Computer Security

Hybrid Cryptography using AES, RSA and Camellia

Submitted By:

K Balavignesh Reddy	AV.EN.U4AIE22016
R Kabin Dev	AV.EN.U4AIE22015
P Gagan Devesh	AV.EN.U4AIE22027

B. Tech (AIE) SIXTH SEMESTER

Under the Guidance of

Dr. Aditya Kumar Sahu

Department of Computer Science and Engineering

Amrita School of Computing

Amrita Vishwa Vidyapeetham, Amaravati Campus

Table of Contents

Abstract

In today's digital landscape, securing sensitive data against cyber threats has become more critical than ever. Traditional cryptographic techniques, while effective, often come with trade-offs in terms of performance, scalability, and security robustness. This project presents a hybrid cryptographic framework that leverages the strengths of three encryption algorithms—AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and Camellia—to provide a multi-layered security approach for data encryption and decryption.

The proposed method integrates AES and Camellia for symmetric encryption, ensuring fast and efficient encryption of data, while RSA is utilized for key exchange, guaranteeing secure transmission of encryption keys. AES, a widely used symmetric algorithm, provides high-speed encryption with minimal computational overhead. Camellia, another powerful symmetric cipher, offers additional security by diversifying the encryption process, making it more resistant to cryptanalysis. RSA, an asymmetric encryption algorithm, securely encrypts the symmetric keys, preventing unauthorized access during transmission.

This framework supports the encryption and decryption of various file formats, including text, images, audio, and video, making it versatile for real-world applications. The system is designed to first split the input data into two parts, encrypting one with AES and the other with Camellia. The encrypted symmetric keys are then securely transmitted using RSA. On the receiver's end, the encrypted keys are decrypted using the RSA private key, and the original data is reconstructed by decrypting both AES and Camellia-encrypted parts.

A major highlight of this approach is its balance between security and efficiency. The hybrid model reduces the risks associated with single-algorithm encryption while maintaining high processing speeds suitable for large files, including multimedia data. Additionally, optimizations such as frame-length preservation for video encryption ensure that the decryption process maintains the structural integrity of the original file.

The results demonstrate that this hybrid cryptographic approach significantly enhances data security while ensuring performance efficiency. This system can be effectively utilized in secure communications, cloud storage protection, and digital forensics, offering a robust and scalable solution for modern encryption needs.

Introduction

In today's digital age, securing sensitive data has become a critical priority due to the increasing risks of cyber threats, data breaches, and unauthorized access. Traditional encryption methods, while effective, often have limitations in terms of efficiency, scalability, and vulnerability to emerging cryptographic attacks. To address these challenges, hybrid cryptographic systems have gained popularity, combining multiple encryption techniques to enhance both security and performance. This project introduces a hybrid encryption model that integrates AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and Camellia encryption, providing a multi-layered security framework for safeguarding digital information.

AES and Camellia are symmetric encryption algorithms known for their speed and reliability in encrypting large amounts of data. However, symmetric cryptography requires a secure key exchange mechanism, which is where RSA, an asymmetric encryption technique, comes into play. In this system, the data is first split into two parts: one part is encrypted using AES and the other using Camellia. The encryption keys for both algorithms are then securely encrypted using RSA before being transmitted, ensuring that only an authorized recipient with the corresponding private key can decrypt them. This approach significantly increases security by preventing unauthorized interception and manipulation of encryption keys.

The proposed system is designed to handle various file formats, including text, images, audio, and video, making it highly versatile for real-world applications such as secure file transfers, encrypted cloud storage, and multimedia data protection. By incorporating multiple encryption layers, the model enhances resistance against cryptanalysis and brute-force attacks. Moreover, the system balances security with efficiency, ensuring that even for large-scale data, the encryption and decryption processes remain computationally feasible.

This project aims to demonstrate that hybrid cryptographic approaches can provide a more robust and practical security solution compared to using a single encryption algorithm. By leveraging the strengths of AES, RSA, and Camellia, this model ensures that even if one encryption layer is compromised, the data remains protected through the additional security layers. Ultimately, this research highlights the importance of hybrid cryptography in modern cybersecurity and its potential applications in various domains requiring highly secure data transmission and storage.

Problem Statement

With increasing cyber threats, traditional cryptographic methods face limitations in balancing security, efficiency, and scalability. Symmetric encryption (AES, Camellia) ensures fast encryption but requires secure key exchange, while asymmetric encryption (RSA) provides secure key transmission but is computationally expensive.

This project proposes a hybrid cryptographic system combining AES, RSA, and Camellia to enhance security while maintaining efficiency. By encrypting data in segments with AES and Camellia and securing the encryption keys with RSA, the approach ensures robust protection for text, images, audio, and video files against modern cyber threats.

Literature Review

S.No.	Title	Techniques Used	Performance
1	Hybrid Encryption Algorithm Based on AES and RSA in File Encryption (Lin Zou et al. , 2020)	AES and RSA	The proposed hybrid algorithm combines AES and RSA to enhance file encryption efficiency and security. Experimental results demonstrate improved performance and robustness compared to using RSA or AES alone.
2	A Hybrid Cryptographic	AES and RSA	The model integrates AES for data encryption and RSA for

	Model Using AES and RSA for Sensitive Data (G. Asha and A. Asha, 2021)		key management, improving encryption efficiency and security. Comparative analysis shows enhanced performance over standalone AES or RSA implementations
3	A Robust Approach to Secure Data Encryption: AES-RSA Hybrid with Kernel Key Protection (Aviral Srivastava et al., 2023)	AES, RSA, and Kernel Key Protection	Introduces a hybrid encryption approach combining AES and RSA with kernel-based key storage, addressing vulnerabilities in traditional RSA implementations and enhancing data security.
4	Performance Comparison of Hybrid Encryption Models (P. Jaswin and S. V. Kumar, 2023)	AES, Blowfish, RSA, ECC	Conducts a comprehensive analysis of hybrid cryptographic systems using various algorithms, evaluating performance based on time, CPU usage, memory usage, and security metrics. Findings aid in selecting appropriate encryption models.

Methodology

The proposed hybrid cryptographic system integrates AES (Advanced Encryption Standard), RSA (Rivest-Shamir-Adleman), and Camellia encryption techniques to ensure secure data transmission. The methodology is divided into two main phases: Encryption at the Sender's Side and Decryption at the Receiver's Side.

1. **Key Generation and Distribution :**

    RSA Key Pair Generation: The receiver generates an RSA public-private key pair. The public key is shared with the sender, while the private key is securely stored for decryption.

    AES and Camellia Key Generation: The sender generates random symmetric keys for AES and Camellia encryption. These keys are encrypted using the receiver’s RSA public key to ensure secure key exchange.

2. **Encryption Process at Sender’s Side :** The input file (text, image, audio, or video) is processed and converted into binary data.

