Configuration Manual

Enhancing security efficiency through the integration of Elliptic Curve Cryptography with Audio Steganography

MSc Research Project
Practicum 2

Nishant Bhosale
Student ID: 22227377

School of Computing
National College of Ireland

Supervisor: Khadija Hafeez

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	…Nishant Sandeep Bhosale
Student ID:	22227377
Programme:	MSc In Cybersecurity **Year:** 2023
Module:	Practicum 2
Lecturer:	Khadija Hafeez
Submission Due Date:	12th August 2024
Project Title:	Enhancing security efficiency through the integration of Elliptic Curve Cryptography with Audio Steganography
Word Count:	…1460…… **Page Count:** ……12…….…….……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:

Date: 12th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Nishant Bhosale
Student ID: 22227377

1 Introduction

This configuration manual provides a comprehensive guide for the installation, configuration, and practical usage of the prototype application developed to enhance security efficiency through the integration of Elliptic Curve Cryptography (ECC) with Audio Steganography. The manual covers software and hardware requirements, detailed installation instructions, key generation, encryption-decryption processes, embedding-extraction procedures, practical usage steps, and performance evaluation.

2 Environment

2.1 Hardware Requirements

- **Processor**: Intel Core™ i3 or higher (i5 recommended for faster processing)
- **RAM**: 4 GB or higher
- **Storage**: Minimum 120 GB HDD (SSD recommended for better performance)
- **Graphics**: Integrated or dedicated graphics card (NVIDIA GeForce GTX series recommended)
- **Audio Interface**: High-definition audio codec, supporting 24-bit/96 kHz playback
- **Additional Hardware**: Microphone and speakers for audio testing

2.2 Software Requirements

- **Operating System**: Windows 10 or higher, Ubuntu 20.04 or higher
- **Python Version**: Python 3.8 or higher
- **Required Python Libraries**:
  - `numpy`
  - `matplotlib`
  - `pycryptodome`
  - `wave`
  - `tkinter`
  - `scipy`
  - `cryptography`
  - `psutil`

  (Install the required packages using pip: pip install numpy matplotlib pycryptodome wave tkinter scipy cryptography psutil)

- **Integrated Development Environment (IDE)**: PyCharm or Visual Studio Code
- **Version Control**: Git (for version control, collaboration, and backup)

3 Installation Steps

3.1 Python Installation

1. **Windows**:
   - Download the latest version of Python from the official Python website.
   - Run the installer and ensure that "Add Python to PATH" is checked before installation.
   - Verify installation by opening Command Prompt and typing `python --version`.
2. **Ubuntu**:
   - Open the terminal.
   - Run the following commands to install Python:

     ```
     sudo apt update
     sudo apt install python3 python3-pip
     ```

   - Verify installation by typing `python3 --version`.

3.2 IDE Installation

- **PyCharm**:
  - Download from the official JetBrains website.
  - Follow the installation prompts.
  - Open PyCharm and configure it to use the installed Python interpreter.
- **Visual Studio Code**:
  - Download from the official VS Code website.
  - Install the Python extension for Visual Studio Code via the Extensions Marketplace.

3.3 Cloning the Repository

1. **Using Git**:
   - Open the terminal (Command Prompt on Windows, Terminal on Ubuntu).
   - Clone the repository containing the project:

     ```
     git clone https://github.com/your-repository-url.git
     ```

   - Navigate to the project directory:

     ```
     cd your-repository-folder
     ```

2. **Manually Downloading**:
   - Download the project ZIP file from the repository.
   - Extract the contents to a desired location on your system.

3.4 Installing Required Python Libraries

- Navigate to the project directory in your terminal or command prompt.
- Run the following command to install all required dependencies:

```
pip install -r requirements.txt
```

4 Key Generation

The system supports both RSA and ECC for key generation. Below are the steps to generate keys:

4.1 RSA Key Generation

1. **Run the RSA key generation script**:

```
python key_gen.py --type rsa --output rsa_keys/
```

   - This will generate RSA public and private keys in the `rsa_keys` directory.
2. **Verify the keys**:
   - Ensure that `private.pem` and `public.pem` files are created in the specified directory.

