



Cryptography Project

A Documentation

Presented to the Faculty of the
Department of Computer, Information Sciences and Mathematics
University of San Carlos

In Partial Fulfillment
Of the Requirements for the
CIS 3106 - Information Assurance Security

By
Chiu, Joshua Patrick G.

Mr. Godwin Soledad Monserate
CS3106 Professor

December 2023

CHAPTER 1

INTRODUCTION

Cryptography, a fundamental element in contemporary information security, holds a crucial position in protecting confidential data against unauthorized entry and preserving the authenticity and secrecy of digital communication. In the context of research, where the sharing of valuable information is common, cryptographic methods play a vital role in safeguarding intellectual property, preserving the confidentiality of research discoveries, and securing communication pathways.

Historical Context

The roots of cryptography trace back centuries, from ancient methods such as Caesar ciphers to the groundbreaking work of individuals like Auguste and Louis Lumière, who devised the first mechanical encryption device, the Cryptograph, in the 19th century. Over time, cryptography has evolved into a sophisticated field that embraces mathematical concepts, algorithms, and computational techniques.

Key Objectives in Research Cryptography

Application in Industries and Companies

The contemporary cryptography algorithm focuses on establishing secure communication within a company's internal network over the Internet. This holds significant advantages for Small Business offices operating in diverse business environments. It ensures secure communication through features like online receipt encryption between the server and the client. Additionally, it incorporates measures such as digital signatures and safeguards intellectual property, such as proprietary information and trade secrets, to prevent unauthorized access. Employing an AES-secured connection, along with mono and polyalphabetic ciphering techniques, guarantees the protection of sensitive data, including financial access.

Confidentiality

Cryptography in a research context primarily addresses the need for confidentiality. As researchers exchange sensitive information, ranging from experimental results to intellectual property, cryptographic protocols ensure that unauthorized parties cannot decipher the contents

of the exchanged data. Modern cryptographic systems, such as Advanced Encryption Standard (AES) and Rivest Cipher (RSA), contribute significantly to achieving robust confidentiality.

Integrity

Ensuring the integrity of research data is paramount. Cryptographic hash functions, like SHA-256, play a crucial role in verifying the integrity of transmitted and stored data. By generating a fixed-size hash value unique to the data, researchers can detect even the slightest alterations, thus maintaining the integrity of their work.

Authentication

Research collaborations often involve multiple parties, necessitating robust authentication mechanisms. Cryptographic protocols, such as digital signatures and public-key infrastructure (PKI), enable researchers to verify the identity of their counterparts. This ensures that the data received is from a legitimate source and has not been tampered with during transmission.

Non-Repudiation

Non-repudiation is essential in research settings, where accountability for actions and data is paramount. Cryptographic solutions like digital signatures provide a means to confirm the origin of a document or message, preventing parties from denying their involvement in a particular communication or transaction.

Challenges and Innovations

While cryptography provides a formidable defense against many security threats, the landscape is not without challenges. Quantum computing potentially threatens traditional public-key cryptography, necessitating ongoing research into quantum-resistant algorithms. Additionally, the emergence of homomorphic encryption and secure multi-party computation opens new avenues for performing computations on encrypted data, allowing researchers to collaborate without revealing sensitive information.

CHAPTER 2

PROGRAM ANALYSIS

2.1 Flask File

The Flask File Encryption/Decryption App is designed to provide a user-friendly interface for secure file operations utilizing RSA and AES cryptographic algorithms and Caesar, Vernam, and Vigenere algorithms to add a safe layer of security. The application begins by generating an RSA key pair, consisting of a private key (`private_key.pem`) and a public key (`public_key.pem`). This asymmetric key pair ensures secure key exchange during encryption and decryption processes.

