PQMessenger: A Web-Based Hybrid Post-Quantum Encrypted Messaging System

Swechchha Adhikari* Shrijan Poudel†

Abstract

Post-quantum cryptography has emerged as a critical field to address the threats posed by quantum computers to classical cryptographic algorithms. To demonstrate its practical feasibility, this paper presents PQMessenger, a proof-of-concept web-based messaging platform implementing a hybrid encryption framework that combines the CRYSTALS-Kyber key encapsulation mechanism (KEM) with AES-CBC for data confidentiality. We describe the system's full-stack design, including its frontend, backend, and cryptographic architecture, as well as key engineering challenges related to session management, cross-platform library integration, and secure key storage. Our goal is to bridge theoretical post-quantum security with practical web deployment, and to provide an open-source foundation for future educational and experimental use.

1 Introduction and Motivation

The rise of quantum computing poses a major threat to existing digital security systems. Public-key encryption algorithms, such as RSA, rely on mathematical problems that are infeasible for classical computers to solve efficiently [4]. Quantum computers, however, exploit superposition and entanglement through *quantum bits (qubits)*, allowing them to evaluate many solutions simultaneously. Shor's algorithm, introduced in 1994, uses these capabilities to factor large integers and compute discrete logarithms in polynomial time, compromising standard encryption methods like RSA, Diffie–Hellman, and ECC [5]. Although current quantum hardware remains limited, experimental implementations have already demonstrated successful attacks on weakened cryptographic variants [2], which underscores the urgency of developing quantum-resistant solutions.

Post-quantum cryptography (PQC) aims to secure digital communication against such quantum adversaries by designing cryptographic primitives that rely on problems believed to remain hard even for quantum computers. Among the families of PQC schemes, lattice-based cryptography stands out for its strong theoretical guarantees and practical efficiency. It relies on the computational hardness of high-dimensional lattice problems such as the *Shortest Vector Problem (SVP)* and the *Learning with Errors (LWE)* problem, for which no efficient

*Brigham Young University, Provo, Utah, USA. Email: adhikar6@byu.edu
†St. Xavier's College, Maitighar, Kathmandu, Nepal. Email: 023a124@sxc.edu.np

classical or quantum algorithms are known. Within NIST's ongoing PQC standardization effort, the lattice-based key encapsulation mechanism (KEM) CRYSTALS–Kyber was selected for its balance of security, performance, and implementation simplicity [1]. Kyber's security is based on the *Module Learning with Errors (MLWE)* problem, whose hardness is grounded in over a decade of lattice cryptography research [1].

In this work, we explore these concepts through the practical implementation of the Kyber KEM within a web-based messaging application, **PQMessenger**. [1] We begin by reviewing the theoretical foundations, then provide an overview of the application's methodology and implementation, followed by a discussion of challenges, key takeaways, and future work. The source code for the implementation is publicly available at PQMessenger.

2 Background and Theoretical Foundations

2.1 Preliminaries

We first introduce the notation, definitions, and mathematical background used throughout this work.

Notation

- Vectors and matrices are denoted by bold lowercase and uppercase letters, respectively (e.g., $\mathbf{s}$, $\mathbf{A}$).
- Polynomials in the ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$ are written as lowercase letters (e.g., $a(x)$).
- Modular arithmetic modulo q is denoted by $[\,\cdot\,]_q$.
- $\lceil \cdot \rfloor$ denotes rounding to the nearest integer.

Lattices

A *lattice* $\mathcal{L} \subset \mathbb{R}^n$ is a discrete additive subgroup generated by integer combinations of a basis $\{b_1, \ldots, b_n\}$:

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} z_i b_i \mid z_i \in \mathbb{Z} \right\}.$$

Key computational problems on lattices, also known as lattice hard problems include:

- **Shortest Vector Problem (SVP):** find a nonzero lattice vector of minimal Euclidean length.
- **Closest Vector Problem (CVP):** given a target vector, find the lattice vector closest to it.

[1] This work originated from a six-week virtual project-based learning program for high school and gap year students in Nepal, called Uunchai. We thank our collaborators from that program for their early contributions and discussions.

Learning With Errors (LWE)

The *LWE problem* over $\mathbb{Z}_q$ is: given a matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ and a vector $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \in \mathbb{Z}_q^m$ (where $\mathbf{s}$ is secret and $\mathbf{e}$ is a small error vector), recover $\mathbf{s}$. The hardness of LWE underpins lattice-based cryptography.

