



Trinity College Dublin

Coláiste na Tríonóide, Baile Átha Cliath

The University of Dublin

CSU33032 Advanced Computer Networks

Project #2: Securing the Cloud

Abdelaziz Abushark, Std# 20332134

April 10th, 2023

Contents

1. Introduction	2
2. Requirements	2
3. Summary of the system architecture	2
4. Protocol	5
4. Design:	7
5. Implementation	8
5.1 User Interface – GUI	8
5.2 Functionality	9
User Authentication and Adding	10
Browsing Files	11
File Encryption and Uploading	12
File Downloading and Decryption	13
Status Updates and SSL usage	14
6. Code	15
7. References	20

1. Introduction

We have been assigned a project with an interesting objective - create and implement a proper key management system for your application that uses public-key certificates and enables users to add and delete users from groups as well as share files securely. You are free to utilize any open-source cryptographic libraries when developing your application for desktop or mobile platforms.

2. Requirements

As part of this project, it is imperative to create an avant-garde cloud storage application that effectively caters to the diverse and dynamic needs of renowned cloud storage platforms like Dropbox, Box, Google Drive, and Office365, among others. To achieve this, the application must ensure that every file uploaded to the cloud is safeguarded against unauthorized access, such that only authorized individuals, who are part of the exclusive "Secure Cloud Storage Group," can decrypt uploaded files. Additionally, the application should allow any member of the group to upload encrypted files to the cloud service, with an assurance of complete encryption of the files, thereby rendering them unintelligible to any unauthorized users. This necessitates the design and implementation of a key management system that utilizes public-key certificates, which empowers users of the system to share files securely, as well as add or remove users from the group as needed.

Furthermore, the project demands the implementation of the application for either a desktop or mobile platform, with a preference for the utilization of open-source cryptographic libraries to ensure an enhanced level of security for the users. It is also necessary to ensure that the application's design incorporates user-friendliness and ease of use, without compromising the high level of security that is required. Finally, the application must provide seamless integration with popular cloud storage platforms, enabling users to upload and access files with ease and convenience. To accomplish these requirements, it is crucial to conduct extensive research and development, as well as rigorous testing and optimization of the application, to ensure a reliable and efficient system that meets the expectations of the users.

3. Summary of the system architecture

The proposed cloud storage application architecture consists of a secure system that ensures that all files uploaded to the cloud are encrypted and accessible only to members of the exclusive "Secure Cloud Storage Group." The system employs a key management system that utilizes public-key certificates to enable secure sharing of files and the addition or removal of users from the group. The application is designed for either desktop or mobile platforms and incorporates open-source cryptographic libraries for enhanced security. The design of the system prioritizes user-friendliness and ease of use while maintaining a high level of security. The system seamlessly integrates with popular cloud storage platforms, allowing users to upload and access files with ease and convenience. Rigorous research, development, testing, and optimization are required to ensure a reliable and efficient system that meets the expectations of the users.

Data encryption is frequently used in communications-related activities to prevent anyone other than the intended recipients from understanding the contents of text, messages, or even files. In addition, it enables recipients to demonstrate that a message originated from a specific sender and wasn't intercepted or altered.

Tools for public key cryptography are primarily reliant on end-to-end encryption. Under typical conditions, symmetric encryption can be used to do this. The issue with this strategy is that the sender wishes to use a channel to transmit a message to the intended recipient. Yet there are several unreliable and nosy intermediates in this channel. In other words, it is continuously the target of outsider eavesdropping and danger. Thus, it is essential to encrypt the message before transmission and have it decrypted when the payload data is received by the recipient. This ensures that even if the communication is intercepted in the middle of transmission, only the intended recipient will be able to understand its contents. This method is known as symmetric encryption because it uses identical encryption keys on both sides which is shown in figure 1.

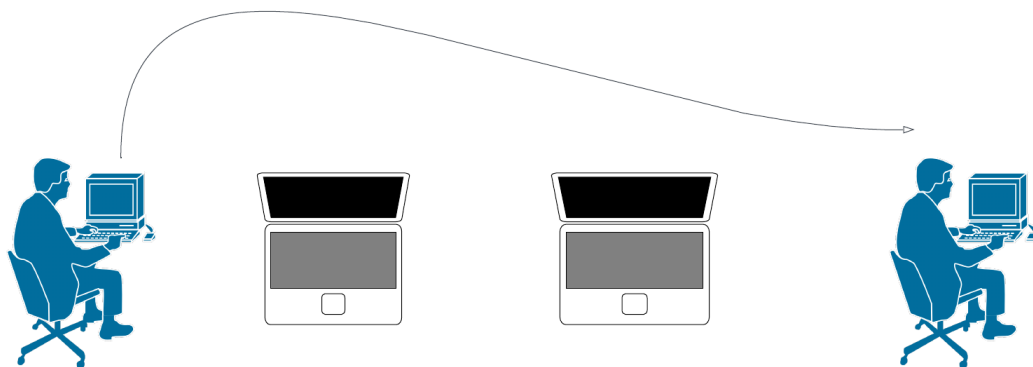


Figure [1]

Symmetric cryptography, however, doesn't address the issue of someone being able to simply eavesdrop and steal the symmetric key being passed from the sender to the receiver, therefore this doesn't even come close to providing sufficient confidence on the security of the transmission. This enables the listener to carry out middle-of-the-message attacks by intercepting the communication and changing its content. Information contained in the payload could be read, changed, and deleted by the attacker before being sent to the recipient. As a result, the recipient will instead receive the incorrect message which is shown in figure 2.