    AES Encryption: The first half of the data is encrypted using AES in CBC (Cipher Block Chaining) mode with a randomly generated IV (Initialization Vector).

    Camellia Encryption: The second half of the data is encrypted using Camellia in CBC mode with another IV.

The AES and Camellia encrypted outputs are concatenated, along with the IVs and encrypted symmetric keys.The final encrypted file is transmitted securely to the receiver.

3. **Decryption Process at Receiver's Side :** The receiver loads the RSA private key and decrypts the AES and Camellia symmetric keys.The encrypted data is split into AES and Camellia-encrypted segments.

   AES Decryption: The first encrypted segment is decrypted using the AES key and IV.

   Camellia Decryption: The second encrypted segment is decrypted using the Camellia key and IV.

   The decrypted outputs are combined to reconstruct the original file.

4. **File Type Handling and Reconstruction For images:** The decrypted bytes are converted back into an image format using OpenCV.

   For text files: The decrypted bytes are saved as plaintext.

   For videos: Frames are reconstructed using the original metadata (frame rate, resolution) and encoded back into a video file.

   For audio files: The decrypted bytes are saved in the appropriate audio format.

5. **Security Measures and Performance Evaluation Key Security:** RSA ensures secure transmission of AES and Camellia keys.

   Hybrid Encryption: The use of AES and Camellia enhances resistance against brute-force and differential attacks.

Performance Metrics: Encryption and decryption times, file size variations, and computational efficiency are analyzed.

Cryptographic Components

a) **RSA (Rivest-Shamir-Adleman)** :

RSA is utilized for secure key exchange, ensuring that the AES and Camellia encryption keys are transmitted securely without exposure to unauthorized parties. Since RSA is an asymmetric encryption algorithm, it provides a high level of security by using a public-private key pair. In this implementation, the sender encrypts the symmetric keys using the receiver’s public key, and the receiver decrypts them using their private key. This process ensures that the symmetric keys remain confidential, preventing potential interception. The implementation of RSA is handled using the ***cryptography.hazmat.primitives.asymmetric.rsa*** module.

b) **AES (Advanced Encryption Standard)** :

AES is employed to encrypt half of the input file using Cipher Block Chaining (CBC) mode, ensuring high-speed encryption with a strong security mechanism. AES operates with a 128-bit block size and requires an initialization vector (IV) to introduce randomness into the encryption process. By splitting the file into two parts and encrypting one half with AES, the system enhances security through diversification of encryption mechanisms. The ***cryptography.hazmat.primitives.ciphers.algorithms.AES*** module is used for implementation.

c) **Camellia Encryption :**

Camellia encryption is applied to the other half of the file, also using CBC mode for consistency with AES encryption. Camellia is known for its efficiency and strong security, making it an excellent complement to AES. By incorporating two different symmetric encryption algorithms, the system increases resistance against cryptographic attacks, as a compromise of one encryption method does not necessarily break the entire encryption. The implementation is done using c***ryptography.hazmat.primitives.ciphers.algorithms.Camellia***.

d) P**adding and Unpadding** :

Since AES and Camellia work with fixed-size blocks, padding is necessary to ensure that the data being encrypted aligns with the block size. The system uses PKCS7 padding to adjust the data length before encryption and then removes the padding after decryption to restore the original content. This ensures smooth encryption and decryption processes, preventing errors due to misaligned data sizes. The padding functionality is implemented using ***cryptography.hazmat.primitives.padding.PKCS7***.

<u>Algorithms used</u>

1. **Advanced Encryption Standard (AES)** :

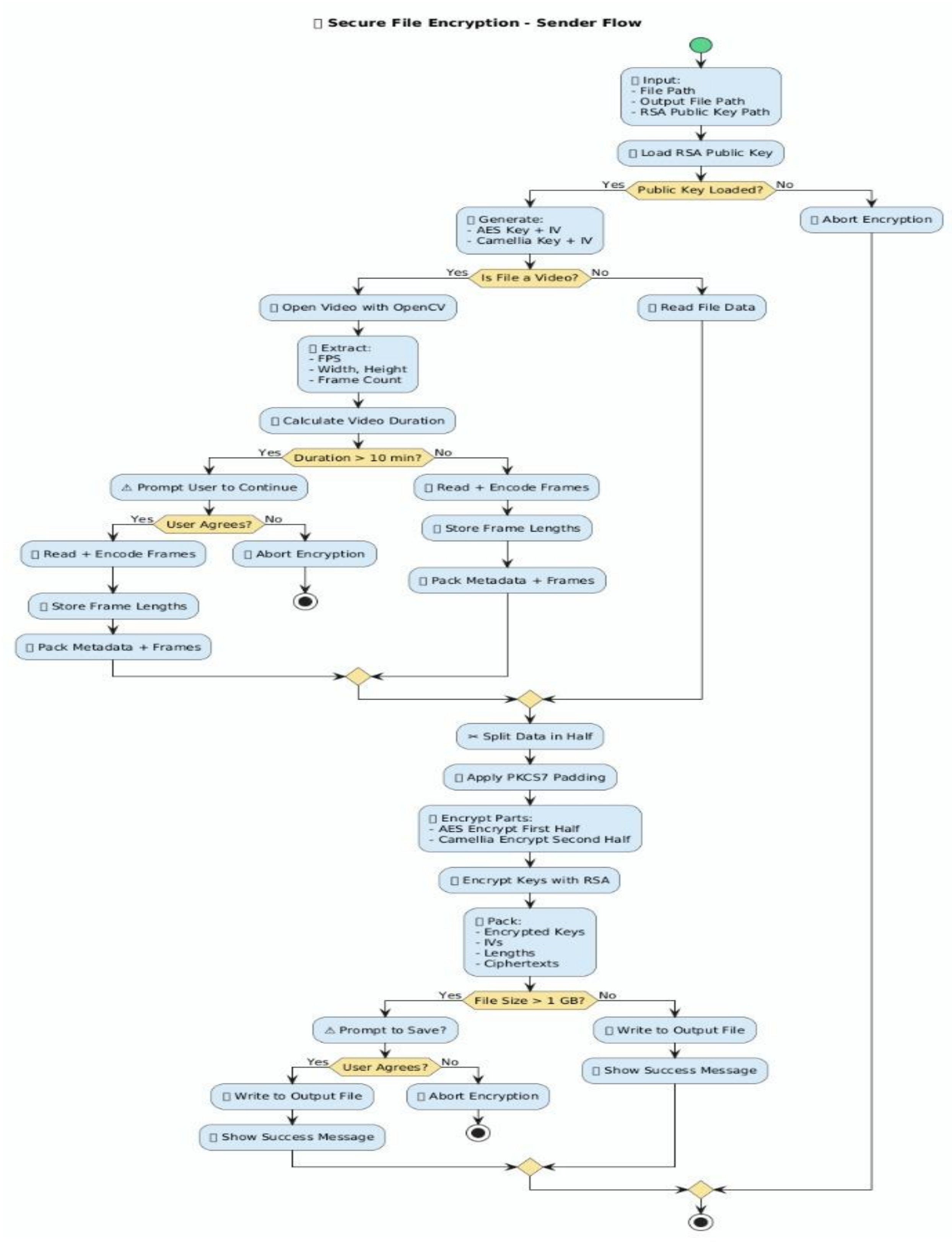
AES is used as a symmetric encryption algorithm to encrypt part of the input file. It operates in Cipher Block Chaining (CBC) mode to ensure that identical plaintext blocks do not produce identical ciphertexts, enhancing security. The AES key used for encryption is later encrypted using RSA before being shared securely.