Module-LWE (MLWE)

Module-LWE generalizes LWE to modules over R_q. Given a public matrix $\mathbf{A} \in R_q^{k \times l}$ and a secret vector $\mathbf{s} \in R_q^l$, the problem is to distinguish $\mathbf{A}\mathbf{s} + \mathbf{e}$ from uniform, where $\mathbf{e} \in R_q^k$ is an error vector sampled from a small distribution.

Number Theoretic Transform (NTT)

The NTT is a fast Fourier-like transform for polynomials over R_q, allowing efficient polynomial multiplication. Using NTT reduces the complexity from $O(n^2)$ to $O(n \log n)$.

Error Sampling

Kyber samples errors from a discrete distribution (usually a centered binomial distribution) to hide secrets and ensure IND-CPA security. For $x \in \mathbb{Z}$, let $\psi_\eta(x)$ denote the probability of sampling x.

Compression

A compression function $\text{Compress}_q(\cdot, d)$ reduces each coefficient modulo q to d bits while maintaining correctness with high probability. This reduces key and ciphertext sizes.

2.2 Kyber Overview

Kyber is a lattice-based key encapsulation mechanism built on the **Module Learning With Errors (MLWE)** problem. MLWE extends the LWE problem to modules over polynomial rings $\mathbb{Z}_q[x]/(x^n + 1)$, enabling efficient polynomial arithmetic while preserving security based on worst-case lattice problems such as the Shortest Vector Problem (SVP) and Closest Vector Problem (CVP). The Number Theoretic Transform (NTT) is used to accelerate polynomial operations, and modular arithmetic with noise sampling ensures that secret values remain hidden.

2.3 Key Generation

Kyber generates a public-private key pair such that the public key hides the secret within a noisy linear system. Two seeds are sampled to deterministically generate the public matrix and secret vectors. The public key is then compressed for efficiency, while the secret key consists of the secret vector.

Algorithm 1 Kyber.CPA.KeyGen()

Sample seeds $\rho, \sigma \leftarrow \{0,1\}^{256}$
Generate public matrix $\mathbf{A}$ from ρ
Sample secret and error vectors $(\mathbf{s}, \mathbf{e})$ from σ
Compute $\mathbf{t} \leftarrow \text{Compress}_q(\mathbf{As} + \mathbf{e}, d_t)$
return $pk \leftarrow (\mathbf{t}, \rho),\ sk \leftarrow \mathbf{s}$

2.4 Encapsulation

To encapsulate a message, Kyber samples fresh polynomials for randomness, reconstructs the public matrix, and computes two ciphertext components using linear combinations of the public matrix and the secret polynomials. Compression reduces the ciphertext size while allowing correct decapsulation.

Algorithm 2 Kyber.CPA.Enc(pk, m)

Sample fresh seed and polynomials (r, e_1, e_2)
Reconstruct $\mathbf{A}$ from pk
Compute $u \leftarrow \text{Compress}_q(A^T r + e_1, d_u)$
Compute $v \leftarrow \text{Compress}_q(t^T r + e_2 + \lceil q/2 \rceil \cdot m, d_v)$
return $c \leftarrow (u, v)$

2.5 Decapsulation

Decapsulation reverses the encapsulation process. The ciphertext components are decompressed, and the secret key is used to recover the original message by computing a linear combination and compressing the result.

Algorithm 3 Kyber.CPA.Dec(sk, c)

Decompress u, v from c
Compute $m \leftarrow \text{Compress}_q(v - s^T u, 1)$
return m

3 System Design and Implementation

We built an end-to-end encrypted messaging website using CRYSTALS-Kyber as a Key Encapsulation Mechanism (KEM). The frontend is powered by React, with framer motion as the animation library and tailwind CSS for styling. It also uses Axios as the HTTP client to make requests to the backend. The backend is powered by Flask, with Firestore as the real-time database and Redis handling session cache. Users are authenticated using hashed passwords and JWT-based session tokens, while their private keys are securely encrypted with a password-derived symmetric key. Messaging uses a hybrid cryptographic approach:

Kyber encrypts session-specific AES keys and AES-CBC encrypts the actual message content.

The Kyber implementation used in this system was compiled from the publicly available reference code provided by the CRYSTALS project on GitHub [3]. The RESTful APIs of the site connect the frontend and backend, and social features such as friend requests, notifications, and message retrieval are all stored efficiently in Firestore.

