends against this attack on the server side by using a parameterized query. The use of `?` as a placeholder in the SQL statement, along with passing the actual values as a tuple (username,), ensures that user input is treated as data rather than executable SQL code. This means that even if an attacker attempts to inject SQL, it will be interpreted as a string literal and matched against the data in the database. If the input is incorrect, the attacker will not gain access to any functions. This strategy is consistently applied throughout the server-side code, effectively preventing SQL injection. Additionally, the code incorporates input sanitization. By utilizing parameterized queries, inputs are inherently sanitized. The database driver ensures that the input is properly escaped, preventing any injected SQL from being executed. Moreover, the code includes error handling, which means that if an unexpected condition occurs (such as attempting to fetch a non-existent user), the system will fail gracefully. This prevents the exposure of database details and mitigates the risk of further manipulation.

Also, illegal characters will be detected for filenames. When an attacker tries to upload files with invalid file name, e.g., `../file.txt`, which is used in directory traversal attack, the system will detect it and return detected message to defend the attack, as explained in Section 5.5

7 Future works

We propose several possible directions for future works to improve the usability and security of our system, which are not implemented in our current version. Most of them are related to non-functional requirements such as performance, usability, and security. We also think of additional features that can be added to our system to enhance functionality and user experience.

7.1 User interface

For simplicity rather that user experience, we use the command line interface (CLI) to implement the user interface of our system, which may result in various border cases regarding the user input. We can further improve the accessibility of our system by implementing a graphical user interface (GUI) with the help of libraries such as `tkinter` or `PyQt`.

7.2 Client-server communication

We do not implement protocols for stable and secure communication between the client-side and the server-side applications, and the implementation of request and response is not strictly classified into GET or POST methods, as explained in 4.5 We might revise our implementation of client-server communication by following the modern paradigm such as the transport layer security (TLS) protocol to defend external attackers.

7.3 Refinement of the design of database schemas

As discussed in Section 4.6, our schemas are generally applicable without strict constraints on foreign keys and data types whose implementation might enhance the performance of the database.

For example, there is an interchangeable use of `user_id` and `username`, or `file_id` and `filename`, among different table, which may lead to ambiguity in writing complex SQL queries. We can simplify the database schemas by normalization to deduce the redundant attributes or deduce the size of each table to improve the maintainability and the amount of unit operations in each execution of SQL queries.

7.4 Software architecture

Though the major software architecture is appropriate for our system from a high-level perspective, we can further improve the modularity of our system by separating the client-side and server-side applications into different modules. For example, we can implement the model-view-controller (MVC) pattern to separate the logics of different functions in the client-side application and the server-side application respectively.

7.5 Key derivation

In our practice, we derive an AES key from a user's password with a randomly generated salt with only 1 iteration. We may increase the number of iterations to enhance the security of the key derivation process, or include `PBKDF2` library to implement a more stable key derivation function.

7.6 Recovery and cancellation of operations

Our system includes functions to delete an uploaded file from the database, which is irreversible. Since the log mechanism is well developed and records the critical operations comprehensively, it is easy to develop a function to recover the deleted file by restoring the file data into a cache from the log to reduce the cost of an undesired irreversible action accidentally taken by the user. We can also implement a function to cancel the last operation taken by the user, which is not implemented in our current version.

8 Conclusion

We sincerely thank you for your time and effort to read our project report and all other project materials. We hope that our project can provide a secure and user-friendly file management system for users to manage their files and accounts in a safe manner. Best wishes to you and have a nice day!