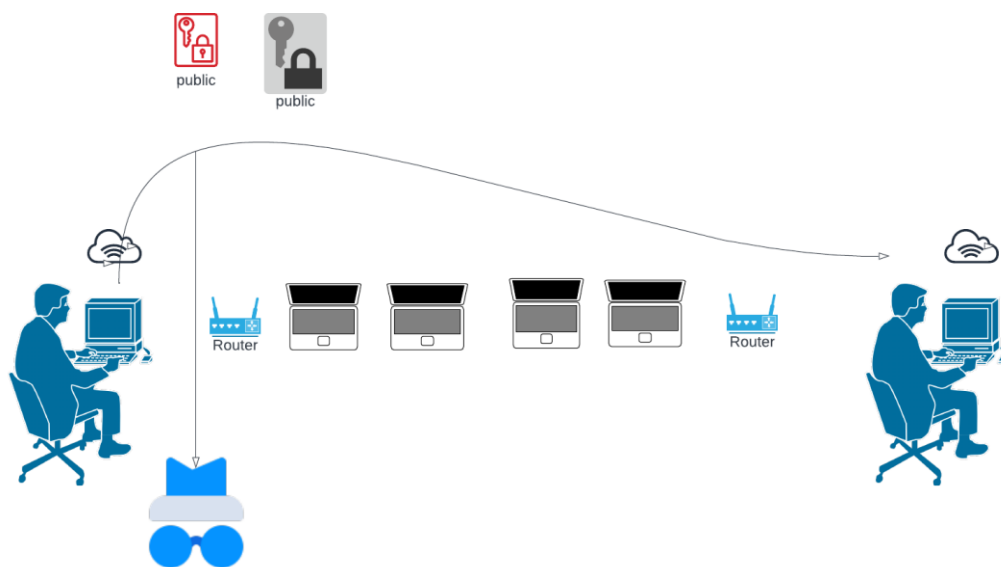


Figure [2]

Another encryption must be used to make the symmetric key resistant to the assaults to make up for this weakness. Encrypting the symmetric key with a public key and decrypting it with the associated private key is one option that can be used. The goal is to have the sender use their public key to encrypt the keys or communications while the recipient broadcasts their public key. Just the intended can read the contents as a result, while intermediaries can only access the metadata, which includes the subject line, dates, sender, and receiver which is shown in figure 3.

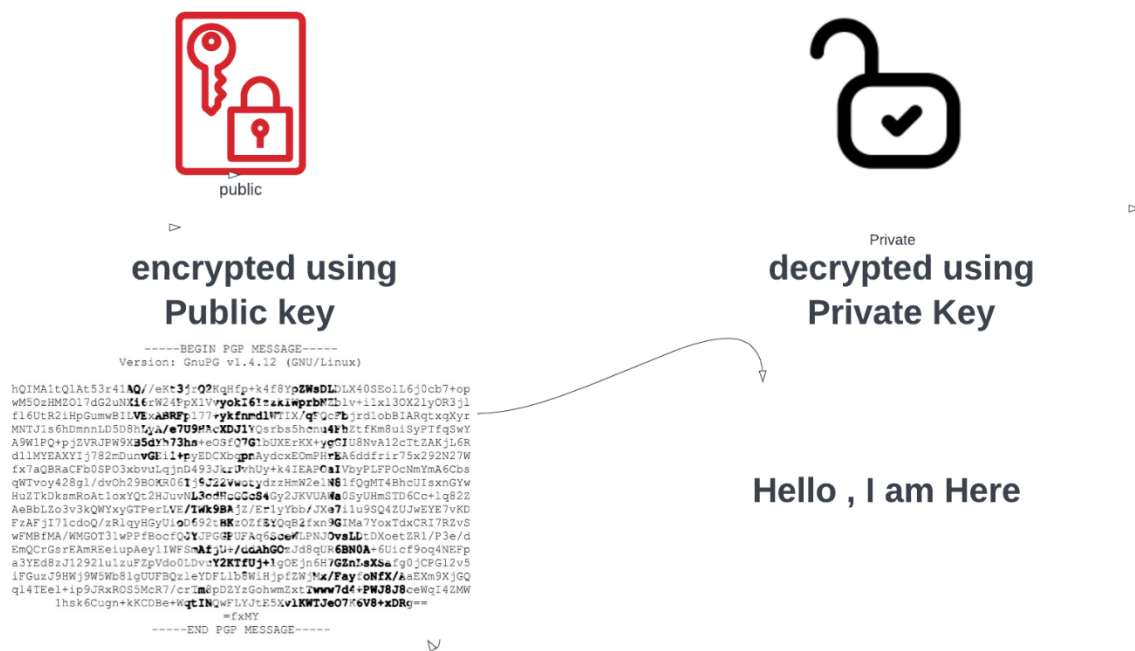


Figure [3]

In addition it uses RSA algorithm, a widely used public-key cryptosystem for secure data transfer, is the primary basis of the encryption process. To encrypt and decrypt messages, RSA uses the mathematical features of huge prime numbers. The secure cloud storage.py file, which contains the SecureCloudStorage class, is where the encryption procedure is carried out. Using the PyCryptodome library, this class defines methods for encrypting and decrypting data using the RSA algorithm.

The User class is set up in the user.py file to hold user data like an email address, login credentials, and a public-private key pair. With a default key size of 2048 bits, the generate key pair() method creates a fresh pair of RSA keys. While the private key is required for decryption, the public key is utilized for encryption. The system is made more secure by associating each user with a special key pair.

A mechanism for managing users, encrypting and decrypting data, and uploading and downloading files to and from Google Drive is implemented by the SecureCloudStorage class. The user email and the data to be encrypted are inputs for the encrypt data() method. It imports the user's public key and uses the PKCS1 OAEP cipher to encrypt the data with the RSA algorithm. The encrypted data and the user's email are inputs into the decrypt data() method. It uses the same cipher to decrypt the data after importing the user's private key.

4. Protocol

The code consists of three files: `main.py`, `user.py`, and `secure_cloud_storage.py`. The application uses the tkinter library to create a graphical user interface (GUI) for a secure cloud storage application that interacts with Google Drive. The code also employs the Python Cryptography Toolkit (PyCrypto) library to perform encryption and decryption.

