UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Department of Information Engineering

BACHELOR'S DEGREE IN INFORMATION ENGINEERING

Implementation and Performance Evaluation of Quantum-Safe Cryptography in C: A Comparative Study Using LibOQS and OpenSSL

Supervisor: Prof. Rizzo Luigi

Candidate: Biruk Kiros Meles

Academic year

2023 – 2024

Abstract

This thesis focuses on the analysis and testing of some encryption and signature algorithms, resistant to quantum computer attacks, known as quantum safe algorithms, recently evaluated by NIST (National Institute of Standards and Technology), aiming to protect data against future quantum computing threats. The study objective is to evaluate the performance of some quantum safe encryption, digital signature and key generation algorithms implemented using the liboqs library and compare these performances with the ones of traditional algorithms implemented using the OpenSSL library presenting a comparative analysis of these solutions, and proposing their applications in some real scenarios. Specifically, we developed applications and measured the time/performance for key encapsulation, decapsulation, encryption, decryption, signature creation and validation processes.

Our methodology involved implementing various, quantum-resistant and not, cryptographic algorithms in both libraries and recording the time taken to complete each cryptographic operation. The results revealed significant differences in execution times among the algorithms tested. These findings highlight the trade-offs between computational efficiency and security provided by each approach.

The study concludes that while both liboqs and OpenSSL offer viable quantum-safe solutions, the choice of an algorithm should be guided by specific application requirements, prioritizing either speed or security. Future research should focus on further optimization and real-world testing of these algorithms to enhance their practical applicability.

Riassunto

Questa tesi si concentra sull'analisi e sul test di alcuni algoritmi di crittografia e firma, resistenti agli attacchi informatici quantistici, noti come Quantum Secure Algorithms, recentemente valutati dal NIST (National Institute of Standards and Technology), con l'obiettivo di proteggere i dati dalle future minacce del calcolo quantistico. L'obiettivo dello studio è valutare le prestazioni di alcuni algoritmi di crittografia quantistica sicura, firma digitale e generazione di chiavi implementati utilizzando la libreria liboqs e confrontare queste prestazioni con quelle degli algoritmi tradizionali implementati utilizzando la libreria OpenSSL presentando un'analisi comparativa di queste soluzioni e proponendo le proprie applicazioni in alcuni scenari reali. Nello specifico, abbiamo sviluppato applicazioni e misurato tempo/prestazioni per i processi di incapsulamento, decapsulamento, crittografia, decrittografia, creazione di firme e convalida delle chiavi.

La nostra metodologia prevedeva l'implementazione di vari algoritmi crittografici, sia resistenti che non resistenti ai quanti, in entrambe le librerie e la registrazione del tempo impiegato per completare ciascuna operazione crittografica. I risultati hanno rivelato differenze significative nei tempi di esecuzione tra gli algoritmi testati. Questi risultati evidenziano i compromessi tra efficienza computazionale e sicurezza forniti da ciascun approccio.

Lo studio conclude che, sebbene sia liboqs che OpenSSL offrano soluzioni praticabili di sicurezza quantistica, la scelta di un algoritmo dovrebbe essere guidata da requisiti applicativi specifici, dando priorità alla velocità o alla sicurezza. La ricerca futura dovrebbe concentrarsi su un’ulteriore ottimizzazione e test nel mondo reale di questi algoritmi per migliorarne l’applicabilità pratica.

Contents

Introduction

Imagine yourself in ancient Egypt, around 1900 BC, where the high priests used cryptography to protect sacred messages. They employed non-standard hieroglyphs, not for the common eye, but to obscure religious texts. While this method seemed impenetrable at the time, it wasn't designed for complex adversaries—its real challenge was its unfamiliar symbols, which could still be deciphered by those trained to read them.

Fast forward to ancient Greece and Rome, around 500 BC. Here, cryptography took on a new form with the Scytale, a tool used by the Spartans for military communication. A strip of parchment was wrapped around a rod of a specific diameter, and only when wrapped around an identical rod would the message make sense. The Scytale gave the Spartans an edge in battle, but its simplicity meant that it could be broken with trial and error – just test enough rods, and the message would reveal itself.

By the time we reach Julius Caesar in the 1st century BC, cryptography had evolved with the famous Caesar cipher. Caesar shifted letters in the alphabet by a fixed number, effectively scrambling his messages. While brilliant for its era, the cipher was eventually broken by a method known as frequency analysis. Skilled cryptanalysts realized that in any language, certain letters occur more frequently than others—'E' is more common than 'X', for example. By tracking letter frequencies, they could undo the cipher and read the hidden text.

Fast forward again to the Renaissance period, and cryptography had become more widespread, particularly in European courts and during wars. Substitution ciphers and polyalphabetic ciphers became standard. These systems added more layers of complexity, such as using different alphabets to encrypt different parts of the message. Yet even these advanced methods were eventually cracked by patterns—repetition in letters and phrases allowed skilled cryptanalysts to break through.

No cryptography method faced as big a test as the Enigma machine during World War II. In the 20th century, the Germans believed Enigma, with its rotating mechanical wheels and near-endless combinations, was unbreakable. However, the Allies, through a combination of human intelligence, mathematical breakthroughs, and known-plaintext attacks, cracked Enigma, altering the course of the war.As the digital age dawned in the mid-20th century, cryptography faced a new challenge: computers. In the 1970s, the Data Encryption Standard (DES) was developed to protect digital information. DES became the global standard, trusted to secure everything from government data to commercial transactions. However, with growing computational power, hackers began to exploit DES's vulnerabilities through brute-force attacks, where every possible key was tested until the correct one was found.Recognizing these weaknesses, cryptographers developed more robust methods. In 2001, the Advanced Encryption Standard (AES) was introduced, offering a more secure and efficient encryption algorithm. Around the same time, the RSA algorithm emerged, based on the mathematical difficulty of factoring large prime numbers. RSA and AES became the pillars of modern cryptography, securing everything from online banking to military communications.

From ancient symbols to the power of prime numbers, cryptography has continuously evolved, adapting to the threats and technologies of its time. However, with the rise of quantum computing on the horizon, even AES and RSA face the possibility of being broken, driving cryptographers to prepare for the next frontier in securing information – *Quantum Safe Cryptography(QSC).*

1. Quantum Computing

1.1 Overview

Quantum computing leverages quantum mechanics to solve complex problems that classical computers can't handle efficiently. Unlike classical bits, which represent either 0 or 1, quantum bits (qubits) can exist in multiple states simultaneously, enabling massive parallelism and processing power.

1.2 Key Characteristics of Quantum Computing

1.2.1 Superposition

Superposition allows qubits to exist in multiple states (0 and 1) simultaneously, unlike classical bits that are limited to a single state.

A qubit in superposition is described as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where $|0\rangle$ and $|1\rangle$ are the two basis states, and α and β are complex numbers representing the probabilities of being in either state. The total probability must equal 1 ($|\alpha|^2 + |\beta|^2 = 1$) [1].

Superposition allows qubits to explore multiple simultaneously, leading to exponential speedups in areas like optimization, cryptography, and database searching. Quantum algorithms like Shor's and Grover's exploit superposition and quantum interference to find solutions more efficiently than classical methods.

1.2.2 Entanglement

Entanglement occurs when qubits become quantum mechanically linked, so the state of one qubit directly affects the state of another, even over large distances [2].

- **Quantum Teleportation**: Transfers information between entangled qubits without physical movement, allowing for remote data exchange [2].
- **Error Correction**: Entanglement aids in detecting and correcting errors in quantum systems, enhancing system stability and reliability.

Entanglement enables faster, more efficient information processing by synchronizing qubits. It allows quantum computers to explore multiple solutions simultaneously, improving problem-solving speed and security in areas such as quantum communication and algorithm efficiency.

1.2.3 Quantum Interference

Quantum interference involves the combination of probability amplitudes from overlapping quantum states. It amplifies the probability of correct solutions (constructive interference) and diminishes incorrect ones (destructive interference) [2] [3]. For instance, in Grover's algorithm, interference boosts the likelihood of the correct search result, while in Shor's algorithm [4]. It aids in faster integer factorization. This efficiency accelerates problem-solving by focusing on promising solutions [1].

1.3 The Power of Quantum Computing

Classical computers process information sequentially using bits (0 or 1), whereas quantum computers use qubits, which can exist in superposition, allowing them to process multiple possibilities at once.

Quantum computers leverage superposition, entanglement, and quantum interference to solve certain problems much faster than classical computers. They are particularly well-suited for specific tasks, such as optimization and quantum simulations.[4] In the next, sections it is discussed how quantum computers excel relative to classical computers in the field of cryptography.

2. Why quantum computing is a threat to modern day cryptography

This section illustrates how quantum computing influences the widely used RSA public key algorithm.

2.1. Understanding RSA Encryption

RSA encryption is based on the challenge of factoring a large number *N*, which is the product of two large prime numbers, *p* and *q*. The public key (*e,N*) is used for encryption, while the private key (*d,N*) is used for decryption. The private key d is calculated from *p* and *q*, but because these prime factors are kept confidential, it is computationally difficult to derive the private key from the public key 5. Classical computers will take exponential time to factorize the large number *N* [4] [6].

2.2. Shor's Algorithm Unveiled: Quantum Computing's Breakthrough in Factoring RSA Encryption

Shor's algorithm, created by Peter W. Shor, is a significant breakthrough in quantum computing, as it efficiently solves integer factorization and affects encryption [8].

The algorithm can be used as the following to break RSA [4]:

- **Step 1: Choose *N*** In RSA, *N* is the product of two primes, so Shor's algorithm starts with the goal of factoring *N*.
- **Step 2: Find a Suitable a** Shor's algorithm chooses a random number *a* that is less than and relatively prime to *N*, which will help in the factorization process.
- **Step 3: Quantum Period Finding** Using **quantum parallelism** and the **Quantum Fourier Transform (QFT)**, Shor's algorithm finds the "period" *r* of the function $f(x)=a^x \bmod N$. The period *r* is the smallest value such that $a^r \bmod N = 1$ [4] [7].
- **Step 4: Classical Post-Processing**: Once the period *r* is found, it is related to the prime factors of *N*. Classical computations help to extract these factors using the relationship between *a*, *r*, and *N*.
- **Step 5: Extract the Prime Factors**: Using the period r, compute $gcd\left(a^{r/2}-1, N\right)$ and $gcd\left(a^{r/2}-1, N\right)$. One of these will give a nontrivial factor of N [8].

These greatest common divisor calculations reveal the prime factors *p* and *q* [4] [8].

RSA's security relies on the fact that, with classical computers, factoring *N* is extremely difficult and time-consuming for large numbers, which takes an exponential time [4] [6]. However, Shor's algorithm, running on a quantum computer, factors *N* efficiently in **polynomial time**. Once *p* and

q are determined, the attacker can compute the private key *d*, making it possible to decrypt any encrypted messages and completely break the RSA encryption [4] [5].

3. Transitioniong to Post-Quantum Cryptography

3.1 Current Advances, Threats, and the Road Ahead

In 2019, Google and NASA announced they had achieved quantum supremacy, claiming their quantum computer performed a calculation in seconds that would take a classical supercomputer thousands of years [9]. In 2023, IBM unveiled their new quantum computer, IBM Condor, featuring 1,121 physical qubits and utilizing cross-resonance gate technology [10].

Regarding cryptographic threats, it was estimated that breaking RSA with Shor's Algorithm requires [11] [12]:

- **RSA-2048**: 6,190 logical qubits
- **RSA-7680**: 23,239 logical qubits
- **ECDSA-256**: 2619 logical qubits.

The current concern is that malicious actors may store encrypted data now for future decryption once quantum computers advance, while the transition to post-quantum cryptography, as noted by NIST, could take years [13].

3.2 Standardization efforts

The National Institute of Standards and Technology (NIST) began its post-quantum cryptography (PQC) standardization process in 2016 to establish algorithms resilient against quantum attacks. This process involved multiple evaluation rounds of candidate algorithms based on their security and performance.

AS of August 12, 2024, The Secretary of Commerce has approved three new FIPS for post-quantum cryptography [26]:

- **FIPS 203** focuses on a key encapsulation mechanism (KEM) derived from the CRYSTALS-KYBER submission. KEMs enable secure key exchanges between two parties over a public channel, and current schemes are described in NIST Special Publications (SP) 800-56A and 800-56B.
- **FIPS 204** specifies a digital signature standard based on the CRYSTALS-Dilithium submission, while **FIPS 205** defines a stateless hash-based digital signature standard derived from the SPHINCS+ submission. Digital signatures are essential for verifying data integrity and the authenticity of signatories. Existing signature schemes are outlined in FIPS 186-5 and SP 800-208. NIST is also working on a digital signature algorithm derived from FALCON as an additional option.

3.3 Transitioning challenges

A NIST report titles “Getting Ready for Post-Quantum Cryptography” [13] highlights several challenges in transitioning to post-quantum cryptography (PQC), a process expected to take 5 to 15 years or more, with greater complexity than traditional cryptographic updates due to significant differences in methods. Key challenges include:

1. Implementation and Infrastructure Modifications: Upgrading cryptographic libraries, hardware, and protocols to support PQC, replacing existing algorithms like RSA and ECC.
2. Performance and Scalability Issues: PQC’s larger key sizes and computational demands can reduce performance, especially in constrained environments like IoT.
3. Key Management and Storage: Larger keys strain storage and transmission systems, complicating key distribution and affecting overall performance.
4. Interoperability and Legacy Systems: Ensuring compatibility with older systems requires hybrid solutions combining classical and quantum-resistant algorithms.
5. Operational Migration Challenges: Organizations need structured plans to gradually migrate to PQC without disrupting operations or compromising security.

4. Code implementation for comparison

4.1 Overview

This section explains the implementation of code designed to demonstrate and measure the performance of different cryptographic algorithms. Specifically, it compares the speed and efficiency of algorithms from the **liboqs** library and **OpenSSL** in the context of Public Key Infrastructure (PKI).

The code evaluates how each library handles various cryptographic operations, providing insights into their performance differences. This comparison is crucial for understanding the trade-offs between quantum-resistant algorithms and traditional cryptographic methods, helping to guide the selection of appropriate tools for secure communications in a post-quantum world.