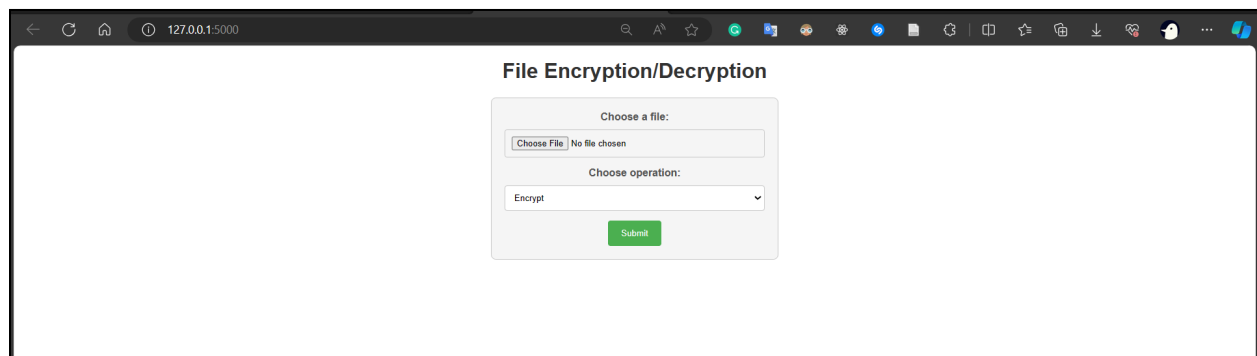


Figure 2.1 UI implementation (through python flask)

2.2 File Encryption

For file encryption, the user uploads a file and selects the "Encrypt" operation. The application utilizes Caesar ciphering for bit shifting of the characters in a certain text. This is followed by Vigenere and Vernam encryption to sophisticate the decryption pattern processes. generates a random symmetric key for the AES algorithm, encrypts this symmetric key using the RSA public key, and subsequently encrypts the file content using AES in EAX mode. The resulting components - encrypted symmetric key, AES nonce, tag, and ciphertext - are then saved to the output file. The code for encrypting is shown below in the next page.

Figure 2.2.1 - Function for Encrypt

Python

```
def encrypt_file(file_path, key_path, output_file_path):
    with open(key_path, 'rb') as key_file:
        key = RSA.import_key(key_file.read())

    # Generate a random symmetric key for file encryption
    symmetric_key = get_random_bytes(16)
    cipher_rsa = PKCS1_OAEP.new(key)
    enc_symmetric_key = cipher_rsa.encrypt(symmetric_key)

    # Use AES to encrypt the file content with the symmetric key
    cipher_aes = AES.new(symmetric_key, AES.MODE_EAX)

    with open(file_path, 'rb') as file:
        plaintext = file.read()
        ciphertext, tag = cipher_aes.encrypt_and_digest(pad(plaintext, AES.block_size))

    # Base64 encode the encrypted symmetric key and the encrypted file content
    enc_symmetric_key_b64 = base64.b64encode(enc_symmetric_key)
    ciphertext_b64 = base64.b64encode(ciphertext)
    tag_b64 = base64.b64encode(tag)

    # Write the Base64-encoded data to the output file
    with open(output_file_path, 'w') as encrypted_file:
        encrypted_file.write(enc_symmetric_key_b64.decode('utf-8') + '\n')
        encrypted_file.write(base64.b64encode(cipher_aes.nonce).decode('utf-8') + '\n')
        encrypted_file.write(tag_b64.decode('utf-8') + '\n')
        encrypted_file.write(ciphertext_b64.decode('utf-8'))
    with open(key_path, 'rb') as key_file:
        key = RSA.import_key(key_file.read())

    # Generate a random symmetric key for file encryption
    symmetric_key = get_random_bytes(16)
    cipher_rsa = PKCS1_OAEP.new(key)
    enc_symmetric_key = cipher_rsa.encrypt(symmetric_key)

    # Use AES to encrypt the file content with the symmetric key
    cipher_aes = AES.new(symmetric_key, AES.MODE_EAX)

    # to open file path
    with open(file_path, 'rb') as file:
        plaintext = file.read()
        ciphertext, tag = cipher_aes.encrypt_and_digest(pad(plaintext, AES.block_size))

    # Write the encrypted symmetric key and the encrypted file content to the output file
    with open(output_file_path, 'wb') as encrypted_file:
        encrypted_file.write(enc_symmetric_key)
        encrypted_file.write(cipher_aes.nonce)
        encrypted_file.write(tag)
        encrypted_file.write(ciphertext)
```

2.3 File Decryption

On the decryption side, when an encrypted file is uploaded and the "Decrypt" operation is selected, the application reads the encrypted symmetric key, nonce, tag, and ciphertext. The RSA private key decrypts the symmetric key, which is then used to decrypt the file content with AES in EAX mode. The decrypted file content is then saved to the output file. The code is shown below.