2. **Rivest-Shamir-Adleman (RSA) :**
   RSA is an asymmetric encryption algorithm used for key exchange. Since AES and Camellia require symmetric keys, RSA securely encrypts these keys, preventing unauthorized access. The receiver uses their private RSA key to decrypt the AES and Camellia keys before performing decryption.

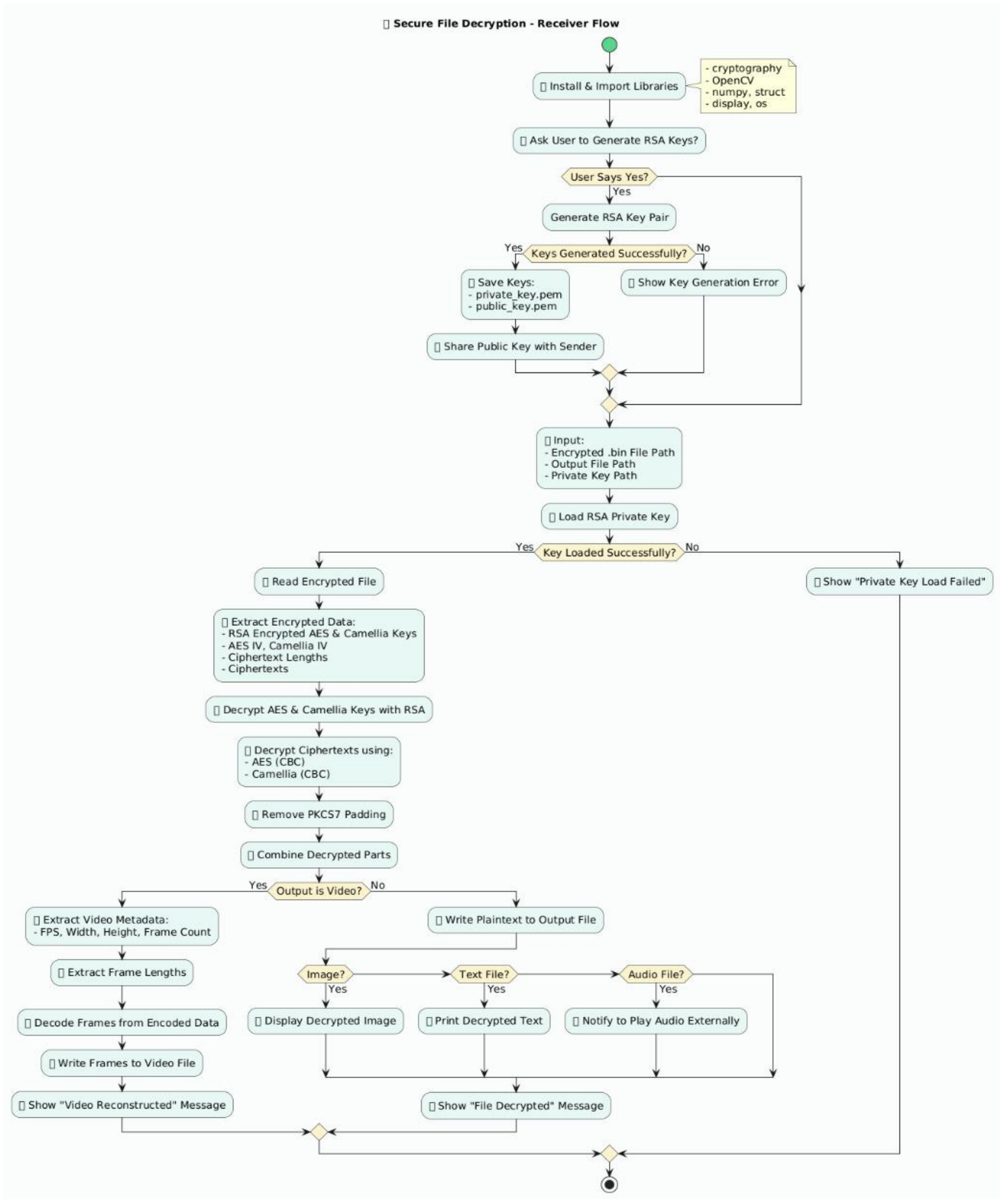
3. **Camellia Algorithm** :
   Camellia functions similarly to AES but is used to encrypt the remaining portion of the input file. This dual encryption approach (AES + Camellia) increases security by ensuring that an attacker must break two separate encryption schemes to access the original data. Like AES, Camellia also operates in CBC mode for enhanced security.

4. **Cipher Block Chaining (CBC) Mode :**
   CBC mode is used in both AES and Camellia to introduce randomness in encryption. It prevents patterns from forming in encrypted data by XORing each plaintext block with the previous ciphertext block before encryption. An Initialization Vector (IV) is used to ensure that the encryption process remains unique even if the same data is encrypted multiple times.