(Diagram here: RSA key generation process flow diagram with file outputs)

4.2 ECC Key Generation

1. **Run the ECC key generation script**:

```
python key_gen.py --type ecc --curve secp256k1 --output ecc_keys/
```

   - This will generate ECC public and private keys in the `ecc_keys` directory.
2. **Verify the keys**:
   - Ensure that `ecc_private_key.pem` and `ecc_public_key.pem` files are created in the specified directory.

5 Implementation

5.1 Encryption

5.1.1 RSA Encryption

1. **Algorithm**: RSA with OAEP Padding
2. **Hash Function**: SHA-256
3. **Process**:
   - Load the RSA public key from the `.pem` file.
   - Encrypt the plaintext message.
   - Save the ciphertext in a binary format.

```
def rsa_encrypt_message():
    from Crypto.Cipher import PKCS1_OAEP
    from Crypto.PublicKey import RSA
    from Crypto.Hash import SHA256
    public_key = RSA.import_key(open("public.pem").read())
    cipher = PKCS1_OAEP.new(public_key, hashAlgo=SHA256)
    ciphertext = cipher.encrypt(b"Your message here")
    with open("ciphertext.bin", "wb") as cipher_file:
        cipher_file.write(ciphertext)
```

5.1.2 ECC Encryption

1. **Algorithm**: Elliptic Curve Integrated Encryption Scheme (ECIES)
2. **Symmetric Encryption**: AES-GCM
3. **Process**:
    - Generate an ephemeral key pair.
    - Derive a shared secret using the recipient's public key.
    - Encrypt the message using AES-GCM.
    - Save the ciphertext along with the ephemeral public key.

```
def ecc_encrypt_message():
    from cryptography.hazmat.primitives.kdf.hkdf import HKDF
    from cryptography.hazmat.primitives.hashes import SHA256
    from cryptography.hazmat.primitives.kdf.hkdf import HKDFExpand
    from cryptography.hazmat.primitives import hashes, serialization
    from cryptography.hazmat.primitives.asymmetric import ec
    from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

    # Load recipient's public key
    with open("ecc_public_key.pem", "rb") as key_file:
        public_key = serialization.load_pem_public_key(key_file.read())

    # Generate ephemeral key pair
    ephemeral_private_key = ec.generate_private_key(ec.SECP256K1())
    shared_secret = ephemeral_private_key.exchange(ec.ECDH(), public_key)

    # Derive a symmetric key
    symmetric_key = HKDF(
        algorithm=SHA256(),
        length=32,
        salt=None,
        info=b'handshake data'
    ).derive(shared_secret)

    # Encrypt message
    aesgcm = algorithms.AES(symmetric_key)
    encryptor = Cipher(aesgcm, modes.GCM()).encryptor()
    ciphertext = encryptor.update(b"Your message here") + encryptor.finalize()

    with open("ecc_ciphertext.bin", "wb") as cipher_file:
        cipher_file.write(ciphertext)
```

5.2 Steganography

5.2.1 Embedding the Encrypted Message into Audio

1. **Technique**: Least Significant Bit (LSB)
2. **Process**:
    - Convert the encrypted message to a binary sequence.
    - Embed the binary sequence into the LSB of each audio sample.
    - Save the modified audio as a stego audio file.

```
def embed_message_in_audio():
    import wave
    message_bits = to_binary("Your encrypted message here")
    audio = wave.open("audio.wav", "rb")
    frames = bytearray(list(audio.readframes(audio.getnframes())))

    for i in range(len(message_bits)):
        frames[i] = (frames[i] & 254) | int(message_bits[i])

    with wave.open("stego_audio.wav", "wb") as stego_audio:
        stego_audio.setparams(audio.getparams())
        stego_audio.writeframes(frames)
```