4.2 liboqs implementation

The Open Quantum Safe (OQS) project [15], part of the Linux Foundation's Post-Quantum Cryptography Alliance, is driving the shift to quantum-resistant cryptography with its open-source tools. Central to this effort is **liboqs**, an open-source C library that integrates quantum-safe algorithms into the widely-used OpenSSL library.

The liboqs C library provides implementations of various quantum-safe cryptographic algorithms, including key encapsulation mechanisms (KEMs) and digital signature algorithms. Below is a summary of selected algorithms from liboqs, including their key sizes, ciphertext sizes, and brief descriptions.

4.2.1 Key Encapsulation Mechanisms (KEMs)

4.2.1.1 Used algorithms

A. Kyber768

- **Description**:Kyber is a NIST selected(standaradized) IND-CCA2-secure key encapsulation mechanism (KEM), whose security is based on the hardness of solving the learning-with-errors (LWE) problem over module lattices [16].
- **size(bytes)**:
    - Public key size : 1184
    - Secret key : 2400
    - Ciphertext : 1088
    - Shared secret : 32 [17]

B. frodokem_640_aes

- **Description**: FrodoKEM is a family of key-encapsulation mechanisms that are designed to be conservative yet practical post-quantum constructions whose security derives from cautious parameterizations of the well-studied learning with errors problem, which in turn

has close connections to conjectured-hard problems on generic, algebraically unstructured lattices [18].

- **size(bytes)**:
    - Public key size : 9616
    - Secret key : 19888
    - Ciphertext : 9720
    - Shared secret : 16 [17]

C. bike_l1

- **Description**: BIKE is a, round 4 NIST candidate, code-based key encapsulation mechanism based on QC-MDPC (Quasi-Cyclic Moderate Density Parity-Check) codes [19].
- **size(bytes)**:
    - Public key size : 1541
    - Secret key : 5223
    - Ciphertext : 1572
    - Shared secret : 32 [17]

D. classic_mceliece_348864

- **Description**: McEliece is a NIST round 4 candidate cryptosystem that relies on error-correcting codes, specifically Goppa codes, to encrypt data. Its security is based on the difficulty of decoding these codes without the private key, making it resistant to both classical and quantum attacks [20].
- **size(bytes)**:
    - Public key size : 261120
    - Secret key : 6492
    - Ciphertext : 96
    - Shared secret : 32 [17]

E. hqc_128

- **Description**: HQC (Hamming Quasi-Cyclic) is a NIST round 4 candidate code-based public key encryption scheme developed to offer protection against attacks from both classical and quantum computers [21].
- **size(bytes)**:
    - Public key size : 2249
    - Secret key : 2305
    - Ciphertext : 4433
    - Shared secret : 64 [17]

4.2.1.2 The Implementation of liboqs library in C

The performance of the algorithms is demonstrated by the code listed in the appendix A.

Input

The code accepts data stored in a file named **"input.txt"**.

Output

A CSV file is generated, such as **"Time_all_KEMs.csv"**, **"tkyber.csv"**, **"thqc.csv"**, etc., based on the chosen algorithm from the menu. The data in these files contain the time measurements for encapsulation, decapsulation, and the average of these times for multiple runs. The output format, which is consistent for every algorithm presented in this project, is show below:

alg_name,Encryption Time (ms),Decryption Time (ms)
kyber_768,0.042000,0.046000
kyber_768,0.065000,0.025000
Mean_kyber_768,0.053500,0.035500

Figure 4.1. Output format of Appendix A when kyber768 is selected and run 2 times

This output format is consistent for every algorithm presented in this project.

In the following, we will detail the implementation of the main functions for the Kyber-768 algorithm. The same approach is used for other algorithms, with the primary difference being the substitution of the word "kyber_768" with those specific to each algorithm.

a. kyber_encrypt

This function returns a double representing the time taken to encapsulate data using the Kyber-768 encryption scheme. The core operation, performed by **OQS_KEM_kyber_768_encaps**, generates a ciphertext and a shared secret shared_secret_A based on the provided public_key. If the encryption fails, it logs an error and returns -1. Otherwise, it returns the time taken in milliseconds.

```
double kyber_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t
        *public_key);
        1double kyber_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t
*public_key) {
        2 clock_t start = clock();
        3 OQS_STATUS status = OQS_KEM_kyber_768_encaps(ciphertext, shared_secret_A,
public_key);
        4 clock_t end = clock();
        5 if (status != OQS_SUCCESS) {
        6 fprintf(stderr, "Error: Kyber encryption failed.\n");
        7       return -1;
        8   }
        9   return measure_time_encrypt(start, end);
        10}
```

Figure 4.2 function kyber_encrypt from Appendix A

b. kyber_decrypt

This function decrypts the ciphertext and retrieves shared_secret_B. The time taken for decryption is returned as a double. If decryption fails, it logs an error and returns -1.

```
        double kyber_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t
        *secret_key)
        1double kyber_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t
*secret_key) {
        2   clock_t start = clock();
        3   OQS_STATUS status = OQS_KEM_kyber_768_decaps(shared_secret_B, ciphertext,
secret_key);
        4   clock_t end = clock();
        5    if (status != OQS_SUCCESS) {
        6       fprintf(stderr, "Error: Kyber decryption failed.\n");
        7       return -1;
        8   }
        9   return measure_time_decrypt(start, end);
        10}
```

Figure 4.3 function kyber_decrypt from Appendix A

c. case_kyber

The case_kyber function performs Kyber-768 encryption and decryption multiple times, recording the time for each process and outputting the results to a CSV file. It verifies that the encryption and decryption operations are successful by checking if shared_secret_A and shared_secret_B match.

Parameters:

- **file_to_write_to**: The name of the CSV file for timing results.
- **k**: Number of iterations.
- **input_buffer**: Input data used for encryption.
- **enc_times**: Array to store encryption times.
- **dec_times**: Array to store decryption times.
- **bytes_read**: Number of bytes read from the input buffer.

Process:

1. **Key Pair Generation**: Creates public and secret keys for Kyber-768.
2. **Encryption and Decryption Loop**:
    - **Encryption**: Encrypts input, generates a shared secret and ciphertext, and stores the time.
    - **Decryption**: Decrypts ciphertext, retrieves the shared secret, and stores the time.
    - **Verification**: Ensures that the shared secrets from both encryption and decryption match.
3. **CSV Output**: Saves the times to a CSV file.
4. **Error Handling**: Logs errors for failed key generation, encryption, or decryption.

```
1 int case_kyber(const char *file_to_write_to, int k, uint8_t *input_buffer, double
*enc_times,double                *   dec_times, size_t bytes_read ) {
2     uint8_t public_key[OQS_KEM_kyber_768_length_public_key];
3     uint8_t secret_key[OQS_KEM_kyber_768_length_secret_key];
4     uint8_t shared_secret_A[OQS_KEM_kyber_768_length_shared_secret];

5     // Key pair generation
6     OQS_STATUS status = OQS_KEM_kyber_768_keypair(public_key, secret_key);
7     if (status != OQS_SUCCESS) {
8         fprintf(stderr, "Error: Kyber key pair generation failed.\n");
9         return 1;
10    }

11    // Initialize encryption and decryption time arrays

12    // Loop for each run
13    for (int i = 0; i < k; ++i) {
```

```
        uint8_t ciphertext[OQS_KEM_kyber_768_length_ciphertext];
        memcpy(ciphertext, input_buffer, bytes_read * sizeof(uint8_t));

        // Encryption
        enc_times[i] = kyber_encrypt(ciphertext, shared_secret_A, public_key);

        // Decryption
        uint8_t shared_secret_B[OQS_KEM_kyber_768_length_shared_secret];
        dec_times[i] = kyber_decrypt(shared_secret_B, ciphertext, secret_key);

        // Print and compare input and decrypted content
        printf("Kyber - Run %d:\n", i + 1);
        printf("Shared_secret_A: "); printHex(shared_secret_A,
    OQS_KEM_kyber_768_length_shared_secret);
        printf("Shared_secret_B: "); printHex(shared_secret_B,
    OQS_KEM_kyber_768_length_shared_secret);
        printf("Shared keys 1 and 2 Equal?: %s\n", memcmp(shared_secret_A,
shared_secret_B,    OQS_KEM_kyber_768_length_shared_secret) == 0 ? "\033[0;32mOK\
033[0m" : "\033[0;31mNot OK\    033[0m");
    }

    // Write times to CSV file
    write_to_csv("kyber_768", file_to_write_to, enc_times, dec_times, k);
}
```

Figure 4.4 function case_kyber from Appendix A

4.2.2 OQC Digital Signature Algorithms

4.2.2.1 Used Algorithms

a. Dilithium2

- **Description:** Dilithium is a NIST selected digital signature scheme that provides strong security against chosen message attacks, relying on the difficulty of solving lattice problems over module lattices. This means an attacker with access to a signing oracle cannot generate a signature for a new message or alter an existing signature [22].
- **Size (bytes):**
    - Public key size: 1312
    - Secret key: 2528
    - Signature: 2420 [17]

b. FALCON512

- **Description:** Falcon is another NIST selected digital signature scheme based on the hardness of lattice-based problems, similar to Dilithium, but uses a different mathematical structure called NTRU lattices [23].
- **Size (bytes):**
    - Public key size: 1312
    - Secret key: 2528
    - Signature: 2420 [17]

4.2.2.2 Implementation of Code

The code provided in **Appendix B** benchmarks the performance of two post-quantum signature algorithms, Dilithium2 and Falcon-512, by measuring the time taken for signing and verifying a message.

Input:

1. **User Input**:
    - Algorithm choice:
        - 1 for Dilithium2
        - 2 for Falcon-512
        - 3 for both
    - Number of runs to repeat signing/verifying operations.
2. **File Input**:
    - **input.txt**: Contains the message to be signed (up to 1024 bytes).

Process:

1. **Algorithm Initialization**: Sets up the selected algorithm and generates key pairs.
2. **Signing/Verifying**:
    - Signs the message using the private key.
    - Verifies the signature with the public key.
    - Measures time for both operations in milliseconds.
3. **Benchmarking**:
    - Timing data is recorded and averaged across runs.
    - Results are stored in CSV files:
        - “dilithium2_time.csv”, “falcon512_time.csv”, or “time_of_both_algorithms.csv”.

Output:

CSV files contain:

- Algorithm name
- Signing and verifying times (ms)

- Average times across runs.

The main functions are described below.

a. dilithium2_sign

The dilithium2_sign function signs a message using the Dilithium2 scheme and returns the time taken in double format. It uses OQS_SIG_sign to generate the signature with the provided secret_key, measuring the process with clock(). On failure, it logs an error and returns -1; otherwise, it returns the signing time.

```
1.  double dilithium2_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t  *signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read,uint8_t *secret_key)
{
2.      OQS_STATUS rc;
3.
4.      // Generate keypair
4.      rc = OQS_SIG_keypair(sig, public_key, secret_key);
6.      if (rc != OQS_SUCCESS) {
7.          fprintf(stderr, "ERROR: OQS_SIG_keypair failed!\n");
8.          exit(1);
9.      }
10.
11.     // Display the message to be signed
12.     printf("Message to be signed: ");
13.     for (size_t i = 0; i < bytes_read; i++) {
14.         printf("%02X", message[i]);
14.     }
16.     printf("\n");
17.
18.     // Sign the message
19.     clock_t start = clock();
20.     rc = OQS_SIG_sign(sig, signature, signature_len, message, message_len, secret_key);
21.     clock_t end = clock();
22.     if (rc != OQS_SUCCESS) {
23.         fprintf(stderr, "ERROR: OQS_SIG_sign failed!\n");
24.         exit(1);
24.     }
26.
27.     // Display the generated signature
28.     printf("Generated signature: ");
29.     for (size_t i = 0; i < *signature_len; i++) {
30.         printf("%02X", signature[i]);
31.     }
```

```
    printf("\n");

    return measure_time(start, end);
}
```

Figure 4.5 code for function dilithium2_sign of **Appendix B**

b.dilithium2_verify

The dilithium2_verify function verifies a message's signature using the Dilithium2 scheme and returns the verification time in double format. It uses OQS_SIG_verify to check the signature with the public_key, measuring the process with clock(). On failure, it logs an error and returns -1; otherwise, it returns the verification time.

```
double dilithium2_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read) {
    clock_t start = clock();
    OQS_STATUS rc = OQS_SIG_verify(sig, message, message_len, signature, signature_len, public_key);
    clock_t end = clock();

    // Display the message being verified
    printf("Message being verified: ");
    for (size_t i = 0; i < bytes_read; i++) {
        printf("%02X", message[i]);
    }
    printf("\n");

    // Display verification status
    if (rc == OQS_SUCCESS) {
```

```
14.         printf("Verification successful.\n");
16.     } else {
17.         printf("Verification failed.\n");
18.     }
19.
20.     return measure_time(start, end);
21. }
```

Figure 4.6 code for function **dilithium2**_verify of **Appendix B**

4.3 OpenSSL library for classical algorithms

4.3.1 Encryption

4.3.1.1 Algortihms used

a. AES-256-CBC

AES-256-CBC is a variant of the Rijndael encryption algorithm, with a fixed block size of 128 bits and a key size of 256 bits [24].

b. DES-EDE3-CBC

DES-EDE3-CBC is a variant of the DES encryption algorithm that uses three 56-bit DES keys (168 bits in total) in an Encrypt-Decrypt-Encrypt (EDE) sequence to enhance security [25].

4.3.1.2 Code Implementation

The C program in **Appendix C** benchmarks AES-256-CBC and DES-EDE3-CBC encryption and decryption using OpenSSL. It measures and logs the time taken for these operations in CSV files.

Key Features:

1. **Initialization:** Sets up OpenSSL, generates random key and IV.
2. **User Input:** Prompts for number of runs.
3. **File Preparation:** Creates CSV files for time logs.
4. **Operations:**
    - For each buffer size:
        - Generates random data.
        - Encrypts/decrypts with AES-256-CBC and DES-EDE3-CBC.

- Measures and logs time.

Output: Hex representation of plaintext, ciphertext, decrypted text, and timing logs in CSV files (Openssl_algs_all.csv, destime.csv, aestime.csv).

a. aes_256_cbc_encrypt

The aes_256_cbc_encrypt function encrypts data using AES-256-CBC.