Here is a high-level description of the protocol design:

1. `main.py`

The GUI is created using the tkinter library, with themed tkinter (ttk) and ttkthemes for a better look and feel. The GUI consists of the following components:

- Email input field: for the user to enter their email address.
- Authenticate & Add User button: when clicked, it runs the `authenticate_and_add_user` function, which authenticates the user with Google Drive API and adds the user to the `SecureCloudStorage` instance.
- File path input field: for the user to input the file path or select a file using the Browse button.
- Browse button: when clicked, it runs the `browse_file` function, which opens a file dialog to select a file, and inserts the selected file's path into the `file_path_entry` field.
- Upload File button: when clicked, it runs the `upload_file` function, which uploads and encrypts the selected file to Google Drive, and displays the uploaded file's ID in the `status_label`.
- File ID input field: for the user to input the file ID of a file they want to download from Google Drive.
- Download File button: when clicked, it runs the `download_file` function, which downloads and decrypts the specified file from Google Drive, and displays the decrypted file's path in the `status_label`.
- Status label: displays the status of the application.

Email input field:

```
email_label = ttk.Label(main_frame, text='Email:')
email_label.grid(column=0, row=0, padx=5, pady=10, sticky=tk.W)
email_entry = ttk.Entry(main_frame)
email_entry.grid(column=1, row=0, padx=5, pady=10, sticky=(tk.W, tk.E))
```

Authenticate & Add User button:

```
authenticate_button = ttk.Button(main_frame, text='Authenticate & Add User',
                                command=authenticate_and_add_user)
authenticate_button.grid(column=0, row=1, columnspan=2, pady=10)
```

File path input field and Browse button:

```
file_path_label.grid(column=0, row=2, padx=5, pady=10, sticky=tk.W)
file_path_entry = ttk.Entry(main_frame)
file_path_entry.grid(column=1, row=2, padx=5, pady=10, sticky=(tk.W, tk.E))
browse_button = ttk.Button(main_frame, text='Browse', command=browse_file)
browse_button.grid(column=0, row=3, columnspan=2, pady=10)
```

Upload File button:

```
upload_button = ttk.Button(main_frame, text='Upload File', command=upload_file)
upload_button.grid(column=0, row=4, columnspan=2, pady=10)
```

File ID input field:

```
file_id_label = ttk.Label(main_frame, text='File ID:')
file_id_label.grid(column=0, row=5, padx=5, pady=10, sticky=tk.W)
file_id_entry = ttk.Entry(main_frame)
file_id_entry.grid(column=1, row=5, padx=5, pady=10, sticky=(tk.W, tk.E))
```

2. user.py

- The User class represents a user with their email, Google Drive API credentials, and RSA public and private keys.
- The init method initializes the user object. If public and private keys are not provided, it calls the generate_key_pair method to generate a new key pair.
- The generate_key_pair method generates a new RSA key pair using the Crypto.PublicKey.RSA module from the PyCrypto library. The default key size is 2048 bits, but it can be changed by passing a different key_size value.

init method and generate_key_pair method:

```
class User:
    def __init__(self, email, credentials, public_key=None, private_key=None):
        self.email = email
        self.credentials = credentials
        self.public_key = public_key
        self.private_key = private_key

        if not self.public_key or not self.private_key:
            self.generate_key_pair()

    def generate_key_pair(self, key_size=2048):
        key = RSA.generate(key_size)
        self.private_key = key.export_key().decode()
        self.public_key = key.publickey().export_key().decode()
```

3. secure_cloud_storage.py

- The SecureCloudStorage class is responsible for managing users and their data in the application. It maintains a dictionary of users, keyed by their email addresses.
- The add_user and remove_user methods are used to add and remove users from the dictionary.
- The encrypt_data method takes the data and user_email as input, imports the user's public key, creates a PKCS1_OAEP cipher, and encrypts the data using the user's public key.
- The decrypt_data method takes the encrypted data and user_email as input, imports the user's private key, creates a PKCS1_OAEP cipher, and decrypts the encrypted data using the user's private key.

- The `upload_to_drive` method reads the data from the specified file, encrypts it using the `encrypt_data` method, writes the encrypted data to a new file, creates a Google Drive API service using the user's credentials, and uploads the encrypted file to Google Drive.
- The `download_from_drive` method creates a Google Drive API service using the user's credentials, downloads the specified encrypted file from Google Drive, decrypts the data using the `decrypt_data` method, and writes the decrypted data to a new file.

`add_user` and `remove_user` methods:

```
class SecureCloudStorage:
    def __init__(self):
        self.users = {}

    def add_user(self, user):
        self.users[user.email] = user

    def remove_user(self, user_email):
        if user_email in self.users:
            del self.users[user_email]
```

`encrypt_data` and `decrypt_data` methods:

```
def encrypt_data(self, data, user_email):
    if user_email not in self.users:
        raise ValueError('User not found.')
```

Overall, the code creates a GUI for a secure cloud storage application that allows users to authenticate with Google Drive, upload and download encrypted files, and manage users and their encryption keys using the `SecureCloudStorage` class. The encryption and decryption process uses RSA public key cryptography with the `PyCrypto` library to ensure secure data storage and retrieval.

4. Design:

The code you provided consists of three parts - `main.py`, `secure_cloud_storage.py`, and `user.py`. The `main.py` file is the main entry point of the program, which imports modules, defines functions, and creates a graphical user interface (GUI) using the `Tkinter` library. The `secure_cloud_storage.py` file contains functions related to secure cloud storage, such as adding a user, uploading and downloading files, and authentication. The `user.py` file defines a `User` class that represents a user with an email, credentials, and keys for secure communication.