5.2.2 Extracting the Encrypted Message from Audio

1. **Process**:
    - Load the stego audio file.
    - Extract the LSBs to reconstruct the binary sequence.
    - Convert the binary sequence back to the encrypted message.

```
def extract_message_from_audio():
    import wave
    audio = wave.open("stego_audio.wav", "rb")
    frames = bytearray(list(audio.readframes(audio.getnframes())))

    message_bits = [str((frame & 1)) for frame in frames[:len(frames)]]
    message_binary = ''.join(message_bits)

    with open("extracted_message.txt", "wb") as message_file:
        message_file.write(from_binary(message_binary))
```

5.3 Decryption

5.3.1 RSA Decryption

1. **Process**:
    - Load the private key.
    - Decrypt the ciphertext using RSA.
    - Display the plaintext message.

```
def rsa_decrypt_message():
    from Crypto.Cipher import PKCS1_OAEP
    from Crypto.PublicKey import RSA
    private_key = RSA.import_key(open("private.pem").read())
    cipher = PKCS1_OAEP.new(private_key)
    with open("ciphertext.bin", "rb") as cipher_file:
        plaintext = cipher.decrypt(cipher_file.read())
    print(plaintext.decode())
```

5.3.2 ECC Decryption

1. **Process**:
    - Load the recipient's private key.
    - Derive the shared secret using the ephemeral public key.
    - Decrypt the message.
    - Display the plaintext message.

```
def ecc_decrypt_message():
    from cryptography.hazmat.primitives.kdf.hkdf import HKDF
    from cryptography.hazmat.primitives.hashes import SHA256
    from cryptography.hazmat.primitives.kdf.hkdf import HKDFExpand
    from cryptography.hazmat.primitives import serialization
    from cryptography.hazmat.primitives.asymmetric import ec
    from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

    with open("ecc_private_key.pem", "rb") as key_file:
        private_key = serialization.load_pem_private_key(key_file.read(), None)

    with open("ecc_ciphertext.bin", "rb") as cipher_file:
        ciphertext = cipher_file.read()

    ephemeral_public_key = # Extract from ciphertext
    shared_secret = private_key.exchange(ec.ECDH(), ephemeral_public_key)

    symmetric_key = HKDF(
        algorithm=SHA256(),
        length=32,
        salt=None,
        info=b'handshake data'
    ).derive(shared_secret)

    aesgcm = algorithms.AES(symmetric_key)
    decryptor = Cipher(aesgcm, modes.GCM()).decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()
    print(plaintext.decode())
```

6. Practical Usage Steps

6.1 Running the Application

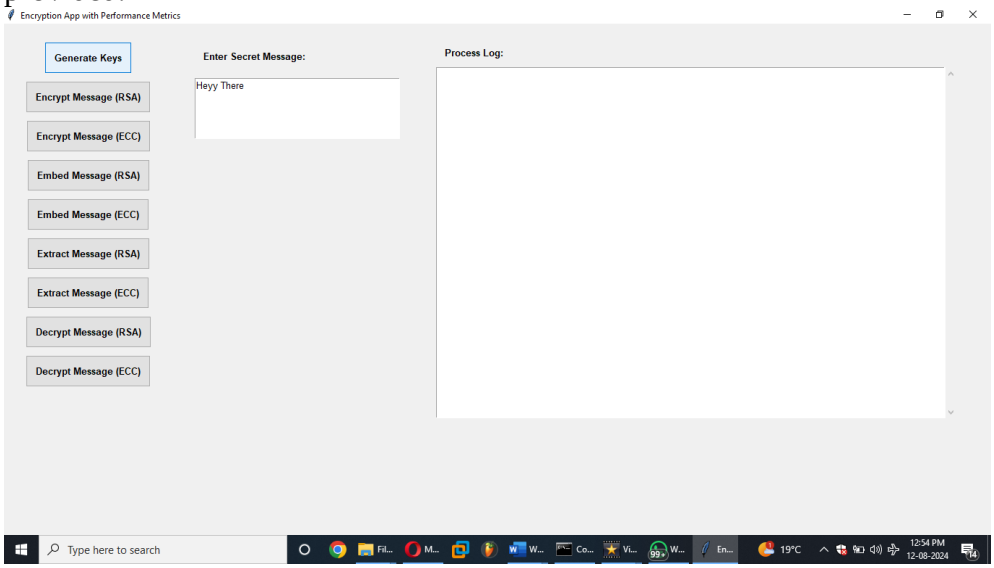
1. **Start the GUI Application**:
    - In the terminal or command prompt, navigate to the project directory.
    - Run the following command to start the application:

```
python app.py
```

