Post-Quantum Secure Messaging Web Application

Group Members

Jannat Butt	21I-0259
Misha Fakhar	21I-2718

GitHub Repository:
https://github.com/Jannat-Butt/Infosecurity_projects

PythonAnywhere Deployment:
https://imjannatbutt.pythonanywhere.com/

April 30, 2025

Contents

1 Introduction

With the rise of quantum computing, classical encryption algorithms are vulnerable to quantum attacks. This project demonstrates a secure web application that uses **post-quantum cryptography (PQC)** techniques to ensure encrypted communication remains secure even in the presence of quantum adversaries.

The web application uses **CRYSTALS-Kyber768**, a lattice-based Key Encapsulation Mechanism (KEM) standardized by NIST, in combination with **AES-GCM** for message confidentiality.

2 System Architecture

The secure messaging system is built using the Flask web framework and consists of the following components:

- Kyber768 KEM for key exchange.
- AES-GCM symmetric encryption for message confidentiality.
- HTML/CSS frontend for user interaction.
- Flask server logic to manage encryption and decryption.

3 Key Generation

When a user clicks `Generate Keys`, the system uses the Kyber768 algorithm to generate:

- A public key (used for encryption)
- A private key (used for decryption)

These keys are encoded in Base64 and displayed on the webpage.

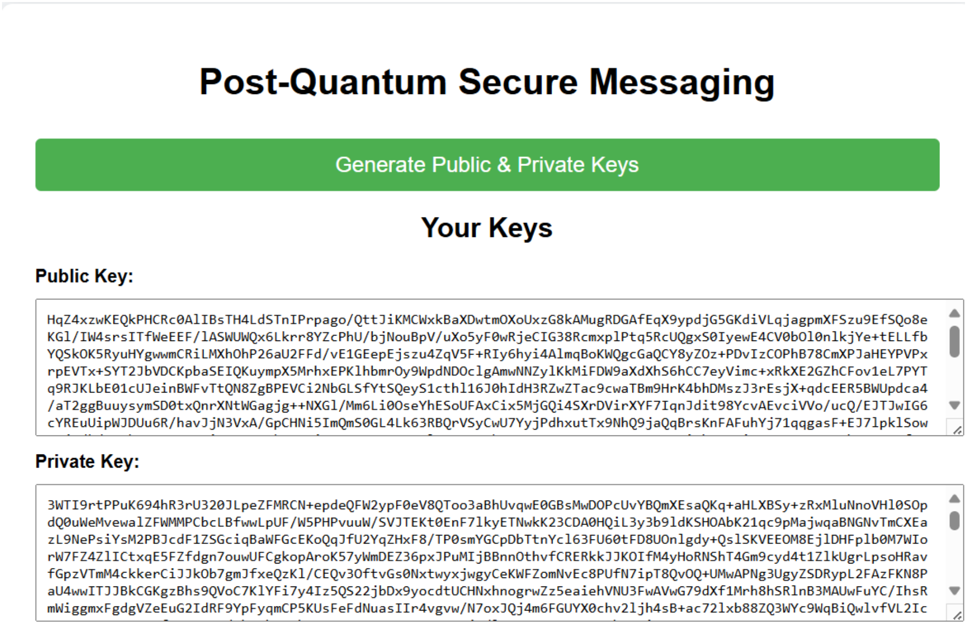

Figure 1: Key Generation

4 Encryption Process

The encryption process consists of two stages:

4.1 1. Key Encapsulation (Kyber768)

The sender uses the receiver's public key to perform the `encap()` operation, which yields:

- `ciphertext_kem`: Encapsulated shared secret
- `shared_secret`: A 32-byte secret key derived during encapsulation

4.2 2. Symmetric Encryption (AES-GCM)

The shared secret is used to derive a 256-bit AES key (first 32 bytes). Then, the plaintext message is encrypted using AES-GCM:

- A random 96-bit IV (initialization vector) is generated
- GCM mode ensures both confidentiality and integrity
- The resulting token includes IV, tag, and ciphertext