- **Inputs:** input_buffer, buffer_size, key, iv
- **Outputs:** encrypted_input, encrypted_len
- **Process:**
    - Prints input data in hex
    - Measures and logs encryption time
    - Encrypts with AES-256-CBC
    - Prints encrypted data in hex
    - Returns encryption time in milliseconds.

```
1 double aes_256_cbc_encrypt(uint8_t *input_buffer, size_t buffer_size, uint8_t *encrypted_input,    int *encrypted_len, uint8_t *key, uint8_t *iv) {
2 display_content("Input Content (aes_256_cbc)", input_buffer, buffer_size);

3 clock_t start_enc = clock();
4 EVP_CIPHER_CTX *ctx_enc = EVP_CIPHER_CTX_new();
5 EVP_EncryptInit_ex(ctx_enc, EVP_aes_256_cbc(), NULL, key, iv);
6 int out_len;
7  EVP_EncryptUpdate(ctx_enc, encrypted_input, &out_len, input_buffer, buffer_size);
8 int final_len;
9 EVP_EncryptFinal_ex(ctx_enc, encrypted_input + out_len, &final_len);
10 EVP_CIPHER_CTX_free(ctx_enc);
11 clock_t end_enc = clock();
12 *encrypted_len = out_len + final_len;

13 display_content("Encrypted Content (aes_256_cbc)", encrypted_input, *encrypted_len);

14 return measure_time(start_enc, end_enc);
15  }
```

Figure 4.7 function aes_256_cbc_encrypt of **Appendix C**

b. aes_256_cbc_decrypt

The aes_256_cbc_decrypt function decrypts data using AES-256-CBC**.**

- **Inputs:** encrypted_input, encrypted_len, key, iv, output_filename
- **Outputs:** Returns decryption time in milliseconds.
- **Process:**
    - Measures and logs decryption time
    - Decrypts data
    - Prints decrypted data in hex
    - Saves decrypted data to file

```
1 double aes_256_cbc_decrypt(uint8_t *encrypted_input, int encrypted_len, const char
*output_filename, uint8_t *key, uint8_t *iv) {
2 clock_t start_dec = clock();
3 EVP_CIPHER_CTX *ctx_dec = EVP_CIPHER_CTX_new();
4 EVP_DecryptInit_ex(ctx_dec, EVP_aes_256_cbc(), NULL, key, iv);
5 uint8_t decrypted_text[BUFFER_SIZE + EVP_MAX_BLOCK_LENGTH];
6 int out_len_dec;
7 EVP_DecryptUpdate(ctx_dec, decrypted_text, &out_len_dec, encrypted_input,
encrypted_len);
8 int final_len_dec;
9 EVP_DecryptFinal_ex(ctx_dec, decrypted_text + out_len_dec, &final_len_dec);
10 EVP_CIPHER_CTX_free(ctx_dec);
11 clock_t end_dec = clock();
12 int decrypted_len = out_len_dec + final_len_dec;
13
14 display_content("Decrypted Content (aes_256_cbc)", decrypted_text, decrypted_len);
15
16 FILE *output_file = fopen(output_filename, "w");
17 if (output_file != NULL) { 18: fwrite(decrypted_text, 1, decrypted_len, output_file);
19 fclose(output_file);
20 }
21
22 return measure_time(start_dec, end_dec);
23 }
```

Figure 4.8 function **aes_256_cbc_decrypt** of **Appendix C**

4.3.2 Signature

Two variants using SHA-256 hashing algorithm are tested for signature and verification, SHA-256 is a one-way hash function that processes a message to produce a 256-bit message digest. It supports messages smaller than 2^{64} bits, operates on 512-bit blocks, and uses a 32-bit word size [28]. Both variants are implemented in this project to compare classic signature algorithms with those from liboqs.

4.3.2.1 Algorithms used

a. sha256WithRSAEncryption
Uses SHA-256 with the RSA(2048-bit key) digital signature scheme.

b. sha256WithECDSA
Uses SHA-256 with ECDSA (secp256r1, 256-bit key) signature scheme.

4.3.2.2 Implementation of Code

This code listed in **Appendix C** benchmarks RSA and ECDSA algorithms using OpenSSL, measuring signing and verification times.

- **Algorithms:**
    - RSARSA(2048-bit key) with sha256
    - ECDSA(secp256r1, 256-bit key) with sha256
- **Benchmarking:**
    - Records signing and verification times in milliseconds
    - Outputs to rsa_time.csv, ecdsa_time.csv, and time_of_both_algorithms.csv
- **I/O:**
    - Reads from input.txt (default 1024 bytes)
    - Writes results in CSV format
- **Operations:**
    - RSA and ECDSA signing/verification with SHA-256
- **Functions:**
    - rsa_sign, rsa_verify, ecdsa_sign, ecdsa_verify
    - measure_time, calculate_mean, write_to_csv
    - read_file, print_uint8_t

a. rsa_sign
Inputs:

- uint8_t *message: Message to sign
- size_t message_len: Length of the message

- uint8_t *signature: Buffer for the signature
- size_t *signature_len: Length of the signature
- EVP_PKEY *pkey: RSA private key

- **Process:**

1. Initialize context with EVP_MD_CTX_new().
2. Configure for SHA-256 and RSA using EVP_DigestSignInit().
3. Feed message data with EVP_DigestSignUpdate().
4. Determine signature size with EVP_DigestSignFinal() (null buffer).
5. Generate signature with EVP_DigestSignFinal() (actual buffer).
6. Measure signing time.
7. Free context with EVP_MD_CTX_free().

- **Outputs:**

- uint8_t *signature: Generated RSA signature
- size_t *signature_len: Length of the signature
- double: Signing time in milliseconds

double rsa_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t *signature_len, EVP_PKEY *pkey) {

```
0: EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
1: if (mdctx == NULL) {
2: fprintf(stderr, "Error: EVP_MD_CTX_new failed.\n");
3: exit(1);
4: }
5: if (EVP_DigestSignInit(mdctx, NULL, EVP_sha256(), NULL, pkey) <= 0) {
6: fprintf(stderr, "Error: EVP_DigestSignInit failed.\n");
7: exit(1);
8: }
9: if (EVP_DigestSignUpdate(mdctx, message, message_len) <= 0) {
10: fprintf(stderr, "Error: EVP_DigestSignUpdate failed.\n");
11: exit(1);
12: }
```

```
13: size_t req = 0;
14: if (EVP_DigestSignFinal(mdctx, NULL, &req) <= 0) {
15: fprintf(stderr, "Error: EVP_DigestSignFinal (preliminary) failed.\n");
16: exit(1);
17: }
18: *signature_len = req; 1
9: clock_t start = clock();
20: if (EVP_DigestSignFinal(mdctx, signature, signature_len) <= 0) {
21: fprintf(stderr, "Error: EVP_DigestSignFinal failed.\n");
22: exit(1);
23: }
24: clock_t end = clock();
25: EVP_MD_CTX_free(mdctx);
26: return measure_time(start, end);
```

Figure 4.9 function **double rsa_sign** of code in **Appendix D**

b. rsa_verify

Inputs:

- uint8_t *message: Signed message data
- size_t message_len: Length of message
- uint8_t *signature: Signature to verify
- size_t signature_len: Length of signature
- EVP_PKEY *pkey: RSA public key

Process

1. Initialize context with EVP_MD_CTX_new().
2. Configure for SHA-256 and RSA using EVP_DigestVerifyInit().
3. Feed message data with EVP_DigestVerifyUpdate().
4. Measure time and verify signature with EVP_DigestVerifyFinal().

5. Check result: returns 1 (success), 0 (failure), or error.
6. Free context with EVP_MD_CTX_free().

Outputs

- double: Verification time in milliseconds
- Console message: Success or failure

```
1: double rsa_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t signature_len, EVP_PKEY *pkey) {
2: EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
3: if (mdctx == NULL) {
4: fprintf(stderr, "Error: EVP_MD_CTX_new failed.\n");
5: exit(1);
6: }
7: if (EVP_DigestVerifyInit(mdctx, NULL, EVP_sha256(), NULL, pkey) <= 0) {
8: fprintf(stderr, "Error: EVP_DigestVerifyInit failed.\n");
9: exit(1);
10: }
11: if (EVP_DigestVerifyUpdate(mdctx, message, message_len) <= 0) {
12: fprintf(stderr, "Error: EVP_DigestVerifyUpdate failed.\n");
13: exit(1);
14: }
15: clock_t start = clock();
16: int rc = EVP_DigestVerifyFinal(mdctx, signature, signature_len);
17: clock_t end = clock();
18: if (rc == 1) {
19: printf("Verification successful.\n");
20: } else if (rc == 0) {
21: printf("Verification failed.\n");
22: } else {
```

```
23: fprintf(stderr, "Error: EVP_DigestVerifyFinal failed.\n");
24: exit(1);
25: }
26: EVP_MD_CTX_free(mdctx);
27: return measure_time(start, end);
28: }
```

Figure 4.10 function **rsa_verify** of **Appendix D**

=

5. Project results and conclusion

In this section, we will compare the time performance of different algorithms. The tests were performed on a Lenovo IdeaPad 330-15IKB, powered by a Core i5 8th gen 2.5 GHz processor, 1 TB HDD, and 4GB of RAM, running Ubuntu 22.04 LTS. The version of liboqs used was 0.10.0 [27], released on March 23, 2024, and OpenSSL version 3.0.13, released on January 30, 2024. Both libraries, liboqs and OpenSSL, were evaluated under these system conditions through the implementation discussed in **Section 4.**

5.1 Encryption/decryption comparison

Since the encapsulation and decapsulation functions require input data, a 1-byte input text was used for the Key Encapsulation Mechanisms (KEMs), including Kyber, Frodo, HQC, BIKE, and McEliece. For classical algorithms like AES and DES, measurements were taken using input sizes that matched the shared key sizes of the post-quantum cryptographic algorithms for comparison purposes. For example, in Table 5.2, the first row and second column shows the time performance when an input of size equal to Kyber768's shared key size was used for encryption with **AES_256_CBC**.

	Encapsulation	**Decapsulation**
Kyber768	0.023936	0.015892
hqc128	4.567625	5.927775
frodkem768	0.890537	0.860615
bike_l1	0.048726	0.807621
mceliece348864	0.055108	0.101444

Table 5.1 Time performance of QSC algorithsms(KEMs)

Input size (QSC algorithms' shared key sizes**)**	**AES_256_CBC**		**DES_256_CBC**	
	Encryption	**Decryption**	**Encryption**	**Decryption**
Kyber768	0.006318	0.004096	0.008393	0.006211
hqc128	0.008566	0.005178	0.016096	0.012008
frodkem768	0.008357	0.004987	0.010699	0.007507
bike_l1	0.007996	0.005031	0.01146	0.008245
mceliece348864	0.009813	0.006066	0.009813	0.006066

Table 5.2 Time performance of Classical algorithms with various input sizes

5.1.1 Encryption

Below, we will compare Kyber768 with AES-256-CBC. For Kyber's encapsulation mechanism, a plaintext input size of 1-byte was used. In contrast, a randomly generated buffer, with a size equal to Kyber's shared key, was used as the input for AES-256-CBC encryption.

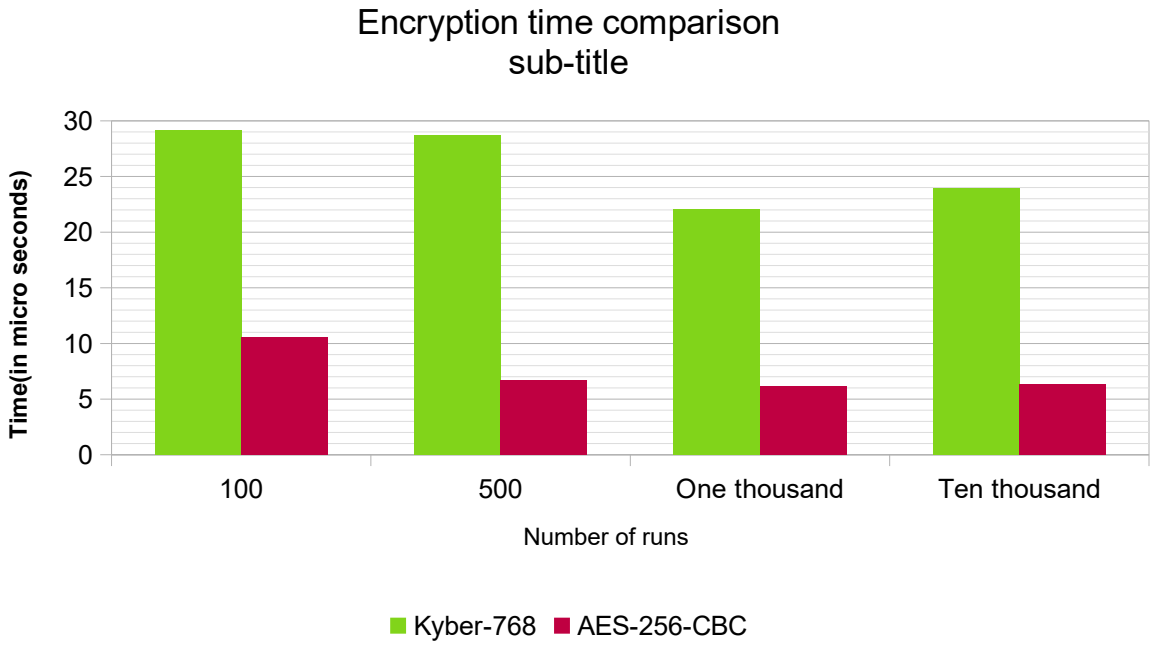

Figure 5.1 encryption time between kyber768 and aes256cbc

5.1.2 Decryption

The ciphertexts generated by both encryption mechanisms, Kyber768 and AES-256-CBC, were decrypted using their respective methods, and the decryption times were measured and compared.