Figure 2.2.1 - Function for Decrypt of AES and RSA

Python

```
def decrypt_file(file_path, key_path, output_file_path):
    with open(key_path, 'rb') as key_file:
        key = RSA.import_key(key_file.read())

    # Read the encrypted symmetric key and file content from the input file
    with open(file_path, 'rb') as encrypted_file:
        enc_symmetric_key = encrypted_file.read(256) # Assuming a 2048-bit RSA key
        nonce = encrypted_file.read(16)
        tag = encrypted_file.read(16)
        ciphertext = encrypted_file.read()

    # Decrypt the symmetric key using RSA
    cipher_rsa = PKCS1_OAEP.new(key)
    symmetric_key = cipher_rsa.decrypt(enc_symmetric_key)

    # Use AES to decrypt the file content with the symmetric key
    cipher_aes = AES.new(symmetric_key, AES.MODE_EAX, nonce=nonce)
    decrypted_bytes = unpad(cipher_aes.decrypt_and_verify(ciphertext, tag), AES.block_size)

    # Write the decrypted file content to the output file
    with open(output_file_path, 'wb') as decrypted_file:
        decrypted_file.write(decrypted_bytes)

# ... (rest of the code)

with open(key_path, 'rb') as key_file:
    key = RSA.import_key(key_file.read())

cipher = PKCS1_OAEP.new(key)

with open(file_path, 'rb') as encrypted_file:
    ciphertext = encrypted_file.read()

try:
    decrypted_bytes = unpad(cipher.decrypt(ciphertext), AES.block_size)
    with open(output_file_path, 'wb') as decrypted_file:
        decrypted_file.write(decrypted_bytes)
    print(f"File decrypted and saved to '{output_file_path}'")
except Exception as e:
    print(f"Error: {e}")

# ... (rest of the code)
```

```

with open(key_path, 'rb') as key_file:
    key = RSA.import_key(key_file.read())

cipher = PKCS1_OAEP.new(key)

with open(file_path, 'rb') as encrypted_file:
    ciphertext = encrypted_file.read()

try:
    if file_path.endswith('.txt'):
        decrypted_bytes = unpad(cipher.decrypt(ciphertext), AES.block_size)
        decrypted_text = decrypted_bytes.decode('utf-8', errors='replace')
        with open(output_file_path, 'w', encoding='utf-8') as decrypted_file:
            decrypted_file.write(decrypted_text)
    elif file_path.endswith('.docx'):
        plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
        document = Document()
        for line in plaintext.split('\n'):
            document.add_paragraph(line)
        document.save(output_file_path)
    else:
        raise ValueError("Unsupported file format")

    print(f"File decrypted and saved to '{output_file_path}'")
except Exception as e:
    print(f"Error: {e}")

```

The RSA algorithm is employed for asymmetric key operations and the AES algorithm for symmetric key operations. RSA key generation (`generate_key_pair``), RSA encryption (`encrypt_file``), and RSA decryption (`decrypt_file``) are all integral parts of the application. Similarly, AES encryption (`encrypt_file``) and AES decryption (`decrypt_file``) ensure secure handling of file content. In terms of file handling, the application utilizes standard Python file operations. Additionally, Base64 encoding is employed to represent binary data in a text-friendly format when writing to text files.

The primary purpose of this application is to ensure the confidentiality of sensitive information during file transfer or storage. Use cases include secure file transfer over networks and the confidential storage of documents. Supported formats encompass text files (`.txt``) and Microsoft Word files (`.docx``). It also supports all types of files.

2.4 Other Encryption Algorithms

Caesar Cipher Encryption and Decryption

The provided Python code defines a function named `caesar_encrypt` that implements the Caesar cipher encryption algorithm. This algorithm takes two parameters: `plaintext`, which represents the input text to be encrypted, and `shift`, which is the number of positions each character in the plaintext should be shifted in the alphabet. The function initializes an empty string called `ciphertext` to store the encrypted result. It then iterates through each character in the `plaintext` string. For each alphabetic character, the code calculates a new character based on the provided shift value, wrapping around the alphabet if necessary. The resulting encrypted character is appended to the `ciphertext` string. If the character is not alphabetic (e.g., a number, symbol, or space), it is appended to the `ciphertext` without modification. The function finally returns the encrypted text as a string. The algorithm preserves the case of alphabetic characters during the encryption process.