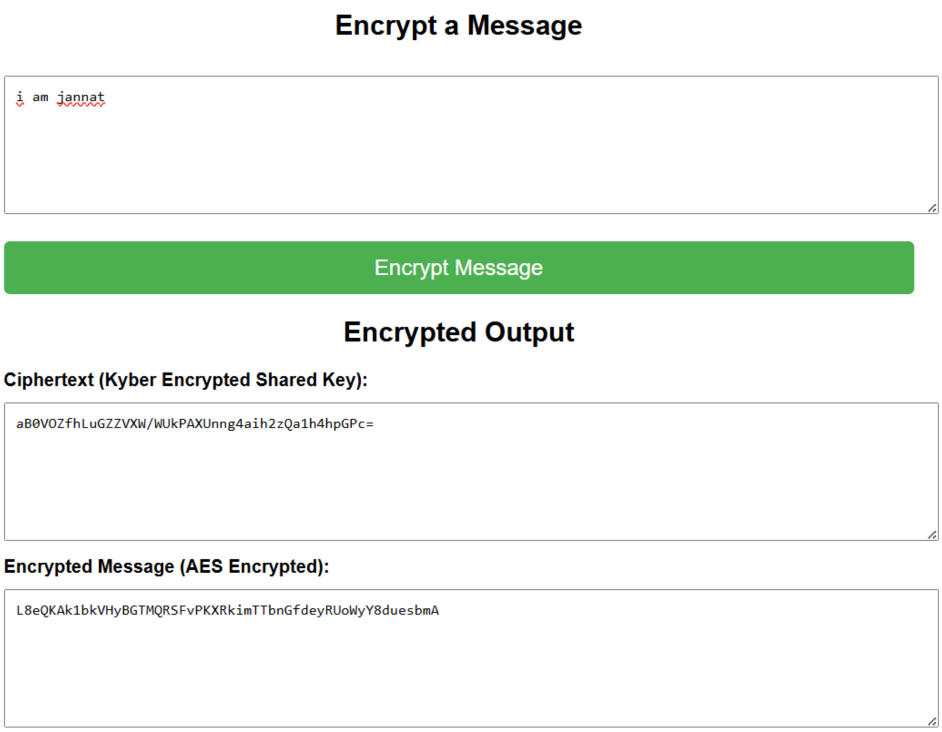

Figure 2: Message Encryption

5 Decryption Process

The decryption also has two stages:

5.1 1. Symmetric Decryption (AES-GCM)

Instead of recalculating the shared secret, the server reuses the `shared_secret` from the previous encryption step. It extracts the AES key and decrypts the message using AES-GCM.

Note: In this implementation, the shared secret derived during encryption is reused during decryption for simplicity. In a production environment, the correct approach is to use the `decap()` function with the private key and `ciphertextkem` to re-derive the shared secret securely. This enhances security and aligns with best cryptographic practices.

5.2 2. Message Retrieval

The decrypted message is then displayed in a readable format.

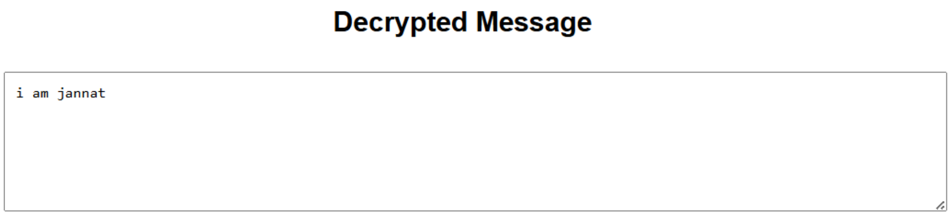

Figure 3: Message Decryption

6 Kyber768: The PQC Algorithm

Kyber768 is a quantum-secure algorithm based on the **Module-LWE** problem, which remains hard even for quantum computers.

6.1 Algorithm Overview

- Based on structured lattice problems
- Uses polynomial arithmetic over rings
- Offers strong security and high efficiency
- Selected by NIST for post-quantum standardization

6.2 Security Parameters

- Security level: **128-bit post-quantum security**
- Key size (public): 1.1KB
- Ciphertext size: 1KB
- Shared secret: 32 bytes

7 AES-GCM: Symmetric Encryption

AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) is used for encrypting messages with the shared key.

- Combines encryption and authentication
- Fast and secure for real-time applications
- Requires IV (nonce), tag, and ciphertext

8 Technology Stack

- **Frontend:** HTML, CSS
- **Backend:** Python, Flask
- **Crypto Libraries:**
  - `pypqc` for Kyber768
  - `cryptography` for AES-GCM
- **Security Middleware:** Flask-Talisman
- **Environment Management:** python-dotenv

9 Sample Code Snippets

9.1 AES Encryption Function

```
def aes_encrypt(key, plaintext):
    iv = secrets.token_bytes(12)
    encryptor = Cipher(
        algorithms.AES(key),
        modes.GCM(iv),
        backend=default_backend()
    ).encryptor()
    ciphertext = encryptor.update(plaintext.encode()) + encryptor.
       finalize()
    return base64.b64encode(iv + encryptor.tag + ciphertext).decode('
       utf-8')
```

Listing 1: AES-GCM Encryption

9.2 Kyber Key Generation

```
public_key, private_key = kyber.keypair()
public_key_b64 = base64.b64encode(public_key).decode('utf-8')
private_key_b64 = base64.b64encode(private_key).decode('utf-8')
```

Listing 2: Generate Kyber Keys

10 Conclusion

This project effectively demonstrates the integration of post-quantum cryptography into a real-world web application. By combining Kyber768 KEM with AES-GCM, we provide both forward secrecy and authenticated encryption, ensuring robust communication security even in a post-quantum world.