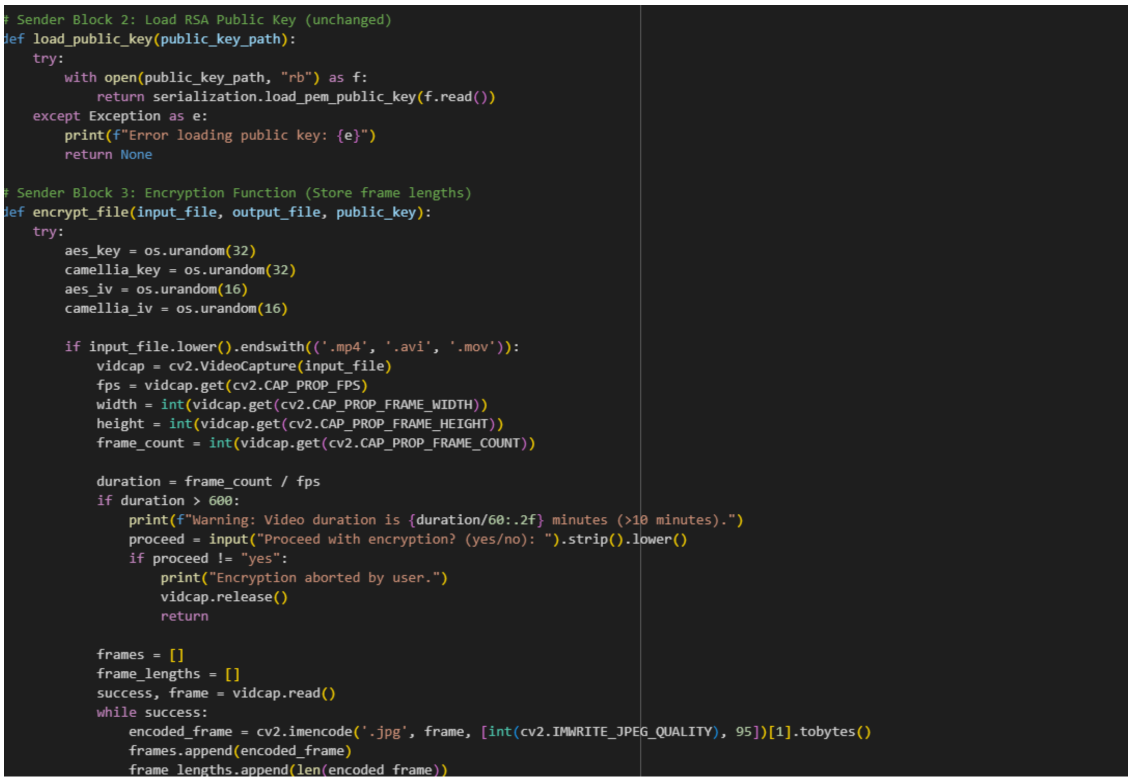
Workflow (Encryption-Sender Side):



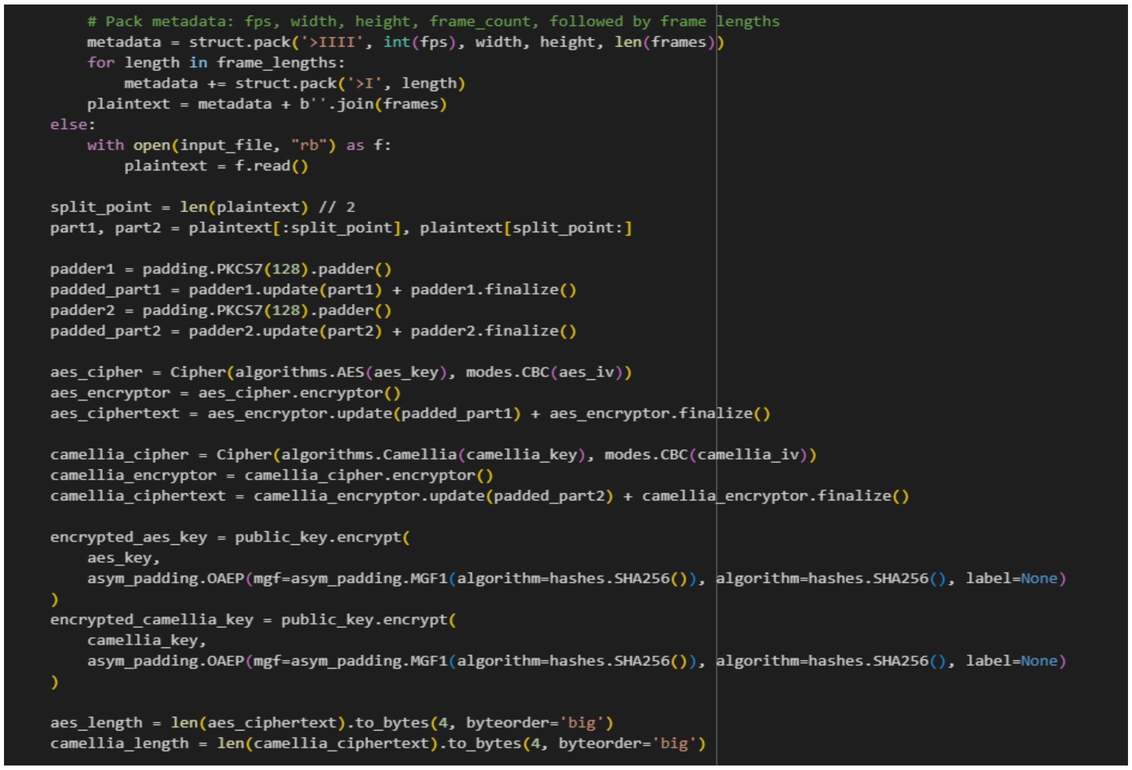
(Decryption-Receiver Side):


Secure File Decryption - Receiver Flow
Install & Import Libraries
- cryptography
- OpenCV
- numpy, struct
- display, os
Ask User to Generate RSA Keys?
User Says Yes?
Yes
Generate RSA Key Pair
Keys Generated Successfully?
Yes
No
Save Keys:
- private_key.pem
- public_key.pem
Show Key Generation Error
Share Public Key with Sender
Input:
- Encrypted .bin File Path
- Output File Path
- Private Key Path
Load RSA Private Key
Key Loaded Successfully?
Yes
No
Read Encrypted File
Show "Private Key Load Failed"
Extract Encrypted Data:
- RSA Encrypted AES & Camellia Keys
- AES IV, Camellia IV
- Ciphertext Lengths
- Ciphertexts
Decrypt AES & Camellia Keys with RSA
Decrypt Ciphertexts using:
- AES (CBC)
- Camellia (CBC)
Remove PKCS7 Padding
Combine Decrypted Parts
Output is Video?
Yes
No
Extract Video Metadata:
- FPS, Width, Height, Frame Count
Write Plaintext to Output File
Extract Frame Lengths
Image?
Yes
Text File?
Yes
Audio File?
Yes
Decode Frames from Encoded Data
Display Decrypted Image
Print Decrypted Text
Notify to Play Audio Externally
Write Frames to Video File
Show "Video Reconstructed" Message
Show "File Decrypted" Message