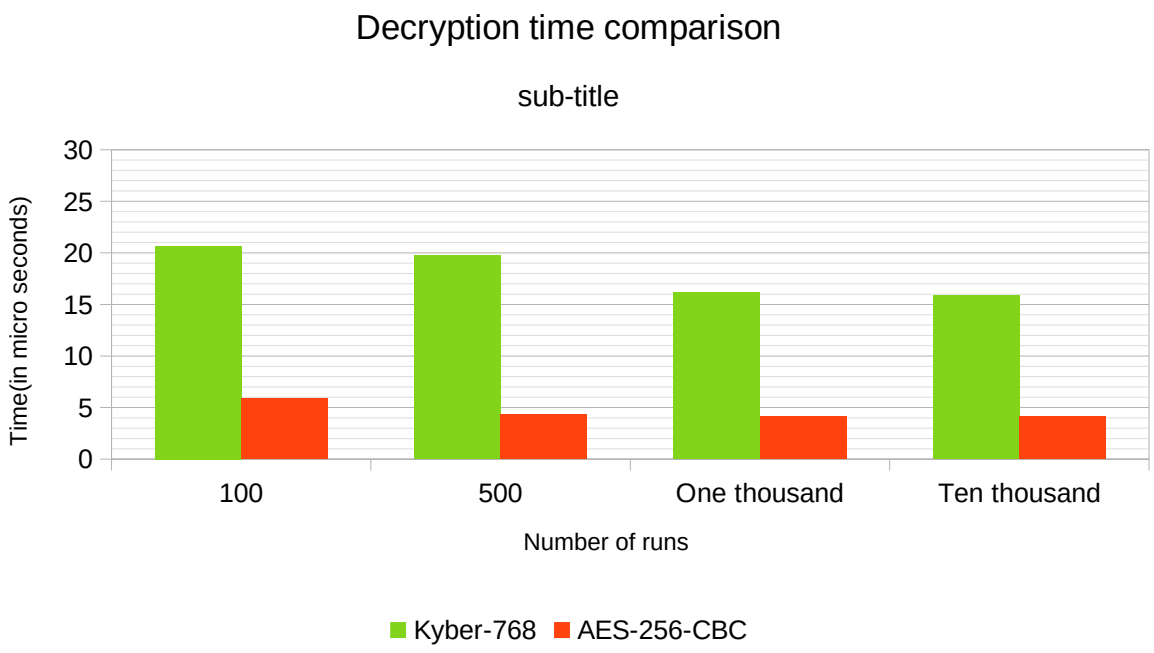

Figure 5.2 Decryption time comparison of kyber768 and aes256cbc

5.2 Signature/Verification

The operations were executed 10,000 times, and the recorded times, in milliseconds, are presented in the table below. This data provides a detailed comparison of performance across different algorithms based on the repeated executions.

Algorithms	Signature	Verification
Dilithium2	0.081364	0.031961
Falcon-512	0.3254	0.063503
sha256_RSA	0.601328	0.019635
sha256_ECDSA	0.028242	0.072797

Table 5.3 time measurement of used signature/verification algorithms(in milliseconds)

Using the data from the outputs of code in Appendix B and D, we will compare the performance between two of the used algorithms, Falcon512 and SHA256 with RSA. Both algorithms were used to sign and verify a 6-byte message from the input file, and their performance in terms of signing and verification times was compared.

5.2.1 Signature

Below is a visual comparison of the signature time performance between Falcon512 and SHA256 with RSA. The comparison was conducted with four different repetitions, illustrating the performance differences between the two algorithms.

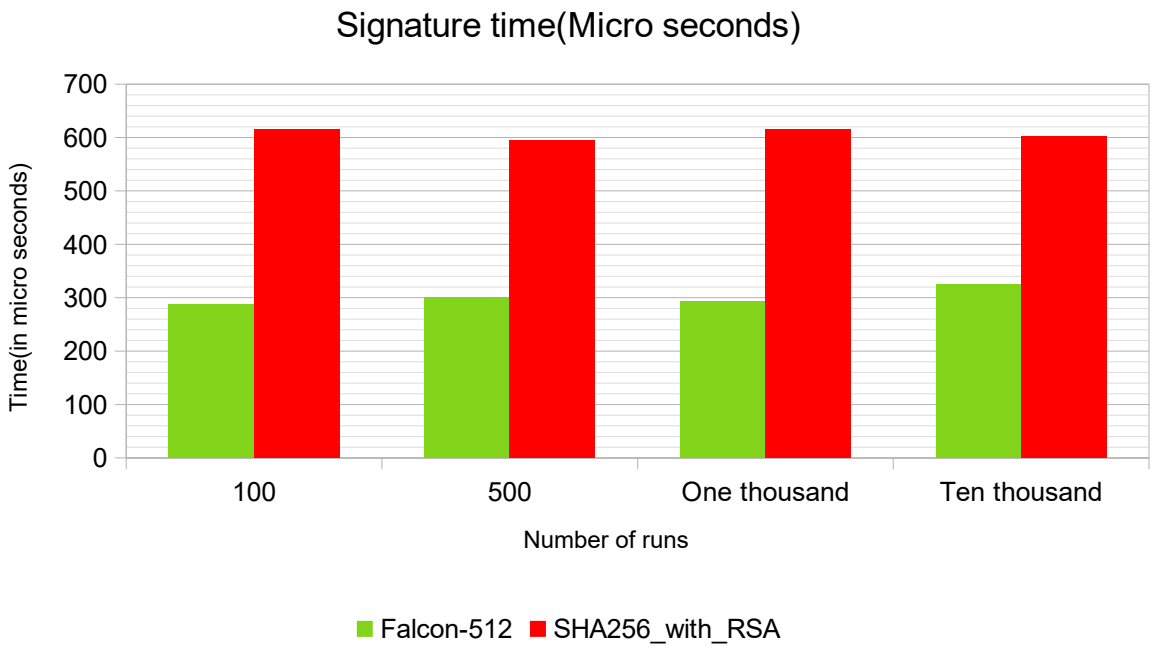

Figure 5.3 Signature time comparison between Falcon512 and sha256with RSA

5.2.2 Verification

The signature generated earlier was verified using the respective methods of both Falcon512 and SHA256 with RSA, with the verification times measured and compared for both algorithms.

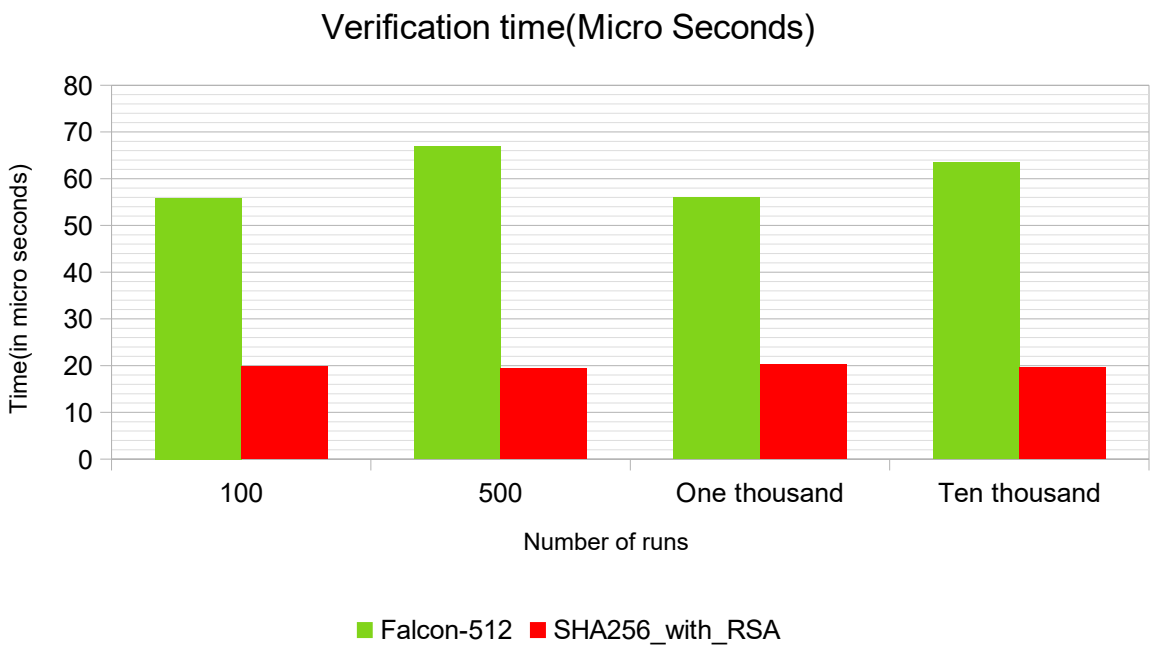

Figure 5.4 Verification time comparison between Falcon512 and sha256withRSA

5.3 Summary and Conclusion

The performance data provided highlights several key differences between post-quantum cryptographic algorithms and classical cryptographic algorithms in terms of encryption, decryption, signature, and verification times.

5.3.1. Encapsulation and Decapsulation

According to Table 5.1 and Table 5.2**,** additionally as shown in Figure 5.5:

- **Kyber768 (Quantum)**: The fastest performance in both encapsulation (0.023936 ms) and decapsulation (0.015892 ms), indicating that it is one of the most efficient quantum-safe key encapsulation mechanisms (KEMs).
- **AES_256_CBC (Classical)**: When using the shared key size derived from Kyber768, AES encryption and decryption times were 0.006318 ms and 0.004096 ms, respectively. This shows that classical AES encryption and decryption are faster than quantum-safe Kyber768, though Kyber provides future-proof security.
- **HQC128 (Quantum)**: Significantly slower, with encapsulation (4.567625 ms) and decapsulation (5.927775 ms), reflecting a higher computational demand compared to other quantum-safe algorithms.

- **DES_256_CBC (Classical)**: For HQC128's shared secret size, DES encryption and decryption times were 0.016096 ms and 0.012008 ms, respectively. Even though DES is slower than AES, it still outperforms HQC128 significantly.
- **FrodoKEM768 (Quantum)**: Moderate performance with encapsulation (0.890537 ms) and decapsulation (0.860615 ms), offering a balance between security and performance.
- **AES_256_CBC (Classical)**: For FrodoKEM's shared key size, AES showed encryption (0.008357 ms) and decryption (0.004987 ms) times, much faster than FrodoKEM.
- **BIKE_L1 (Quantum)**: Encapsulation (0.048726 ms) and decapsulation (0.807621 ms) times indicate that it performs relatively fast in encapsulation but slower in decapsulation.
- **DES_256_CBC (Classical)**: For BIKE_L1's shared key size, DES encryption (0.01146 ms) and decryption (0.008245 ms) times are faster than BIKE_L1, making classical DES more efficient in this context.
- **McEliece348864 (Quantum)**: Encapsulation (0.055108 ms) and decapsulation (0.101444 ms) times reflect its efficiency in quantum-safe encryption, though still slower than classical counterparts.
- **AES_256_CBC (Classical)**: When using McEliece's shared secret size, AES encryption and decryption were 0.009813 ms and 0.006066 ms, respectively, showing better performance compared to McEliece.

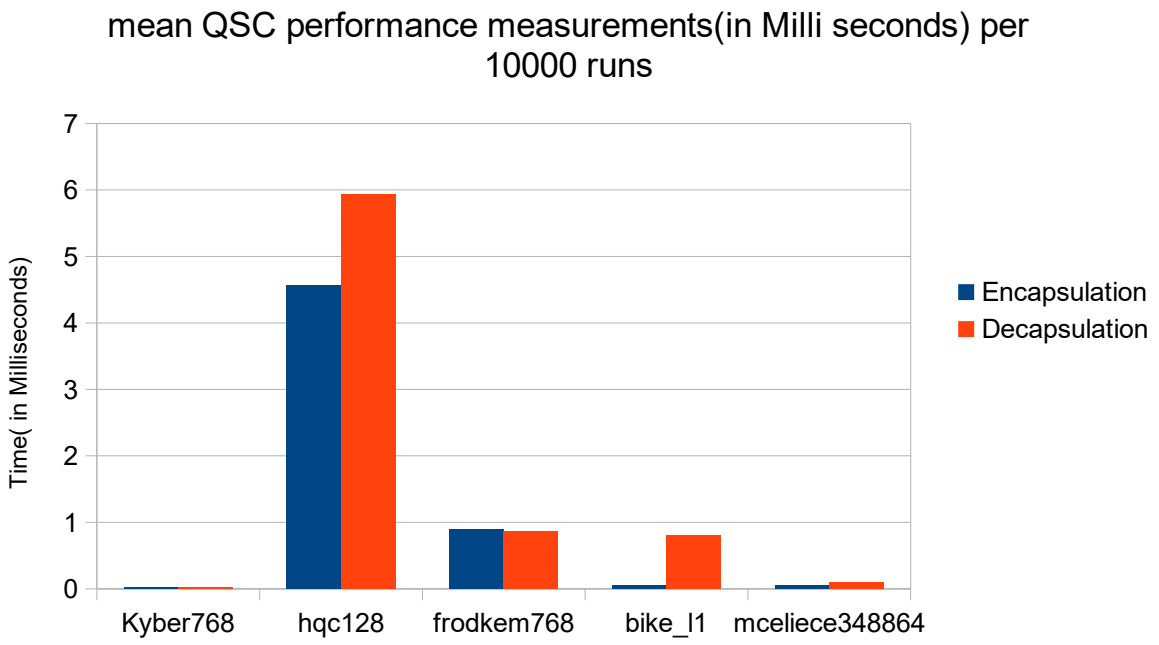

Figure 5.5 Mean times recorder for QSC algorithms for 10,000 runs

5.3.2. Signature and Verification

According to Table 5.3 and Figure 5.6 below:

- Among quantum-safe algorithms, **Dilithium2** shows fast signature times (0.081364 ms) and even faster verification times (0.031961 ms), making it efficient for both operations.
- **Falcon-512** is slower in signature generation (0.3254 ms) but still has competitive verification times (0.063503 ms).