In `main.py`, the first lines of code import several modules that are used in the program, including `tkinter`, `ttk`, `ThemedTk`, `filedialog`, `SecureCloudStorage`, `User`, and `InstalledAppFlow`. The `get_user_credentials()` function defined in this module reads the client secret file and initiates a flow to authenticate the user.

Next, the `authenticate_and_add_user()` function is defined, which obtains the user's email and credentials, creates a `User` object, and adds it to the `SecureCloudStorage` object (`scs`). This function is called when the user clicks the 'Authenticate & Add User' button in the GUI.

Similarly, the `browse_file()`, `upload_file()`, and `download_file()` functions are defined to handle file selection, uploading, and downloading, respectively. These functions are triggered by the corresponding buttons in the GUI.

The Tkinter library is used to create a GUI in `main.py`. The `ThemedTk` class is used to create a themed window, and a `ttk.Frame` is used to group GUI elements together. The GUI consists of several labels, entry widgets, and buttons to input user credentials, browse files, and upload and download files. The status label at the bottom of the GUI displays the current status of the program.

In `secure_cloud_storage.py`, the `get_user_credentials()` function is again defined to obtain user credentials. The `authenticate_and_add_user()`, `browse_file()`, `upload_file()`, and `download_file()` functions are defined to interact with Google Drive to authenticate users, select, and transfer files.

In `user.py`, the `User` class is defined with the constructor that initializes the email, credentials, public key, and private key of the user. If the public and private keys are not already set, then the `generate_key_pair()` method is called to create a new key pair for the user. The method uses the `PyCrypto` library to generate a new RSA key pair with a default key size of 2048 bits.

In summary, the code you provided is a Python program that provides a GUI for a secure cloud storage system. It defines functions to authenticate users, select, upload and download files, and generate public and private keys for users. The code makes use of the Tkinter library for GUI elements, and the `PyCrypto` library for key generation. The `SecureCloudStorage` class is the central point of interaction with Google Drive in which all explained in the figure below.

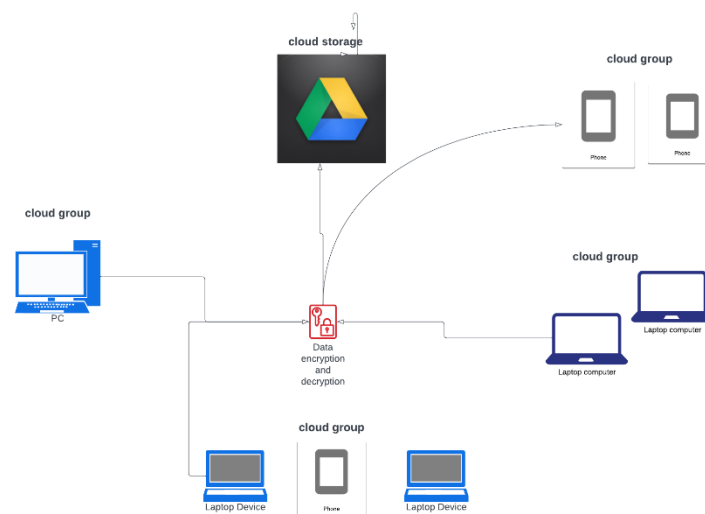


Figure [4]

5. Implementation

5.1 User Interface – GUI

The user interface of this program is built using the Tkinter library and themed using the `ttkthemes` package. It provides a simple, clean, and user-friendly experience for interacting with

the Secure Cloud Storage application. The interface consists of a form with various input fields and buttons that allow users to perform tasks like authenticating and adding a user, browsing for files, and uploading or downloading files to and from their Google Drive.

The UI features an email input field for the user to input their email address, followed by an "Authenticate & Add User" button that authenticates the user and adds them to the SecureCloudStorage instance. Next, there is a file path input field along with a "Browse" button that enables the user to select a file from their local storage. An "Upload File" button is available to upload the selected file to Google Drive in an encrypted format.

A separate input field is provided for entering the File ID of a previously uploaded file, which is needed when the user wants to download and decrypt the file. To complete this action, the user clicks on the "Download File" button. Finally, a status label is displayed at the bottom of the UI to provide real-time feedback on the various operations being performed by the application.

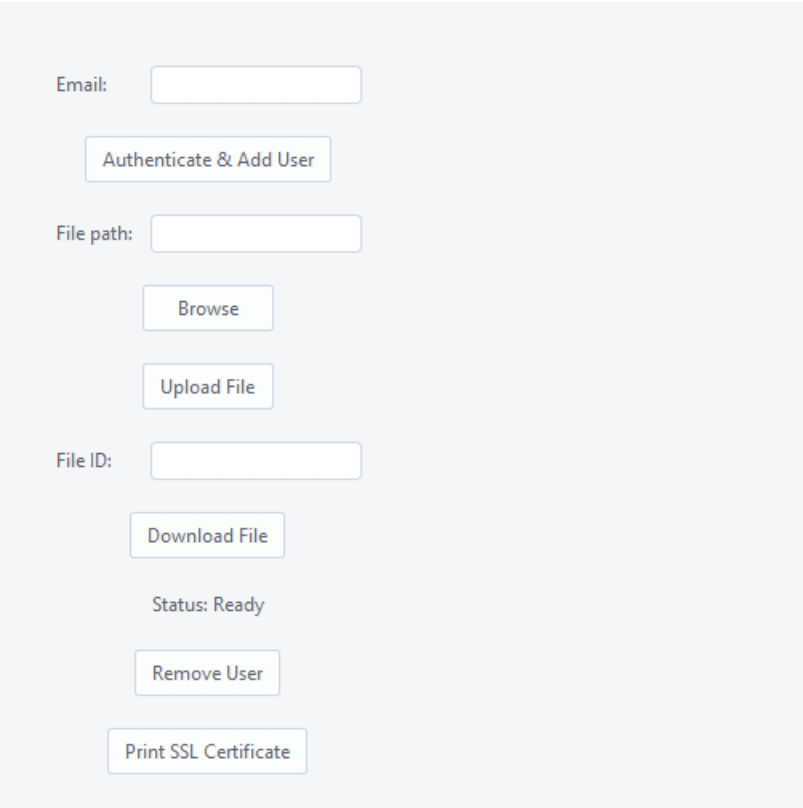
The image shows a web application window titled "Key management system". The interface is a light gray form with several input fields and buttons. At the top, there is an "Email:" label followed by a text input field. Below this is a button labeled "Authenticate & Add User". Next is a "File path:" label followed by another text input field. Below the file path field is a "Browse" button. This is followed by an "Upload File" button. Then, there is a "File ID:" label followed by a text input field. Below the File ID field is a "Download File" button. At the bottom of the form, there is a "Status: Ready" label, a "Remove User" button, and a "Print SSL Certificate" button. The window has standard OS controls (minimize, maximize, close) in the top right corner.