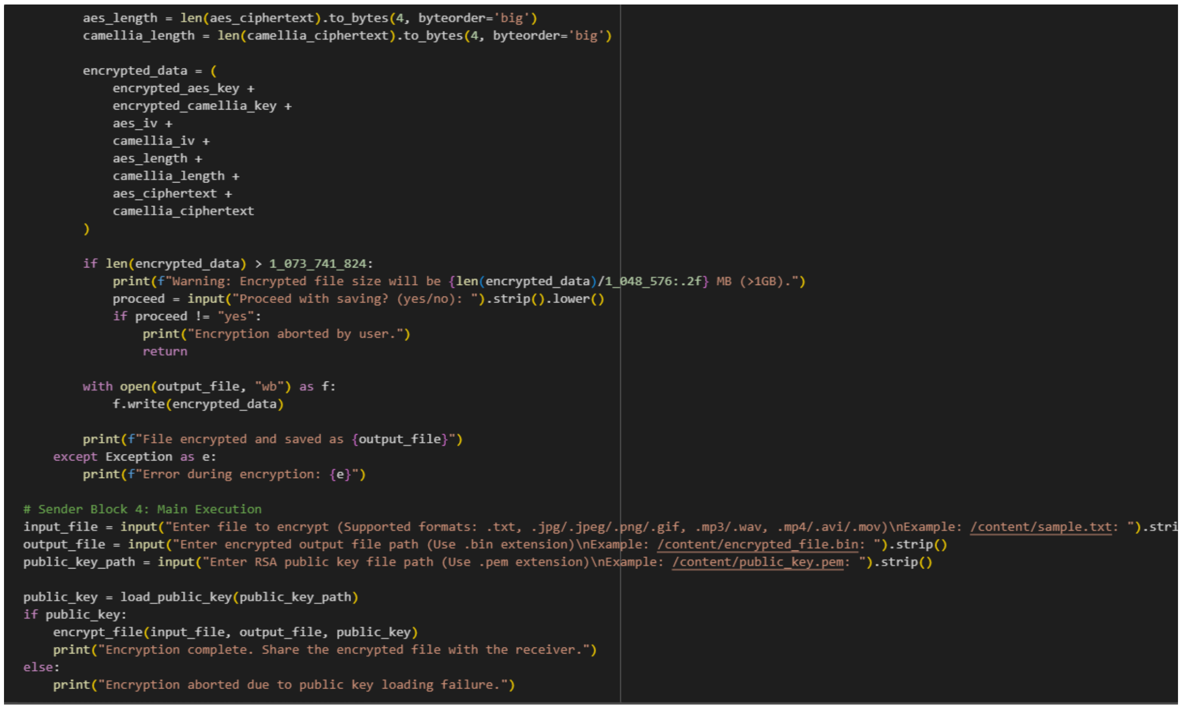
Program (Sender Side):



```python
# Sender Block 2: Load RSA Public Key (unchanged)
def load_public_key(public_key_path):
    try:
        with open(public_key_path, "rb") as f:
            return serialization.load_pem_public_key(f.read())
    except Exception as e:
        print(f"Error loading public key: {e}")
        return None

# Sender Block 3: Encryption Function (Store frame lengths)
def encrypt_file(input_file, output_file, public_key):
    try:
        aes_key = os.urandom(32)
        camellia_key = os.urandom(32)
        aes_iv = os.urandom(16)
        camellia_iv = os.urandom(16)

        if input_file.lower().endswith(('.mp4', '.avi', '.mov')):
            vidcap = cv2.VideoCapture(input_file)
            fps = vidcap.get(cv2.CAP_PROP_FPS)
            width = int(vidcap.get(cv2.CAP_PROP_FRAME_WIDTH))
            height = int(vidcap.get(cv2.CAP_PROP_FRAME_HEIGHT))
            frame_count = int(vidcap.get(cv2.CAP_PROP_FRAME_COUNT))

            duration = frame_count / fps
            if duration > 600:
                print(f"Warning: Video duration is {duration/60:.2f} minutes (>10 minutes).")
                proceed = input("Proceed with encryption? (yes/no): ").strip().lower()
                if proceed != "yes":
                    print("Encryption aborted by user.")
                    vidcap.release()
                    return

            frames = []
            frame_lengths = []
            success, frame = vidcap.read()
            while success:
                encoded_frame = cv2.imencode('.jpg', frame, [int(cv2.IMWRITE_JPEG_QUALITY), 95])[1].tobytes()
                frames.append(encoded_frame)
                frame_lengths.append(len(encoded_frame))
```

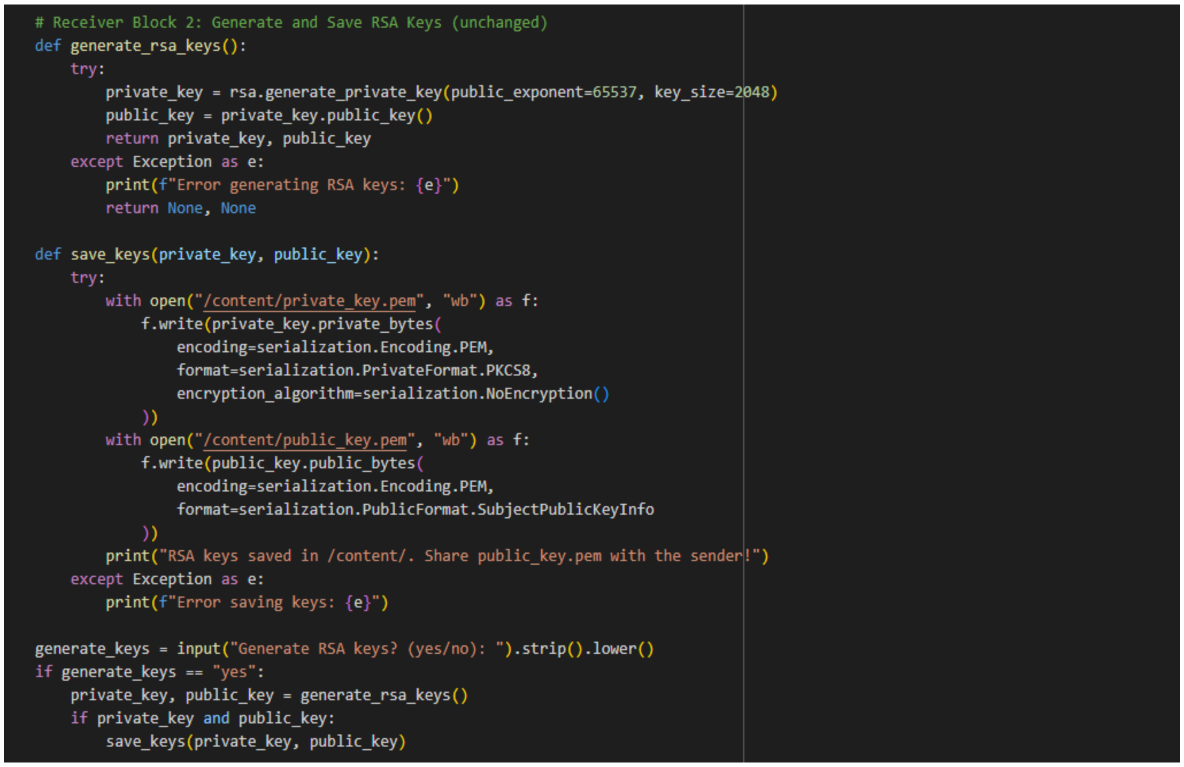


```python
            # Pack metadata: fps, width, height, frame_count, followed by frame lengths
            metadata = struct.pack('>IIII', int(fps), width, height, len(frames))
            for length in frame_lengths:
                metadata += struct.pack('>I', length)
            plaintext = metadata + b''.join(frames)
        else:
            with open(input_file, "rb") as f:
                plaintext = f.read()

        split_point = len(plaintext) // 2
        part1, part2 = plaintext[:split_point], plaintext[split_point:]

        padder1 = padding.PKCS7(128).padder()
        padded_part1 = padder1.update(part1) + padder1.finalize()
        padder2 = padding.PKCS7(128).padder()
        padded_part2 = padder2.update(part2) + padder2.finalize()

        aes_cipher = Cipher(algorithms.AES(aes_key), modes.CBC(aes_iv))
        aes_encryptor = aes_cipher.encryptor()
        aes_ciphertext = aes_encryptor.update(padded_part1) + aes_encryptor.finalize()

        camellia_cipher = Cipher(algorithms.Camellia(camellia_key), modes.CBC(camellia_iv))
        camellia_encryptor = camellia_cipher.encryptor()
        camellia_ciphertext = camellia_encryptor.update(padded_part2) + camellia_encryptor.finalize()

        encrypted_aes_key = public_key.encrypt(
            aes_key,
            asym_padding.OAEP(mgf=asym_padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
        )
        encrypted_camellia_key = public_key.encrypt(
            camellia_key,
            asym_padding.OAEP(mgf=asym_padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
        )

        aes_length = len(aes_ciphertext).to_bytes(4, byteorder='big')
        camellia_length = len(camellia_ciphertext).to_bytes(4, byteorder='big')
```