3.1 Frontend Architecture

The frontend is organized into distinct layers, built with modern JavaScript frameworks and libraries:

- **Routing Layer**: React Router manages client-side navigation, defining routes for login, registration, dashboard, and chat views.
- **Authentication Layer**: AuthContext provides global authentication state and ensures that only authorized users can access protected routes. This makes use of local-storage to store authentication token in the browser itself and the username for session management and API requests. (All of which are validated by the backend layer)
- **UI Layer**: Tailwind CSS provides utility-first styling for responsive design, while Framer Motion enables smooth animations and transitions. React Icons supply standardized iconography.
- **State and Data Layer**: Context and utility hooks manage user state and responsive behavior. Axios is used for secure communication with the Flask backend.
- **Component Layer**: Reusable components implement key features such as messaging, friend requests, notifications, and user profiles.

3.2 Backend Architecture

The backend is structured into several logical parts and layers:

3.2.1 Authentication Layer

Responsible for registering new users and verifying logins, this layer calls the ***Security Layer*** to hash, encrypt, and validate sensitive data before storing them in the Firestore database. It is also responsible for responding to any **POST** or **GET** requests made to the backend server and validates the requests using JWT-encoded session tokens.

Upon new user registration, this layer generates user-specific public and private keys and securely stores them in the database. It also generates a session-based AES key during login, which is used to encrypt messages and protect user data throughout the session.

The generated AES key is then **encrypted for the user** using the CRYSTALS-Kyber KEM. A shared secret and its ciphertext are derived using `kyber_encapsulate` with the user's public key. The shared secret is then used to derive a symmetric wrapping key after

applying SHA-256. It is then used to encrypt the session AES key using AES-CBC and a newly generated Initialization Vector (IV).

This ensures that we do not need to re-encrypt the AES key for the sender repeatedly during the current session.

Now, the generated AES key, the encrypted AES key for the user (i.e., the **sender**) for the session, the ciphertext for the shared secret, the initialization vector along with the decrypted `private key` of the user are temporarily stored in the Redis cache for later use.

During messaging operations, this encrypted AES key (for the sender) is retrieved and stored in the message document.

3.2.2 Security Layer

This layer manages encryption, decryption, and user credential validation. Passwords must be at least 8 characters long and include uppercase, lowercase, numeric, and special characters. Usernames must be 3–20 characters, containing only letters, numbers, or underscores, with no spaces or other special symbols.

It manages password hashing using the `bcrypt` library and validates user credentials during login attempts using the `bcrypt.checkpw()` method. Furthermore, this layer is responsible for encrypting and decrypting users' *private keys* upon request from the authentication layer.

Encryption and Decryption of Secret Keys: To ensure secure storage and transmission of sensitive cryptographic materials, the system employs a two-step process involving key derivation and AES encryption.

1. **Key Derivation:**
    - Utilizes the PBKDF2HMAC (Password-Based Key Derivation Function 2) algorithm with SHA-256.
    - Takes the user's password, a unique salt, and a defined number of iterations (specified in the environment configuration, 100000 by default).
    - Produces a strong 32-byte symmetric key derived from the password.
2. **AES Encryption:**
    - Employs AES encryption in CBC (Cipher Block Chaining) mode.
    - The derived key and a random Initialization Vector (IV) are used to create the cipher.
    - The user's secret key (in hexadecimal format) is converted to bytes and padded using PKCS7 to align with AES block size.
    - The padded secret key is then encrypted with AES, resulting in an encrypted secret key (stored as bytes).
3. **AES Decryption:**

- Uses the same PBKDF2HMAC parameters (password, salt, and iteration count) to regenerate the decryption key.
- The AES cipher (CBC mode) decrypts the stored ciphertext using the derived key and IV.
- PKCS7 padding is removed to recover the original secret key bytes.
- The final output is the decrypted secret key represented as a hexadecimal string.

Summary:

- **Encryption:** secret_key (hex) + password + salt + IV $\xrightarrow{\text{PBKDF2 + AES-CBC}}$ encrypted secret key
- **Decryption:** encrypted secret key + password + salt + IV $\xrightarrow{\text{PBKDF2 + AES-CBC}}$ decrypted secret key

This design ensures that even if an encrypted secret key is compromised, it remains computationally infeasible to decrypt it without the correct user password, thereby maintaining the confidentiality and integrity of stored keys.

