

PROJECT ENTITLED:  
DOCUMENTATION OF SYMMETRIC AND ASYMMETRIC ENCRYPTION:  
IMPLEMENTATION AND PERFORMANCE ANALYSIS  
BY  
DAMOAH BASHIRU

## **Abstract**

This report documents the practical implementation and analysis of symmetric and asymmetric encryption techniques, specifically the Data Encryption Standard (DES) and RSA algorithms. The lab exercises demonstrate how plaintext data can be securely transformed into ciphertext and subsequently decrypted using secret and public/private keys. Performance testing was conducted to evaluate the efficiency of both encryption methods across files of varying sizes and key lengths. Additionally, a basic biometric-based access simulation was implemented to restrict access to encrypted data, illustrating how authentication mechanisms can enhance security. The report provides insights into the advantages, limitations, and potential security considerations of each cryptographic method.

## Table of Contents

Abstract.....	2
DES Lab Report.....	3
1. Background.....	3
2. Lab Procedure.....	3
3. Python GUI Implementation.....	4
4. Experiment / Exercise Question (Q17).....	5
5. Conclusion.....	5
RSA Encryption Lab Report.....	6
1. Introduction.....	6
2. Theory.....	6
3. Implementation.....	6
4. Procedure.....	9
5. Observations.....	11
6. Conclusion.....	12
Biometric-Based Access to Encrypted Data.....	13
1. Introduction.....	13
2. Biometric Authentication Implementation.....	13
3. Biometric Input.....	16
4. Secret File Integration.....	16
5. Conclusion.....	17

# DES Lab Report

## 1. Background

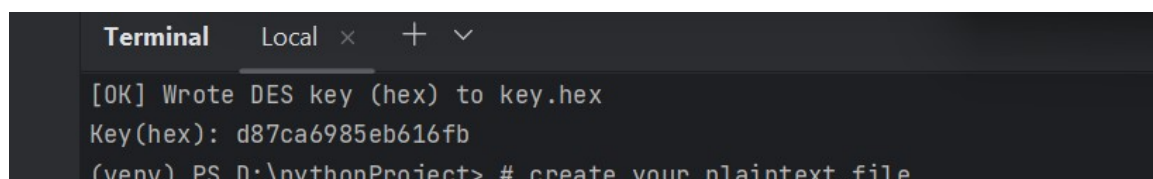
The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of digital data. Originally developed by IBM in the 1960s as Lucifer, it was refined into the Data Encryption Algorithm (DEA) and adopted as the standard in 1976. DES operates on 64-bit blocks of data using a 56-bit key, performing a series of permutations, substitutions, and rounds (16 Feistel rounds).

Although DES is relatively fast, its small key size makes it vulnerable to brute-force attacks, which is why it has been superseded by AES in modern applications.

## 2. Lab Procedure

The following steps were performed in the DES lab using the Python GUI tool:

1. Generate a DES key (8 bytes / 56-bit) and save it in hexadecimal format.
2. Select a plaintext file (.open) to encrypt.
3. Encrypt the file using the generated key to produce a ciphertext file (.close).
4. Share the ciphertext file and key with a teammate.
5. Receive a teammate's ciphertext file and key, then decrypt it to retrieve the original plaintext.

A screenshot of a terminal window with a dark background. The title bar at the top shows 'Terminal' and 'Local' with window control icons. The terminal output shows a confirmation message, the generated key in hexadecimal, and a prompt for the next step.

```
Terminal Local x + v
[OK] Wrote DES key (hex) to key.hex
Key(hex): d87ca6985eb616fb
(venv) PS D:\pythonProject> # create your plaintext file
```

Figure 1: Hexadecimal Key Generated (DES)

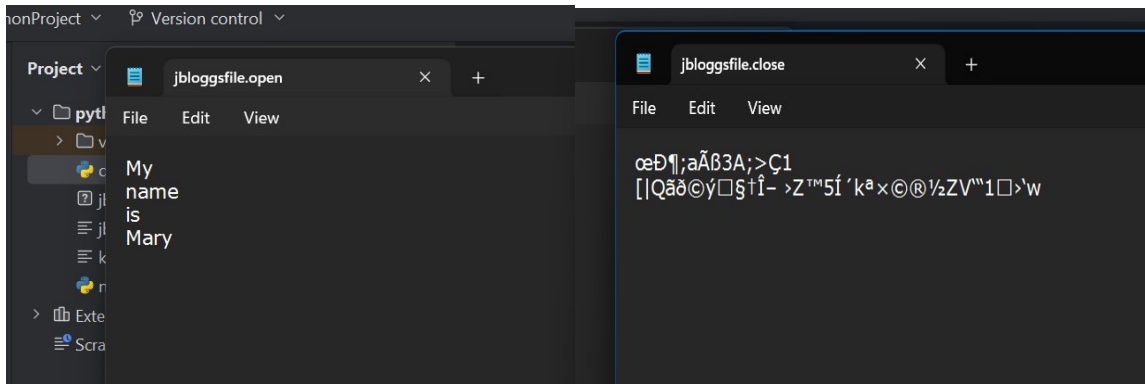


Figure 2: plaintext file

Figure 3: cipher text using KEY

### 3. Python GUI Implementation

A Python GUI was implemented using customtkinter to perform DES encryption and decryption. The GUI allows users to generate keys, browse input files, select output locations, encrypt/decrypt files and view log messages. DES encryption was performed in ECB mode with PKCS#7 padding.