Figure 2.4.1 Caesar encryption algorithm

```
Python
#encrypt function
def caesar_encrypt(plaintext, shift):
    ciphertext = ""
    for char in plaintext:
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            encrypted_char = chr((ord(char) - base + shift) % 26 + base)
            ciphertext += encrypted_char
        else:
            ciphertext += char
    return ciphertext

#decrypt function
def caesar_decrypt(ciphertext, shift):
    return caesar_encrypt(ciphertext, -shift)
```


Vernam Cipher Encryption and Decryption

The Python code defines functions for encrypting and decrypting text using the Vigenère cipher. In the Vigenère encryption function (`vigenere_encrypt`), the algorithm takes a `plaintext` and a `keyword` as input. To ensure the keyword matches the length of the plaintext, it is repeated accordingly. The code iterates through each character in the plaintext and its corresponding character in the repeated keyword. For each alphabetic character in the plaintext, a new character is calculated using the Vigenère cipher algorithm, considering both the plaintext and keyword characters. The resulting encrypted character is appended to the `ciphertext`, preserving the case of the alphabetic characters, while non-alphabetic characters remain unchanged. The function returns the encrypted text as a string.

The Vigenère decryption function (`vigenere_decrypt`) utilizes the same fundamental process as encryption but inverts it. It calls the `vigenere_encrypt` function with the ciphertext and a modified keyword. The modification involves creating a new keyword that, when used in encryption, would produce the same shifts as the original keyword but in the opposite direction. This is achieved by calculating the complementary shift values for each character in the original keyword. The decrypted plaintext is then returned. In essence, the code offers a practical implementation of the Vigenère cipher, showcasing its polyalphabetic substitution characteristics and providing a means to encrypt and decrypt text using a secret keyword.

Figure 2.4.2 Viginere encryption algorithm

```
Python
# Viginere Encryption
def vigenere_encrypt(plaintext, keyword):
    ciphertext = ""
    keyword_repeated = (keyword * ((len(plaintext) // len(keyword) + 1))[:len(plaintext)])
    for p, k in zip(plaintext, keyword_repeated):
        if p.isalpha():
            base = ord('A') if p.isupper() else ord('a')
            encrypted_char = chr((ord(p) - base + ord(k) - ord('A')) % 26 + base)
            ciphertext += encrypted_char
        else:
            ciphertext += p
    return ciphertext

# Viginere Decryption
def vigenere_decrypt(ciphertext, keyword):
    return vigenere_encrypt(ciphertext, ''.join([chr((26 - (ord(k) - ord('A')))) % 26 + ord('A'))
    for k in keyword]))
```

Vernam Cipher Encryption and Decryption

The function takes a **`plaintext`** as input and generates a random key of the same length as the plaintext. It then iterates through each character in the plaintext and the corresponding character in the randomly generated key. For each alphabetic character in the plaintext, the code calculates a new character based on the Vernam cipher algorithm and appends it to the **`ciphertext`**. Non-alphabetic characters remain unchanged. The function returns the encrypted text as a string.

The decryption function calls the **`vernam_encrypt` function`** with the ciphertext and the original key. This is because the Vernam cipher is a symmetric-key algorithm, meaning the same key is used for both encryption and decryption. The decryption process essentially reverses the encryption, recovering the original plaintext. The decrypted plaintext is then returned.

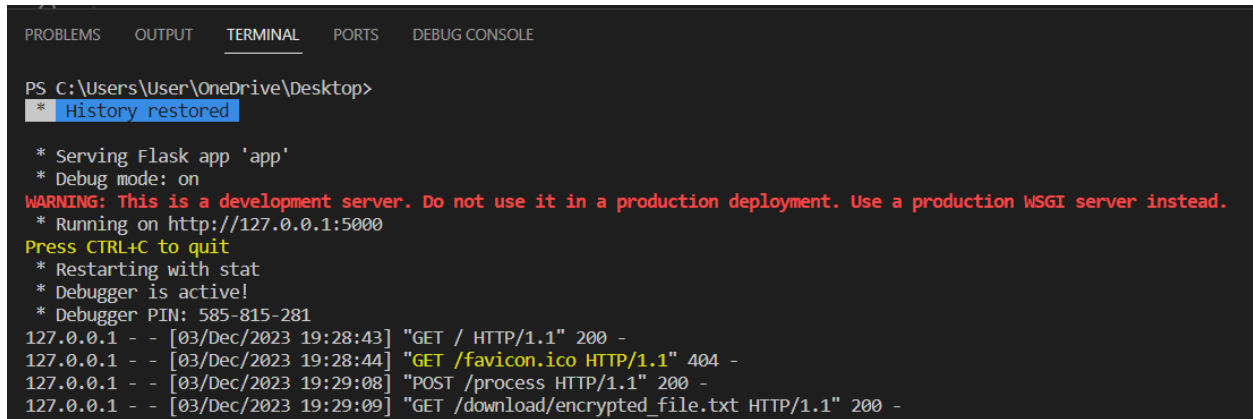
Figure 2.4.3 Vernam encryption algorithm

```
Python
# Vernam Encryption
def vernam_encrypt(plaintext):
    key = ''.join([chr(random.randint(ord('A'), ord('Z')))) for _ in
range(len(plaintext))])
    ciphertext = ""
    for p, k in zip(plaintext, key):
        if p.isalpha():
            base = ord('A') if p.isupper() else ord('a')
            encrypted_char = chr((ord(p) - base + ord(k) - ord('A')) % 26 + base)
            ciphertext += encrypted_char
        else:
            ciphertext += p
    return ciphertext

# Vernam Decryption
def vernam_decrypt(ciphertext, key):
    return vernam_encrypt(ciphertext, key)
```