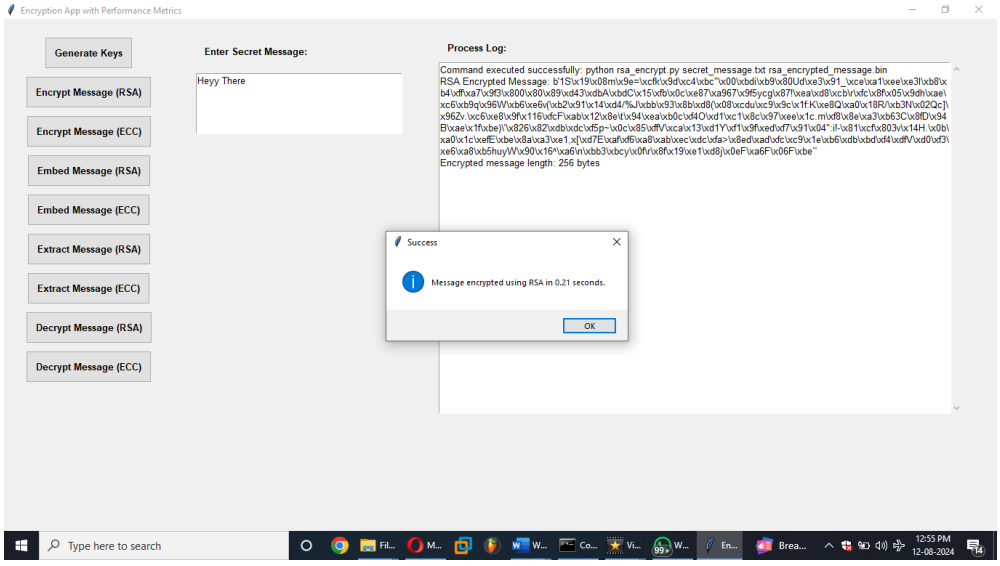
- This will launch the graphical user interface (GUI).

6.2 Using the Encoder

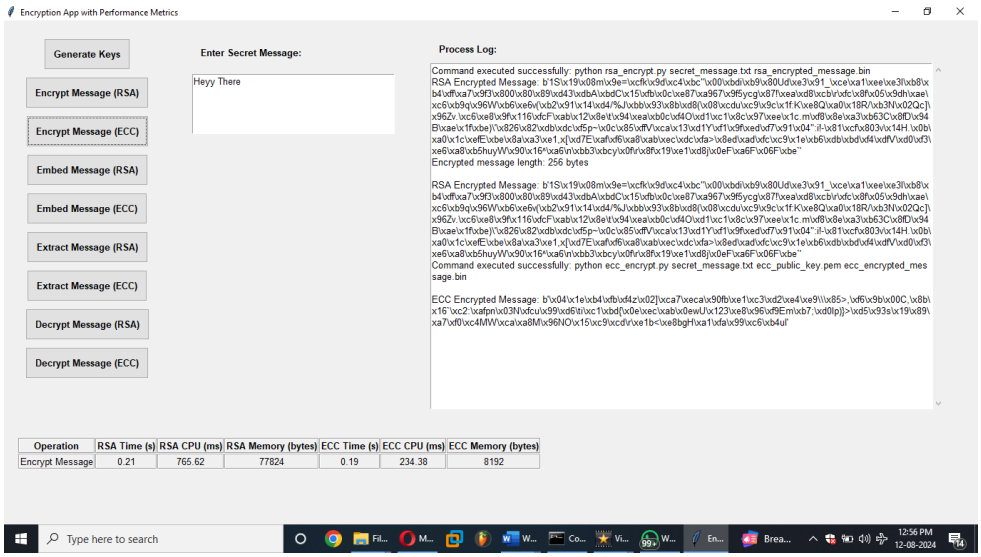
1. **Input the Secret Text**:
   - Enter the secret message in the text box provided.