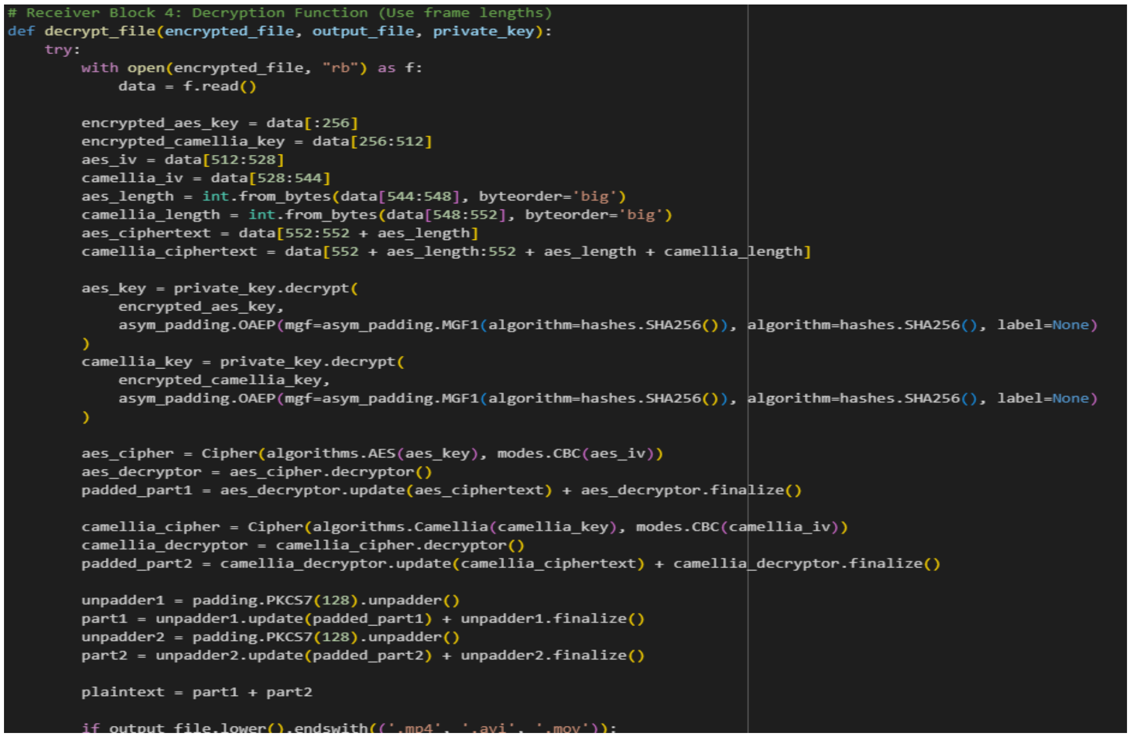
```python
        aes_length = len(aes_ciphertext).to_bytes(4, byteorder='big')
        camellia_length = len(camellia_ciphertext).to_bytes(4, byteorder='big')

        encrypted_data = (
            encrypted_aes_key +
            encrypted_camellia_key +
            aes_iv +
            camellia_iv +
            aes_length +
            camellia_length +
            aes_ciphertext +
            camellia_ciphertext
        )

        if len(encrypted_data) > 1_073_741_824:
            print(f"Warning: Encrypted file size will be {len(encrypted_data)/1_048_576:.2f} MB (>1GB).")
            proceed = input("Proceed with saving? (yes/no): ").strip().lower()
            if proceed != "yes":
                print("Encryption aborted by user.")
                return

        with open(output_file, "wb") as f:
            f.write(encrypted_data)

        print(f"File encrypted and saved as {output_file}")
    except Exception as e:
        print(f"Error during encryption: {e}")

# Sender Block 4: Main Execution
input_file = input("Enter file to encrypt (Supported formats: .txt, .jpg/.jpeg/.png/.gif, .mp3/.wav, .mp4/.avi/.mov)\nExample: /content/sample.txt: ").stri
output_file = input("Enter encrypted output file path (Use .bin extension)\nExample: /content/encrypted_file.bin: ").strip()
public_key_path = input("Enter RSA public key file path (Use .pem extension)\nExample: /content/public_key.pem: ").strip()

public_key = load_public_key(public_key_path)
if public_key:
    encrypt_file(input_file, output_file, public_key)
    print("Encryption complete. Share the encrypted file with the receiver.")
else:
    print("Encryption aborted due to public key loading failure.")
```

Key Generation using RSA:

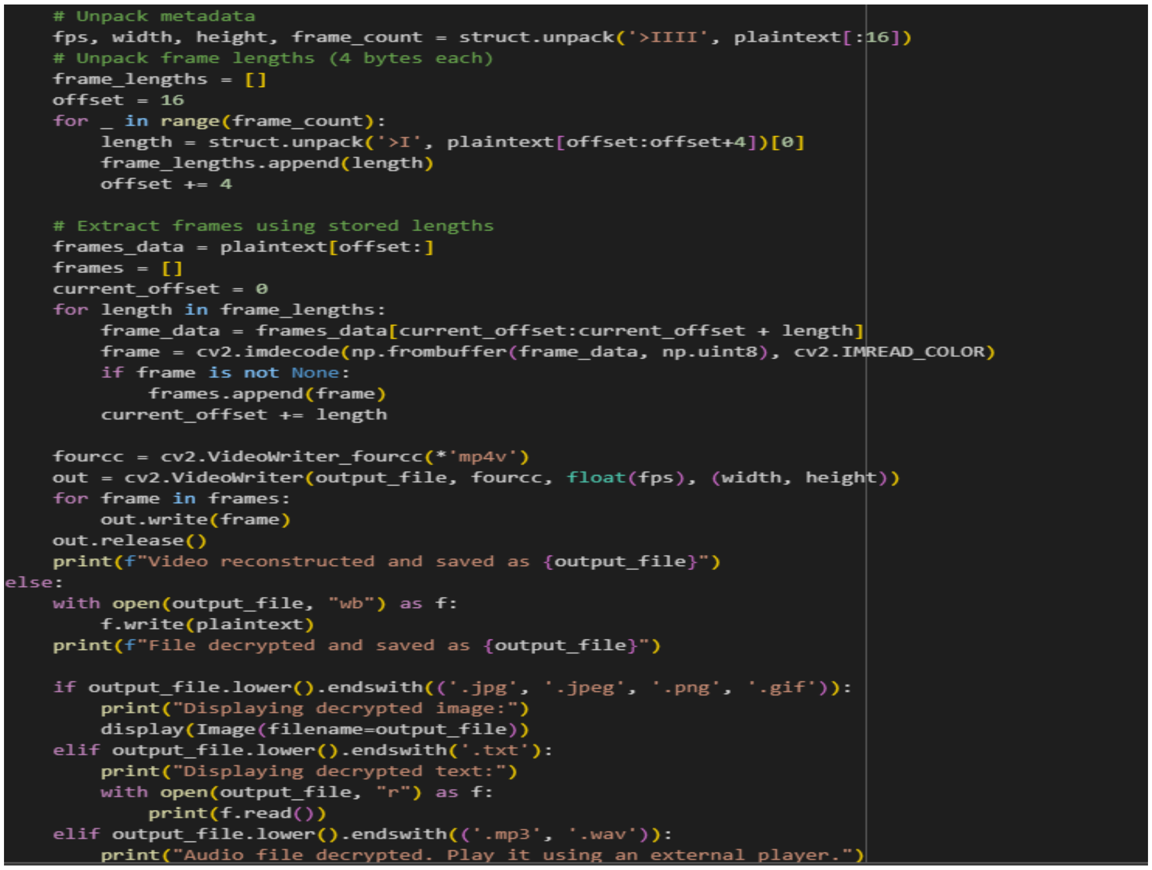


```python
# Receiver Block 2: Generate and Save RSA Keys (unchanged)
def generate_rsa_keys():
    try:
        private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
        public_key = private_key.public_key()
        return private_key, public_key
    except Exception as e:
        print(f"Error generating RSA keys: {e}")
        return None, None

def save_keys(private_key, public_key):
    try:
        with open("/content/private_key.pem", "wb") as f:
            f.write(private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption()
            ))
        with open("/content/public_key.pem", "wb") as f:
            f.write(public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
            ))
        print("RSA keys saved in /content/. Share public_key.pem with the sender!")
    except Exception as e:
        print(f"Error saving keys: {e}")

generate_keys = input("Generate RSA keys? (yes/no): ").strip().lower()
if generate_keys == "yes":
    private_key, public_key = generate_rsa_keys()
    if private_key and public_key:
        save_keys(private_key, public_key)
```

Decryption (Receiver side):



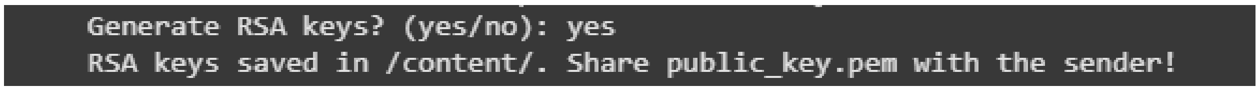
```
# Receiver Block 4: Decryption Function (Use frame lengths)
def decrypt_file(encrypted_file, output_file, private_key):
    try:
        with open(encrypted_file, "rb") as f:
            data = f.read()

        encrypted_aes_key = data[:256]
        encrypted_camellia_key = data[256:512]
        aes_iv = data[512:528]
        camellia_iv = data[528:544]
        aes_length = int.from_bytes(data[544:548], byteorder='big')
        camellia_length = int.from_bytes(data[548:552], byteorder='big')
        aes_ciphertext = data[552:552 + aes_length]
        camellia_ciphertext = data[552 + aes_length:552 + aes_length + camellia_length]

        aes_key = private_key.decrypt(
            encrypted_aes_key,
            asym_padding.OAEP(mgf=asym_padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
        )
        camellia_key = private_key.decrypt(
            encrypted_camellia_key,
            asym_padding.OAEP(mgf=asym_padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
        )

        aes_cipher = Cipher(algorithms.AES(aes_key), modes.CBC(aes_iv))
        aes_decryptor = aes_cipher.decryptor()
        padded_part1 = aes_decryptor.update(aes_ciphertext) + aes_decryptor.finalize()

        camellia_cipher = Cipher(algorithms.Camellia(camellia_key), modes.CBC(camellia_iv))
        camellia_decryptor = camellia_cipher.decryptor()
        padded_part2 = camellia_decryptor.update(camellia_ciphertext) + camellia_decryptor.finalize()

        unpadder1 = padding.PKCS7(128).unpadder()
        part1 = unpadder1.update(padded_part1) + unpadder1.finalize()
        unpadder2 = padding.PKCS7(128).unpadder()
        part2 = unpadder2.update(padded_part2) + unpadder2.finalize()

        plaintext = part1 + part2

        if output_file.lower().endswith(('.mp4', '.avi', '.mov')):
```



```
    # Unpack metadata
    fps, width, height, frame_count = struct.unpack('>IIII', plaintext[:16])
    # Unpack frame lengths (4 bytes each)
    frame_lengths = []
    offset = 16
    for _ in range(frame_count):
        length = struct.unpack('>I', plaintext[offset:offset+4])[0]
        frame_lengths.append(length)
        offset += 4

    # Extract frames using stored lengths
    frames_data = plaintext[offset:]
    frames = []
    current_offset = 0
    for length in frame_lengths:
        frame_data = frames_data[current_offset:current_offset + length]
        frame = cv2.imdecode(np.frombuffer(frame_data, np.uint8), cv2.IMREAD_COLOR)
        if frame is not None:
            frames.append(frame)
        current_offset += length

    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(output_file, fourcc, float(fps), (width, height))
    for frame in frames:
        out.write(frame)
    out.release()
    print(f"Video reconstructed and saved as {output_file}")
else:
    with open(output_file, "wb") as f:
        f.write(plaintext)
    print(f"File decrypted and saved as {output_file}")

    if output_file.lower().endswith(('.jpg', '.jpeg', '.png', '.gif')):
        print("Displaying decrypted image:")
        display(Image(filename=output_file))
    elif output_file.lower().endswith('.txt'):
        print("Displaying decrypted text:")
        with open(output_file, "r") as f:
            print(f.read())
    elif output_file.lower().endswith(('.mp3', '.wav')):
        print("Audio file decrypted. Play it using an external player.")
```