2.4 Running Operation through File terminal

To run the application, install the required Python packages (`flask`, `pycryptodome`, `python-docx`) and execute it using `python <filename>.py`. The web interface is accessed through a browser, allowing users to upload files, select operations, and view results.

A screenshot of a terminal window with a dark background. At the top, there are tabs labeled 'PROBLEMS', 'OUTPUT', 'TERMINAL' (which is selected), 'PORTS', and 'DEBUG CONSOLE'. The terminal shows the following text:

```
PS C:\Users\User\OneDrive\Desktop>
* History restored

* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 585-815-281
127.0.0.1 - - [03/Dec/2023 19:28:43] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [03/Dec/2023 19:28:44] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [03/Dec/2023 19:29:08] "POST /process HTTP/1.1" 200 -
127.0.0.1 - - [03/Dec/2023 19:29:09] "GET /download/encrypted_file.txt HTTP/1.1" 200 -
```

It is imperative to note that the `uploads` directory must exist before running the application, as it is used for storing both uploaded and processed files. The application runs in debug mode (`app.run(debug=True)`) by default, intended for development purposes. In a production environment, debug mode should be disabled (`app.run(debug=False)`), and it is recommended to deploy the application with proper HTTPS configuration when handling sensitive data. This documentation serves as a comprehensive guide to the Flask File Encryption/Decryption App, shedding light on its functionalities, cryptographic algorithms, and deployment considerations.