Figure [5]

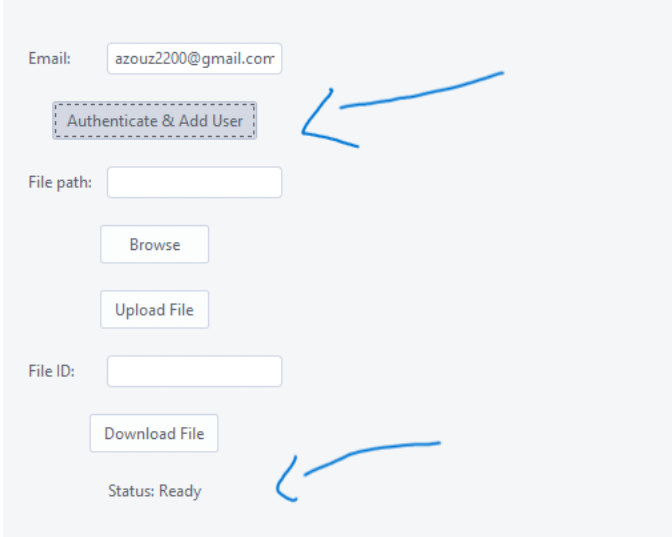
5.2 Functionality

To ensure clear understanding of the functionalities and protocol, I will first provide a detailed explanation in written paragraphs. This will allow for a comprehensive understanding of the different elements and processes involved. However, I also recognize that visual aids can greatly aid in the comprehension and retention of information. Therefore, we will supplement the written explanation with relevant images to help illustrate and visualize the different functionalities and protocol. This combined approach of written explanations and visual aids will enable a more effective and efficient learning experience, ensuring that you have a thorough understanding of the topic at hand.

User Authentication and Adding

The user authentication functionality is initiated when the "Authenticate & Add User" button is clicked. The `authenticate_and_add_user` function is called, which retrieves the user's email from the email input field and obtains their Google credentials using the `get_user_credentials` function. The `get_user_credentials` function employs the `InstalledAppFlow` from the `google_auth_oauthlib.flow` library to facilitate the authentication process using the specified client secret file and required Google Drive scopes.

Once the user's credentials are obtained, a new `User` object is created using the email and credentials. The user's public and private RSA key pair is generated within the `User` class if not provided during instantiation. The new user is then added to the `SecureCloudStorage` instance, which maintains a dictionary of all authenticated users and then you do the same to add any use to your secure cloud group.



The screenshot shows a web application interface with the following elements:

- Email:** A text input field containing the email address `azouz2200@gmail.com`.
- Authenticate & Add User:** A button with a dashed border, highlighted by a blue arrow pointing to it from the right.
- File path:** A text input field.
- Browse:** A button located below the File path field.
- Upload File:** A button located below the Browse button.
- File ID:** A text input field.
- Download File:** A button located below the File ID field.
- Status: Ready** A status indicator at the bottom, highlighted by a blue arrow pointing to it from the right.

Figure [6]

```
C:\Users\Abdelaziz Abushark\Desktop\cloud4>python main.py
Please visit this URL to authorize this application: https://accounts.google.com/o/oauth2/auth?response_type=code&cli
ent_id=153409804727-grbcm3b9cjhdio2cukerou52c0nd4r.apps.googleusercontent.com&redirect_uri=http%3A%2F%2Flocalhost%3
A62696%2F&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive&state=mVrmHoY9B7EWA9R3UUKr2DJxEWGelg&access_type=offl
ine
```

Figure [7]

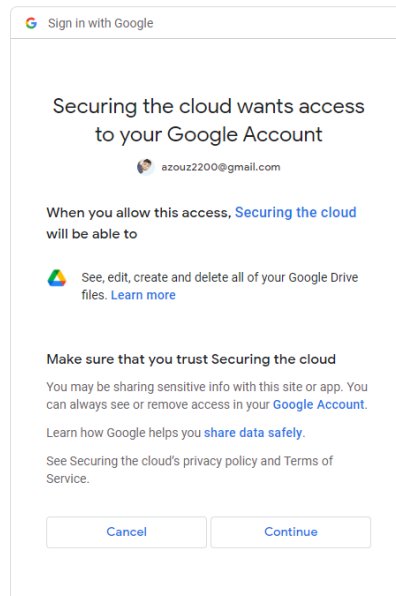


Figure [8]

The authentication flow has completed. You may close this window.

Figure [9]

User azouz2200@gmail.com authenticated and added to the group .

Figure [10]

Browsing Files

When users click the "Browse" button, the `browse_file` function is called. This function utilizes the `filedialog.askopenfilename` method from the Tkinter library to open a file dialog allowing users to navigate their local file system and choose a file for uploading. The chosen file's path is then inserted into the file path input field.