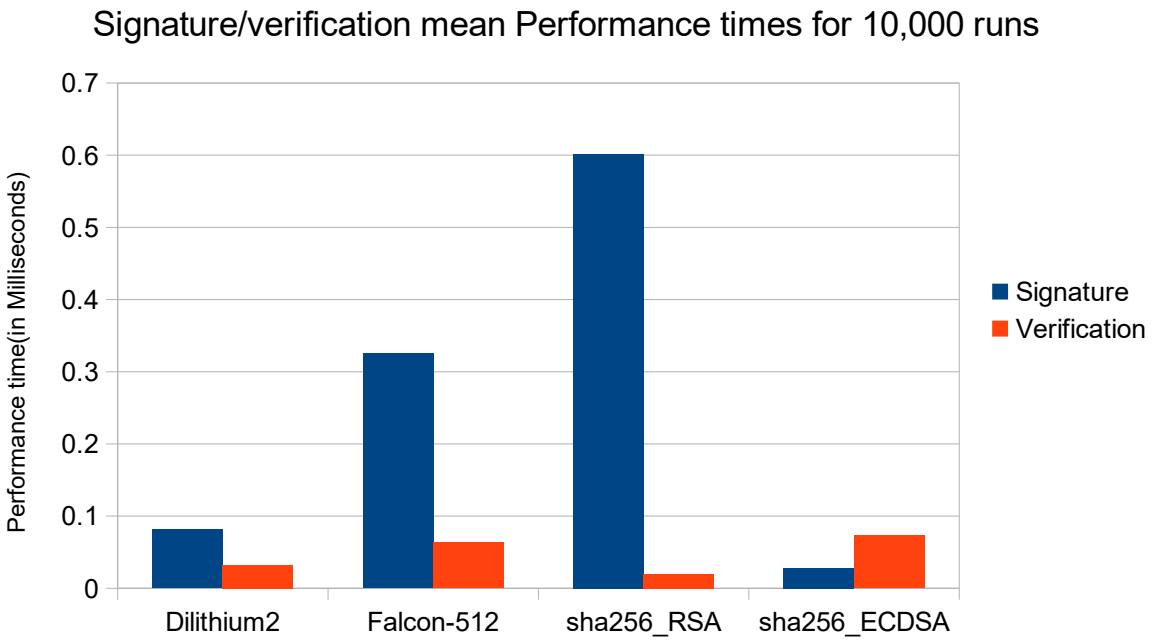

Figure 5.6 Mean performance times(in Milliseconds) for 10,000 runs

5.3.2 Overall Insights

- **Quantum-Safe Algorithms**: Kyber768 and Dilithium2 exhibit the best performance for encapsulation and signatures among the post-quantum options, showing they are efficient candidates for future cryptographic standards.
- **Classical Algorithms**: While classical algorithms like AES and RSA are still faster in some verification tasks, post-quantum algorithms like Falcon and Dilithium are closing the performance gap, particularly in verification tasks.

- **Trade-offs**: Quantum-safe algorithms, although slower in some cases, offer critical resistance to future quantum attacks, making them essential for long-term cryptographic security.

The data shows that as quantum computing progresses, post-quantum algorithms are becoming increasingly viable, though some algorithms still have higher computational costs than classical methods.

Bibliography

1.Nielsen, M. A., & Chuang, I. L. (2010). Quantum computation and quantum information. Cambridge University Press.

2. Singh, R., & Bodile, R. M. (2024). A quick guide to quantum communication. [PDF file]. https://arxiv.org/html/2402.15707v1

3. Griffiths, D. J., & Schroeter, D. F. (2018). Quantum mechanics. Cambridge University Press.

4. Singh, S., & Sakk, E. (2024). Implementation and analysis of Shor's algorithm to break RSA cryptosystem security. TechRxiv. https://doi.org/10.36227/techrxiv.170259160.05374043/v2

5. Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. MIT Laboratory for Computer Science.

6. Lenstra, A. K., Lenstra, H. W., Jr., Manasse, M. S., & Pollard, J. M. (1992). The number field sieve. https://www.researchgate.net/publication/225158770_The_number_field_sieve

7. Camps, D., Van Beeumen, R., & Yang, C. (2024). Quantum Fourier transform revisited. TechRxiv. https://doi.org/10.36227/techrxiv.170259160.05374043/v2

8.Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. https://arxiv.org/abs/quant-ph/9508027

9. Tavares, F. (2024, January 8). Google and NASA achieve quantum supremacy. NASA. https://www.nasa.gov/technology/computing/google-and-nasa-achieve-quantum-supremacy/

10. IBM Quantum. (2024, January 8). Quantum roadmap 2033. IBM. https://www.ibm.com/quantum/blog/quantum-roadmap-2033

11. Gagliardoni, T. (2021, August 24). Quantum attack resource estimate: Using Shor's algorithm to break RSA vs DH/DSA vs ECC.https://research.kudelskisecurity.com/2021/08/24/quantum-attack-resource-estimate-using-shors-algorithm-to-break-rsa-vs-dh-dsa-vs-ecc/

12. Gheorghiu, V., & Mosca, M. (2024). A resource estimation framework for quantum attacks against cryptographic functions - Recent developments. EvolutionQ Inc.

13. Barker, W., Polk, W., & Souppaya, M. (2021). Cryptography: Exploring challenges associated with adopting and using post-quantum cryptographic algorithms. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.CSWP.15

14. National Institute of Standards and Technology. (2024, August 13). Announcing approval of three federal information processing standards (FIPS) for post-quantum cryptography. https://csrc.nist.gov/News/2024/postquantum-cryptography-fips-approved

15. Open Quantum Safe. "Home." Accessed September 15, 2024. https://openquantumsafe.org/.

16. PQ-Crystals. "Kyber." Accessed September 15, 2024. https://pq-crystals.org/kyber/index.shtml.

17. Open Quantum Safe. "algorithms" Accessed September 15, 2024. https://openquantumsafe.org/liboqs/algorithms

18. FrodoKEM. "FrodoKEM." Accessed September 15, 2024. https://frodokem.org/

19. BikeSuite. "BikeSuite ." Accessed September 15, 2024.https://bikesuite.org/

20. Classic_Mceliece. " Classic_Mceliece ." Accessed September 15, 2024.https://classic.mceliece.org/

21.HQC. " HQC." Accessed September 15, 2024. https://pqc-hqc.org/

22.PQ-Crystals. "dilithium ." Accessed September 15, 2024. https://pq-crystals.org/dilithium/

23. Falcon. "About Falcon." Accessed September 15, 2024. https://falcon-sign.info/.

24. Wikipedia. "Advanced Encryption Standard." Last modified September 15, 2024. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

25. Kummert, H. "The PPP Triple-DES Encryption Protocol (3DESE)." RFC 2420. September 1998. 2024. https://www.rfc-editor.org/rfc/rfc2420.html.

26. National Institute of Standards and Technology (NIST). "FIPS PUB 180-4: Secure Hash Standard (SHS)." August 2015. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf.

27. Open Quantum Safe. "liboqs Release 0.10.0." GitHub, March 23, 2024. https://github.com/open-quantum-safe/liboqs/releases/tag/0.10.0.

28. Secure Hash Algorithms. In *Wikipedia, The Free Encyclopedia*. Retrieved September 17, 2024, from https://en.wikipedia.org/wiki/Secure_Hash_Algorithms