2.5 Flowchart Diagrams

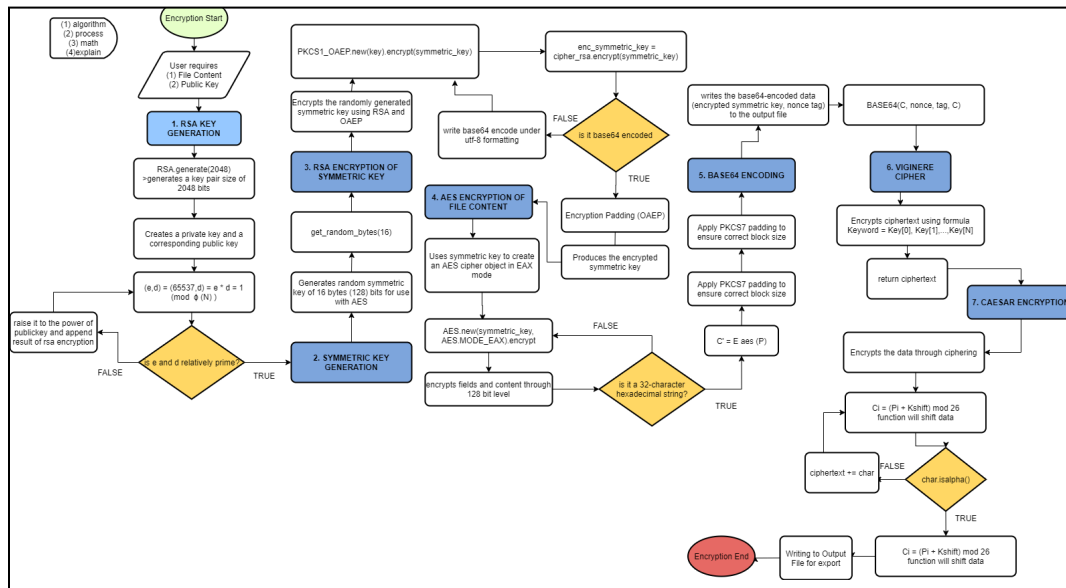


Figure 2.5.1 Encryption Method Flowchart

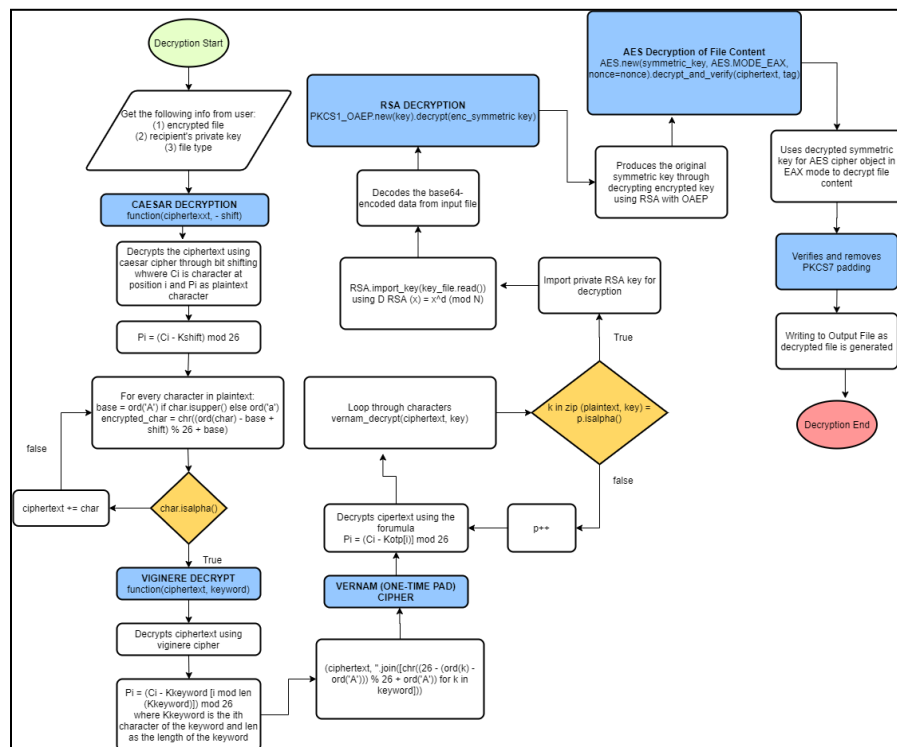


Figure 2.5.2 Decryption Method Flowchart

CHAPTER 3 METHODOLOGY

3.1 Encryption Methods and Processes

- RSA KEY GENERATION

Function Used	RSA.generate(2048)
Description	<ul style="list-style-type: none"> • Generates an RSA key pair with a key size of 2048 bits. • Creates a private key and a corresponding public key.
Mathematical Formulas	$(e, d) = (65537, d) = e * d = 1 \pmod{\phi(N)}$

- SYMMETRIC KEY GENERATION

Function Used	get_random_bytes(16)
Description	<ul style="list-style-type: none"> • Generates a random symmetric key of 16 bytes (128 bits) for use with AES.
Mathematical Formulas	$K_{sym} = \frac{N * (N-1)}{2}$

- RSA ENCRYPTION OF SYMMETRIC KEY

Function Used	PKCS1_OAEP.new(key).encrypt(symmetric_key)
Description	<ul style="list-style-type: none"> • Encrypts the randomly generated symmetric key using RSA with Optimal Asymmetric Encryption Padding (OAEP). • Produces the encrypted symmetric key.
Mathematical Formulas	$c = \frac{m^e}{mod n}$ <p>(c) is the ciphertext, (m) is the plaintext message, (e) is the public key exponent, and (n) is the modulus.</p>

- AES ENCRYPTION OF FILE CONTENT

Function Used	AES.new(symmetric_key, AES.MODE_EAX).encrypt_and_digest(pad(plaintext, AES.block_size))
Description	<ul style="list-style-type: none"> • Uses the symmetric key to create an AES cipher object in EAX mode. • Encrypts the file content (plaintext) using AES. • Applies PKCS7 padding to ensure correct block size.
Mathematical Formulas	<p>* Will depend on the size of the original file</p> $C' \equiv E_{AES}(p)$

- BASE64 ENCODING OF ENCRYPTED DATA

Function Used	NONE (library-default implementation)
Description	<ul style="list-style-type: none"> • Encodes the encrypted symmetric key, nonce, tag, and ciphertext in Base64 format for easier storage and transmission.
Mathematical Formulas	Base64(C, nonce, tag, C'')

- WRITING TO OUTPUT FILE

Function Used	NONE (library-default implementation)
Description	<ul style="list-style-type: none"> • Writes the Base64-encoded data (encrypted symmetric key, nonce, tag, ciphertext) to the output file.
Mathematical Formulas	Not Applicable

3.2 Decryption Methods and Processes

- RSA KEY IMPORT

Function Used	<code>RSA.import_key(key_file.read())</code>
Description	<ul style="list-style-type: none">• Imports the private RSA key for decryption.
Mathematical Formulas	The actual data are encapsulated within the RSA key object including the Modulus (N), public exponent (e), and private exponent (d).