3.2.3 Session Management Layer

Uses Redis to manage short-lived session identifiers and temporarily store the decrypted private key, session-based AES key, encrypted AES key for the user along with its IV and ciphertext, ensuring rapid access while minimizing the risk of long-term exposure of sensitive information.

3.2.4 Messaging Layer

Responsible for secure end-to-end message exchange between users, implementing a hybrid encryption mechanism that combines post-quantum and symmetric cryptography. It interfaces with both the *Security Layer* for encryption/decryption operations and the *Session Management Layer* (Redis) for temporary key storage and retrieval.

- **Message Sending Process:**
  1. **Fetch User and Friend Data:**
     - Retrieves the current user and friend documents from Firestore using their respective IDs.
     - Extracts the friend's public key for encryption.
  2. **Obtain Session AES Key:**
     - The session AES key, previously established during login, is retrieved from Redis.
     - This key is used for encrypting individual chat messages.

3. **Encrypt the Message:**
   - A random Initialization Vector ($\text{IV}_{\text{message}}$) is generated for the message.
   - The plaintext message is padded using PKCS7 and encrypted with AES in CBC mode using the session AES key.
4. **Wrap the AES Key for the Receiver:**
   - Utilizes CRYSTALS-Kyber to encapsulate the AES key for the receiver.
   - Derives a shared secret (ss_{receiver}) and applies SHA-256 to obtain a symmetric wrapping key.
   - Encrypts the session AES key with this derived key using AES-CBC and a new Initialization Vector ($\text{IV}_{\text{receiver}}$).
5. **Retrieve Sender's Wrapped AES Key:**
   - Obtains the sender's ciphertext, encrypted AES key, and IV from Redis (stored during the login phase).
6. **Store Encrypted Data in Firestore:**
   - Creates a new document in the `messages` collection containing:
     * Encrypted message and its IV.
     * Sender and receiver ciphertexts, encrypted AES keys, and IVs.
     * Sender and receiver user IDs.
     * Timestamp of the message.
7. **Session Expiry Handling:**
   - Resets the session expiry timer in Redis to one hour to maintain active communication securely.

On successful completion of these steps, the message is securely transmitted and stored, ensuring confidentiality even under potential quantum adversaries.

- **Message Retrieval Process:**
  1. **Fetch User and Friend Data:**
     - Retrieves both user and friend documents from Firestore.
  2. **Access User's Private Key:**
     - Loads the user's decrypted private key from Redis (decrypted during login by the Security Layer).
  3. **Query Messages:**
     - Queries the `messages` collection for all messages exchanged between the two users (both sent and received).

- Aggregates and sorts them chronologically.

4. **Decrypt Each Message:**
   - For each message, communicates with the *Security Layer* for decryption, which:
     * Identifies whether the user is the sender or receiver.
     * Decapsulates the appropriate Kyber ciphertext using the user's private key to derive the shared secret.
     * Uses SHA-256 to obtain the AES wrapping key and decrypts the wrapped session AES key.
     * Decrypts the AES-encrypted message using the session key and message IV.
   - Replaces the encrypted message field with the recovered plaintext.
5. **Return Decrypted Messages:**
   - The complete set of decrypted messages is returned as JSON, ready for rendering on the frontend.

3.2.5 Database Layer

Firestore manages persistent data. Two key collections are maintained:

1. `Users`: Stores user IDs, usernames, bcrypt password hashes, public keys, encrypted private keys, salts, and initialization vectors.
2. `Messages`: Stores encrypted message content, sender and receiver IDs, timestamps, and the AES session keys encrypted separately for both parties along with the required data for their decryption.

3.2.6 Kyber Layer

This layer provides a Python wrapper for the kyber library. This allows our code to use the compiled native kyber functions via the use of the ***ctypes***, a foreign function python library. The kyber library in this project has been compiled in two formats, *libkyber.so* for linux and *libkyber.dll* for windows.

This layer works by providing a class, `KyberWrapper`, with methods which either call the functions in the kyber library after the library was successfully loaded or mock the functions when the kyber library couldn't be loaded. The methods are:

- `__init__`: Loads the compiled kyber C library into Python memory, defines argtypes and return types for all the key kyber functions and raises an exception if the library couldn't be loaded.

- `generate_keypair`: Generates `public_key` and `private_key/secret_key` using the `my_crypto_kem_keypair` kyber function, and returns the keys after converting them into hex from bytes.
- `encapsulate`: Takes the `public_key` of the respective user as the parameter, uses the `my_crypto_kem_enc` function to generate a `cipher_text` and the `shared_secret`, and returns those in bytes.
- `decapsulate`: Takes `cipher_text` and `private_key` in bytes as input and uses the `my_crypto_kem_dec` function to get the shared secret and return its value in bytes.