Appendix

Appendix A

```
//this code is used for measurig perfomance of OQC algs
#include <stdio.h>
#include <stdlib.h>
#include <oqs/oqs.h>
#include <oqs/kem_kyber.h>
#include <oqs/kem_hqc.h>
#include <oqs/kem_frodokem.h>
#include <oqs/kem_bike.h>
#include <oqs/kem_classic_mceliece.h>
#include <string.h>
#include <stdint.h>
#include <time.h>

#define BUFFER_SIZE 10000

// Function prototypes
double kyber_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key);
double kyber_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key);
double hqc_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key);
double hqc_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key);
double frodo_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key);
double frodo_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key);
double bike_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key);
double bike_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key);
double mceliece_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key);
double mceliece_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key);
void printHex(const uint8_t *data, size_t data_len);
void OQS_cleanup(void);
int case_kyber(const char *file_to_write_to, int k, uint8_t *input_buffer, double
*enc_times,double *dec_times, size_t bytes_read );
int case_hqc(const char *file_to_write_to, int k, uint8_t *input_buffer, double *enc_times,double
*dec_times, size_t bytes_read );
int case_frodo(const char *file_to_write_to, int k, uint8_t *input_buffer, double
*enc_times,double *dec_times, size_t bytes_read );
int case_bike(const char *file_to_write_to, int k, uint8_t *input_buffer, double *enc_times,double
*dec_times, size_t bytes_read );
```

```
int case_mceliece(const char *file_to_write_to, int k, uint8_t *input_buffer, double
*enc_times,double *dec_times, size_t bytes_read ) ;
// Encryption and Decryption Timer functions
double measure_time_encrypt(clock_t start, clock_t end);
double measure_time_decrypt(clock_t start, clock_t end);

// Function to calculate the mean of an array of doubles
double calculate_mean(double *times, int size) {
    double sum = 0.0;
    for (int i = 0; i < size; ++i) {
        sum += times[i];
    }
    return sum / size;
}

// Function to write encryption and decryption times to a CSV file
void write_to_csv(const char *alg_name, const char *filename, double *enc_times, double
*dec_times, int k) {
    FILE *file = fopen(filename, "a");
    if (file == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }

    for (int i = 0; i < k; ++i) {
        fprintf(file, "%s,%.6f,%.6f\n", alg_name, enc_times[i], dec_times[i]);
    }

    // Calculate and write the mean times
    double mean_enc_time = calculate_mean(enc_times, k);
    double mean_dec_time = calculate_mean(dec_times, k);
    fprintf(file, "Mean_%s,%.6f,%.6f\n", alg_name, mean_enc_time, mean_dec_time);

    fclose(file);
}

int main() {
    OQS_init();
    srand(time(0));

    // Open input file for reading
```

```
FILE *input_file = fopen("input.txt", "rb");
if (input_file == NULL) {
    fprintf(stderr, "Error: Unable to open input file.\n");
    return 1;
}

// Read input from file
uint8_t input_buffer[BUFFER_SIZE];
size_t bytes_read = fread(input_buffer, 1, BUFFER_SIZE, input_file);
fclose(input_file); // Close input file after reading
if (bytes_read == 0) {
    fprintf(stderr, "Error: Failed to read input file.\n");
    return 1;
}
bytes_read = OQS_KEM_kyber_768_length_shared_secret;

// Menu
int choice;
printf("Select an algorithm:\n");
printf("1. Kyber\n");
printf("2. HQC\n");
printf("3. Frodo\n");
printf("4. BIKE\n");
printf("5. McEliece\n");
printf("6. All\n");
printf("Enter your choice: ");
scanf("%d", &choice);

// Prompt for number of runs
int k;
printf("Enter the number of runs: ");
scanf("%d", &k);

// Encryption and Decryption Time Storage
double *enc_times = malloc(k * sizeof(double));
double *dec_times = malloc(k * sizeof(double));

// Perform operations based on choice
switch (choice) {
    case 1: {
    FILE *file4 = fopen("tkyber.csv", "w");
```

```
if (file4 == NULL) {
   fprintf(stderr, "Error: Unable to open output file.\n");
   return 0;
}

fprintf(file4, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
fclose(file4);
       case_kyber("tkyber.csv", k, input_buffer, enc_times, dec_times, bytes_read);
       break;
   }
   case 2: {
   FILE *file4 = fopen("thqc.csv", "w");
if (file4 == NULL) {
   fprintf(stderr, "Error: Unable to open output file.\n");
   return 0;
}

fprintf(file4, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
fclose(file4);

       case_hqc("thqc.csv",k, input_buffer, enc_times, dec_times, bytes_read);
       break;
   }
   case 3: {

   FILE *file4 = fopen("tfrodo.csv", "w");
if (file4 == NULL) {
   fprintf(stderr, "Error: Unable to open output file.\n");
   return 0;
}

fprintf(file4, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
fclose(file4);

       case_frodo("tfrodo.csv",k, input_buffer, enc_times, dec_times, bytes_read);
       break;
   }
   case 4: {
   FILE *file4 = fopen("tbike.csv", "w");
if (file4 == NULL) {
   fprintf(stderr, "Error: Unable to open output file.\n");
```

```
        return 0;
    }

    fprintf(file4, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file4);

            case_bike("tbike.csv",k, input_buffer, enc_times, dec_times, bytes_read);
            break;
        }
        case 5: {
        FILE *file4 = fopen("tmceliece.csv", "w");
    if (file4 == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return 0;
    }

    fprintf(file4, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file4);

            case_mceliece("tmceliece.csv",k, input_buffer, enc_times, dec_times, bytes_read);
            break;
        }
        case 6: {
             FILE *file4 = fopen("Time_All_KEMs.csv", "w");
    if (file4 == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return 0;
    }

    fprintf(file4, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file4);
            case_kyber("Time_All_KEMs.csv", k, input_buffer, enc_times, dec_times, bytes_read);
            case_hqc("Time_All_KEMs.csv", k, input_buffer, enc_times, dec_times, bytes_read);
            case_frodo("Time_All_KEMs.csv", k, input_buffer, enc_times, dec_times, bytes_read);
            case_bike("Time_All_KEMs.csv", k, input_buffer, enc_times, dec_times, bytes_read);
            case_mceliece("Time_All_KEMs.csv", k, input_buffer, enc_times, dec_times,
bytes_read);

            break;
        }
        default:
```

```
            printf("Invalid choice.\n");
            break;
    }


    OQS_cleanup();
    return 0;
}


// Function to measure encryption time
double measure_time_encrypt(clock_t start, clock_t end) {
    return ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;
}

// Function to measure decryption time
double measure_time_decrypt(clock_t start, clock_t end) {
    return ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;
}

// Function to print hexadecimal data
void printHex(const uint8_t *data, size_t data_len) {
    for (size_t i = 0; i < data_len; ++i) {
        printf("%02X ", data[i]);
    }
    printf("\n");
}

// Kyber Encryption
double kyber_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_kyber_768_encaps(ciphertext, shared_secret_A,
public_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: Kyber encryption failed.\n");
        return -1;
    }
    return measure_time_encrypt(start, end);
}
```

```
// Kyber Decryption
double kyber_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_kyber_768_decaps(shared_secret_B, ciphertext,
secret_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: Kyber decryption failed.\n");
        return -1;
    }
    return measure_time_decrypt(start, end);
}

// HQC Encryption
double hqc_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_hqc_128_encaps(ciphertext, shared_secret_A,
public_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: HQC encryption failed.\n");
        return -1;
    }
    return measure_time_encrypt(start, end);
}

// HQC Decryption
double hqc_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_hqc_128_decaps(shared_secret_B, ciphertext,
secret_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: HQC decryption failed.\n");
        return -1;
    }
    return measure_time_decrypt(start, end);
}

// Frodo Encryption
double frodo_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key) {
```

```
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_frodokem_640_aes_encaps(ciphertext, shared_secret_A,
public_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: Frodo encryption failed.\n");
        return -1;
    }
    return measure_time_encrypt(start, end);
}

// Frodo Decryption
double frodo_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_frodokem_640_aes_decaps(shared_secret_B, ciphertext,
secret_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: Frodo decryption failed.\n");
        return -1;
    }
    return measure_time_decrypt(start, end);
}

// BIKE Encryption
double bike_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_bike_l1_encaps(ciphertext, shared_secret_A, public_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: BIKE encryption failed.\n");
        return -1;
    }
    return measure_time_encrypt(start, end);
}

// BIKE Decryption
double bike_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_bike_l1_decaps(shared_secret_B, ciphertext, secret_key);
    clock_t end = clock();
```

```
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: BIKE decryption failed.\n");
        return -1;
    }
    return measure_time_decrypt(start, end);
}

// McEliece Encryption
double mceliece_encrypt(uint8_t *ciphertext, uint8_t *shared_secret_A, uint8_t *public_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_classic_mceliece_348864_encaps(ciphertext,
shared_secret_A, public_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: McEliece encryption failed.\n");
        return -1;
    }
    return measure_time_encrypt(start, end);
}

// McEliece Decryption
double mceliece_decrypt(uint8_t *shared_secret_B, uint8_t *ciphertext, uint8_t *secret_key) {
    clock_t start = clock();
    OQS_STATUS status = OQS_KEM_classic_mceliece_348864_decaps(shared_secret_B,
ciphertext, secret_key);
    clock_t end = clock();
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: McEliece decryption failed.\n");
        return -1;
    }
    return measure_time_decrypt(start, end);
}

void OQS_cleanup(void) {
      OQS_destroy();
}



int case_kyber(const char *file_to_write_to, int k, uint8_t *input_buffer, double
*enc_times,double *dec_times, size_t bytes_read ) {
```

```
    uint8_t public_key[OQS_KEM_kyber_768_length_public_key];
    uint8_t secret_key[OQS_KEM_kyber_768_length_secret_key];
    uint8_t shared_secret_A[OQS_KEM_kyber_768_length_shared_secret];

    // Key pair generation
    OQS_STATUS status = OQS_KEM_kyber_768_keypair(public_key, secret_key);
    if (status != OQS_SUCCESS) {
        fprintf(stderr, "Error: Kyber key pair generation failed.\n");
        return 1;
    }

    // Loop for each run
    for (int i = 0; i < k; ++i) {
        uint8_t ciphertext[OQS_KEM_kyber_768_length_ciphertext];
        memcpy(ciphertext, input_buffer, bytes_read * sizeof(uint8_t));

        // Encryption
        enc_times[i] = kyber_encrypt(ciphertext, shared_secret_A, public_key);

        // Decryption
        uint8_t shared_secret_B[OQS_KEM_kyber_768_length_shared_secret];
        dec_times[i] = kyber_decrypt(shared_secret_B, ciphertext, secret_key);

        // Print and compare input and decrypted content
        printf("Kyber - Run %d:\n", i + 1);
        printf("Shared_secret_A: ");printHex(shared_secret_A,
OQS_KEM_kyber_768_length_shared_secret);
        printf("Shared_secret_B: ");printHex(shared_secret_B,
OQS_KEM_kyber_768_length_shared_secret);
        printf("Shared keys 1 and 2 Equal?: %s\n", memcmp(shared_secret_A, shared_secret_B,
OQS_KEM_kyber_768_length_shared_secret) == 0 ? "\033[0;32mOK\033[0m" : "\
033[0;31mNot OK\033[0m");
    }

    // Write times to CSV file
    write_to_csv("kyber_768",file_to_write_to, enc_times, dec_times, k);



}
```

```
int case_hqc(const char *file_to_write_to, int k, uint8_t *input_buffer, double *enc_times,double
*dec_times, size_t bytes_read ) {
   uint8_t public_key[OQS_KEM_hqc_128_length_public_key];
        uint8_t secret_key[OQS_KEM_hqc_128_length_secret_key];
        uint8_t shared_secret_A[OQS_KEM_hqc_128_length_shared_secret];
        OQS_STATUS status = OQS_KEM_hqc_128_keypair(public_key, secret_key);
        if (status != OQS_SUCCESS) {
           fprintf(stderr, "Error: HQC key pair generation failed.\n");
           return 1;
        }

        clock_t start, end;
        for (int i = 0; i < k; ++i) {

           uint8_t ciphertext[OQS_KEM_hqc_128_length_ciphertext];
           memcpy(ciphertext, input_buffer, bytes_read * sizeof(uint8_t));


           enc_times[i] = hqc_encrypt(ciphertext, shared_secret_A, public_key);
           uint8_t shared_secret_B[OQS_KEM_hqc_128_length_shared_secret];


           dec_times[i] = hqc_decrypt(shared_secret_B, ciphertext, secret_key);

           // Print and compare input and decrypted content
           printf("HQC - Run %d:\n", i + 1);
           printf("Shared_secret_A: ");printHex(shared_secret_A,
OQS_KEM_hqc_128_length_shared_secret);
           printf("Shared_secret_B: ");printHex(shared_secret_B,
OQS_KEM_hqc_128_length_shared_secret);
           printf("Shared keys 1 and 2 Equal?: %s\n", memcmp(shared_secret_A,
shared_secret_B, OQS_KEM_hqc_128_length_shared_secret) == 0 ? "\033[0;32mOK\033[0m" :
"\033[0;31mNot OK\033[0m");
        }

        write_to_csv("hqc_128",file_to_write_to, enc_times, dec_times, k);



}
```

```
int case_frodo(const char *file_to_write_to, int k, uint8_t *input_buffer, double
*enc_times,double *dec_times, size_t bytes_read ) {
   uint8_t public_key[OQS_KEM_frodokem_640_aes_length_public_key];
        uint8_t secret_key[OQS_KEM_frodokem_640_aes_length_secret_key];
        uint8_t shared_secret_A[OQS_KEM_frodokem_640_aes_length_shared_secret];
        OQS_STATUS status = OQS_KEM_frodokem_640_aes_keypair(public_key, secret_key);
        if (status != OQS_SUCCESS) {
           fprintf(stderr, "Error: Frodo key pair generation failed.\n");
           return 1;
        }

        clock_t start, end;
        for (int i = 0; i < k; ++i) {

           uint8_t ciphertext[OQS_KEM_frodokem_640_aes_length_ciphertext];
           memcpy(ciphertext, input_buffer, bytes_read * sizeof(uint8_t));
           frodo_encrypt(ciphertext, shared_secret_A, public_key);

           enc_times[i] = frodo_encrypt(ciphertext, shared_secret_A, public_key);

           uint8_t shared_secret_B[OQS_KEM_frodokem_640_aes_length_shared_secret];


           dec_times[i] = frodo_decrypt(shared_secret_B, ciphertext, secret_key);

           // Print and compare input and decrypted content
           printf("Frodo - Run %d:\n", i + 1);
           printf("Shared_secret_A: ");printHex(shared_secret_A,
OQS_KEM_frodokem_640_aes_length_shared_secret);
           printf("Shared_secret_B: ");printHex(shared_secret_B,
OQS_KEM_frodokem_640_aes_length_shared_secret);
           printf("Shared keys 1 and 2 Equal?: %s\n", memcmp(shared_secret_A,
shared_secret_B, OQS_KEM_frodokem_640_aes_length_shared_secret) == 0 ? "\033[0;32mOK\
033[0m" : "\033[0;31mNot OK\033[0m");
        }

        write_to_csv("frodokem_640",file_to_write_to, enc_times, dec_times, k);


}
```

```
int case_bike(const char *file_to_write_to, int k, uint8_t *input_buffer, double *enc_times,double
*dec_times, size_t bytes_read ) {
   uint8_t public_key[OQS_KEM_bike_l1_length_public_key];
        uint8_t secret_key[OQS_KEM_bike_l1_length_secret_key];
        uint8_t shared_secret_A[OQS_KEM_bike_l1_length_shared_secret];
        OQS_STATUS status = OQS_KEM_bike_l1_keypair(public_key, secret_key);
        if (status != OQS_SUCCESS) {
           fprintf(stderr, "Error: BIKE key pair generation failed.\n");
           return 1;
        }


        for (int i = 0; i < k; ++i) {

           uint8_t ciphertext[OQS_KEM_bike_l1_length_ciphertext]; memcpy(ciphertext,
input_buffer, bytes_read * sizeof(uint8_t));
           bike_encrypt(ciphertext, shared_secret_A, public_key);

           enc_times[i] =          bike_encrypt(ciphertext, shared_secret_A, public_key);


           uint8_t shared_secret_B[OQS_KEM_bike_l1_length_shared_secret];


           dec_times[i] =          bike_decrypt(shared_secret_B, ciphertext, secret_key);

           // Print and compare input and decrypted content
           printf("BIKE - Run %d:\n", i + 1);
           printf("Shared_secret_A: ");printHex(shared_secret_A,
OQS_KEM_bike_l1_length_shared_secret);
           printf("Shared_secret_B: ");printHex(shared_secret_B,
OQS_KEM_bike_l1_length_shared_secret);
           printf("Shared keys 1 and 2 Equal?: %s\n", memcmp(shared_secret_A,
shared_secret_B, OQS_KEM_bike_l1_length_shared_secret) == 0 ? "\033[0;32mOK\033[0m" :
"\033[0;31mNot OK\033[0m");
        }

        write_to_csv("bike_l1",file_to_write_to, enc_times, dec_times, k);


}
```

```
int case_mceliece(const char *file_to_write_to, int k, uint8_t *input_buffer, double *enc_times,
double *dec_times, size_t bytes_read) {
   uint8_t public_key[OQS_KEM_classic_mceliece_348864_length_public_key];
   uint8_t secret_key[OQS_KEM_classic_mceliece_348864_length_secret_key];
   uint8_t shared_secret_A[OQS_KEM_classic_mceliece_348864_length_shared_secret];
   uint8_t shared_secret_B[OQS_KEM_classic_mceliece_348864_length_shared_secret];
   OQS_STATUS status = OQS_KEM_classic_mceliece_348864_keypair(public_key,
secret_key);
   if (status != OQS_SUCCESS) {
      fprintf(stderr, "Error: McEliece key pair generation failed.\n");
      return 1;
   }

   for (int i = 0; i < k; ++i) {
      uint8_t ciphertext[OQS_KEM_classic_mceliece_348864_length_ciphertext];
      memcpy(ciphertext, input_buffer, bytes_read * sizeof(uint8_t));

      // Encryption
      enc_times[i] = mceliece_encrypt(ciphertext, shared_secret_A, public_key);

      // Decryption
      dec_times[i] = mceliece_decrypt(shared_secret_B, ciphertext, secret_key);

      // Print and compare input and decrypted content
      printf("McEliece - Run %d:\n", i + 1);
      printf("Shared_secret_A: ");printHex(shared_secret_A,
OQS_KEM_classic_mceliece_348864_length_shared_secret);
      printf("Shared_secret_B: ");printHex(shared_secret_B,
OQS_KEM_classic_mceliece_348864_length_shared_secret);
      printf("Shared keys 1 and 2 Equal?: %s\n", memcmp(shared_secret_A, shared_secret_B,
OQS_KEM_classic_mceliece_348864_length_shared_secret) == 0 ? "\033[0;32mOK\033[0m" :
"\033[0;31mNot OK\033[0m");
   }

   write_to_csv("classic_mceliece_348864",file_to_write_to, enc_times, dec_times, k);

   return 0;
}
```