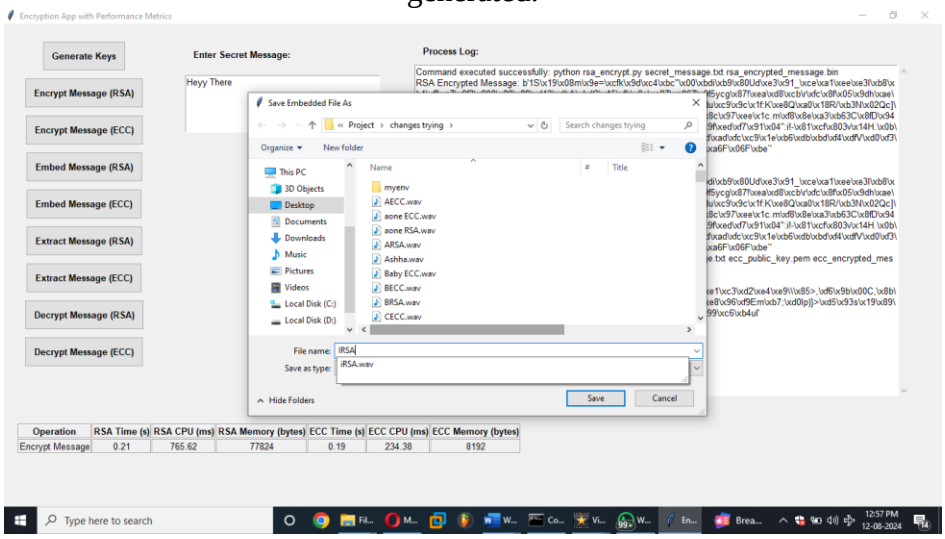
2. **Choose the Encryption Method**:
   - Select either RSA or ECC from the dropdown menu.



3. **Encrypt the Message**:
   - Click on the "Encrypt" button to encrypt the secret message.
   - If using RSA, ensure the public key is correctly loaded. For ECC, ensure the correct curve and public key are selected.