- BASE64 DECODING OF ENCRYPTED DATA

Function Used	NONE (library-default implementation)
Description	<ul style="list-style-type: none">• Decodes the Base64-encoded data (encrypted symmetric key, nonce, tag, ciphertext) from the input file.
Mathematical Formulas	$BASE64^{-1}(C, nonce, tag, C')$

- RSA DECRYPTION OF SYMMETRIC KEY

Function Used	<code>PKCS1_OAEP.new(key).decrypt(enc_symmetric_key)</code>
Description	<ul style="list-style-type: none">• Decrypts the encrypted symmetric key using RSA with OAEP.• Produces the original symmetric key.
Mathematical Formulas	$K_{SYM} \equiv D_{RSA}(C)$

- AES DECRYPTION OF FILE CONTENT

Function Used	AES.new(symmetric_key, AES.MODE_EAX, nonce=nonce).decrypt_and_verify(ciphertext, tag)
Description	<ul style="list-style-type: none"> • Uses the decrypted symmetric key to create an AES cipher object in EAX mode. • Decrypt the file content using AES. • Verifies and removes PKCS7 padding.
Mathematical Formulas	<p>* Will depend on the size of the original file</p> $P \equiv D_{AES}(C')$

- WRITING TO OUTPUT FILE

Function Used	NONE (library-default implementation)
Description	<ul style="list-style-type: none"> • Writes the decrypted file content to the output file.
Mathematical Formulas	Not Applicable

3.3 Libraries Implemented

Library	Purpose
Flask	Web framework for building the application's frontend and handling HTTP requests.
render_template	Function for rendering HTML templates in Flask.
request	Handles HTTP request data in Flask routes.
send_file	Sends files as responses in Flask.
Crypto.PublicKey.RSA	Implements the RSA public-key cryptosystem.
Crypto.Cipher.PKCS1_OAEP	Implements the RSA encryption scheme with OAEP padding.
Crypto.Cipher.AES	Provides AES encryption and decryption.
Crypto.Random.get_random_bytes	Generates cryptographically secure random bytes.

Crypto.Util.Padding	Implements padding schemes for block ciphers.
os	Provides a way to interact with the operating system, used for file operations and directory creation.
mimetypes	Maps filename extensions to MIME types.
base64	Provides base64 encoding and decoding.

3.4 Function Details

Library	Purpose
generate_key_pair()	<ul style="list-style-type: none"> • Purpose: <ul style="list-style-type: none"> - Generates an RSA key pair (public and private keys) with a key size of 2048 bits. It then writes the private and public keys to 'private_key.pem' and 'public_key.pem' files, respectively. • Usage: <ul style="list-style-type: none"> - Run this function to create the initial RSA key pair.
encrypt_file(file_path, key_path, output_file_path)	<ul style="list-style-type: none"> • Purpose: <ul style="list-style-type: none"> - Encrypts a file using a hybrid encryption scheme, where the file content is symmetrically encrypted with AES, and the symmetric key is encrypted with RSA. • Parameters: <ul style="list-style-type: none"> - file_path: Path to the file to be encrypted. - key_path: Path to the RSA public key file. - output_file_path: Path to save the encrypted file
decrypt_file(file_path, key_path, output_file_path)	<ul style="list-style-type: none"> • Purpose: Decrypts a file that was encrypted using the encrypt_file function. It reverses the process by first decrypting the symmetric key with the RSA private key and then decrypting the file content with AES. • Parameters: <ul style="list-style-type: none"> - file_path: Path to the encrypted file. - key_path: Path to the RSA private key file. - output_file_path: Path to save the decrypted file.
index()	<ul style="list-style-type: none"> • Purpose: <ul style="list-style-type: none"> - Renders the main page of the web application. • Usage: <ul style="list-style-type: none"> - Accessed when the root route ('/') is visited.
process()	<ul style="list-style-type: none"> • Purpose: <ul style="list-style-type: none"> - Handles form submissions, including file uploads, and performs either encryption or decryption based on the user's selection. It then renders the result