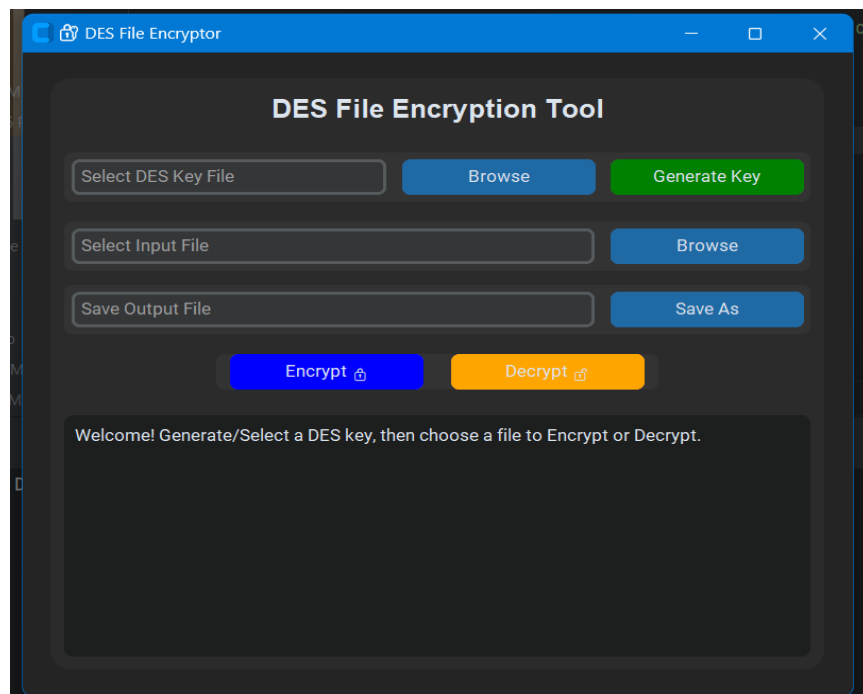
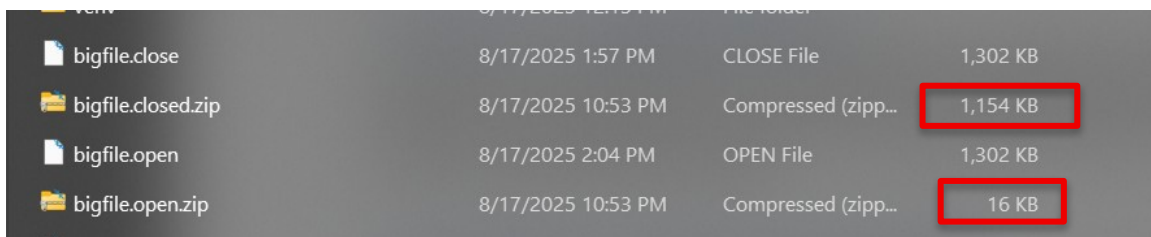


Figure 4: GUI Encryptor Interface for the DES

## 4. Experiment / Exercise Question (Q17)

A large file (1MB) named bigfile.open was created. It was encrypted to produce bigfile.close. Both files were then compressed using WinZip.

The **compressed plaintext file** was **smaller due to redundancy** being removed by compression. The **encrypted file** appeared **random and high-entropy**, so compression did not significantly reduce its size.



bigfile.close	8/17/2025 1:57 PM	CLOSE File	1,302 KB
bigfile.closed.zip	8/17/2025 10:53 PM	Compressed (zipp...	1,154 KB
bigfile.open	8/17/2025 2:04 PM	OPEN File	1,302 KB
bigfile.open.zip	8/17/2025 10:53 PM	Compressed (zipp...	16 KB

Figure 5: Before and After of encrypted files with varying ZIP file size

## 5. Conclusion

The lab demonstrated symmetric key encryption using DES. Key generation, file encryption, and decryption were successfully implemented using a Python GUI. The experiment highlighted why compression should be applied before encryption to reduce file size.

# RSA Encryption Lab Report

## 1. Introduction

The RSA algorithm, named after Rivest, Shamir, and Adleman, is a widely-used public-key cryptosystem. It uses asymmetric encryption with a public key for encryption and a private key for decryption.

This lab focuses on:

1. Generating RSA keys
2. Encrypting and decrypting files
3. Measuring encryption/decryption performance

The implementation uses Python and CustomTkinter to provide a GUI-based tool.