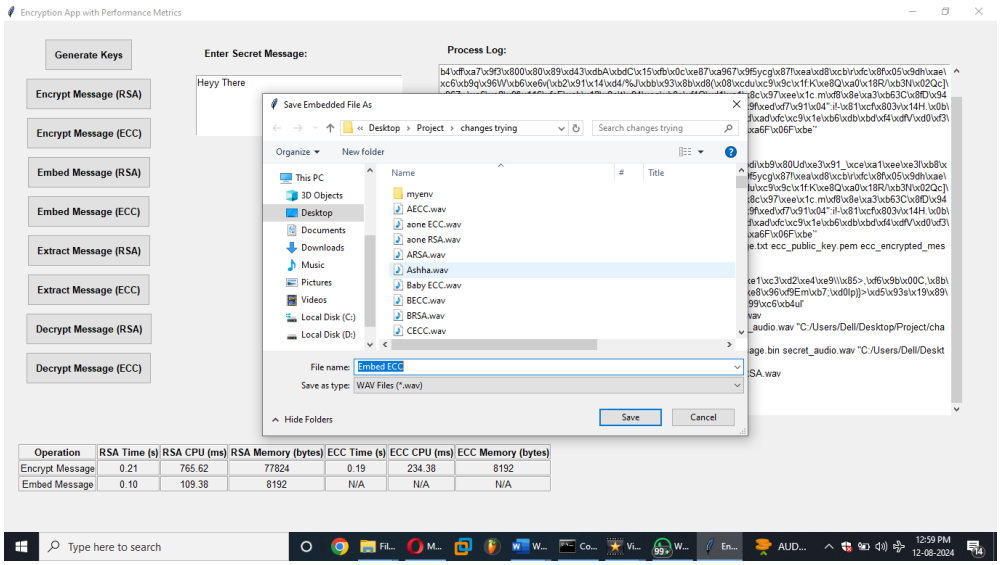


4. **Embed the Encrypted Message**:
   - Click on the "Embed" button to embed the encrypted message into the audio file.
   - Specify the number of LSBs to use for embedding (default is 1 LSB).
     - After embedding, the stego audio file will be automatically generated.

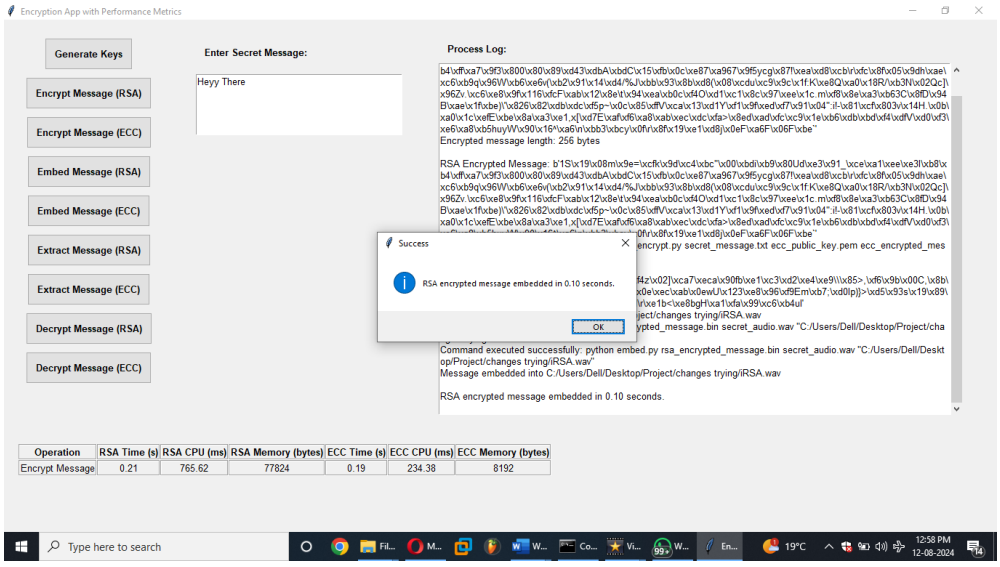


5. **Save the Stego Audio File**:
   - Click on "Save" to store the stego audio file in the desired directory.

For ECC



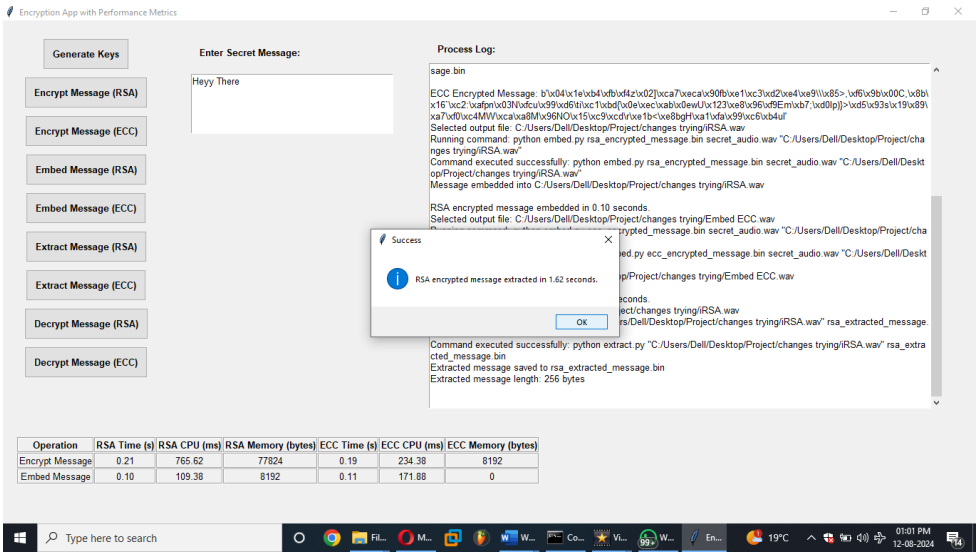
For RSA.