3.3 Workflow

When a new user registers, the Authentication Layer in the backend generates a Kyber key-pair. The public key is stored in Firestore, while the private key is encrypted with the password-derived key and stored securely. For login, credentials are verified against the stored bcrypt hash by Security Layer, and upon success, the private key is temporarily decrypted and cached in Session Management Layer along with the session-based AES key in Authentication Layer

After a successful log-in, authentication layer then directs the user to the *dashboard page*. There, the user can now :

- Start a conversation or send a message.
- Add new friends.
- Check notifications.
- View friend requests.

Notifications and friend requests are handled by Authentication Layer. This works by querying all the friend request / notification docs from the database which contains the specific user id, checking their status and sending them to front-end in the following way:

- Friend-request page if it is an incoming friend request.
- Notification page if the friend request has been accepted.

The message sending and retrieval process is handled by the Messaging Layer. This layer interacts with Security Layer and Database Layer to encrypt / decrypt the messages and securely store them, respectively.

When a user logs out, the JWT encoded token is removed from the browser's local-storage, the user value in the *Authentication Layer* of the front-end is set to null and hence, *Auth Context* then directs the user to the login page.

3.4 System Overview Diagram

To illustrate, Figure 1 shows the high-level architecture of the system.

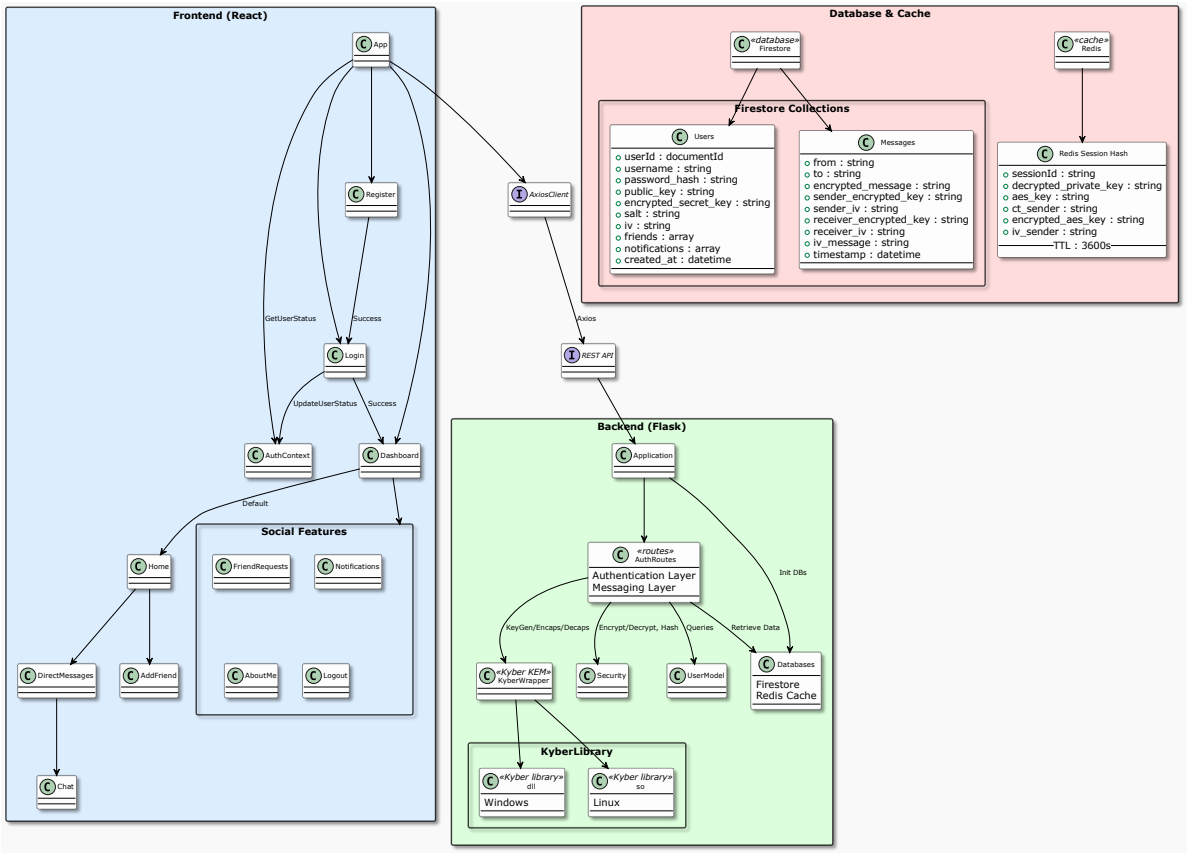

Figure 1: High-level architecture of the encrypted messaging platform.