Appendix B

```
//liboqs signature and verification

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include <oqs/oqs.h>

#define MESSAGE_LEN 1024


/* Function prototypes */
double measure_time(clock_t start, clock_t end);
double calculate_mean(double *times, int length);
void write_to_csv(const char *alg_name, const char *filename, double *sign_times, double
*verify_times, int k);
double dilithium2_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t
*signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read,uint8_t *secret_key);
double dilithium2_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t
signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read);
double falcon512_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t
*signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read,uint8_t *secret_key);
double falcon512_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t
signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read);

int main() {
    int choice;
    int runs;
    printf("Choose the algorithm:\n1. Dilithium2\n2. Falcon-512\n3. Both\nEnter your choice: ");
    scanf("%d", &choice);

    printf("Enter the number of runs: ");
    scanf("%d", &runs);

    double dilithium_sign_times[runs], dilithium_verify_times[runs];
    double falcon_sign_times[runs], falcon_verify_times[runs];
```

```
uint8_t message[MESSAGE_LEN];

size_t signature_len;


FILE *input_file = fopen("input.txt", "rb");
if (input_file == NULL) {
    fprintf(stderr, "Error: Unable to open input file.\n");
    return 1;
}

// Read input from file

size_t bytes_read = fread(message, 1, MESSAGE_LEN, input_file);
fclose(input_file); // Close input file after reading
if (bytes_read == 0) {
    fprintf(stderr, "Error: Failed to read input file.\n");
    return 1;
}


FILE *file3 = fopen("time_of_both_algorithms.csv", "w");
if (file3 == NULL) {
    fprintf(stderr, "Error: Unable to open output file.\n");
    return;
}
    fprintf(file3, "Alg_name,Signing Time (ms),Verifying Time (ms)\n");
       fclose(file3);

if (choice == 1 || choice == 3) {
    OQS_SIG *dilithium_sig = OQS_SIG_new(OQS_SIG_alg_dilithium_2);
    if (dilithium_sig == NULL) {
        fprintf(stderr, "Error: OQS_SIG_new failed for Dilithium2.\n");
        exit(1);
    }
       uint8_t signature[OQS_SIG_dilithium_2_length_signature];
    uint8_t public_key[OQS_SIG_dilithium_2_length_public_key];
    uint8_t secret_key[OQS_SIG_dilithium_2_length_secret_key];
    FILE *file2 = fopen("dilithium2_time.csv", "w");
if (file2 == NULL) {
```

```
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }
                fprintf(file2, "Alg_name,Signing Time (ms),Verifying Time (ms)\n");
            fclose(file2);


        for (int i = 0; i < runs; i++) {
            dilithium_sign_times[i] = dilithium2_sign(message, MESSAGE_LEN, signature,
&signature_len, dilithium_sig, public_key, bytes_read, secret_key);
            dilithium_verify_times[i] = dilithium2_verify(message, MESSAGE_LEN, signature,
signature_len, dilithium_sig, public_key, bytes_read);
        }
        OQS_SIG_free(dilithium_sig);
        write_to_csv("Dilithium2", "dilithium2_time.csv", dilithium_sign_times,
dilithium_verify_times, runs);
        write_to_csv("Dilithium2", "time_of_both_algorithms.csv", dilithium_sign_times,
dilithium_verify_times, runs);
    }

    if (choice == 2 || choice == 3) {
        OQS_SIG *falcon_sig = OQS_SIG_new(OQS_SIG_alg_falcon_512);
        if (falcon_sig == NULL) {
            fprintf(stderr, "Error: OQS_SIG_new failed for Falcon-512.\n");
            exit(1);
        }
        FILE *file4 = fopen("falcon512_time.csv", "w");
    if (file4 == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
              fprintf(file4, "Alg_name,Signing Time (ms),Verifying Time (ms)\n");
    }
            fclose(file4);
            uint8_t signature[OQS_SIG_falcon_512_length_signature];
        uint8_t public_key[OQS_SIG_falcon_512_length_public_key];
        uint8_t secret_key[OQS_SIG_falcon_512_length_secret_key];

        for (int i = 0; i < runs; i++) {
            falcon_sign_times[i] = falcon512_sign(message, MESSAGE_LEN, signature,
&signature_len, falcon_sig, public_key, bytes_read, secret_key);
            falcon_verify_times[i] = falcon512_verify(message, MESSAGE_LEN, signature,
```

```
signature_len, falcon_sig, public_key, bytes_read);
        }
        OQS_SIG_free(falcon_sig);
        write_to_csv("Falcon-512", "falcon512_time.csv", falcon_sign_times, falcon_verify_times,
runs);
        write_to_csv("Falcon-512", "time_of_both_algorithms.csv", falcon_sign_times,
falcon_verify_times, runs);
    }



    return 0;
}

// Function to measure time
double measure_time(clock_t start, clock_t end) {
    return ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;
}

// Function to calculate mean
double calculate_mean(double *times, int length) {
    double sum = 0.0;
    for (int i = 0; i < length; ++i) {
        sum += times[i];
    }
    return sum / length;
}

// Function to write times to CSV
void write_to_csv(const char *alg_name, const char *filename, double *sign_times, double
*verify_times, int k) {
    FILE *file = fopen(filename, "a");
    if (file == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }

    // Write the times for each run
    for (int i = 0; i < k; ++i) {
        fprintf(file, "%s,%.6f,%.6f\n", alg_name, sign_times[i], verify_times[i]);
    }
```

```
    // Calculate and write the mean times
    double mean_sign_time = calculate_mean(sign_times, k);
    double mean_verify_time = calculate_mean(verify_times, k);
    fprintf(file, "Mean_%s,%.6f,%.6f\n", alg_name, mean_sign_time, mean_verify_time);

    fclose(file);
}

// Function to sign using Dilithium2
double dilithium2_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t
*signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read,uint8_t *secret_key) {
    OQS_STATUS rc;

    // Generate keypair
    rc = OQS_SIG_keypair(sig, public_key, secret_key);
    if (rc != OQS_SUCCESS) {
        fprintf(stderr, "ERROR: OQS_SIG_keypair failed!\n");
        exit(1);
    }

    // Display the message to be signed
    printf("Message to be signed: ");
    for (size_t i = 0; i < bytes_read; i++) {
        printf("%02X", message[i]);
    }
    printf("\n");

    // Sign the message
    clock_t start = clock();
    rc = OQS_SIG_sign(sig, signature, signature_len, message, message_len, secret_key);
    clock_t end = clock();
    if (rc != OQS_SUCCESS) {
        fprintf(stderr, "ERROR: OQS_SIG_sign failed!\n");
        exit(1);
    }

    // Display the generated signature
    printf("Generated signature: ");
    for (size_t i = 0; i < *signature_len; i++) {
        printf("%02X", signature[i]);
```

```
    }
    printf("\n");

    return measure_time(start, end);
}

// Function to verify using Dilithium2
double dilithium2_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t
signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read) {
    clock_t start = clock();
    OQS_STATUS rc = OQS_SIG_verify(sig, message, message_len, signature, signature_len,
public_key);
    clock_t end = clock();

    // Display the message being verified
    printf("Message being verified: ");
    for (size_t i = 0; i < bytes_read; i++) {
        printf("%02X", message[i]);
    }
    printf("\n");

    // Display verification status
    if (rc == OQS_SUCCESS) {
        printf("Verification successful.\n");
    } else {
        printf("Verification failed.\n");
    }

    return measure_time(start, end);
}


// Function to sign using Falcon-512
double falcon512_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t
*signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read,uint8_t *secret_key) {
    OQS_STATUS rc;

    // Generate keypair
    rc = OQS_SIG_keypair(sig, public_key, secret_key);
    if (rc != OQS_SUCCESS) {
        fprintf(stderr, "ERROR: OQS_SIG_keypair failed!\n");
```

```
        exit(1);
    }

    // Display the message to be signed
    printf("Message to be signed: ");
    for (size_t i = 0; i < bytes_read; i++) {
        printf("%02X", message[i]);
    }
    printf("\n");

    // Sign the message
    clock_t start = clock();
    rc = OQS_SIG_sign(sig, signature, signature_len, message, message_len, secret_key);
    clock_t end = clock();
    if (rc != OQS_SUCCESS) {
        fprintf(stderr, "ERROR: OQS_SIG_sign failed!\n");
        exit(1);
    }

    // Display the generated signature
    printf("Generated signature: ");
    for (size_t i = 0; i < *signature_len; i++) {
        printf("%02X", signature[i]);
    }
    printf("\n");

    return measure_time(start, end);
}


// Function to verify using Falcon-512
double falcon512_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t
signature_len, OQS_SIG *sig, uint8_t *public_key, int bytes_read) {
    clock_t start = clock();
    OQS_STATUS rc = OQS_SIG_verify(sig, message, message_len, signature, signature_len,
public_key);
    clock_t end = clock();

    // Display the message being verified
    printf("Message being verified: ");
    for (size_t i = 0; i < bytes_read; i++) {
```

```
        printf("%02X", message[i]);
    }
    printf("\n");

    // Display verification status
    if (rc == OQS_SUCCESS) {
        printf("Verification successful.\n");
    } else {
        printf("Verification failed.\n");
    }

    return measure_time(start, end);
}
```

Appendix C

//OpenSSL Encryptio/decryption

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <time.h>

// Buffer sizes for different schemes
#define kyber_768_SIZE 32
#define hqc_128_SIZE 64
#define frodokem_640_aes_SIZE 16
#define bike_l1_SIZE 32
#define classic_mceliece_348864_SIZE 32
#define BUFFER_SIZE 10000

// Function prototypes
double aes_256_cbc_encrypt(uint8_t *input_buffer, size_t buffer_size, uint8_t *encrypted_input,
int *encrypted_len, uint8_t *key, uint8_t *iv);
double aes_256_cbc_decrypt(uint8_t *encrypted_input, int encrypted_len, const char
*output_filename, uint8_t *key, uint8_t *iv);
double des_ede3_cbc_encrypt(uint8_t *input_buffer, size_t buffer_size, uint8_t
*encrypted_input, int *encrypted_len, uint8_t *key, uint8_t *iv);
double des_ede3_cbc_decrypt(uint8_t *encrypted_input, int encrypted_len, const char
*output_filename, uint8_t *key, uint8_t *iv);
void display_content(const char *label, uint8_t *content, int length);
double measure_time(clock_t start, clock_t end);
double calculate_mean(double *times, int length);
void write_to_csv(const char *alg_name, const char *filename, double *enc_times, double
*dec_times, int k, const char *buffer_name);
void generate_random_buffer(uint8_t *buffer, size_t size);

// Function to display content as hexadecimal
void display_content(const char *label, uint8_t *content, int length) {
    printf("%s:", label);
    for (int i = 0; i < length; ++i) {
        printf("%02x", content[i]);
    }
```

```
    printf("\n");
}

int main() {
    // Initialize OpenSSL library
    OpenSSL_add_all_algorithms();

    uint8_t encrypted_input[BUFFER_SIZE + EVP_MAX_BLOCK_LENGTH];
    int encrypted_len;
    uint8_t key[EVP_MAX_KEY_LENGTH];
    uint8_t iv[EVP_MAX_IV_LENGTH];

    // Generate random key and IV
    RAND_bytes(key, EVP_MAX_KEY_LENGTH);
    RAND_bytes(iv, EVP_MAX_IV_LENGTH);

    int runs;
    printf("Enter the number of runs: ");
    scanf("%d", &runs);

    // Define buffer sizes and corresponding names
    size_t buffer_sizes[] = {kyber_768_SIZE, hqc_128_SIZE, frodokem_640_aes_SIZE,
bike_l1_SIZE, classic_mceliece_348864_SIZE};
    const char *buffer_names[] = {"kyber_768 ", "hqc_128", "frodokem_640_aes", "bike_l1",
"classic_mceliece_348864"};
    const int num_schemes = sizeof(buffer_sizes) / sizeof(buffer_sizes[0]);

    double aes_enc_times[runs], aes_dec_times[runs];
    double des_enc_times[runs], des_dec_times[runs];

    FILE *file_all = fopen("Openssl_algs_all.csv", "w");
    if (file_all == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return 1;
    }
    fprintf(file_all, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file_all);

    FILE *file_des = fopen("destime.csv", "w");
    if (file_des == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
```

```
        return 1;
    }
    fprintf(file_des, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file_des);

    FILE *file_aes = fopen("aestime.csv", "w");
    if (file_aes == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return 1;
    }
    fprintf(file_aes, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file_aes);

    // Run the algorithms for each buffer size and corresponding name
    for (int s = 0; s < num_schemes; s++) {
        size_t buffer_size = buffer_sizes[s];
        const char *buffer_name = buffer_names[s];

        uint8_t input_buffer[buffer_size];
        generate_random_buffer(input_buffer, buffer_size);

        for (int i = 0; i < runs; i++) {
            aes_enc_times[i] = aes_256_cbc_encrypt(input_buffer, buffer_size, encrypted_input,
&encrypted_len, key, iv);
            aes_dec_times[i] = aes_256_cbc_decrypt(encrypted_input, encrypted_len,
"decrypted_aes.txt", key, iv);
            printf("AES Run %d (buffer %s) encryption time %.6f\n", i + 1, buffer_name,
aes_enc_times[i]);
            printf("AES Run %d (buffer %s) decryption time %.6f\n", i + 1, buffer_name,
aes_dec_times[i]);
        }
        write_to_csv("aes_256_cbc", "aestime.csv", aes_enc_times, aes_dec_times, runs,
buffer_name);
        write_to_csv("aes_256_cbc", "Openssl_algs_all.csv", aes_enc_times, aes_dec_times, runs,
buffer_name);

        for (int i = 0; i < runs; i++) {
            des_enc_times[i] = des_ede3_cbc_encrypt(input_buffer, buffer_size, encrypted_input,
&encrypted_len, key, iv);
            des_dec_times[i] = des_ede3_cbc_decrypt(encrypted_input, encrypted_len,
"decrypted_des.txt", key, iv);
```

```
            printf("DES Run %d (buffer %s) encryption time %.6f\n", i + 1, buffer_name,
des_enc_times[i]);
            printf("DES Run %d (buffer %s) decryption time %.6f\n", i + 1, buffer_name,
des_dec_times[i]);
        }
        write_to_csv("des-ede-3-cbc", "destime.csv", des_enc_times, des_dec_times, runs,
buffer_name);
        write_to_csv("des-ede-3-cbc", "Openssl_algs_all.csv", des_enc_times, des_dec_times, runs,
buffer_name);
    }

    EVP_cleanup();

    return 0;
}

// Function to calculate mean
double calculate_mean(double *times, int length) {
    double sum = 0.0;
    for (int i = 0; i < length; ++i) {
        sum += times[i];
    }
    return sum / length;
}

// Function to write times to CSV
void write_to_csv(const char *alg_name, const char *filename, double *enc_times, double
*dec_times, int k, const char *buffer_name) {
    FILE *file = fopen(filename, "a");
    if (file == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }

    for (int i = 0; i < k; ++i) {
        fprintf(file, "%s_%s,%.6f,%.6f\n", alg_name, buffer_name, enc_times[i], dec_times[i]);
    }

    double mean_enc_time = calculate_mean(enc_times, k);
    double mean_dec_time = calculate_mean(dec_times, k);
    fprintf(file, "Mean_%s_%s,%.6f,%.6f\n", alg_name, buffer_name, mean_enc_time,
```

```
mean_dec_time);

    fclose(file);
}

// Function to measure time
double measure_time(clock_t start, clock_t end) {
    return ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;
}

// Function to generate random buffer
void generate_random_buffer(uint8_t *buffer, size_t size) {
    RAND_bytes(buffer, size);
}

// aes_256_cbc Encryption
double aes_256_cbc_encrypt(uint8_t *input_buffer, size_t buffer_size, uint8_t *encrypted_input,
int *encrypted_len, uint8_t *key, uint8_t *iv) {
    display_content("Input Content (aes_256_cbc)", input_buffer, buffer_size);

    clock_t start_enc = clock();
    EVP_CIPHER_CTX *ctx_enc = EVP_CIPHER_CTX_new();
    EVP_EncryptInit_ex(ctx_enc, EVP_aes_256_cbc(), NULL, key, iv);
    int out_len;
    EVP_EncryptUpdate(ctx_enc, encrypted_input, &out_len, input_buffer, buffer_size);
    int final_len;
    EVP_EncryptFinal_ex(ctx_enc, encrypted_input + out_len, &final_len);
    EVP_CIPHER_CTX_free(ctx_enc);
    clock_t end_enc = clock();
    *encrypted_len = out_len + final_len;

    display_content("Encrypted Content (aes_256_cbc)", encrypted_input, *encrypted_len);

    return measure_time(start_enc, end_enc);
}

// aes_256_cbc Decryption
double aes_256_cbc_decrypt(uint8_t *encrypted_input, int encrypted_len, const char
*output_filename, uint8_t *key, uint8_t *iv) {
    clock_t start_dec = clock();
    EVP_CIPHER_CTX *ctx_dec = EVP_CIPHER_CTX_new();
```