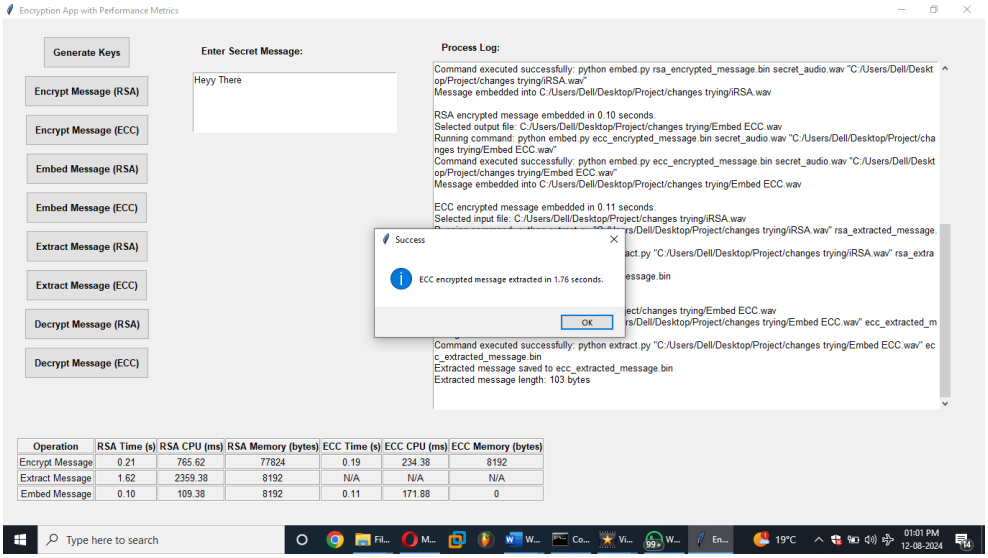
6.3 Using the Decoder

1. **Load the Stego Audio File**:
   - Click on the “Browse” button to load the stego audio file.
2. **Extract the Encrypted Message**:
   - Click on the “Extract” button to retrieve the encrypted message from the stego audio file.