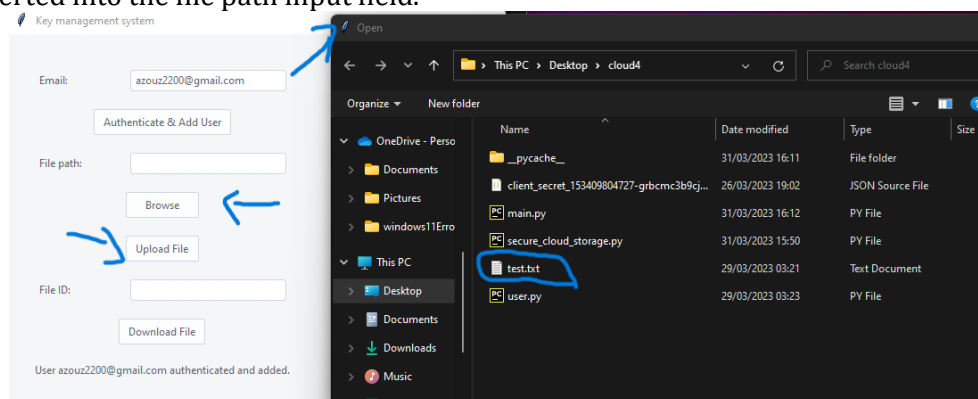


Figure [11]

Email:

File path: ←

Figure [12]

File Encryption and Uploading

The file encryption and uploading process begins when the "Upload File" button is clicked, calling the `upload_file` function. This function retrieves the selected file path and the user's email from the input fields. It then reads the selected file's content as binary data and encrypts it using the `encrypt_data` method of the `SecureCloudStorage` class. This method employs the PKCS1_OAEP encryption scheme from the `Crypto.Cipher` library and the user's public key to encrypt the data.

The encrypted file is saved with an "encrypted_" prefix, and the program uploads it to the user's Google Drive using the `upload_to_drive` method. The `MediaFileUpload` class from the `googleapiclient.http` library is used to perform the file upload, and the Google Drive API (`googleapiclient.discovery.build`) is utilized to create the file in the user's Drive. After a successful upload, the File ID is returned and displayed in the status label.

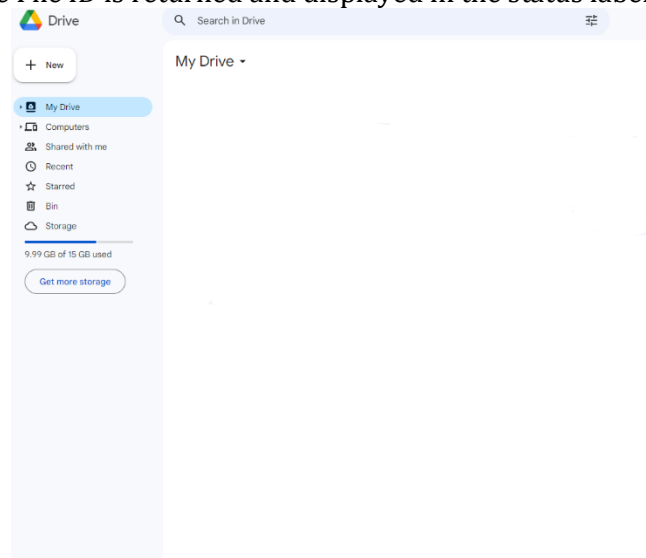


Figure [13]

File ID:

File uploaded with ID: 18/tHf0Tj2V15UtomCrYur2jq_eed3uBa ←

Figure [14]

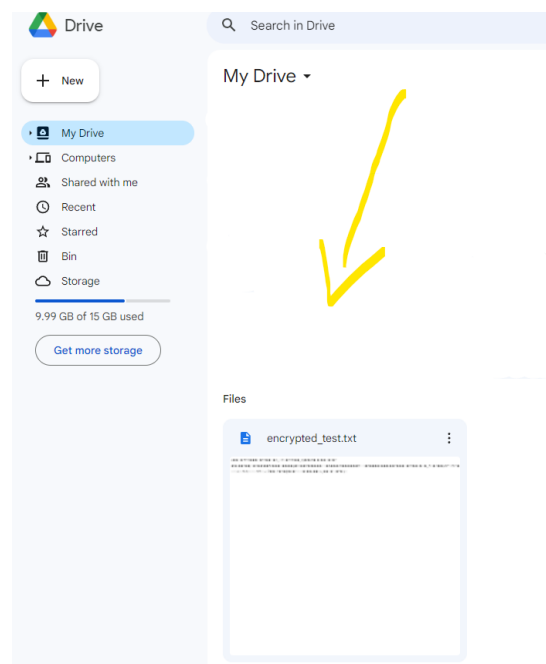


Figure [15]

File Downloading and Decryption

To download and decrypt a file, users need to input the corresponding File ID into the File ID input field and click the "Download File" button. This triggers the `download_file` function, which retrieves the File ID and user's email from the input fields. The `download_from_drive` method is then called to download the encrypted file from Google Drive using the Google Drive API and the `get_media` method.