## 2. Theory

### 2.1 RSA Algorithm

1. Select two large prime numbers  $p$  and  $q$ .
2. Compute modulus:  $N = p * q$
3. Compute Euler's totient:  $\phi(N) = (p-1)*(q-1)$
4. Choose public exponent  $E$  such that  $\gcd(E, \phi(N)) = 1$
5. Compute private exponent  $D$ :  $D \equiv E^{-1} \pmod{\phi(N)}$

Encryption:  $C = M^E \pmod{N}$

Decryption:  $M = C^D \pmod{N}$

## 3. Implementation

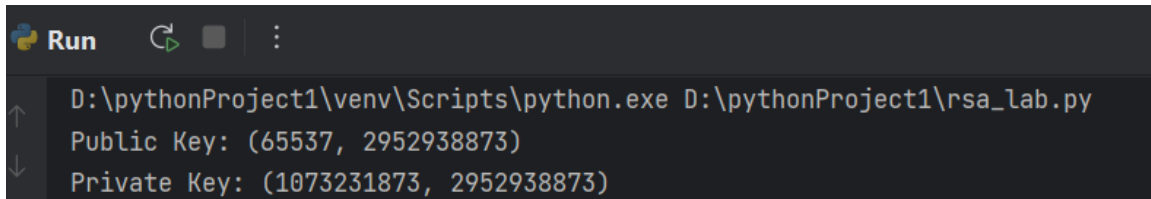
### 3.1 Key Generation

The Python program generates a public/private key pair and allows saving to .key files.

Example Output:

Public Key: (65537, 2952938873)

Private Key: (1073231873, 2952938873)



```
Run
D:\pythonProject1\venv\Scripts\python.exe D:\pythonProject1\rsa_lab.py
Public Key: (65537, 2952938873)
Private Key: (1073231873, 2952938873)
```

Figure 6: Show GUI after generating public/private keys

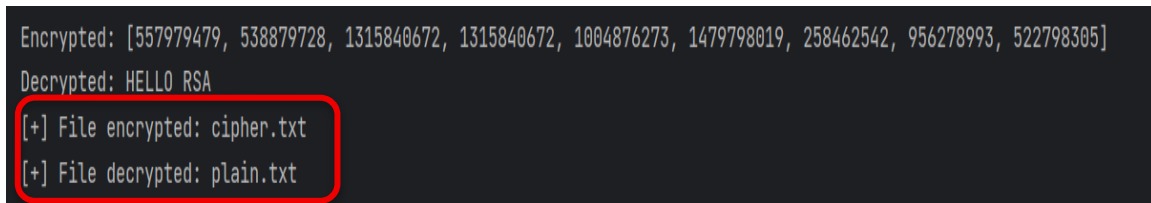
### 3.2 File Encryption & Decryption

Encrypt a file using the public key and decrypt using the private key.

Example Output:

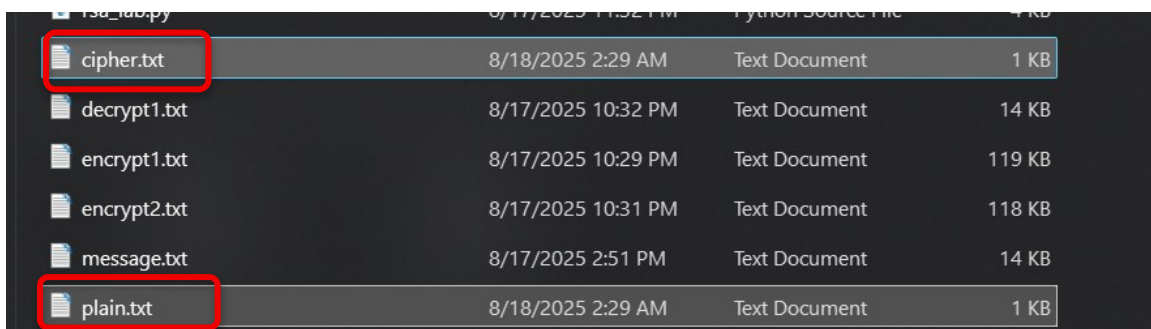
Encrypted: [557979479, 538879728, ...]

Decrypted: HELLO RSA



```
Encrypted: [557979479, 538879728, 1315840672, 1315840672, 1004876273, 1479798019, 258462542, 956278993, 522798305]
Decrypted: HELLO RSA
[+] File encrypted: cipher.txt
[+] File decrypted: plain.txt
```

Figure 7: Encrypting with keys



rsa_lab.py	8/17/2025 11:52 PM	Python Source File	4 KB
cipher.txt	8/18/2025 2:29 AM	Text Document	1 KB
decrypt1.txt	8/17/2025 10:32 PM	Text Document	14 KB
encrypt1.txt	8/17/2025 10:29 PM	Text Document	119 KB
encrypt2.txt	8/17/2025 10:31 PM	Text Document	118 KB
message.txt	8/17/2025 2:51 PM	Text Document	14 KB
plain.txt	8/18/2025 2:29 AM	Text Document	1 KB