For RSA

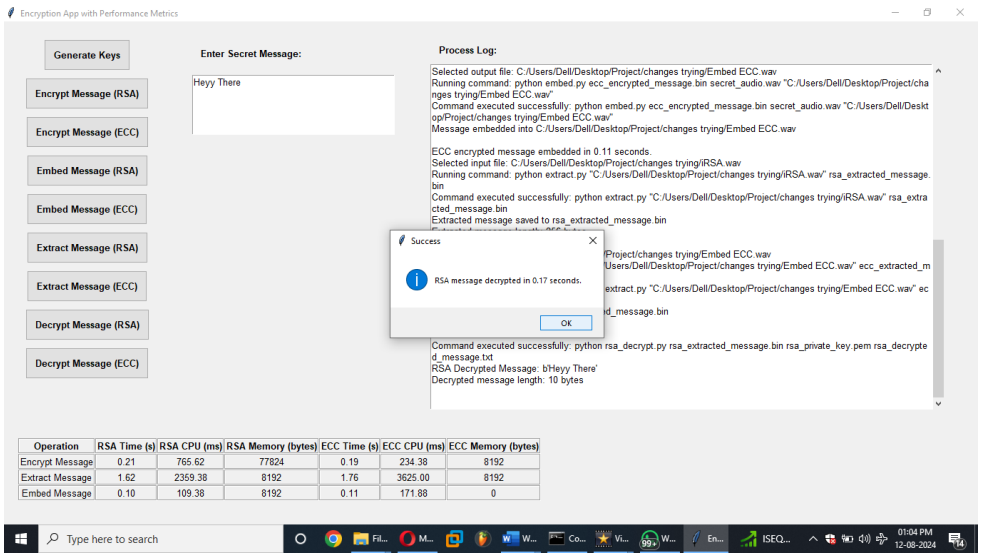


For ECC

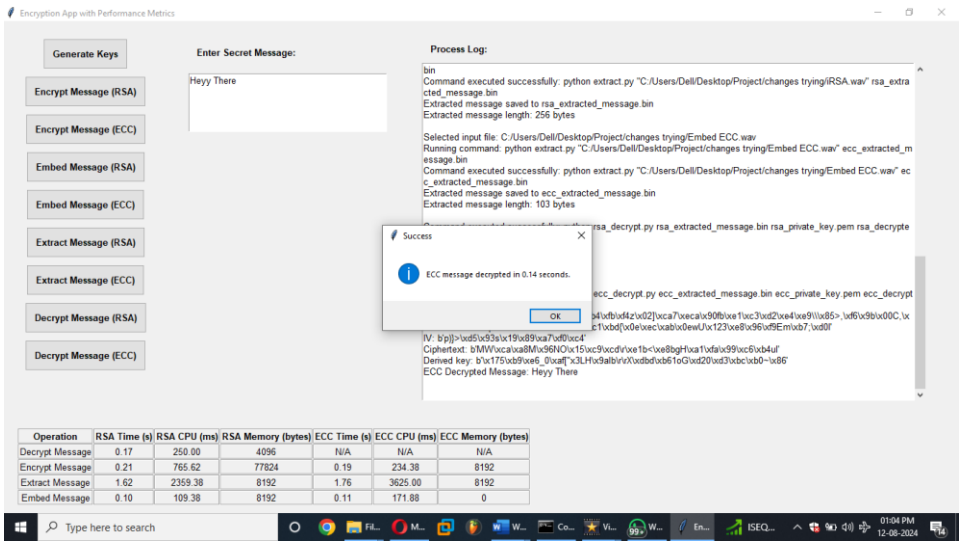


3. **Decrypt the Message**:
    - Click on the "Decrypt" button to decrypt the extracted message.
    - The original plaintext message will be displayed on the screen.

For RSA



For ECC



7. Code Execution

7.1 Running Specific Modules

- **Encrypt a Message Using RSA**:

```
python rsa_encrypt.py --message "Your message here" --output
ciphertext.bin
```

- **Decrypt a Message Using ECC**:

```
python ecc_decrypt.py --input ecc_ciphertext.bin --output
plaintext.txt
```

7.2 Performance Monitoring

- **Real-Time Monitoring**:
  - The application automatically monitors CPU and memory usage during operations.
  - Logs are saved in `performance.log`.
- **Reviewing Logs**:
  - Open the `performance.log` file in any text editor to review resource usage statistics.
  - Metrics include execution time, CPU utilization, and memory consumption.

Operation	RSA Time (s)	RSA CPU (ms)	RSA Memory (bytes)	ECC Time (s)	ECC CPU (ms)	ECC Memory (bytes)
Decrypt Message	0.17	234.38	12288	0.14	171.88	4096
Encrypt Message	0.13	171.88	16384	0.14	171.88	4096
Extract Message	1.43	1875.00	12288	1.42	1750.00	12288
Embed Message	0.10	125.00	12288	0.11	156.25	4096

8. Testing and Evaluation

8.1 Performance Metrics

- **Execution Time**:
  - Measure and compare the time taken by RSA and ECC for encryption, decryption, embedding, and extraction.
- **CPU Usage**:
  - Monitor CPU usage for different operations, ensuring the system remains within acceptable performance limits.
- **Memory Usage**:
  - Track memory consumption during various operations, particularly during large file handling.