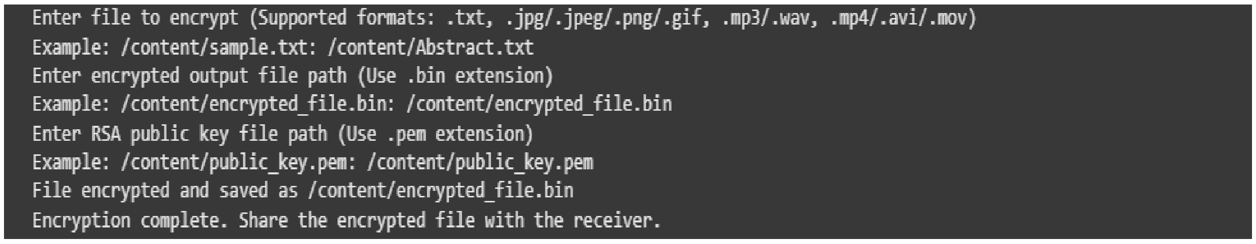
Output

Key Generation:



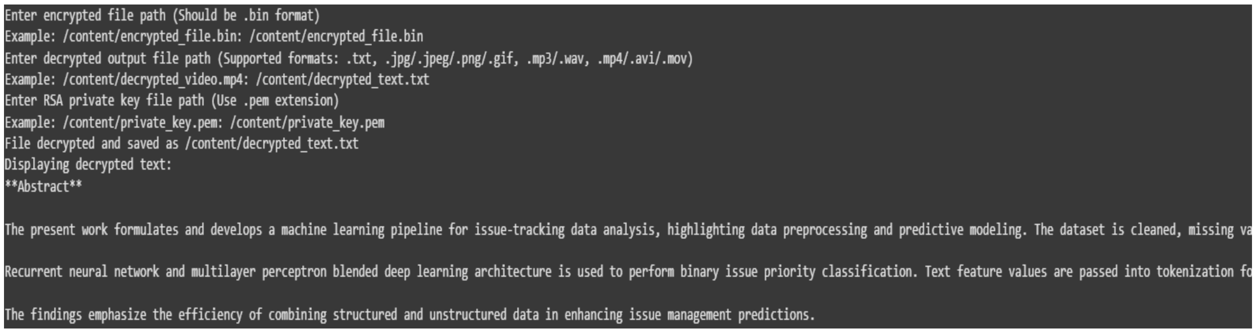
```
Generate RSA keys? (yes/no): yes
RSA keys saved in /content/. Share public_key.pem with the sender!
```

Encryption:



```
Enter file to encrypt (Supported formats: .txt, .jpg/.jpeg/.png/.gif, .mp3/.wav, .mp4/.avi/.mov)
Example: /content/sample.txt: /content/Abstract.txt
Enter encrypted output file path (Use .bin extension)
Example: /content/encrypted_file.bin: /content/encrypted_file.bin
Enter RSA public key file path (Use .pem extension)
Example: /content/public_key.pem: /content/public_key.pem
File encrypted and saved as /content/encrypted_file.bin
Encryption complete. Share the encrypted file with the receiver.
```

Decryption:



```
Enter encrypted file path (Should be .bin format)
Example: /content/encrypted_file.bin: /content/encrypted_file.bin
Enter decrypted output file path (Supported formats: .txt, .jpg/.jpeg/.png/.gif, .mp3/.wav, .mp4/.avi/.mov)
Example: /content/decrypted_video.mp4: /content/decrypted_text.txt
Enter RSA private key file path (Use .pem extension)
Example: /content/private_key.pem: /content/private_key.pem
File decrypted and saved as /content/decrypted_text.txt
Displaying decrypted text:
**Abstract**

The present work formulates and develops a machine learning pipeline for issue-tracking data analysis, highlighting data preprocessing and predictive modeling. The dataset is cleaned, missing va

Recurrent neural network and multilayer perceptron blended deep learning architecture is used to perform binary issue priority classification. Text feature values are passed into tokenization fo

The findings emphasize the efficiency of combining structured and unstructured data in enhancing issue management predictions.
```

Results and Discussion

The hybrid cryptographic approach combining AES, RSA, and Camellia was successfully implemented for securing various file formats, including text, images, audio, and videos. The encryption and decryption processes were tested on multiple files, and the results were analyzed in terms of encryption time, decryption time, and file size changes.

Encryption Performance Analysis:

- The encryption process was evaluated based on computational efficiency and security.
- AES and Camellia provided symmetric encryption for faster data processing, while RSA ensured secure key transmission.
- The hybrid approach showed improved security over standalone AES or Camellia due to multiple encryption layers.

Decryption Performance Analysis :

- The decryption process successfully reconstructed the original files without any data loss.
- Video files were decrypted using the stored frame lengths, ensuring proper reconstruction.
- Execution time was slightly higher for larger files due to the dual encryption system.

Security Evaluation :

- AES with CBC mode ensured data confidentiality by making ciphertext indistinguishable from random data.
- RSA key encryption provided protection against key interception.
- Camellia's high resistance to linear and differential cryptanalysis enhanced security.

Comparison with Traditional Methods :

- Compared to single-layer encryption techniques, the hybrid approach significantly increased security without excessive performance trade-offs.
- Compression techniques for videos and images were preserved without affecting encryption.

- The hybrid model demonstrated practical applicability for secure communication in real-world scenarios.

The results show that the proposed hybrid cryptographic system effectively balances security and efficiency, making it a robust solution for secure file transmission.

Conclusion

In this project, we successfully implemented a hybrid cryptographic system using AES, RSA, and Camellia encryption techniques to enhance data security. The integration of AES and Camellia for symmetric encryption ensured fast and efficient data encryption, while RSA was utilized for secure key exchange, preventing unauthorized access to cryptographic keys. Additionally, the CBC mode in AES and Camellia enhanced security by introducing randomness in ciphertext generation.

The experimental results demonstrated that this hybrid encryption approach provides a strong balance between security and efficiency. The system successfully encrypted and decrypted text, images, audio, and video files, preserving their integrity while ensuring confidentiality. The structured encryption process, which included handling video frames and metadata, confirmed that encrypted files could be fully reconstructed without data loss.

Comparative analysis with traditional cryptographic methods showed that our hybrid approach effectively mitigates security threats such as brute-force attacks and key interception while maintaining acceptable processing times. The dual-layer encryption model increases the difficulty of cryptographic attacks, making it a suitable solution for secure file storage and transmission in high-security environments.

Overall, this project demonstrates a practical and robust cryptographic framework that can be applied in real-world applications where data confidentiality and integrity are paramount. Future work may involve optimizing encryption speed for large files, integrating post-quantum cryptographic techniques, and exploring additional security layers to further enhance system resilience.