Figure 8: Ciphertext and Plaintext files created



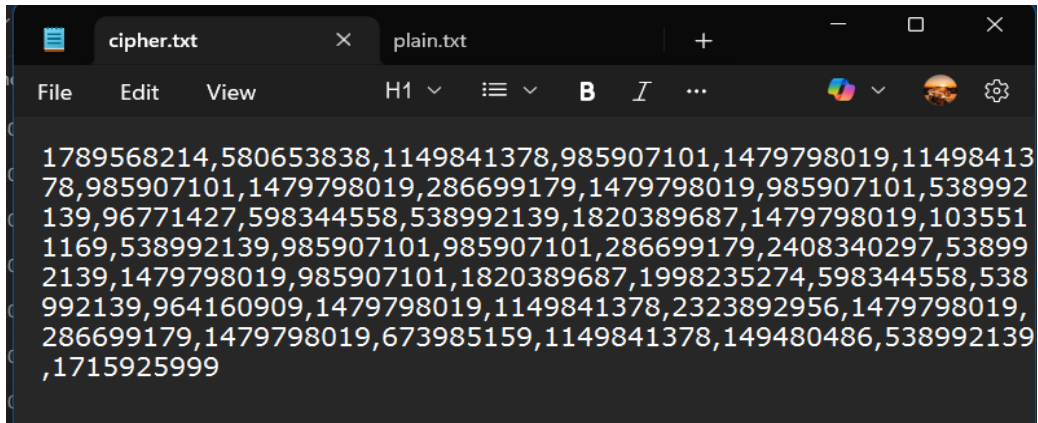


Figure 9: Cipher

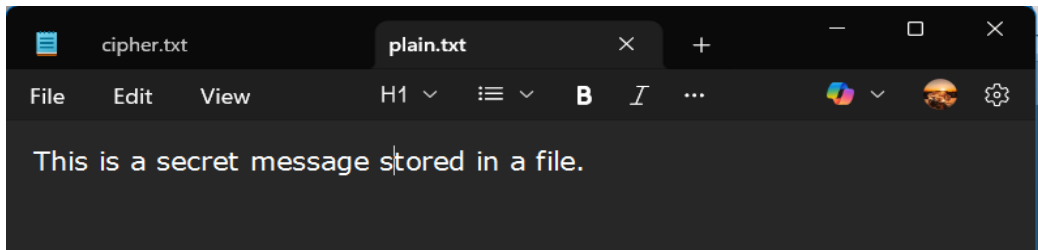


Figure 10: Plaintext

### 3.3 Performance Testing

Multiple files tested for encryption/decryption speed.

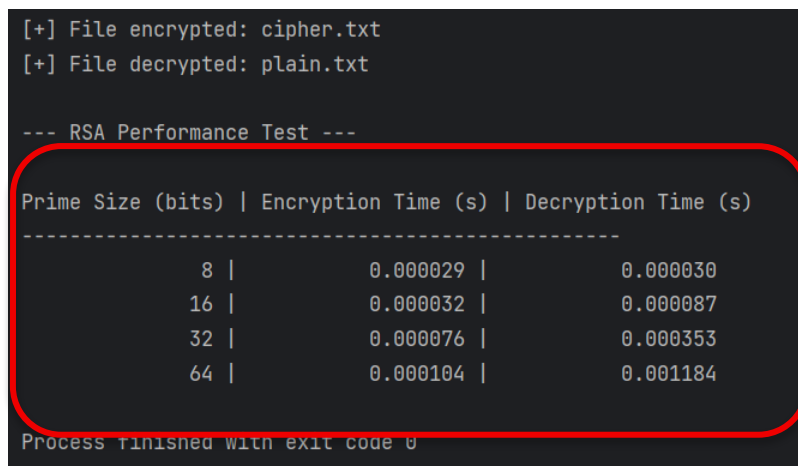


Figure 11: Terminal Performance Measure

## 4. Procedure

1. Extract RSA project files to a working directory.
2. Run the Python program: `python rsa_lab.py`
3. Use the GUI to:
  - Generate public and private keys
  - Encrypt a text file
  - Decrypt the encrypted file
  - Perform performance tests
  - Record times and plot graphs for analysis.