The `decrypt_data` method of the `SecureCloudStorage` class is used to decrypt the downloaded file. This method uses the PKCS1_OAEP decryption scheme from the `Crypto.Cipher` library along with the user's private key to decrypt the data. The decrypted file is saved with a "decrypted_" prefix and the status label is updated to indicate a successful download and decryption.

```
File ID: "18JtHfOtj2V15UtomCrYur2jq_eed3uBa"
```

Figure [16]

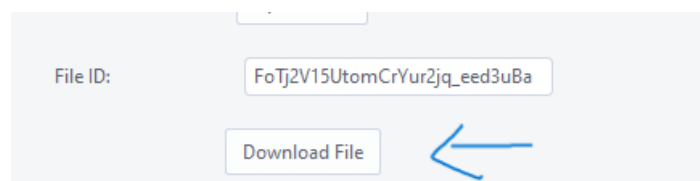


Figure [17]

__pycache__	31/03/2023 16:11	File folder	
client_secret_153409804727-grbcmc3b9cj...	26/03/2023 19:02	JSON Source File	1 KB
encrypted_test.txt	31/03/2023 16:24	Text Document	1 KB
main.py	31/03/2023 16:12	PY File	4 KB
secure_cloud_storage.py	31/03/2023 15:50	PY File	3 KB
test.txt	29/03/2023 03:21	Text Document	1 KB
user.py	29/03/2023 03:23	PY File	1 KB

Figure [18]: before downloading the file and decrypting it

File ID:

FoTj2V15UtomCrYur2jq_eed3

Download File

File downloaded and decrypted: decrypted_test.txt

Figure [19]

__pycache__	31/03/2023 16:11	File folder	
client_secret_153409804727-grbcmc3b9cj...	26/03/2023 19:02	JSON Source File	1 KB
decrypted_test.txt	31/03/2023 16:27	Text Document	1 KB
encrypted_test.txt	31/03/2023 16:24	Text Document	1 KB
main.py	31/03/2023 16:12	PY File	4 KB
secure_cloud_storage.py	31/03/2023 15:50	PY File	3 KB
test.txt	29/03/2023 03:21	Text Document	1 KB
user.py	29/03/2023 03:23	PY File	1 KB

Figure [20]: After downloading the file and decrypting it

Status Updates and SSL usage

The status label serves to communicate the application's operations and outcomes to the user. Throughout the various processes, the config method is used to update the status label's text. Examples of status updates include successful user authentication, file uploads and downloads, decrypted file paths, and error messages that inform users of any issues encountered during the operations as well as remove users from the secure cloud group.

The Google Drive API server and the client are connected securely using the SSL certificate. The SSL certificate details of the Google Drive API server are printed by the print ssl certificate() method to a file called SSLCertificate.txt. The certificate, which provides details about the server's identification, is used to confirm the server's legitimacy and rule out the possibility of a malicious third-party eavesdropping on client and server communications. This contributes to protecting the privacy and accuracy of the data being sent between the client and the server. The DER cert to PEM cert() method is used to convert the certificate from DER format to PEM format, which is a more legible format, while creating the SSL certificate using the Python ssl module.

Status: Ready

File uploaded with ID: 18JtHf0Tj2V15UtomCrYur2jq_eed3uBa

File downloaded and decrypted: decrypted_test.txt

Figure [21]

Remove User

User azouz2200@gmail.com removed.

Figure [22]

Print SSL Certificate

SSL certificate written to file: SSLCertificate.txt





	main.py	05/04/2023 18:39	PY File	5 KB
	secure_cloud_storage.py	03/04/2023 14:17	PY File	3 KB
	SSLCertificate.txt	05/04/2023 19:09	Text Document	3 KB
	test.txt	02/04/2023 18:32	Text Document	1 KB

Figure [23]: shows that I can manually choose to print the SSL Certificate whenever I want whether after uploading the file into drive and encrypt it or when I download and decrypt the file as well as it can do it automatically while uploading and downloading the file without me pressing the button as shown in the demo.

6. Code

imports any necessary libraries.

defines user authentication, file browsing, file uploading, and file downloading features.

creates labels, entries, and buttons for user interaction in the GUI using tkinter and ttkthemes.

runs the GUI's primary event loop.

```
KeyMang.py

import tkinter as tk
from tkinter import ttk
from ttkthemes import ThemedTk
from tkinter import filedialog
from secure_cloud_storage import SecureCloudStorage
from user import User
from google_auth_oauthlib.flow import InstalledAppFlow

def get_user_credentials():
    client_secret_file_path = 'user.json'
    flow = InstalledAppFlow.from_client_secrets_file(client_secret_file_path, scopes=['https://www.googleapis.com/auth/drive'])
    return flow.run_local_server(port=0)

scs = SecureCloudStorage()

def authenticate_and_add_user():
    email = email_entry.get()
    credentials = get_user_credentials()
    user = User(email, credentials)
    scs.add_user(user)
    status_label.config(text=f'User {email} authenticated and added.')

def browse_file():
    file_path = filedialog.askopenfilename()
    file_path_entry.delete(0, tk.END)
    file_path_entry.insert(0, file_path)
```

```

def upload_file():
    file_path = file_path_entry.get()
    encrypted_file = 'encrypted_' + file_path.split('/')[3]
    user_email = email_entry.get()
    file_id = scs.upload_to_drive(user_email, file_path, encrypted_file_path)
    status_label.config(text=f'File uploaded with ID: {file_id}')

def download_file():
    file_id = file_id_entry.get()
    user_email = email_entry.get()
    decrypted_file = 'decrypted_' + file_path_entry.get().split('/')[3]
    scs.download_from_drive(user_email, file_id, decrypted_file)
    status_label.config(text=f'File downloaded and decrypted: {decrypted_file}')

root = ThemedTk(theme="arc")
root.title(' Key management system')

main_frame = ttk.Frame(root, padding="30 30 30 30")
main_frame.grid(column=0, row=0, sticky=(tk.W, tk.E, tk.N, tk.S))
root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)

email_label = ttk.Label(main_frame, text='Email:')
email_label.grid(column=0, row=0, padx=5, pady=10, sticky=tk.W)
email_entry = ttk.Entry(main_frame)
email_entry.grid(column=1, row=0, padx=5, pady=10, sticky=(tk.W, tk.E))

authenticate_button = ttk.Button(main_frame, text='Authenticate & Add User',
command=authenticate_and_add_user)
authenticate_button.grid(column=0, row=1, columnspan=2, pady=10)

file_path_label = ttk.Label(main_frame, text='File path:')
file_path_label.grid(column=0, row=2, padx=5, pady=10, sticky=tk.W)
file_path_entry = ttk.Entry(main_frame)
file_path_entry.grid(column=1, row=2, padx=5, pady=10, sticky=(tk.W, tk.E))

browse_button = ttk.Button(main_frame, text='Browse', command=browse_file)
browse_button.grid(column=0, row=3, columnspan=2, pady=10)

upload_button = ttk.Button(main_frame, text='Upload File', command=upload_file)
upload_button.grid(column=0, row=4, columnspan=2, pady=10)

file_id_label = ttk.Label(main_frame, text='File ID:')
file_id_label.grid(column=0, row=5, padx=5, pady=10, sticky=tk.W)
file_id_entry = ttk.Entry(main_frame)
file_id_entry.grid(column=1, row=5, padx=5, pady=10, sticky=(tk.W, tk.E))

download_button = ttk.Button(main_frame, text='Download File', command=download_file)
download_button.grid(column=0, row=6, columnspan=2, pady=10)

status_label = ttk.Label(main_frame, text='Status: Ready')
status_label.grid(column=0, row=7, columnspan=2, pady=10)
root.mainloop()