```
    EVP_DecryptInit_ex(ctx_dec, EVP_aes_256_cbc(), NULL, key, iv);
    uint8_t decrypted_text[BUFFER_SIZE + EVP_MAX_BLOCK_LENGTH];
    int out_len_dec;
    EVP_DecryptUpdate(ctx_dec, decrypted_text, &out_len_dec, encrypted_input,
encrypted_len);
    int final_len_dec;
    EVP_DecryptFinal_ex(ctx_dec, decrypted_text + out_len_dec, &final_len_dec);
    EVP_CIPHER_CTX_free(ctx_dec);
    clock_t end_dec = clock();
    int decrypted_len = out_len_dec + final_len_dec;

    display_content("Decrypted Content (aes_256_cbc)", decrypted_text, decrypted_len);

    FILE *output_file = fopen(output_filename, "w");
    if (output_file != NULL) {
        fwrite(decrypted_text, 1, decrypted_len, output_file);
        fclose(output_file);
    }

    return measure_time(start_dec, end_dec);
}

// des-ede3-cbc Encryption
double des_ede3_cbc_encrypt(uint8_t *input_buffer, size_t buffer_size, uint8_t
*encrypted_input, int *encrypted_len, uint8_t *key, uint8_t *iv) {
    display_content("Input Content (des-ede3-cbc)", input_buffer, buffer_size);

    clock_t start_enc = clock();
    EVP_CIPHER_CTX *ctx_enc = EVP_CIPHER_CTX_new();
    EVP_EncryptInit_ex(ctx_enc, EVP_des_ede3_cbc(), NULL, key, iv);
    int out_len;
    EVP_EncryptUpdate(ctx_enc, encrypted_input, &out_len, input_buffer, buffer_size);
    int final_len;
    EVP_EncryptFinal_ex(ctx_enc, encrypted_input + out_len, &final_len);
    EVP_CIPHER_CTX_free(ctx_enc);
    clock_t end_enc = clock();
    *encrypted_len = out_len + final_len;

    display_content("Encrypted Content (des-ede3-cbc)", encrypted_input, *encrypted_len);

    return measure_time(start_enc, end_enc);
```

```
}

// des-ede3-cbc Decryption
double des_ede3_cbc_decrypt(uint8_t *encrypted_input, int encrypted_len, const char
*output_filename, uint8_t *key, uint8_t *iv) {
    clock_t start_dec = clock();
    EVP_CIPHER_CTX *ctx_dec = EVP_CIPHER_CTX_new();
    EVP_DecryptInit_ex(ctx_dec, EVP_des_ede3_cbc(), NULL, key, iv);
    uint8_t decrypted_text[BUFFER_SIZE + EVP_MAX_BLOCK_LENGTH];
    int out_len_dec;
    EVP_DecryptUpdate(ctx_dec, decrypted_text, &out_len_dec, encrypted_input,
encrypted_len);
    int final_len_dec;
    EVP_DecryptFinal_ex(ctx_dec, decrypted_text + out_len_dec, &final_len_dec);
    EVP_CIPHER_CTX_free(ctx_dec);
    clock_t end_dec = clock();
    int decrypted_len = out_len_dec + final_len_dec;

    display_content("Decrypted Content (des-ede3-cbc)", decrypted_text, decrypted_len);

    FILE *output_file = fopen(output_filename, "w");
    if (output_file != NULL) {
        fwrite(decrypted_text, 1, decrypted_len, output_file);
        fclose(output_file);
    }

    return measure_time(start_dec, end_dec);
}
```

Appendix D

//signature and verfication performance measurement with OpenSSL

```
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/ec.h>
#include <openssl/rsa.h>
#include <openssl/rand.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MESSAGE_LEN 1024

/* Function prototypes */
double measure_time(clock_t start, clock_t end);
double calculate_mean(double *times, int length);
void write_to_csv(const char *alg_name, const char *filename, double *sign_times, double
*verify_times, int k);
double rsa_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t *signature_len,
EVP_PKEY *pkey);
double rsa_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t signature_len,
EVP_PKEY *pkey);
double ecdsa_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t
*signature_len, EVP_PKEY *pkey);
double ecdsa_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t
signature_len, EVP_PKEY *pkey);
void print_uint8_t(const uint8_t *data, size_t len);
void read_file(const char *filename, uint8_t *buffer, size_t *len);

int main() {
    int choice;
    int runs;
    uint8_t message[MESSAGE_LEN];
    size_t message_len = MESSAGE_LEN;
    uint8_t signature[256];
    size_t signature_len;

    // Read the content from the input file
```

```
    read_file("input.txt", message, &message_len);

    printf("Choose the algorithm:\n1. SHA256_with_RSA\n2. SHA256 with ECDSA\n3. Both\
nEnter your choice: ");
    scanf("%d", &choice);

    printf("Enter the number of runs: ");
    scanf("%d", &runs);

    double rsa_sign_times[runs], rsa_verify_times[runs];
    double ecdsa_sign_times[runs], ecdsa_verify_times[runs];
          FILE *file2 = fopen("rsa_time.csv", "w");
    if (file2 == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }

    fprintf(file2, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file2);

    FILE *file3 = fopen("ecdsa_time.csv", "w");
    if (file3 == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }

    fprintf(file3, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file3);

    FILE *file4 = fopen("time_of_both_algorithms.csv", "w");
    if (file4 == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }

    fprintf(file4, "alg_name,Encryption Time (ms),Decryption Time (ms)\n");
    fclose(file4);

    if (choice == 1 || choice == 3) {
        // Generate RSA key pair
        EVP_PKEY *rsa_pkey = EVP_PKEY_new();
```

```
        RSA *rsa = RSA_new();
        BIGNUM *e = BN_new();
        BN_set_word(e, RSA_F4);
        RSA_generate_key_ex(rsa, 2048, e, NULL);
        EVP_PKEY_assign_RSA(rsa_pkey, rsa);
        BN_free(e);
        uint8_t message1[MESSAGE_LEN];
           size_t message_len1 = MESSAGE_LEN;

        for (int i = 0; i < runs; i++) {


           rsa_sign_times[i] = rsa_sign(message, message_len, signature, &signature_len,
rsa_pkey);
           rsa_verify_times[i] = rsa_verify(message, message_len, signature, signature_len,
rsa_pkey);

        }
        EVP_PKEY_free(rsa_pkey);
        write_to_csv("sha256WithRSAEncryption", "rsa_time.csv", rsa_sign_times,
rsa_verify_times, runs);
        write_to_csv("sha256WithRSAEncryption", "time_of_both_algorithms.csv",
rsa_sign_times, rsa_verify_times, runs);
    }

    if (choice == 2 || choice == 3) {
        // Generate ECDSA key pair
        EVP_PKEY *ecdsa_pkey = EVP_PKEY_new();
        EC_KEY *ec_key = EC_KEY_new_by_curve_name(NID_X9_62_prime256v1);
        EC_KEY_generate_key(ec_key);
        EVP_PKEY_assign_EC_KEY(ecdsa_pkey, ec_key);

        for (int i = 0; i < runs; i++) {
           ecdsa_sign_times[i] = ecdsa_sign(message, message_len, signature, &signature_len,
ecdsa_pkey);
           ecdsa_verify_times[i] = ecdsa_verify(message, message_len, signature, signature_len,
ecdsa_pkey);
        }
        EVP_PKEY_free(ecdsa_pkey);
        write_to_csv("sha256WithECDSA", "ecdsa_time.csv", ecdsa_sign_times,
ecdsa_verify_times, runs);
```

```
        write_to_csv("sha256WithECDSA", "time_of_both_algorithms.csv", ecdsa_sign_times,
ecdsa_verify_times, runs);
    }

    return 0;
}

// Function to measure time
double measure_time(clock_t start, clock_t end) {
    return ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;
}

// Function to calculate mean
double calculate_mean(double *times, int length) {
    double sum = 0.0;
    for (int i = 0; i < length; ++i) {
        sum += times[i];
    }
    return sum / length;
}

// Function to write times to CSV
void write_to_csv(const char *alg_name, const char *filename, double *sign_times, double
*verify_times, int k) {
    FILE *file = fopen(filename, "a");
    if (file == NULL) {
        fprintf(stderr, "Error: Unable to open output file.\n");
        return;
    }

    // Write the header
    //fprintf(file, "%s\n", alg_name);
    //fprintf(file, "Algorithm_name,Signing Time (ms),Verifying Time (ms)\n");

    // Write the times for each run
    for (int i = 0; i < k; ++i) {
        fprintf(file, "%s,%.6f,%.6f\n",alg_name, sign_times[i], verify_times[i]);
    }

    // Calculate and write the mean times
    double mean_sign_time = calculate_mean(sign_times, k);
```

```
    double mean_verify_time = calculate_mean(verify_times, k);
    fprintf(file, "Mean_%s,%.6f,%.6f\n", alg_name,mean_sign_time, mean_verify_time);

    fclose(file);
}

// Function to sign using RSA
double rsa_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t *signature_len,
EVP_PKEY *pkey) {
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
    if (mdctx == NULL) {
        fprintf(stderr, "Error: EVP_MD_CTX_new failed.\n");
        exit(1);
    }

    if (EVP_DigestSignInit(mdctx, NULL, EVP_sha256(), NULL, pkey) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignInit failed.\n");
        exit(1);
    }

    if (EVP_DigestSignUpdate(mdctx, message, message_len) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignUpdate failed.\n");
        exit(1);
    }

    size_t req = 0;
    if (EVP_DigestSignFinal(mdctx, NULL, &req) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignFinal (preliminary) failed.\n");
        exit(1);
    }

    *signature_len = req;
    clock_t start = clock();
    if (EVP_DigestSignFinal(mdctx, signature, signature_len) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignFinal failed.\n");
        exit(1);
    }
    clock_t end = clock();

    EVP_MD_CTX_free(mdctx);
    return measure_time(start, end);
```

```
}

// Function to verify using RSA
double rsa_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t signature_len,
EVP_PKEY *pkey) {
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
    if (mdctx == NULL) {
        fprintf(stderr, "Error: EVP_MD_CTX_new failed.\n");
        exit(1);
    }

    if (EVP_DigestVerifyInit(mdctx, NULL, EVP_sha256(), NULL, pkey) <= 0) {
        fprintf(stderr, "Error: EVP_DigestVerifyInit failed.\n");
        exit(1);
    }

    if (EVP_DigestVerifyUpdate(mdctx, message, message_len) <= 0) {
        fprintf(stderr, "Error: EVP_DigestVerifyUpdate failed.\n");
        exit(1);
    }

    clock_t start = clock();
    int rc = EVP_DigestVerifyFinal(mdctx, signature, signature_len);
    clock_t end = clock();

    if (rc == 1) {
        printf("Verification successful.\n");
    } else if (rc == 0) {
        printf("Verification failed.\n");
    } else {
        fprintf(stderr, "Error: EVP_DigestVerifyFinal failed.\n");
        exit(1);
    }

    EVP_MD_CTX_free(mdctx);
    return measure_time(start, end);
}

// Function to sign using ECDSA
double ecdsa_sign(uint8_t *message, size_t message_len, uint8_t *signature, size_t
*signature_len, EVP_PKEY *pkey) {
```

```
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
    if (mdctx == NULL) {
        fprintf(stderr, "Error: EVP_MD_CTX_new failed.\n");
        exit(1);
    }

    if (EVP_DigestSignInit(mdctx, NULL, EVP_sha256(), NULL, pkey) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignInit failed.\n");
        exit(1);
    }

    if (EVP_DigestSignUpdate(mdctx, message, message_len) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignUpdate failed.\n");
        exit(1);
    }

    size_t req = 0;
    if (EVP_DigestSignFinal(mdctx, NULL, &req) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignFinal (preliminary) failed.\n");
        exit(1);
    }

    *signature_len = req;
    clock_t start = clock();
    if (EVP_DigestSignFinal(mdctx, signature, signature_len) <= 0) {
        fprintf(stderr, "Error: EVP_DigestSignFinal failed.\n");
        exit(1);
    }
    clock_t end = clock();

    EVP_MD_CTX_free(mdctx);
    return measure_time(start, end);
}

// Function to verify using ECDSA
double ecdsa_verify(uint8_t *message, size_t message_len, uint8_t *signature, size_t
signature_len, EVP_PKEY *pkey) {
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
    if (mdctx == NULL) {
        fprintf(stderr, "Error: EVP_MD_CTX_new failed.\n");
        exit(1);
```

```
    }

    if (EVP_DigestVerifyInit(mdctx, NULL, EVP_sha256(), NULL, pkey) <= 0) {
        fprintf(stderr, "Error: EVP_DigestVerifyInit failed.\n");
        exit(1);
    }

    if (EVP_DigestVerifyUpdate(mdctx, message, message_len) <= 0) {
        fprintf(stderr, "Error: EVP_DigestVerifyUpdate failed.\n");
        exit(1);
    }

    clock_t start = clock();
    int rc = EVP_DigestVerifyFinal(mdctx, signature, signature_len);
    clock_t end = clock();

    if (rc == 1) {
        printf("Verification successful.\n");
    } else if (rc == 0) {
        printf("Verification failed.\n");
    } else {
        fprintf(stderr, "Error: EVP_DigestVerifyFinal failed.\n");
        exit(1);
    }

    EVP_MD_CTX_free(mdctx);
    return measure_time(start, end);
}

// Function to print the contents of uint8_t array
void print_uint8_t(const uint8_t *data, size_t len) {
    for (size_t i = 0; i < len; ++i) {
        printf("%02X", data[i]);
    }
    printf("\n");
}

// Function to read the content of a file into a buffer
void read_file(const char *filename, uint8_t *buffer, size_t *len) {
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
```

```
        fprintf(stderr, "Error: Unable to open input file.\n");
        exit(1);
    }

    fseek(file, 0, SEEK_END);
    *len = ftell(file);
    fseek(file, 0, SEEK_SET);

    if (*len > MESSAGE_LEN) {
        fprintf(stderr, "Error: Input file is too large.\n");
        exit(1);
    }

    fread(buffer, 1, *len, file);
    fclose(file);
}
```