Figure 12: GUI for RSA

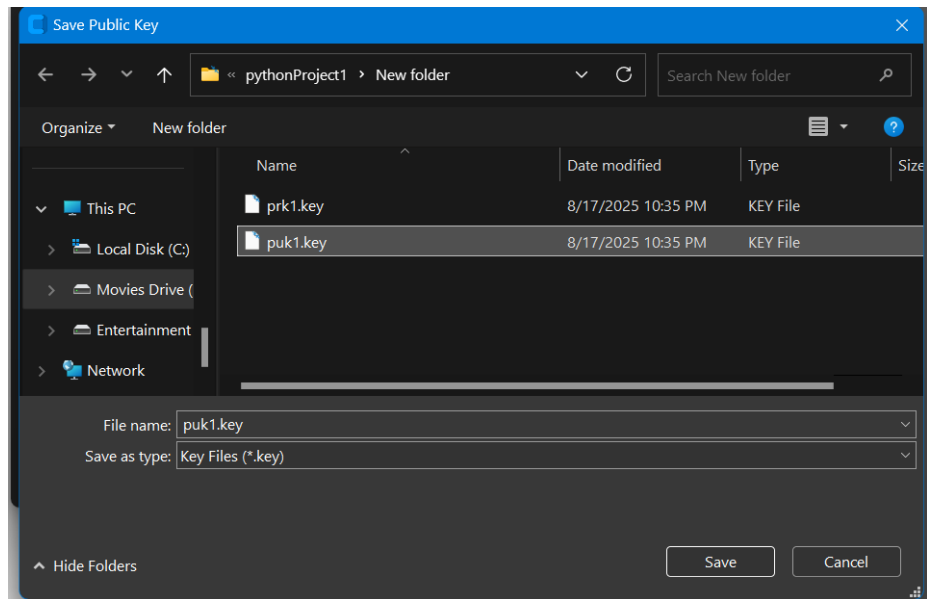


Figure 13: Public Key Gen

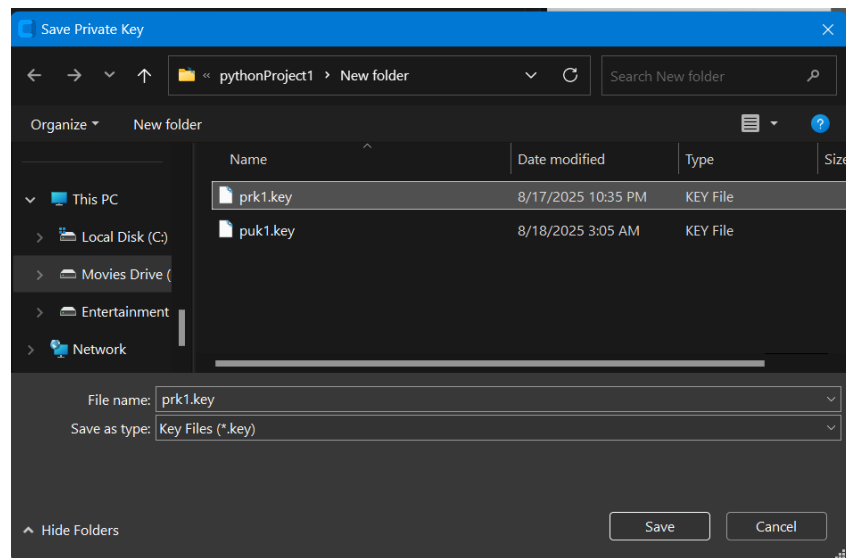


Figure 14: Private Key Gen

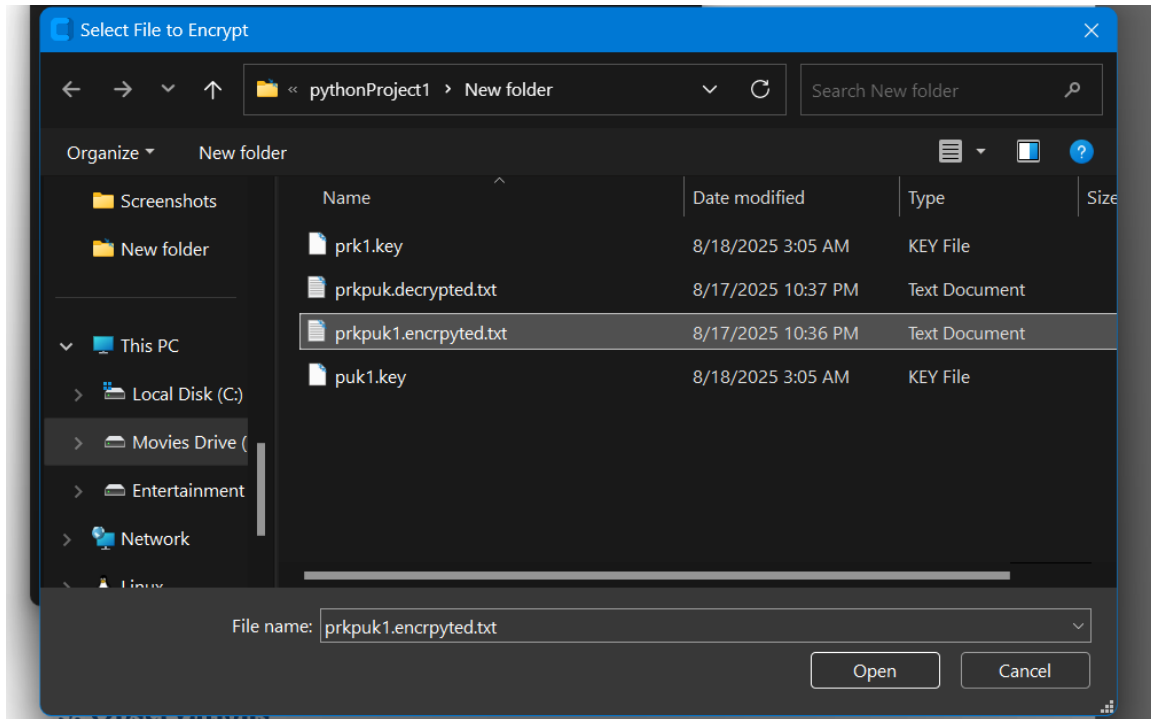


Figure 15: Encrypting a file with Public Key Generated

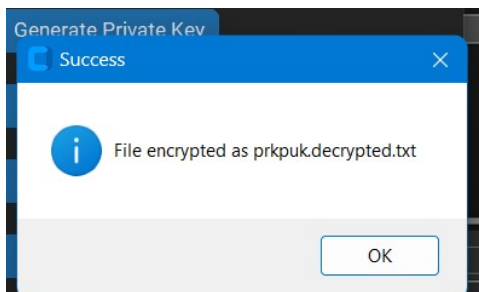


Figure 16: performance Measure

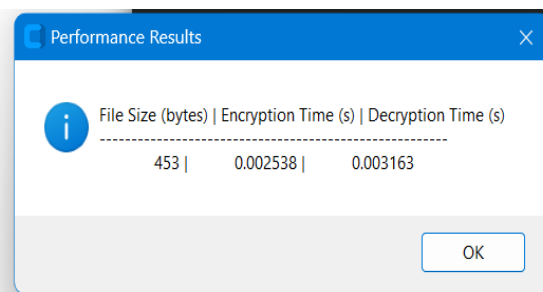


Figure 17: Successful Encryption

## 5. Observations

1. Encryption is faster than decryption due to exponent size differences.
2. Performance slows as prime size increases.
3. RSA is suitable for small messages or files, but not ideal for large data.

## **6. Conclusion**

The lab successfully demonstrated:

- RSA key generation
- File encryption and decryption
- Performance benchmarking

The GUI Python tool provides an interactive way to understand RSA and its practical implications.

# Biometric-Based Access to Encrypted Data

## 1. Introduction

This section demonstrates face recognition-based biometric authentication to control access to a secret file (secret.txt). Using the **webcam** and **MediaPipe**, the system registers a user's face and authenticates them before granting access to the secret key.

## 2. Biometric Authentication Implementation

System Overview:

1. The system captures a face using the webcam and saves it in the folder **'authorized\_face/'**.
2. Authentication is performed by comparing the **live face** with the **registered face** using grayscale image correlation (`numpy.corrcorf`) to compute similarity.
3. A similarity threshold of 0.6 determines a match.

GUI Features:

1. **Register Face:** Captures and saves the user's face.
2. **Authenticate:** Compares the live face to the registered template. If the match is above the threshold, the secret content is displayed.
3. The GUI provides status updates for registration and authentication (successful or failed).