```


defines the SecureCloudStorage class, which has methods for adding and removing users and keeps a dictionary of User objects.
uses the PKCS1 OAEP algorithm from the Crypto library to implement encryption and decryption techniques.
outlines ways to use the Google API Client library to extract and decrypt data from Google Drive as well as upload and upload encrypted items to Google Drive.

Secure_cloud_storage.py

```
import json
import base64
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA
from googleapiclient.discovery import build
from googleapiclient.errors import HttpError
from googleapiclient.http import MediaFileUpload, MediaIoBaseDownload
from io import BytesIO
```

```
class SecureCloudStorage:
```

```
    def connection(self):
        self.users = {}
```

```
    def add_user(self, user):
        self.users[user.email] = user
```

```
    def remove_user(self, user_email):
        if user_email in self.users:
            del self.users[user_email]
```

```
    def encrypt_data(self, data, user_email):
        if user_email not in self.users:
            user = self.users[user_email]
            recipient_key = RSA.import_key(user.public_key)
            cipher_rsa = PKCS1_OAEP.new(recipient_key)
            encrypted_data = cipher_rsa.encrypt(data)
            return encrypted_data
```

```
    def decrypt_data(self, encrypted_data, user_email):
        if user_email not in self.users:
            raise ('User not found.')
```

```
    user = self.users[user_email]
    private_key = RSA.import_key(user.private_key)
    cipher_rsa = PKCS1_OAEP.new(private_key)
    data = cipher_rsa.decrypt(encrypted_data)
    return data
```

```
    def upload_to_drive(self, user_email, file_path, encrypted_file):
        if user_email not in self.users:
            raise ('User not found.')
```

```

encrypted_data = self.encrypt_data(data, user_email)

with open(encrypted_file, 'wb') as f:
    f.write(encrypted_data)

print (f'File ID: "{file.get("id")}"')

return file.get('id')

def download_from_drive(self, user_email, file_id, decrypted_file):
    if user_email not in self.users:
        raise ValueError('User not found.')

    user = self.users[user_email]
    try:
        request = service.files().get_media(fileId=file_id)
        file = request.execute()
        decrypted_data = self.decrypt_data(file, user_email)

        with open(decrypted_file_path, 'wb') as f:
            f.write(decrypted_data)
            print(f'Decrypted file saved to {decrypted_file}')

    except HttpError as error:
        print(f'An error occurred: {error}')
        return None

```

User.py

establishes a user class that symbolizes a user with an email, OAuth login information, and RSA public and private keys.

Contains a technique to create a fresh key pair in the event that the public and private keys are not supplied during startup.

from Crypto.PublicKey import RSA

class User:

```

def api ( self, email, credentials, public_key=None, private_key=None):
    self.public_key = public_key
    self.private_key = private_key
    self.email = email
    self.credentials = credentials

```

```

def generate_key(self, key_size=2048):
    key = RSA.generate(key_size)
    self.private_key = key.export_key().decode()
    self.public_key = key.publickey().export_key().decode()

```

Public Key certificates and SSL usage:

```
from Crypto.PublicKey import RSA
from cryptography import x509
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization

class Certificate:

    def __init__(self, email, credentials, public_key=None, private_key=None, certificate=None):
        self.public_key = public_key
        self.private_key = private_key
        self.email = email
        self.credentials = credentials
        self.certificate = certificate

        if not self.public_key or not self.private_key:
            self.generate_key_pair()

    def load_certificate(self, certificate_pem):
        self.certificate = x509.load_pem_x509_certificate(certificate_pem, default_backend())

    def get_public_key_from_certificate(self):
        if self.certificate:
            return self.certificate.public_key()
        else:
            return None

def print_ssl_certificate(filename='SSLCertificate.txt', host='www.googleapis.com', port=443):
    context = ssl.create_default_context()

    with socket.create_connection((host, port)) as sock:
        with context.wrap_socket(sock, server_hostname=host) as ssock:
            cert = ssl.DER_cert_to_PEM_cert(ssock.getpeercert(binary_form=True))

    with open(filename, 'w') as f:
        f.write(cert)

    print(f"SSL certificate written to file: {filename}")

print_cert_button = ttk.Button(main_frame, text='Print SSL Certificate', command=lambda:
print_ssl_certificate(filename='SSLCertificate.txt'))
print_cert_button.grid(column=0, row=9, columnspan=2, pady=10)
```

7. References

- Advanced Computer Networks Slides.
- <https://www.geeksforgeeks.org/rsa-algorithm-cryptography/>
- <https://www.javatpoint.com/rsa-encryption-algorithm>
- <https://www.preveil.com/blog/end-to-end-encryption/>
- <https://www.geeksforgeeks.org/difference-between-private-key-and-public-key/>