4 Discussion and Key Takeaways

The development of our prototype showed tradeoffs between cryptographic robustness and practical system efficiency. While the algorithm guaranteed strong security, we encountered several engineering challenges including key management, performance, and interoperability across platforms while integrating it within a web-based environment, that led to valuable insights and lessons learned.

We implemented CRYSTALS-KYBER into our program by compiling a shared library from the GitHub repository of its official reference implementation [3]. However, by default, the `Makefile` in the repository only compiles a shared library for Linux environments in the format `kyber.so`. This was not particularly convenient since we were working in a Windows environment. Hence, we compiled a shared library for Windows in the format `kyber.dll` as follows:

- First, we wrapped essential internal functions in a separate C file to hide internal complexity and export a minimal API that could be called by the DLL file. We then marked those specific functions for export from the DLL so they could be accessed by our program.

- Next, the library was compiled into machine code using the GNU Compiler Collection (GCC), producing the `kyber.dll` file.

There was also a challenge in handling a user's private key without compromising their password's security. Since the user's `raw_password` is never stored in the database, their `private_key` can only be decrypted when the user logs in (by entering the raw password). The decrypted key then had to be stored **securely** throughout the session for later use. Hence, we extended our default stack (Flask and Firebase) to include ***Redis*** as a secure session cache with a proper time-to-live (TTL), handled as described in the Messaging Layer.

We also used KYBER to encrypt all the message interactions between users, initially. This resulted in a large amount of delay while sending a message. Moreover, to allow proper interaction between users, the message had to be encrypted two times: once for the sender, and once for the receiver; which further increased the delay. To minimize the delay, we used symmetric encryption to encrypt all the messages using AES. Then, we encrypted the AES key for both sender and receiver respectively. This was further optimized by generating a session-based AES key and encrypting the AES key for the sender during login, as mentioned in Authentication Layer. The Messaging Layer was also optimized to retrive the `encrypted_aes_key_for_sender` from the Redis server.

Additionally, as our system is intended for demonstration purposes, we did not implement a more complex solution such as websockets for real-time updates. Instead, we used *interval polling* to periodically check the backend for new messages, creating the appearance of near real-time updates.

For future work, we plan to conduct a robust performance evaluation of the system. Specifically, we will collect detailed metrics on key exchange latency and measure the time required for key generation, encapsulation, and decapsulation across both the browser (via WebAssembly/JavaScript) and backend (server-side) implementations. We will compare these results against classical baselines such as ECDH and RSA to assess efficiency and practical viability. We also want to analyze end-to-end message delivery latency from the moment a user sends a message to the point it is decrypted, as well as measure CPU and memory utilization on both the browser and server during key exchange and bulk messaging.

5 Conclusion

In this paper, we developed PQMessenger, a web-based messaging application that uses the CRYSTALS-Kyber KEM for key encapsulation and AES-CBC for message encryption. We detailed the design of the frontend and backend architectures, the hybrid encryption workflow, and the secure session management system built with Flask, Redis, and Firestore. We also described the process of compiling and integrating the Kyber library for both Linux and Windows environments, as well as the engineering challenges encountered during implementation and future works. Our results demonstrate a practical deployment of post-quantum encryption methods in a real-world web application setting.

References

[1] Gorjan Alagic, Daniel Apon, David A. Cooper, Quynh H. Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, et al. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process (Update 1). NIST Interagency or Internal Report NIST IR 8413-upd1, National Institute of Standards and Technology, 2022.

[2] Chunlei Hong, Zhi Pei, Qidi Wang, Shuxiao Yang, Jingjing Yu, and Chao Wang. Quantum attack on rsa by d-wave advantage: a first break of 80-bit rsa. *Science China Information Sciences*, 68(2):129501, 2025.

[3] PQ-Crystals. Kyber: Post-Quantum Cryptography Reference Implementation. `https://github.com/pq-crystals/kyber`, 2025. Accessed: 2025-10-15.

[4] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[5] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.