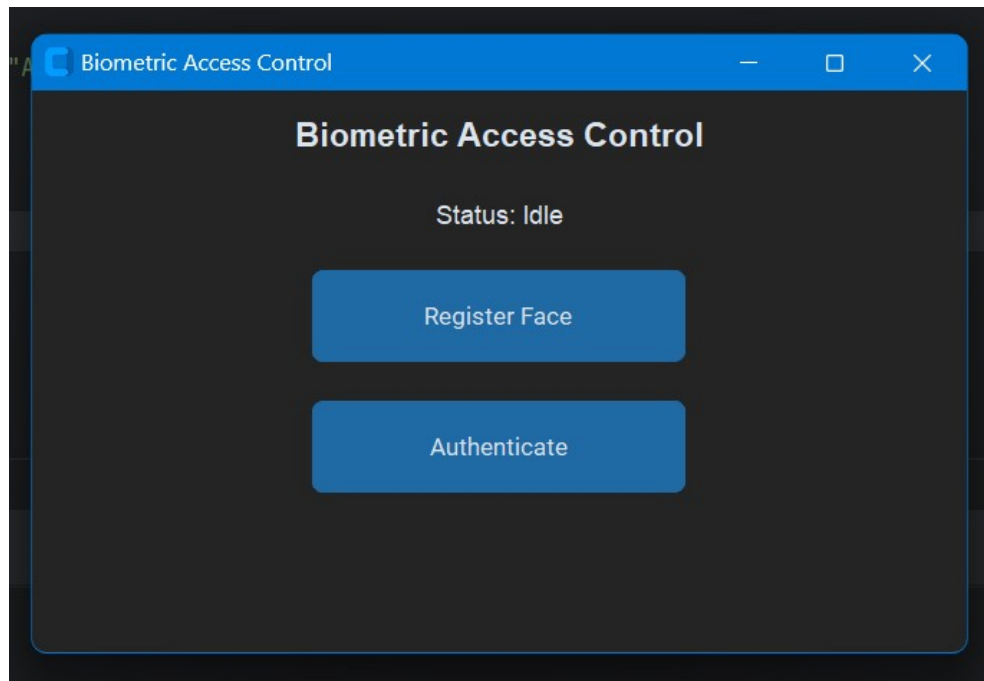


Figure 18: GUI of Face Recognition Access Control

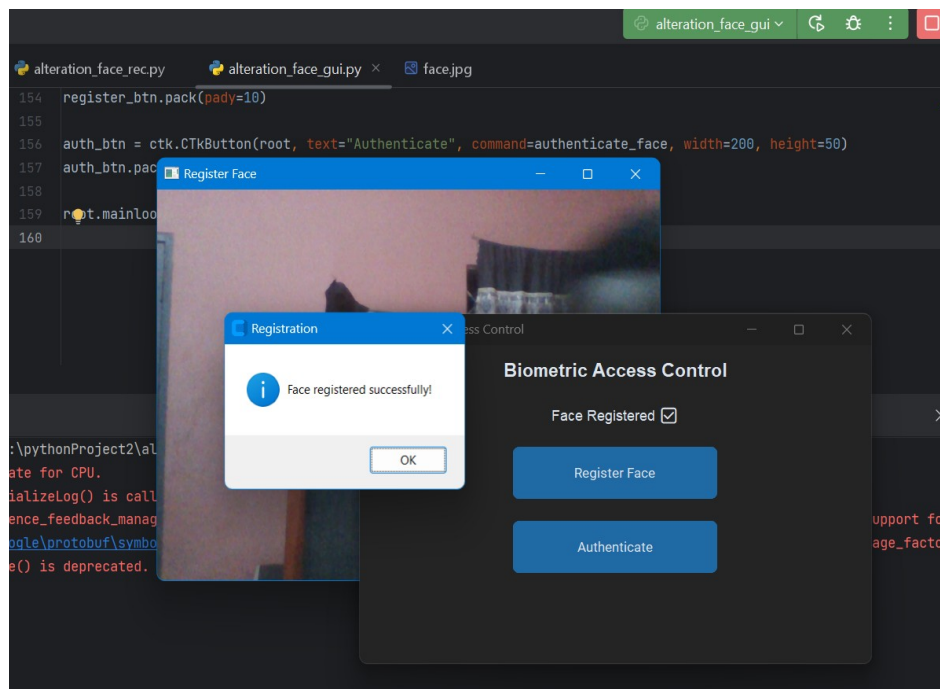


Figure 19: Face Registered

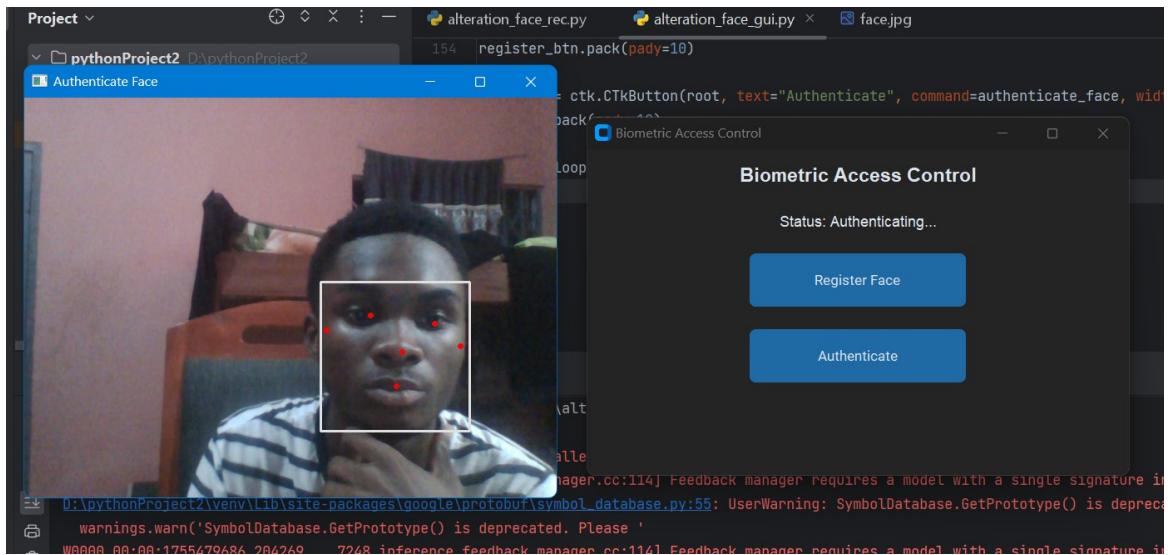


Figure 20: Authenticating Face

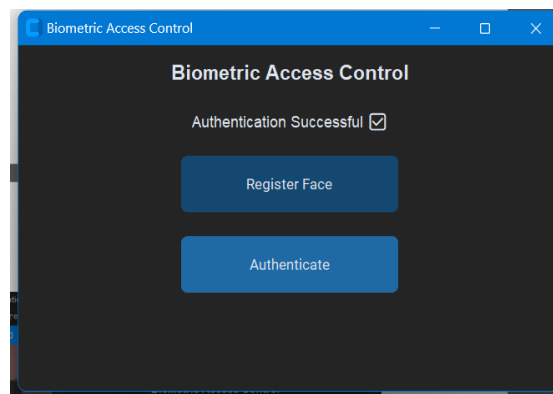


Figure 21: Authentication Successful

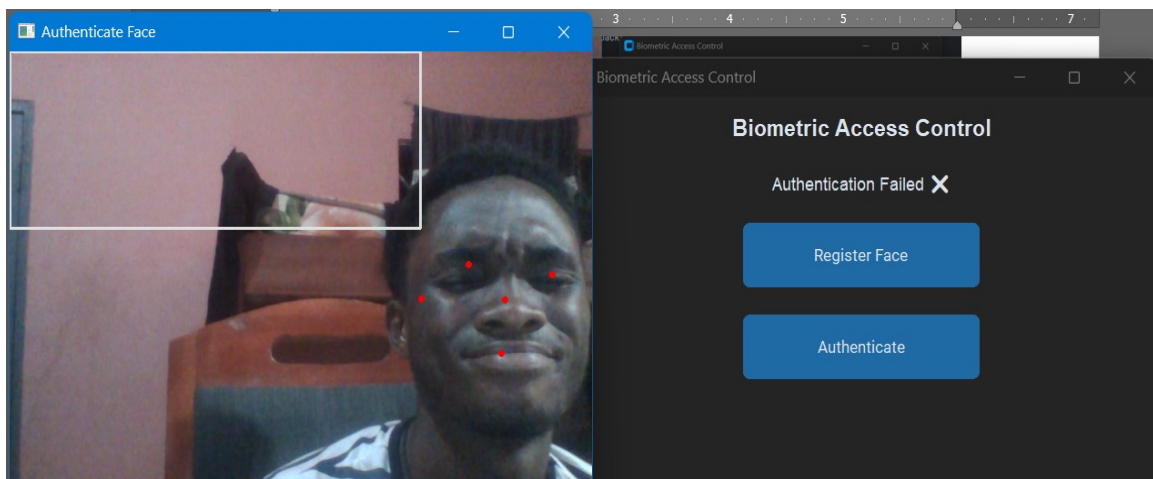


Figure 22: Authentication Failure



### 3. Biometric Input

- a. Type of Biometric Input Simulated: Face recognition via the webcam using MediaPipe.
- b. How it Controls Access: The secret file (secret.txt) is only revealed if the captured face matches the registered face above the similarity threshold.
- c. Potential Security Risks:
  - **Spoofing:** Photos or videos of the authorized user could bypass authentication.
  - **False positives/negatives:** Poor lighting or camera angles may lead to failed authentication or unauthorized access.
  - **Local face storage risk:** Registered face images are stored on disk and could be stolen.
  - **No liveness detection:** The system cannot verify that the face is live, making it vulnerable to replay attacks.