	<p>page.</p> <ul style="list-style-type: none"> • Usage: <ul style="list-style-type: none"> - Triggered when the '/process' route receives a POST request.
download(filename)	<ul style="list-style-type: none"> • Purpose: <ul style="list-style-type: none"> - Sends the requested file for download. • Parameters: <ul style="list-style-type: none"> - filename: The name of the file to be downloaded. • Usage: <ul style="list-style-type: none"> - Triggered when the '/download/<filename>' route is accessed.
main()	<ul style="list-style-type: none"> • Purpose: <ul style="list-style-type: none"> - Initializes the Flask app, creates the 'uploads' directory if it doesn't exist, generates the initial RSA key pair using generate_key_pair(), and starts the app in debug mode. • Usage: <ul style="list-style-type: none"> - Executed when the script is run directly.

3.3 Flow of the Program (in-depth summary)

The provided Python program is a Flask web application utilizing the PyCryptoDome library for secure file encryption and decryption. The application generates an RSA key pair (private and public keys) using the `generate_key_pair` function and saves them as 'private_key.pem' and 'public_key.pem.'

The `encrypt_file` function takes a file, public key, and output file path, employing AES for symmetric encryption and RSA for encrypting the symmetric key. The result is a Base64-encoded file containing the encrypted symmetric key and file content. Conversely, the `decrypt_file` function decrypts an encrypted file using the private key, performing RSA decryption for the symmetric key and subsequently decrypting the file content with AES. The decrypted file is saved.

The web application provides a user interface for file upload, with options to encrypt or decrypt. The `process` route handles user input, initiates the chosen operation, and renders a result page. Users can download the processed file using the `download` route.

The application incorporates error handling for unsupported file formats and decryption errors. It runs in debug mode, creates an 'uploads' directory for storing files, and initializes the RSA keys on execution.

CHAPTER 4

CONCLUSION

Cryptography plays a vital role in upholding the security and confidentiality of digital communication, making it an essential component of contemporary society. Its pivotal significance stems from its capacity to safeguard sensitive data, including personal information, financial transactions, and government correspondence, shielding it from unauthorized access and manipulation. Through the application of sophisticated cryptographic methods like public-key encryption and digital signatures, both individuals and entities can create secure communication channels and ensure the authenticity of digital assets.

In our interconnected global environment, where data flows incessantly through networks, cryptography stands as the bedrock of cybersecurity. It provides protection for critical infrastructure, financial frameworks, and private communications, fostering trust and dependability in digital exchanges. As society increasingly depends on digital platforms for everyday tasks, ranging from online banking to the exchange of healthcare information, the role of cryptography becomes more prominent in addressing the threats associated with cyber risks.

Furthermore, cryptography not only acts as a defense against malicious entities but also nurtures innovation and the expansion of electronic commerce. The existence of secure online transactions, encrypted messaging platforms, and robust authentication mechanisms is all attributable to cryptographic protocols. In summary, cryptography serves as a fundamental pillar of a secure and robust digital society, influencing the landscape of cybersecurity and facilitating the trust necessary for the widespread acceptance of digital technologies. Its ongoing evolution and incorporation into diverse technological facets underscore its profound influence on the interactions of individuals, businesses, and governments in the digital era.

References

- Network Associates, Inc. (2000). PGP 7.0: An Introduction to Cryptography. Retrieved from <https://www.cs.stonybrook.edu/sites/default/files/PGP70IntroToCrypto.pdf>
- Kessler, G. C. (2023). An overview of cryptography. <https://www.garykessler.net/library/crypto.html>
- Raj, A., Beno, & Jaisakthi, R. (2020). Security Enhancement of Information using Multilayered Cryptographic Algorithm. *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*. <https://doi.org/10.1109/icoei48184.2020.9143052>
- Lagnada, A. (2023, May 7). Cryptographic Algorithms: A comparison of security and strength. *KapreSoft*. <https://www.kapresoft.com/software/2023/05/07/cryptography-algorithms-comparison.html>
- Rivest, R. L., Shamir, A., & Adleman, L. M. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2), 120–126.
- Shannon, C. E. (1949). Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28(4), 656–715.
- Singh, S. (1999). *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor.
- National Institute of Standards and Technology (NIST). (2001). FIPS PUB 197: Advanced Encryption Standard (AES).