### 4. Secret File Integration

1. The secret key/message is stored in **secret.txt**.
2. Access is granted only if authentication succeeds.
3. Successful authentication prints the secret content to the console and updates the GUI status.
4. Failed authentication updates the GUI status and denies access.

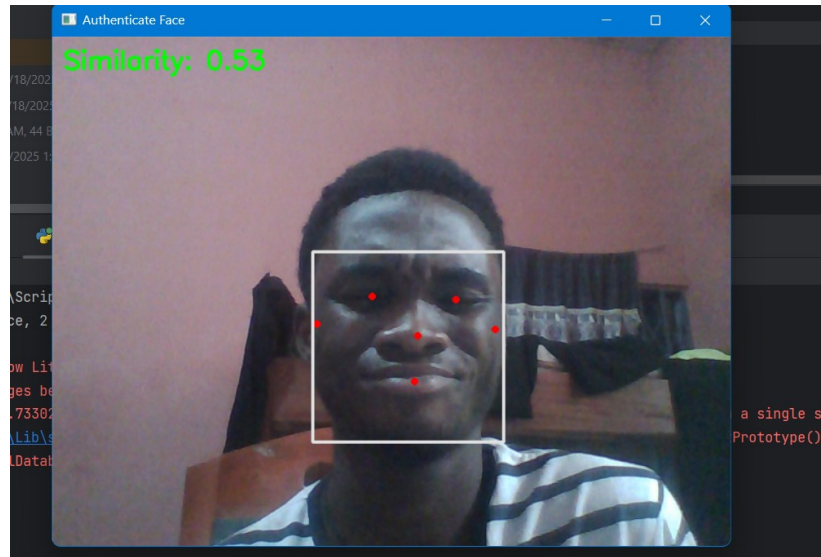


Figure 23:Running Similiarity Test (0.6)

```
W0000 00:00:1755482921.733028 20940 inference_feedback_manager.cc:114] Feedback m
D:\pythonProject2\venv\Lib\site-packages\google\protobuf\symbol_database.py:55: Use
warnings.warn('SymbolDatabase.GetPrototype() is deprecated. Please '
[INFO] Authentication successful!
[SECRET] Access granted! Here is your encrypted message/key:
YOUR_SECRET_MESSAGE_OR_KEY_HERE

Process finished with exit code 0
```

Figure 24: Displays secret message after correct match

## 5. Conclusion

The implementation successfully integrates biometric authentication to protect sensitive data. While suitable for demonstration, a production system should include liveness detection, secure storage, and anti-spoofing mechanisms for enhanced security.